DESIGN AND ANALYSIS OF ALGORITHMS

CSE

IV SEMESTER

PPT ON
DESIGN AND ANALYSIS OF ALGORITHMS
IV SEM (IARE-R18)

UNIT 1
INTRODUCTION

ALGORITHMS

Algorithm Definition

Formal Definition

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms should satisfy the following criteria.

Input: Zero or more quantities are externally supplied.
Output: At least one quantity is produced.
Definiteness: Each instruction is clear and unambiguous.
Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
Effectiveness: Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Areas of study of Algorithm

- **How to devise or design an algorithm**

  It includes the study of various design techniques and helps in writing algorithms using the existing design techniques like divide and conquer.

- **How to validate an algorithm**

  After the algorithm is written it is necessary to check the correctness of the algorithm i.e for each input correct output is produced, known as algorithm validation. The second phase is writing a program known as program proving or program verification.

- **How to analysis an algorithm**

  It is known as analysis of algorithms or performance analysis, refers to the task of calculating time and space complexity of the algorithm.

Areas of study of Algorithm

- **How to test a program**

  It consists of two phases.

  1. Debugging is detection and correction of errors.
  2. Profiling or performance measurement is the actual amount of time required by the program to compute the result.

ALGORITHM SPECIFICATION

ALGORITHM SPECIFICATION

Pseudo-Code for writing Algorithms:

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces {and}.
3. An identifier begins with a letter. The data types of variables are not explicitly declared.
4. Compound data types can be formed with records. Here is an example,

```
Node. Record
{
    data type – 1   data-1;
    . . . . . .
    . . . . . .
    data type – n  data – n;
    node * link;
}
```

Here link is a pointer to the record type node. Individual data items of a record can be accessed with → and period.

ALGORITHM SPECIFICATION

5. Assignment of values to variables is done using the assignment statement.
   <Variable>:= <expression>;

6. There are two Boolean values TRUE and FALSE.
   Logical Operators AND, OR, NOT
   Relational Operators <, <=,>,>=, =, !=

7. The following looping statements are employed.

   For, while and repeat-until

   While Loop:

```
While < condition >do{
        <statement-1>
              .                    .
        <statement-n>
}
```

ALGORITHM SPECIFICATION

For Loop:

for variable: = value-1 to value-2 step step do
{
    <statement-1>
        .
        .
    <statement-n>
}

One step is a key word, other Step is used for increment or decrement

ALGORITHM SPECIFICATION

repeat-until:

repeat{

<statement-1>

.

.

<statement-n>

}until<condition>

ALGORITHM SPECIFICATION

8. A conditional statement has the following forms.

(1) If <condition> then <statement>

(2) If <condition> then <statement-1>

Else <statement-2>

Case statement:

Case

{ :<condition-1>:<statement-1>

.

.

:<condition-n>:<statement-n>

:else:<statement-n+1>

}

ALGORITHM SPECIFICATION

9. Input and output are done using the instructions read & write.

10. There is only one type of procedure:
    Algorithm, the heading takes the form,
        Algorithm Name (<Parameter list>)

EXAMPLE

```
Algorithm Max(A,n)
 // A is an array of size n
{
        Result := A[1];
        for I:= 2 to n do
            if A[I] > Result then
                Result :=A[I];
         return Result;
}
```

Performance Analysis:

Performance Analysis:

- Performance of an algorithm is a process of making evaluative judgment about algorithms.
- Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.
- That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.

- We compare all algorithms with each other which are solving same problem, to select best algorithm.
- To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, execution speed of that algorithm, easy to understand, easy to implement, etc.

Performance Analysis:

Generally, the performance of an algorithm depends on the following elements...

- Whether that algorithm is providing the exact solution for the problem?
- Whether it is easy to understand?
- Whether it is easy to implement?
- How much space (memory) it requires to solve the problem?
- How much time it takes to solve the problem? Etc.,

When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all remaining elements.

- Performance analysis of an algorithm is the process of calculating space required by that algorithm and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

- Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
- Time required to complete the task of that algorithm (**Time Complexity**)

Performance Analysis:

Performance evaluation can be divided into two major phases.

1. Performance Analysis (machine independent)

Space Complexity:

The space complexity of an algorithm is the amount of memory it needs to run for completion.

Time Complexity:

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

2 . Performance Measurement (machine dependent).

Space Complexity

Space Complexity

The space complexity of an algorithm is the amount of memory it needs to run to completion.

The Space Complexity of any algorithm P is given by $S(P)=C+S_P(I)$,C is constant.

- Fixed Space Requirements (C)
  Independent of the characteristics of the inputs and outputs
  - ✓It includes instruction space
  - ✓space for simple variables, fixed-size structured variable, constants

- Variable Space Requirements ($S_P(I)$)
  depend on the instance characteristic I
  - ✓number, size, values of inputs and outputs associated with I
  - ✓recursive stack space, formal parameters, local variables, return address

EXAMPLE 1

Algorithm 1 : Simple arithmetic function

```
Algorithmabc( a,  b,  c)
{
   return a + b + b * c + (a + b - c) / (a + b) + 4.00;
 }
```

$S_P(I)=0$

Hence **S(P)=Constant**

EXAMPLE 2

Algorithm 2: Iterative function for sum a list of numbers

```
Algorithm sum( list[ ],  n)
{
      tempsum = 0;
      for i = 0 ton do
            tempsum += list [i];
     return tempsum;
}
```

In the above example list[] is dependent on n. Hence $S_P(I)=n$. The remaining variables are i, n, tempsum each requires one location. Hence **S(P)=3+n**

EXAMPLE 3

Algorithm 3: Recursive function for sum a list of numbers

```
Algorithmrsum( list[ ],  n)
{
If  (n<=0) then
return 0.0
else
 return rsum(list, n-1) + list[n];

 }
```

In the above example the recursion stack space includes space for formal parameters local variables and return address. Each call to rsum requires 3 locations i.e. for list[ ],n and return address .As the length of recursion is n+1.

S(P)>=3(n+1)

Time Complexity

Time Complexity

The time complexity of an algorithm is the amount of computer time it needs to run to completion.

The time T(P) taken by a program P is the sum of the compile time and the run (or execution)time. The compile time does not depend on the instance characteristics.

$$T(P)=C+T_P(I)$$

It is combination of

-Compile time (C) independent of instance characteristics

-Run (execution) time T_P dependent of instance characteristics

Time complexity is calculated in terms of *program step* as it is difficult to know the complexities of individual operations.

EXAMPLE 1

Algorithm 1 : Iterative function for finding Sum

```
Algorithm sum( list[ ], n)
{
   tempsum := 0; count++; /* for assignment */
   for  i := 1  to n do
    {
        count++;            /*for the for loop */
      tempsum := tempsum + list[i]; count++;  /* for assignment */
    }
count++;        /* last execution of for */
    return tempsum;
count++;        /* for return */
}
```

Hence **T(n)=2n+3**

EXAMPLE 2

Algorithm 2 : Recursive sum

```
Algorithmrsum( list[ ],  n)
{
        count++;      /*for if conditional */
        if (n<=0) {
                count++;  /* for return */
        return 0.0 }
else
return rsum(list, n-1) + list[n];

        count++;/*for return and rsum invocation*/

}
```

T(n)=2n+2

EXAMPLE 3

Algorithm 3: Matrix addition

```
Algorithm add( a[ ][MAX_SIZE],  b[ ][MAX_SIZE],
                        c[ ][MAX_SIZE],  rows, cols )
{
   for i := 1 to rows do {
count++; /* for i for loop */
       for  j :=  1 to cols do {
count++; /* for j for loop */
          c[i][j] := a[i][j] + b[i][j];
count++; /* for assignment statement */
       }
count++;    /* last time of j for loop */
  }
count++;         /* last time of i for loop */
}
```

T(n)=2rows*cols+2*rows+1

Time complexity

Computing Time Complexity

Tabular method for computing Time Complexity :

- ✓ Complexity is determined by using a table which includes steps per execution(s/e) i.e amount by which count changes as a result of execution of the statement.
- ✓ Frequency – number of times a statement is executed.

Example 1

Statement	s/e	Frequency	Total steps
Algorithm sum(list[ ], n)	0	-	0
{	0	-	0
tempsum := 0;	1	1	1
for i := 0 ton do	1	n+1	n+1
tempsum := tempsum + list [i];	1	n	n
return tempsum;	1	1	1
}	0	0	0
Total			2n+3

Example 1 : Iterative function for finding Sum

Example 2 : Recursive sum

Statement	s/e	Frequency		Total steps	
		n=0	n>0	n=0	n>0
Algorithmrsum(list[ ], n)	0	-	-	0	0
{	0	-	-	0	0
If (n<=0) then	1	1	1	1	1
return 0.0;	1	1	0	1	0
else	0	0	0	0	0
return rsum(list, n-1) + list[n];	1+x	0	1	0	1+x
}	0	0	0	0	0
Total				2	2+x

Example 3: Matrix addition

Statement	s/e	Frequency	Total steps
Algorithm add(a,b,c,m,n)	0	-	0
{	0	-	0
for i:=1 to m do	1	m+1	m+1
for j:=1 to n do	1	m(n+1)	mn+m
c[i,j]:=a[i,j]+b[i,j];	1	mn	mn
}	0	-	0
Total			2mn+2m+1

Time complexity Analysis

Time complexity Analysis

•The **worst-case complexity** of the algorithm is the function defined by the maximum number of steps taken on any instance of size *n*. It represents the curve passing through the highest point of each column.

•The **best-case complexity** of the algorithm is the function defined by the minimum number of steps taken on any instance of size *n*. It represents the curve passing through the lowest point of each column.

•Finally, **the average-case complexity** of the algorithm is the function defined by the average number of steps taken on any instance of size *n*.

Example: Sequential Search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

Efficiency of Sequential Search

Case	Total Comparisons
Worst case	n key comparisons
Best case	1 comparison
Average case	(n+1)/2

- Total number of elements in array are n

Asymptotic Notations

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

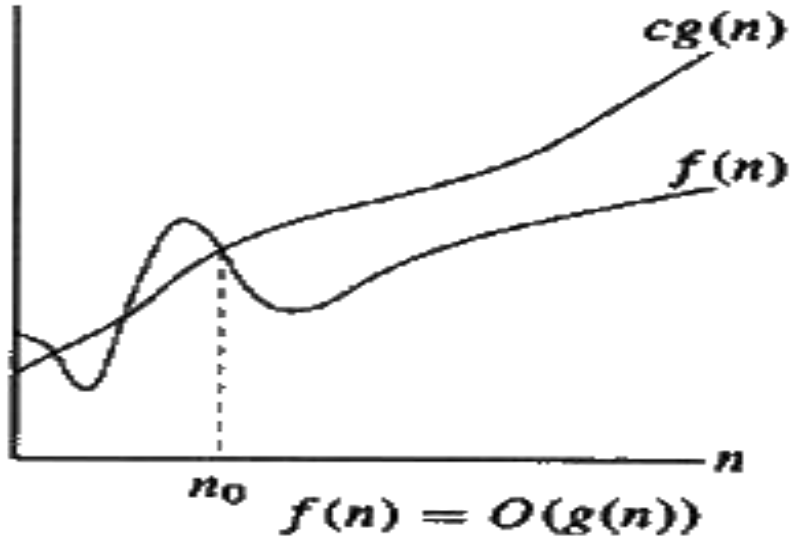
Big oh notation: O

Definition

The function f(n)=O(g(n)) (read as “f of n is big oh of g of n”) iff there exist positive constants c and n_0 such that

f(n)≤C*g(n) for all n, n≥0

The value g(n)is the upper bound value of f(n).



Example:

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent f(n) as O(g(n)) then it must satisfy

f(n) <= C x g(n) for all values of C > 0 and n0>= 1

f(n) <= C g(n)

⇒3n + 2 <= C n

Above condition is always TRUE for all values of C = 4 and n >= 2.

By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

3n+2=O(n) as

3n+2 ≤4n for all n≥2

Examples

```
void printFirstElementOfArray(int  arr[])
 {
        printf("First element of array = %d", arr[0]);
}
```

This function runs in O(1) time (or "constant time") relative to its input. The input array could be 1 item or 1,000 items, but this function would still just require one step.

Examples

```
void printAllElementOfArray(int arr[], int size)
 {
        for (int i = 0; i < size; i++)
        {
           printf("%d\n", arr[i]);
        }
}
```

This function runs in O(n) time (or "linear time"), where nn is the number of items in the array. If the array has 10 items, we have to print 10 times. If it has 1000 items, we have to print 1000 times.

Examples

```
void printAllPossibleOrderedPairs(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
    }
}
```

Here we're nesting two loops. If our array has n items, our outer loop runs n times and our inner loop runs n times for each iteration of the outer loop, giving us n^2 total prints. Thus this function runs in $O(n^2)$ time (or "quadratic time"). If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times.

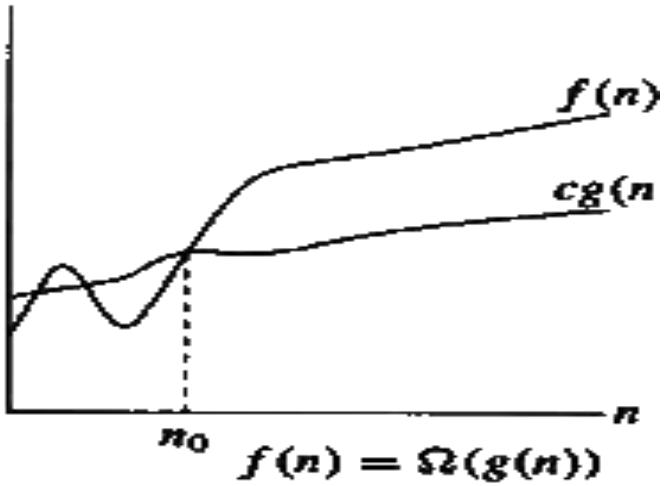
Asymptotic Notations

Algorithm Definition

Omega notation: Ω

The function f(n)=Ω (g(n)) (read as "f of n is Omega of g of n") iff there exist positive constants c and n_0 such that **f(n)≥C*g(n)** for all n, n≥0

The value g(n) is the lower bound value of f(n).



Example:

$3n+2=\Omega(n)$ as

$3n+2 \geq 3n$ for all $n \geq 1$

Consider the following f(n) and g(n)...

$f(n) = 3n + 2$

$g(n) = n$

If we want to represent f(n) as $\Omega(g(n))$ then it must satisfy $f(n) >= C\ g(n)$ for all values of $C > 0$ and $n0 >= 1$

$f(n) >= C\ g(n)$

$\Rightarrow 3n + 2 <= C\ n$

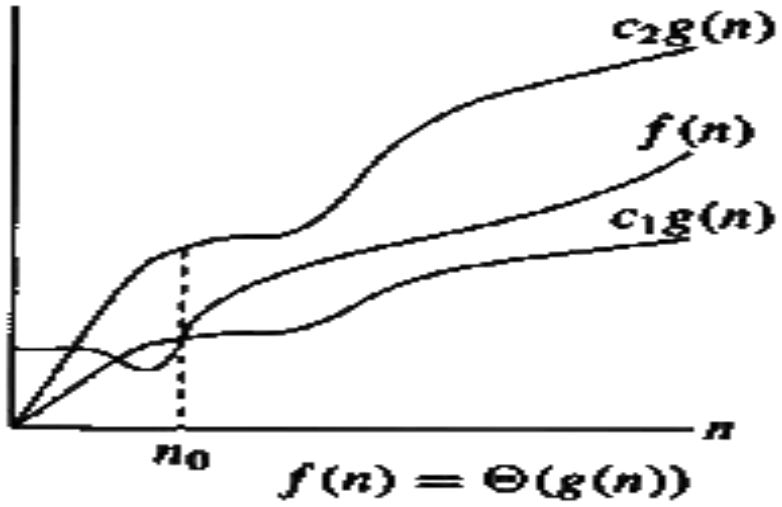
Above condition is always TRUE for all values of $C = 1$ and $n >= 1$.

By using Big - Omega notation we can represent the time complexity as follows...

$3n + 2 = \Omega(n)$

Theta notation: θ

The function f(n)= θ (g(n)) (read as "f of n is theta of g of n") iff there exist positive constants c1, c2 and n_0 such that **C1*g(n) ≤f(n)≤C2*g(n)** for all n, n≥0



Example

Example:

$3n+2=\boldsymbol{\theta}(n)$ as

$3n+2 \geq 3n$ for all $n \geq 2$

$3n+2 \leq 4n$ for all $n \geq 2$

Here $c1=3$ and $c2=4$ and $n_0=2$

Consider the following f(n) and g(n)...

f(n) = 3n + 2

g(n) = n

If we want to represent f(n) as Θ(g(n)) then it must satisfy C1 g(n) <= f(n) <= C2 g(n) for all values of C1, C2 > 0 and n0>= 1

C1 g(n) <= f(n) <= ⇒C2 g(n)

C1 n <= 3n + 2 <= C2 n

Above condition is always TRUE for all values of C1 = 1, C2 = 4 and n >= 1.

By using Big - Theta notation we can represent the time complexity as follows...

$3n + 2 = \Theta(n)$

Asymptotic Notations

Little oh: o

The function f(n)=o(g(n)) (read as “f of n is little oh of g of n”) iff

Lim f(n)/g(n)=0 for all n, n≥0

n→∞

Example:

3n+2=o(n^2) as

Lim ((3n+2)/n^2)=0

n→∞

EXAMPLES

```
void printAllItemsTwice(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
     }
    for (int i = 0; i < size; i++)
   {
        printf("%d\n", arr[i]);
    }
}
```

This is O(2n), which we just call O(n)

EXAMPLES

```
void printAllNumbersThenAllPairSums(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        printf("%d\n", arr[i]);
    }
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d\n", arr[i] + arr[j]);
        }
    }
}
```

Here our runtime is $O(n + n^2)$, which we just call $O(n^2)$

Probabilistic analysis

Probabilistic analysis

- In analysis of algorithms, **probabilistic analysis of algorithms** is an approach to estimate the computational complexity of an algorithm or a computational problem.
- It starts from an assumption about a probabilistic distribution of the set of all possible inputs.
- This assumption is then used to design an efficient algorithm or to derive the complexity of a known algorithm.
- This approach is not the same as that of probabilistic algorithms, but the two may be combined.

- For non-probabilistic, more specifically,
  for deterministic algorithms, the most common types
  of complexity estimates are

1. The average case complexity (**expected time complexity**),
   in which given an input distribution, the expected time of
   an algorithm is evaluated.
2. The **almost always** complexity estimates,
   in which given an input distribution, it is evaluated that the
   algorithm admits given complexity estimate that almost
   surely holds.

- In probabilistic analysis of probabilistic (randomized)
  algorithms, the distributions or averaging for all possible choices
  in randomized steps are also taken into an account, in addition to t
  he input distributions.

Amortized complexity

Amortized complexity

- The actual complexity of an operation is determined by the step count for that operation, and the actual complexity of a sequence of operations is determined by the step count for that sequence.
- The actual complexity of a sequence of operations may be determined by adding together the step counts for the individual operations in the sequence.
- Typically, determining the step count for each operation in the sequence is quite difficult, and instead, we obtain an upper bound on the step count for the sequence by adding together the worst-case step count for each operation.

Amortized complexity

1) Amortized cost of a sequence of operations can be seen as expenses of a salaried person. The average monthly expense of the person is less than or equal to the salary, but the person can spend more money in a particular month by buying a car or something. In other months, he or she saves money for the expensive month.

2) The above Amortized Analysis done for Dynamic Array example is called ***Aggregate Method***. There are two more powerful ways to do Amortized analysis called **Accounting Method** and **Potential Method**. We will be discussing the other two methods in separate posts.

Amortized complexity

3) The amortized analysis doesn't involve probability. There is also another different notion of average case running time where algorithms use randomization to make them faster and expected running time is faster than the worst case running time. These algorithms are analyzed using Randomized Analysis. Examples of these algorithms are Randomized Quick Sort, Quick Select and Hashing.

UNIT 2
SEARCHING AND TRAVERSAL TECHNIQUES

DIVIDE AND CONQUER

DIVIDE AND CONQUER

GENERAL METHOD

• Given a function to compute on ‘n’ inputs the divide-and-conquer strategy suggests splitting the inputs into ‘k’ distinct subsets, 1<k<=n, yielding ‘k’ sub problems.

• These sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole.

• If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied. Often the sub problems resulting from a divide-and-conquer design are of the same type as the original problem.

DIVIDE AND CONQUER

• DAndC(Algorithm) is initially invoked as DandC(P), where ‘p’ is the problem to be solved.

• Small(P) is a Boolean-valued function that determines whether the i/p size is small enough that the answer can be computed without splitting. If this so, the function ‘S’ is invoked.

•Otherwise, the problem P is divided into smaller sub problems.

•These sub problems P1, P2 …P_k are solved by recursive application of DAndC.

DIVIDE AND CONQUER

• Combine is a function that determines the solution to p using the solutions to the ‘k’ sub problems. If the size of ‘p’ is n and the sizes of the ‘k’ sub problems are n1, n2 ….nk, respectively, then the computing time of DAndC is described by the recurrence relation.

$$T(n) = \begin{cases} g(n) & \text{n small} \\ T(n1)+T(n2)+\ldots\ldots\ldots\ldots+T(n_k)+f(n); & \text{otherwise.} \end{cases}$$

Where T(n) is the time for DAndC on any i/p of size ‘n’.

g(n) is the time of compute the answer directly for small i/ps.

f(n) is the time for dividing P & combining the solution to sub problems.

Control Abstract

```
Algorithm DAndC(P)
 {
        if small(P) then return S(P);
        else
        {
              divide P into smaller instances
               P1, P2… Pk, k>=1;
              Apply DAndC to each of these sub problems;
              return combine (DAndC(P1), DAndC(P2),……..,DAndC(Pk));
        }
}
```

DIVIDE AND CONQUER

The complexity of many divide-and-conquer algorithms is given by recurrence relation of the form

$T(n) = T(1)$ $\quad n=1$

$= aT(n/b)+f(n)$ $\quad n>1$

Where a & b are known constants.

We assume that T(1) is known & 'n' is a power of b(i.e., $n=b^k$)

One of the methods for solving any such recurrence relation is called the substitution method.

This method repeatedly makes substitution for each occurrence of the function. T is the right-hand side until all such occurrences disappear.

APPLICATIONS OF DIVIDE AND CONQUER

APPLICATIONS OF DIVIDE AND CONQUER

- ✓ Binary search
- ✓ Quick sort
- ✓ Merge sort
- ✓ Strassen's matrix multiplication.

BINARY SEARCH

- Given a list of n elements arranged in increasing order.

- The problem is to determine whether a given element is present in the list or not. If x is present then determine the position of x, otherwise position is zero.
- Divide and conquer is used to solve the problem.

- The value Small(p) is true if n=1.
- S(P)= i, if x=a[i], a[] is an array otherwise S(P)=0.
- If P has more than one element then it can be divided into sub-problems.

BINARY SEARCH

- Choose an index j and compare x with a_j. then there 3 possibilities
- (i). X=a[j]
- (ii) x<a[j] (x is searched in the list a[1]…a[j-1])
- (iii) x>a[j] (x is searched in the list a[j+1]…a[n]).
- And the same procedure is applied repeatedly until the solution is found or solution is zero.

BINARY SEARCH

```
Algorithm Binsearch(a, n, x)
 // Given an array a[1:n] of elements in non-decreasing
 //order, n>=0,determine whether 'x' is present and
// if so, return 'j' such that x=a[j]; else return 0.
{
      low:=1; high:=n;
      while (low<=high) do
      {
          mid:=[(low + high)/2];
          if (x<a[mid]) then high;
          else if(x>a[mid]) then
                low:=mid+1;
          else return mid;
       }
      return 0;     } //end
```

Example

Example

Let us select the 14 entries.

-15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

- Place them in a[1:14], and simulate the steps Binsearch goes through as it searches for different values of ‘x’.

- Only the variables, low, high & mid need to be traced as we simulate the algorithm.
- We try the following values for x: 151, -14 and 9.
- For 2 successful searches & 1 unsuccessful search.
- Table shows the traces of Binsearch on these 3 steps.

Example

Array Elements

-15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

X=151

low	high	mid
1	14	7
8	14	11
12	14	13
14	14	14

Found

Example

Array Elements

-15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

x=-14

low	high	mid
1	14	7
1	6	3
1	2	1
2	2	2
2	1	

Not found

Example

Array Elements

-15,-6,0,7,9,23,54,82,101,112,125,131,142,151.

x=9

low	high	mid
1	14	7
1	6	3
4	6	5

Found

Time Complexity of binary search

The complexity of binary search is **successful searches** is

- Worst case is θ(log n) or θ(log n)
- Average case is θ(log n) or θ(log n)
- Best case is θ(1) or θ(1)

Unsuccessful search is: θ(log n) for all cases.

QUICKSORT

QUICKSORT

Algorithm QUICKSORT(low, high)

```
// sorts the elements a(low), . . . . . , a(high) which reside in the
global array  A(1 :n) into //ascending order a (n + 1) is considered
to be defined and must be greater than all //elements in a(1 : n);
A(n + 1) = α*/
{
        If( low < high) then
        {
           j := PARTITION(a, low,high+1);
          // J is the position of the partitioning element
          QUICKSORT(low, j –1);
          QUICKSORT(j + 1 ,high);
        }
}
```

QUICKSORT

```
Algorithm PARTITION(a, m, p)
{
        V :=a(m); i :=m; j:=p;
        // a (m) is the partition element
        do
        {
            repeat
                  i  := i +1;
             until  (a(i)≥v);
            repeat
                  j  := j –1;
            until  (a(j)≤v);
            if (i < j) then INTERCHANGE(a, i,j)
         }  while (i ≥j);
```

```
        a[m]  :=a[j];a[j]:=V;
        return j;
}
```

Algorithm INTERCHANGE(a, i, j)

```
{
        p:= a[i];
        a[i]:=a[j];
        a[j]:=p;
}
```

Example

We are given array of n integers to sort:

40	20	10	80	60	50	7	30	100

Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

40	20	10	80	60	50	7	30	100

Partitioning Array

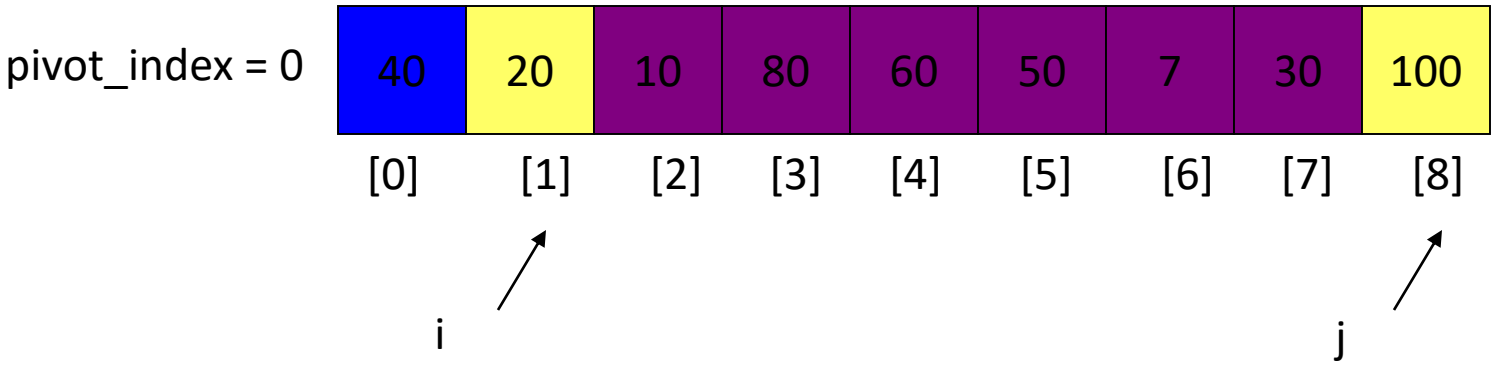
Given a pivot, partition the elements of the array such that the resulting array consists of:

1. One sub-array that contains elements >= pivot
2. Another sub-array that contains elements < pivot

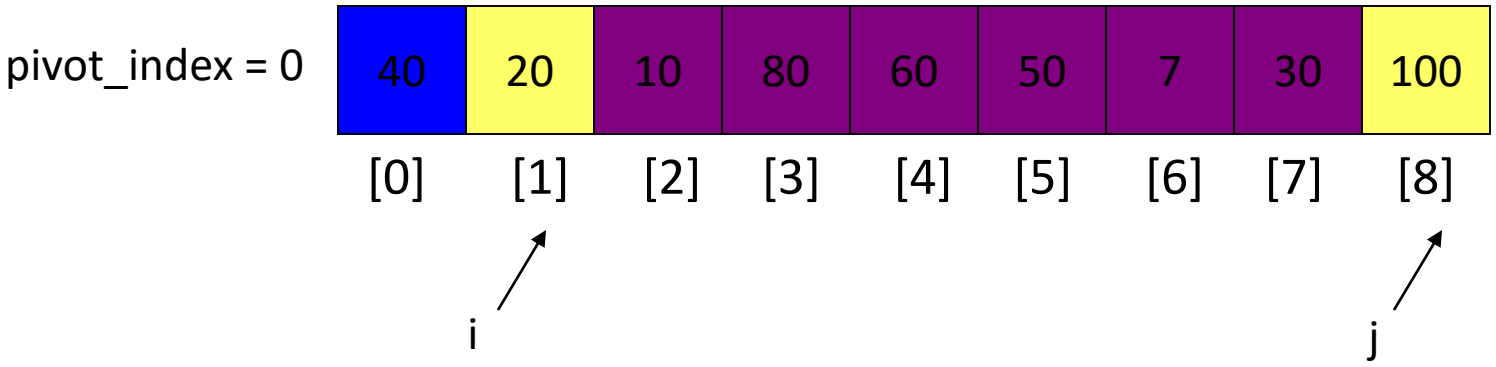
The sub-arrays are stored in the original a array.

Partitioning loops through, swapping elements below/above pivot.

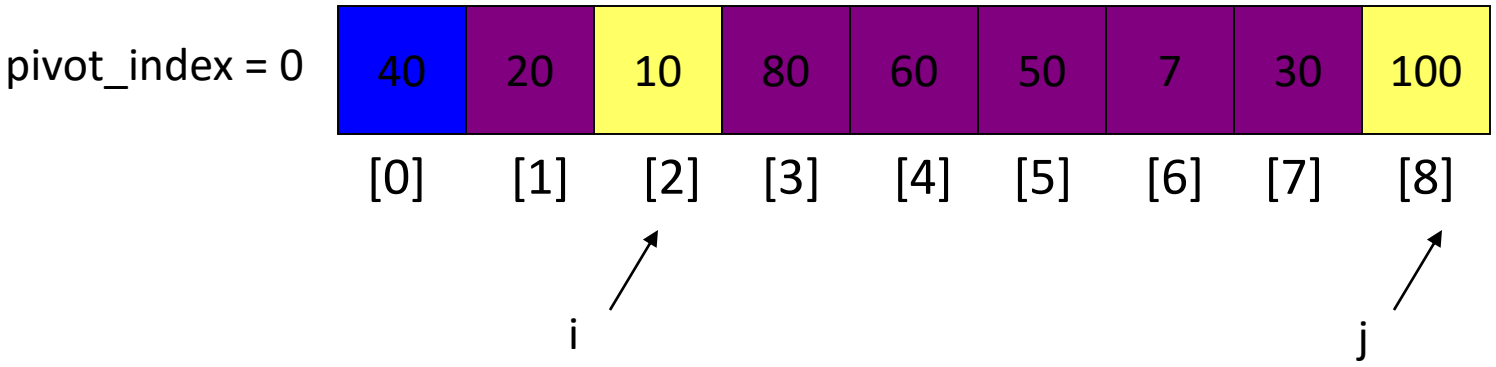
Example



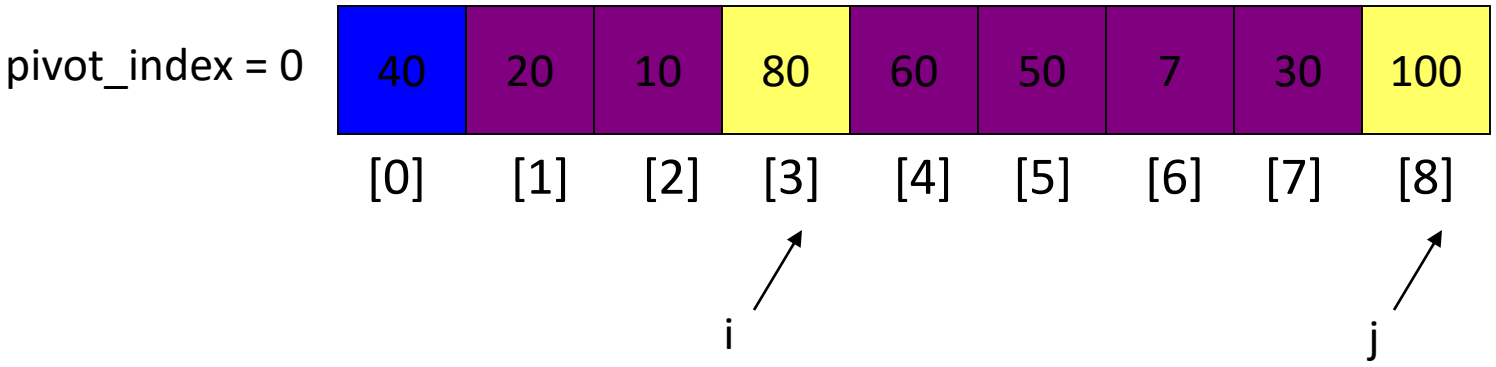
1. While a[i] <= a[pivot]
   ++ i



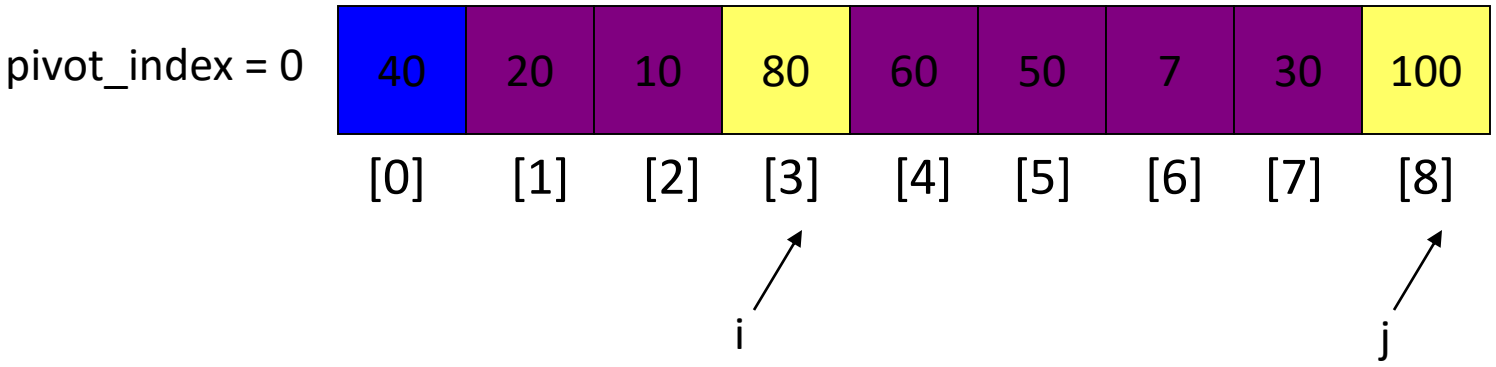
1. While a[i] <= a[pivot]
   ++ i



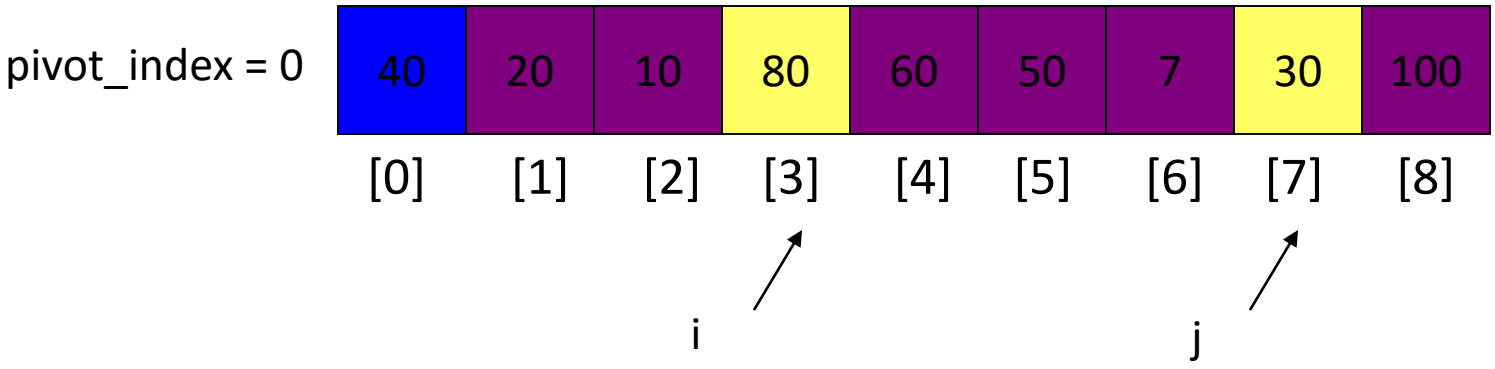
1. While a[i]<= a[pivot]
   ++ i



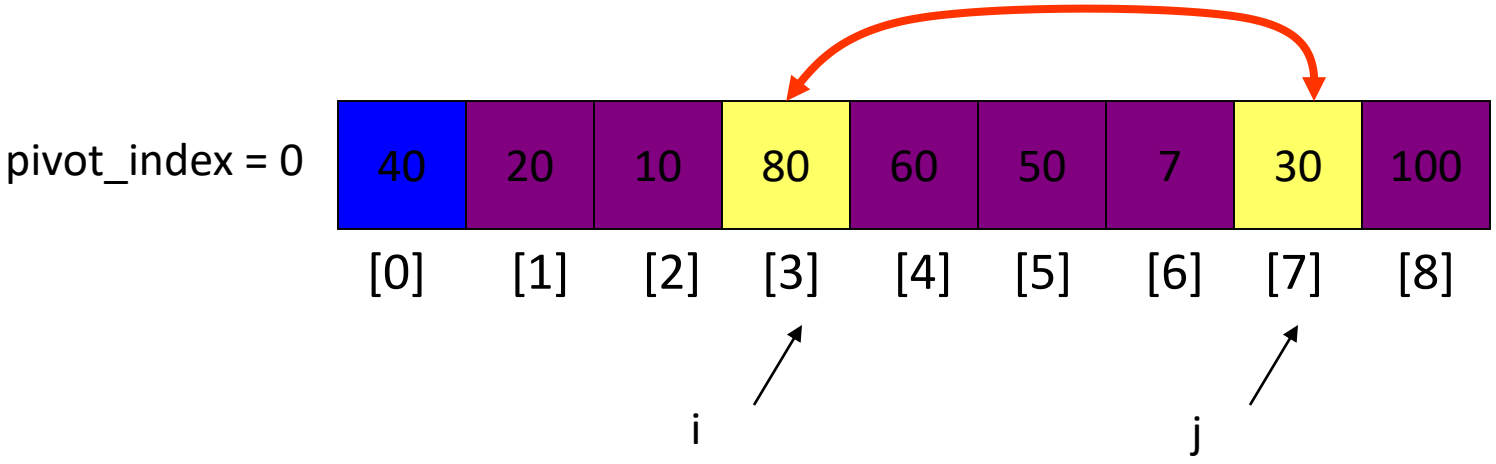
1. While a[i]<= a[pivot]
   ++ i
2. While a[j] > a[pivot]
   -- j



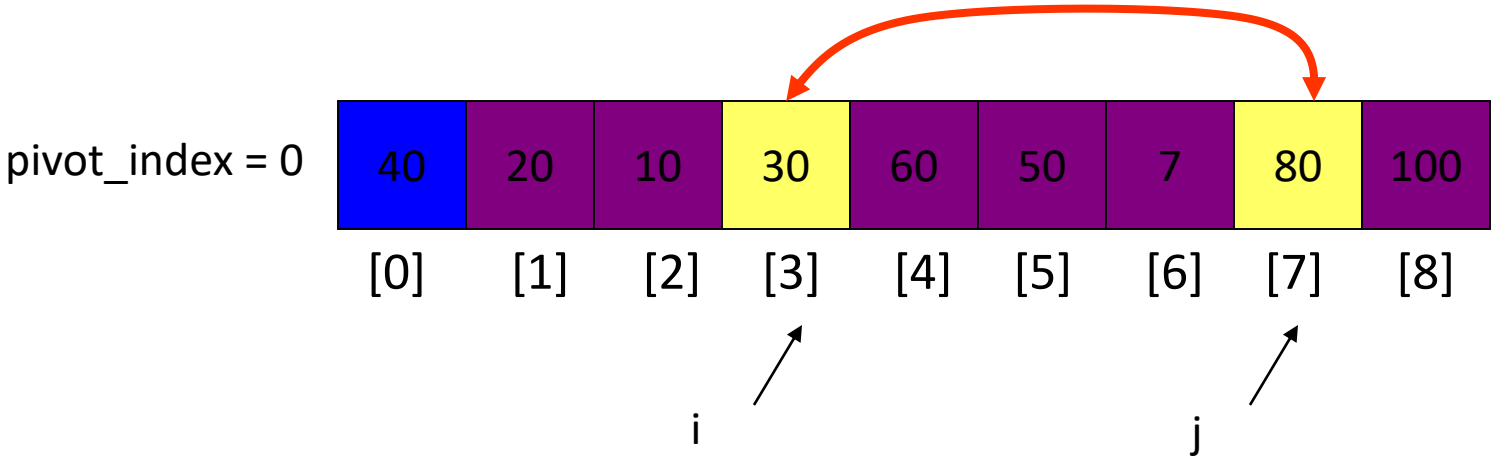
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j



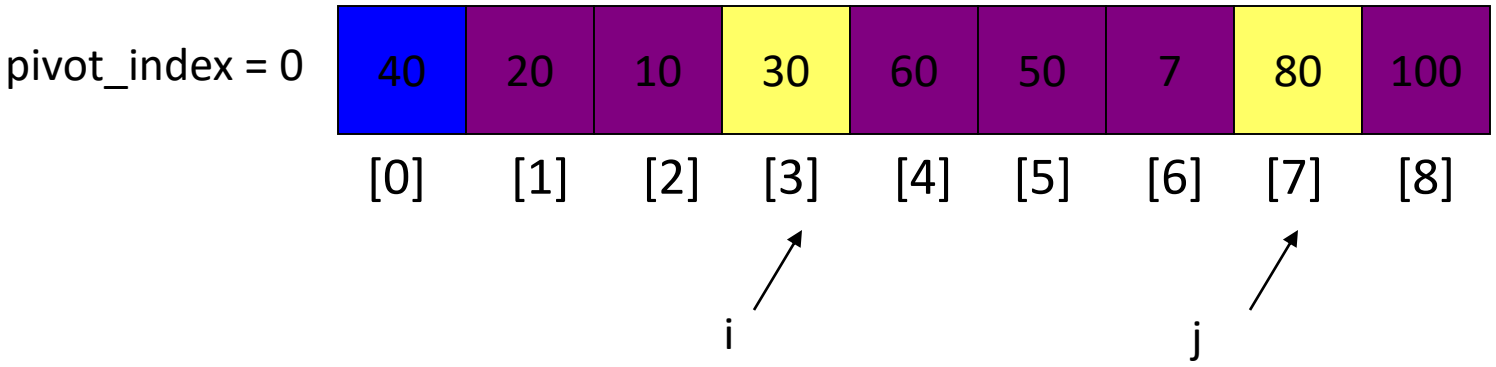
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]



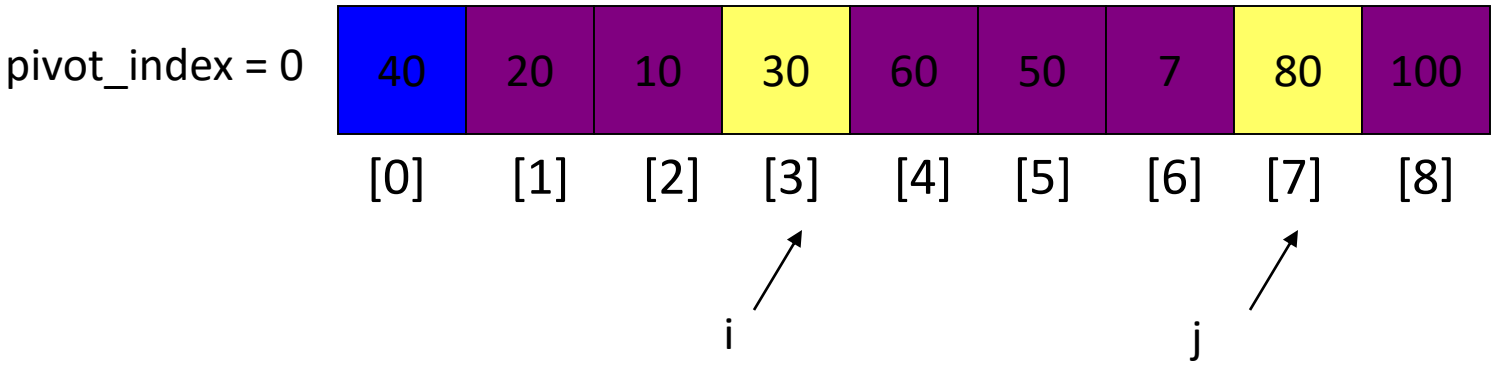
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]



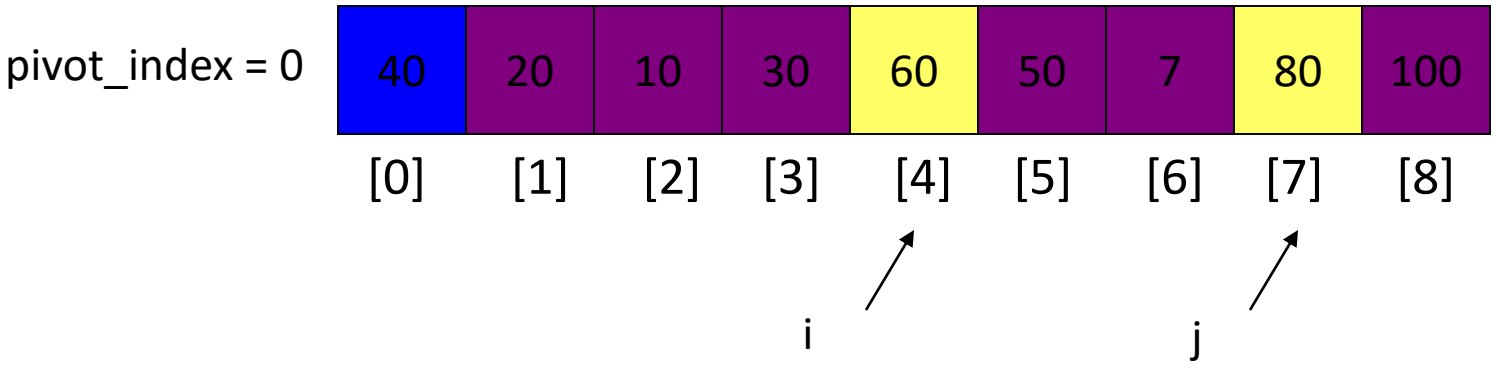
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



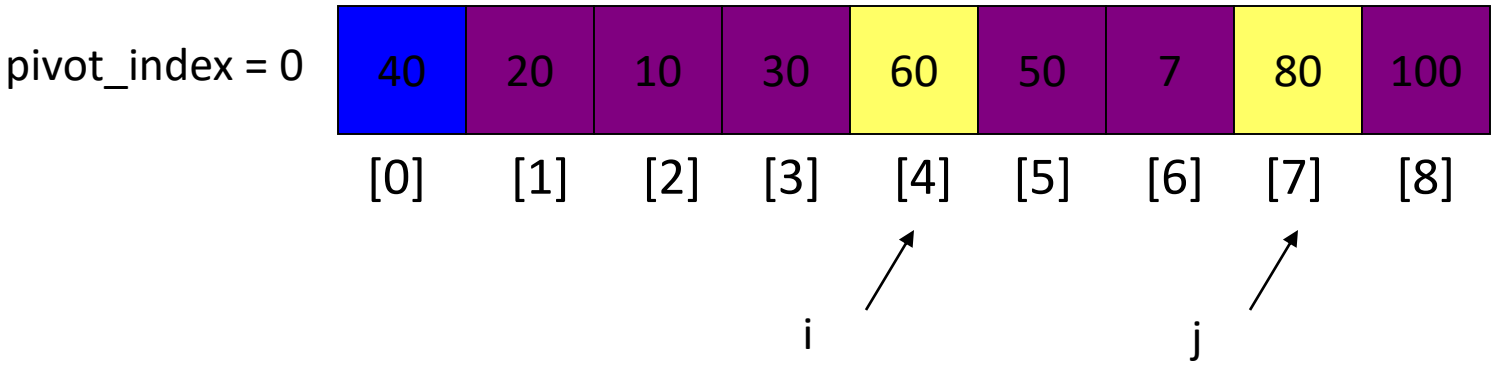
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



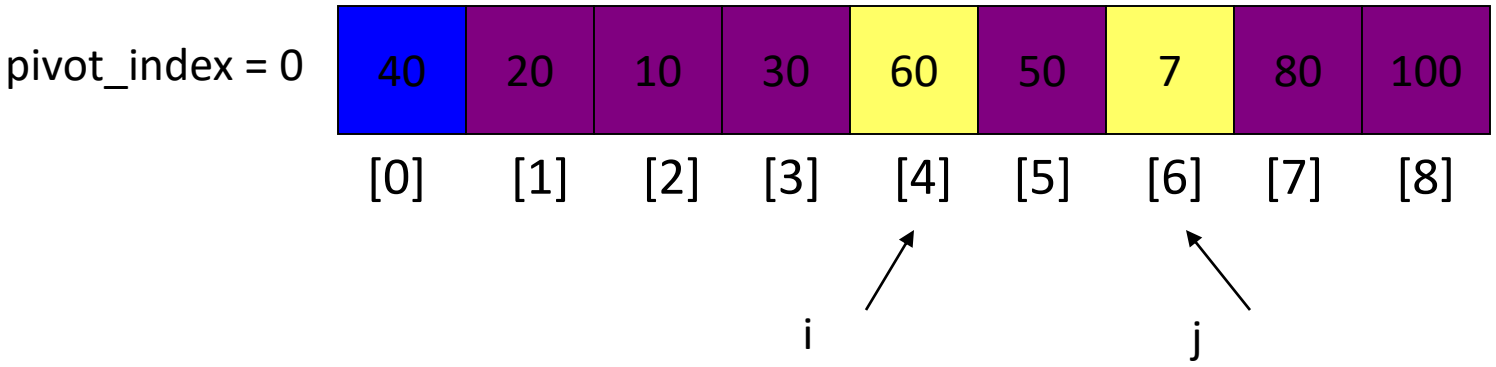
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



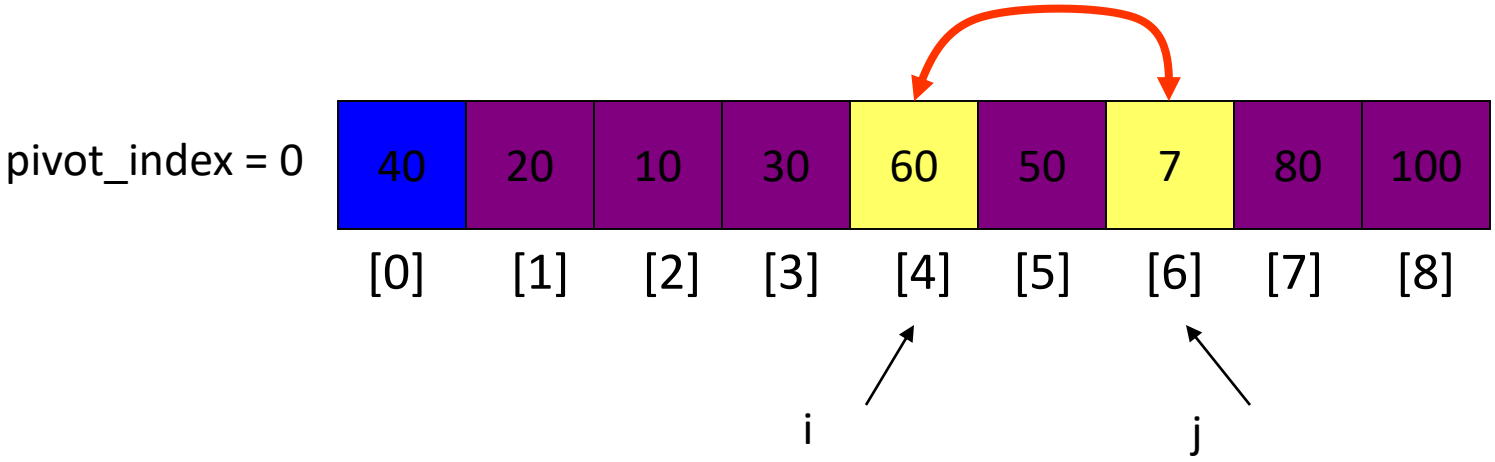
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



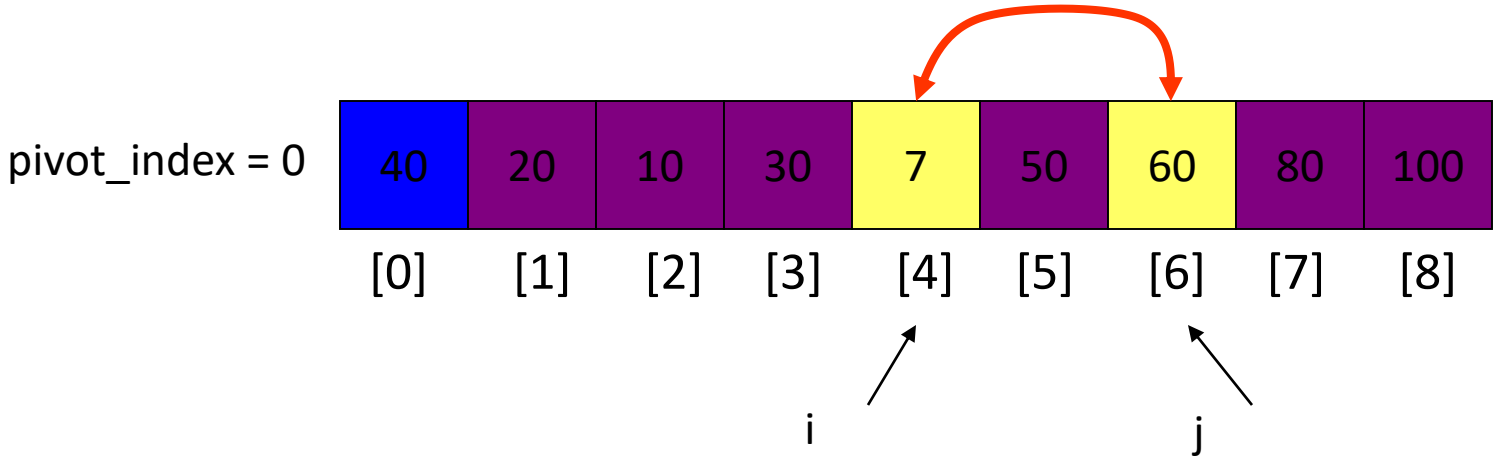
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



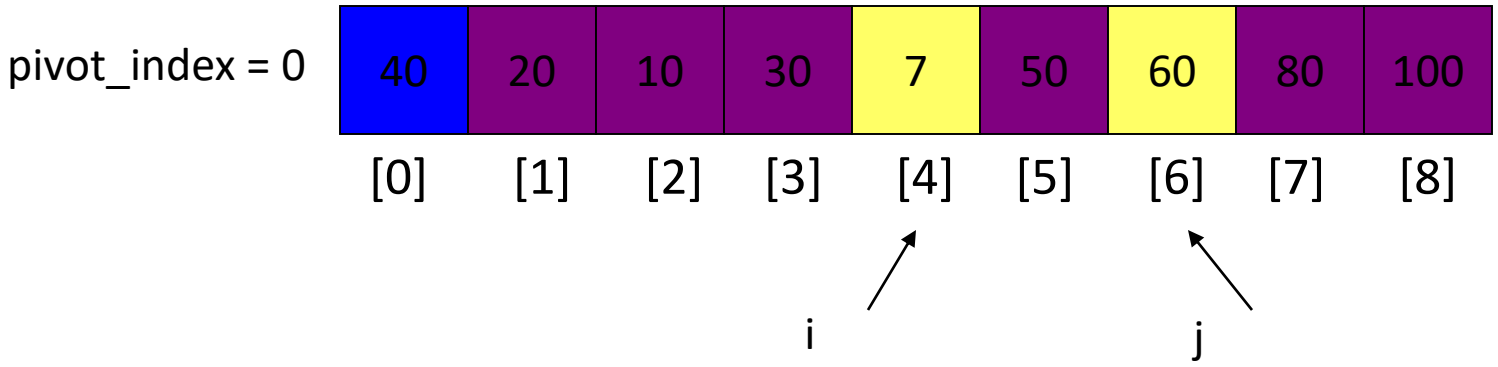
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



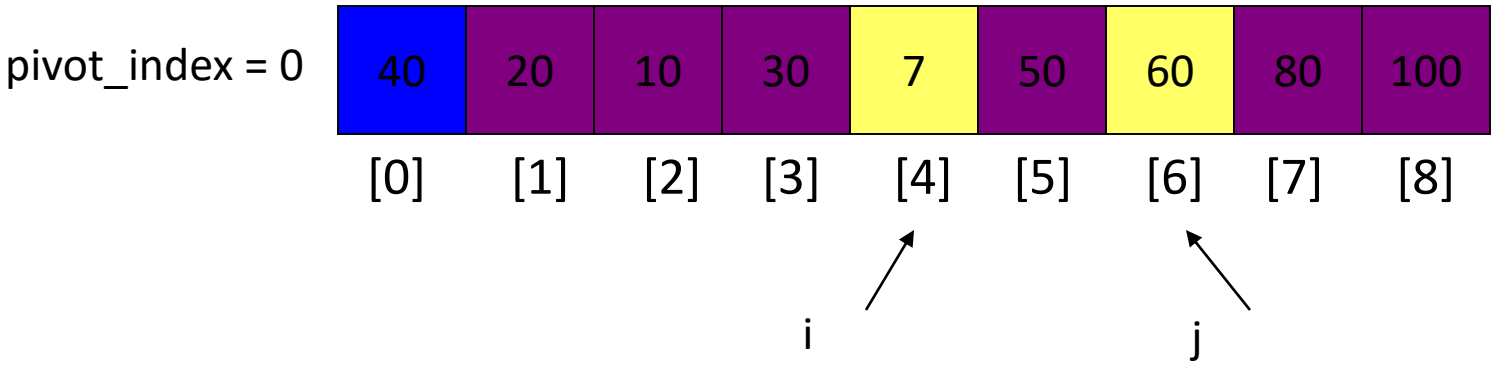
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



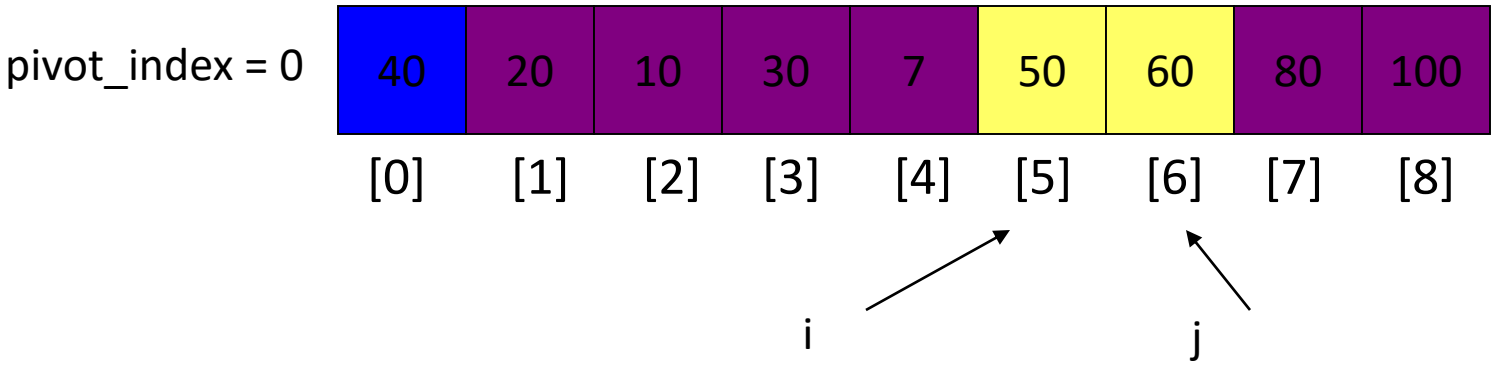
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



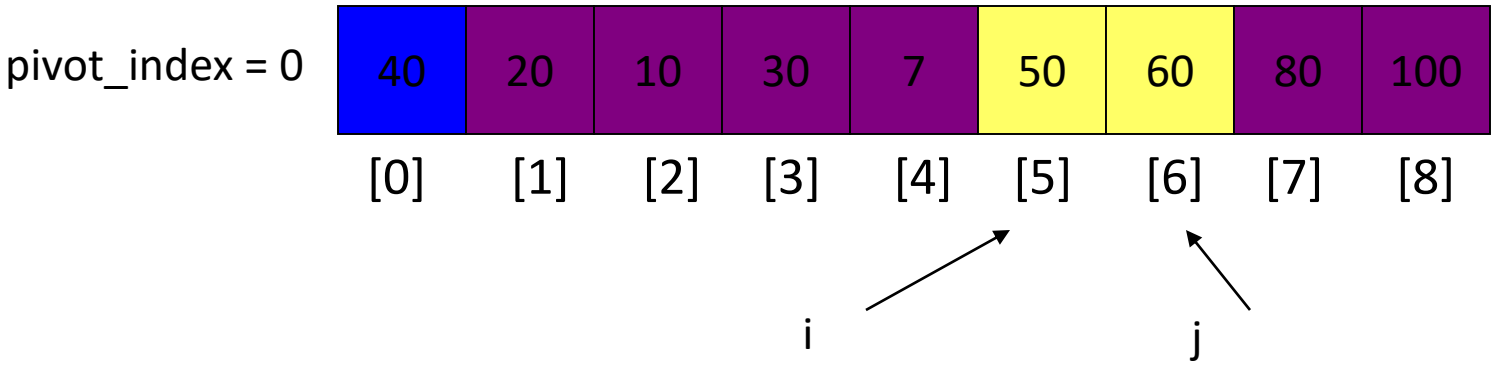
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



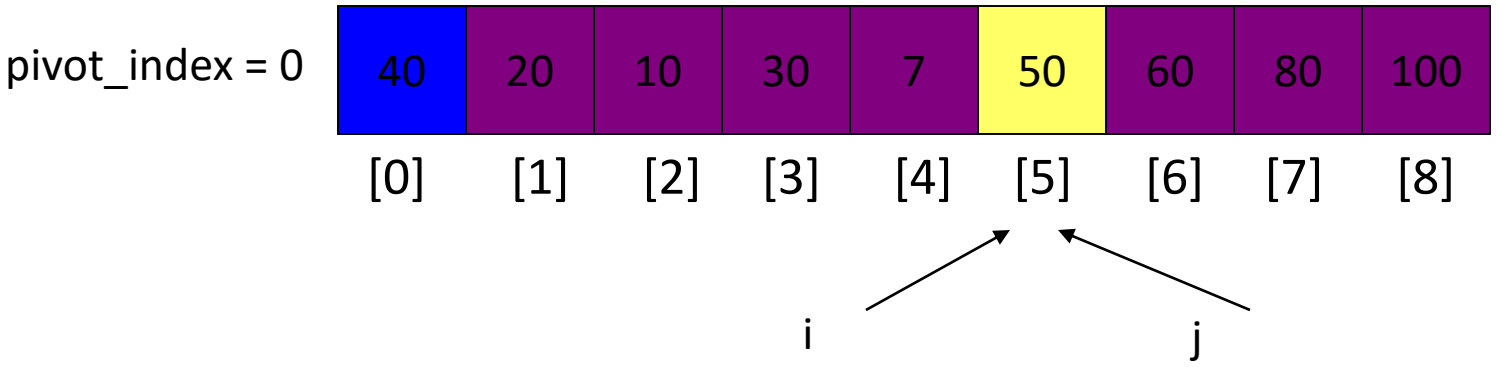
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



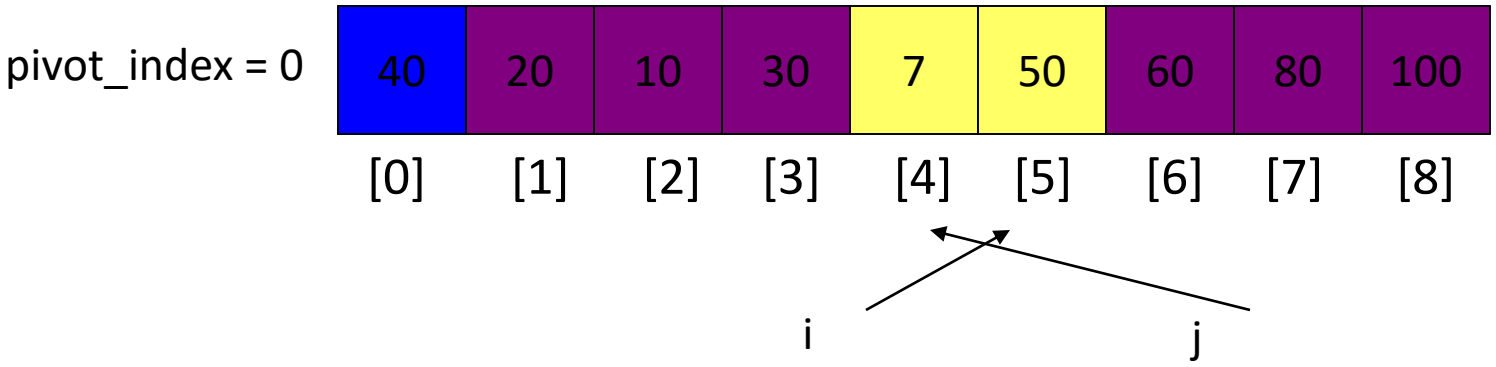
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



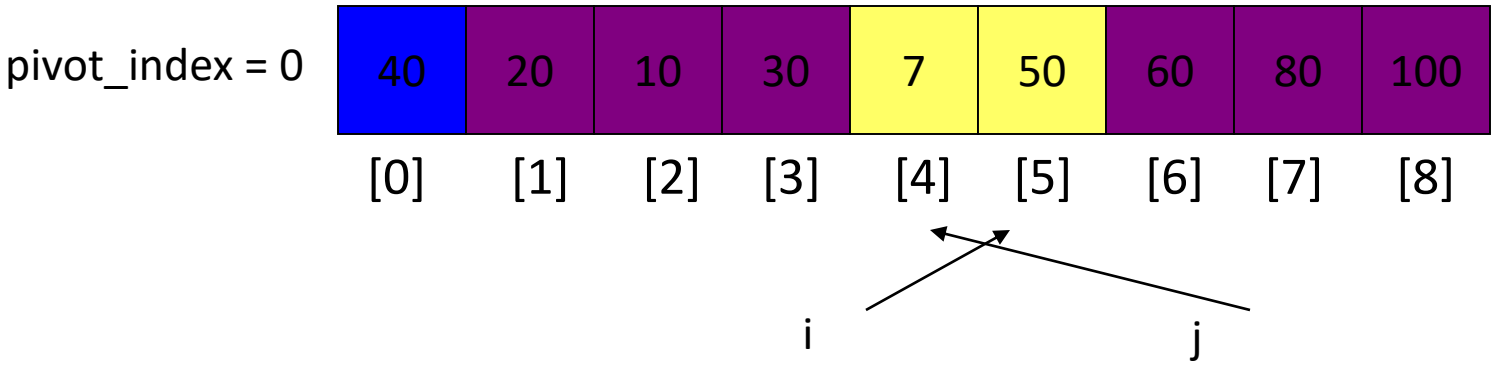
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



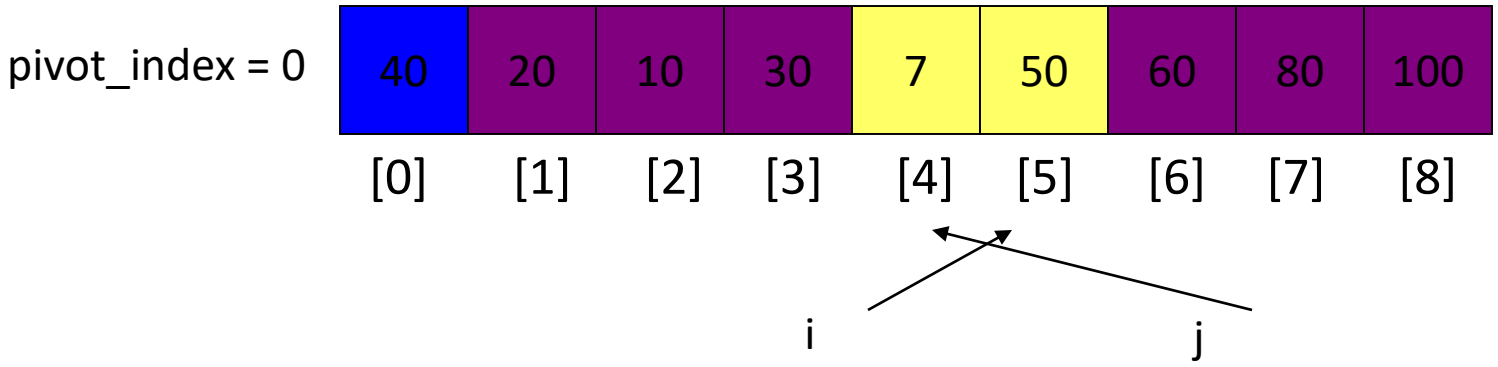
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



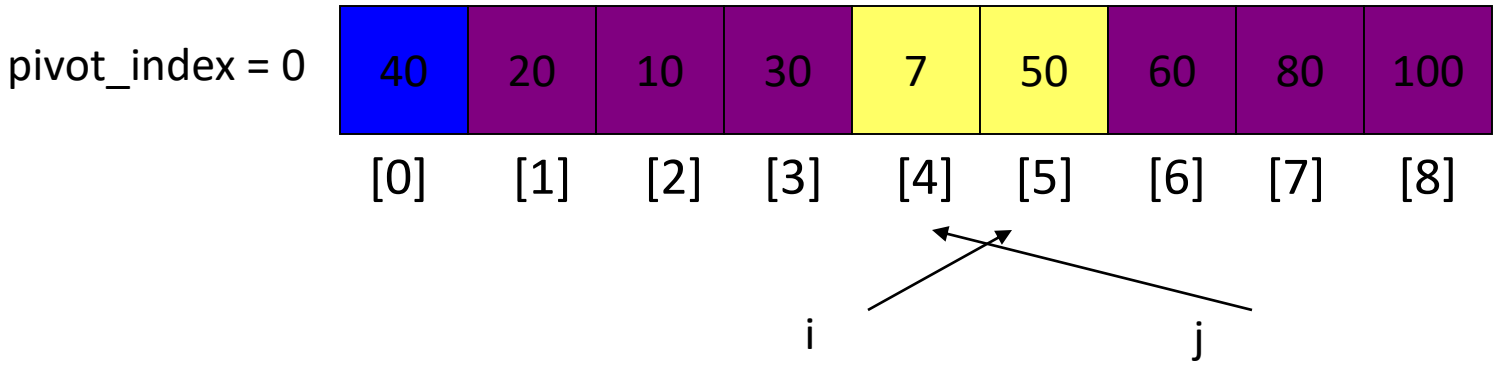
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



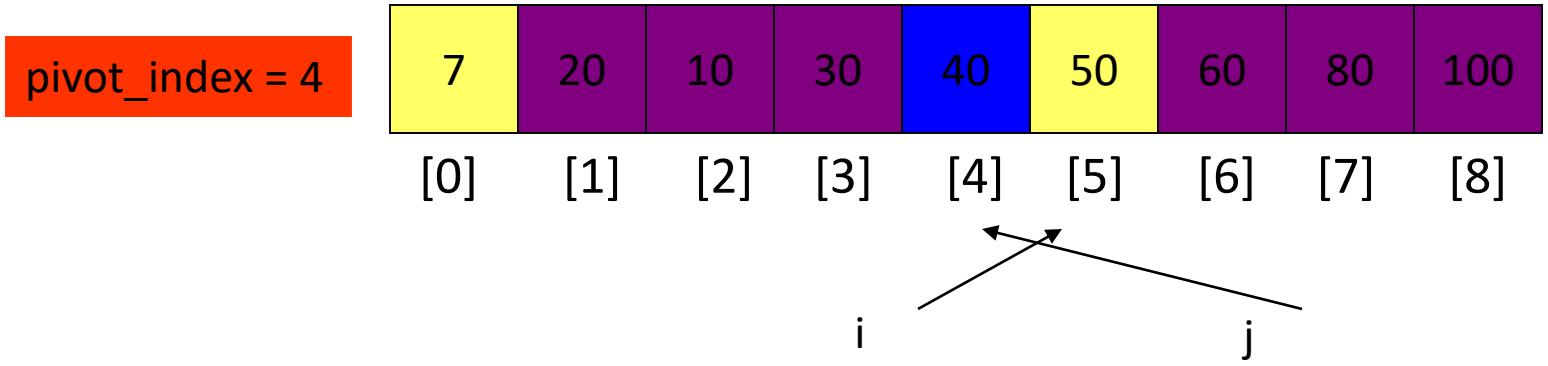
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.



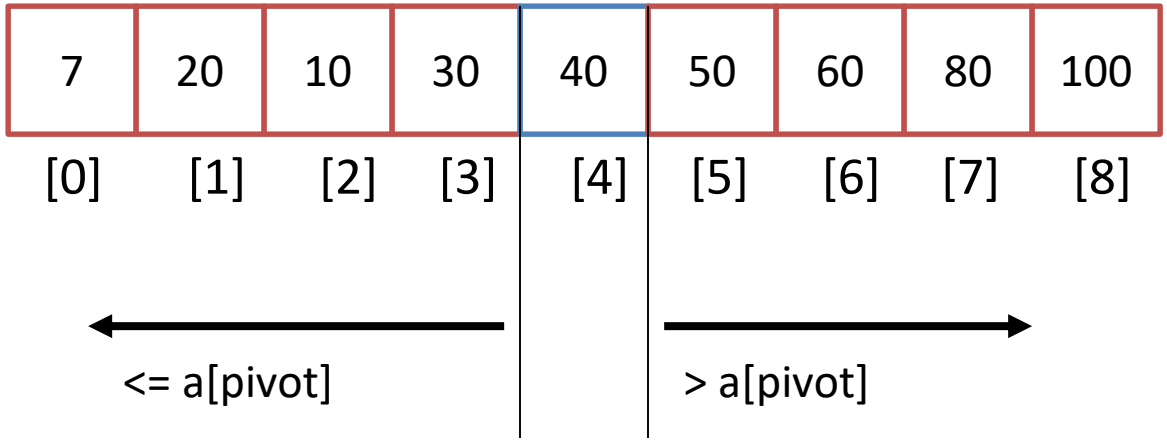
1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.
5. Swap a[j]and a[pivot_index]



1. While a[i]<= a[pivot]
   ++ i
2. While a[j]> a[pivot]
   -- j
3. If i < j
   swap a[i]and a[j]
4. While j > i, go to 1.
5. Swap a[j]and a[pivot_index]



Partition Result



Analysis of Quick Sort

Analysis of Quick Sort

The running time of quick sort is equal to the running time of the two recursive calls plus the linear time spent in the partition (The pivot selection takes only constant time).
This gives the basic quick sort relation:

$$T(n) = T(i) + T(n - i - 1) + Cn - \quad (1)$$

Where, $i = |S1|$ is the number of elements in S1

Analysis of Quick Sort

Worst Case Analysis

The pivot is the smallest element, all the time. Then i=0 and if we ignore T(0)=1, which is insignificant, the recurrences:

$T(n) = T(n-1) + Cn \quad n>1 \quad - \quad (2)$

Using equation – (1) repeatedly, thus

$T(n-1) = T(n-2) + C(n-1)$

$T(n-2) = T(n-3) + C(n-2)$

\- - - - - - - --

$T(2) = T(1) + C(2)$

Adding up all these equations yields

$T(n) = O(n^2)$ - (3)

Analysis of Quick Sort

Best Case Analysis

T (n) = 2 T (n/2) +Cn- (4)

Divide both sides by n and Substitute n/2 for ‘n’

Finally,

Which yields,

T (n) = C n log n + n = **O(n logn)**

T (n) = O(n logn)

Analysis of Quick Sort

Average Case Analysis

The number of comparisons for first call on partition:

Assume left_to_right moves over k smaller element and thus k comparisons. So when right_to_left crosses left_to_right it has made n-k+1 comparisons.

So, first call on partition makes n+1 comparisons. The average case complexity of quick sort is

T(n) = comparisons for first call on quick sort

+

{Σ 1<=nleft, nright<=n [T(nleft) + T(nright)]}n

= (n+1) + 2 [T(0) +T(1) + T(2) + ----- +T(n-1)]/n

Analysis of Quick Sort

$n\,T(n) = n(n+1) + 2\,[T(0) + T(1) + T(2) + \text{-----} + T(n-2) + T(n-1)]$

$(n-1)T(n-1) = (n-1)n + 2\,[T(0) + T(1) + T(2) + \text{-----} + T(n-2)]$

Subtracting both sides:

$nT(n) - (n-1)T(n-1) = [\,n(n+1) - (n-1)n] + 2T(n-1) = 2n + 2T(n-1)$

$nT(n) = 2n + (n-1)T(n-1) + 2T(n-1) = 2n + (n+1)T(n-1)$

$T(n) = 2 + (n+1)T(n-1)/n$

The recurrence relation obtained is:

$T(n)/(n+1) = 2/(n+1) + T(n-1)/n$

Using the method of substitution:

Analysis of Quick Sort

$T(n)/(n+1) = 2/(n+1) + T(n-1)/n$

$T(n-1)/n = 2/n + T(n-2)/(n-1)$

$T(n-2)/(n-1) = 2/(n-1) + T(n-3)/(n-2)$

$T(n-3)/(n-2) = 2/(n-2) + T(n-4)/(n-3)$

. .

. .

$T(3)/4 = 2/4 + T(2)/3$

$T(2)/3 = 2/3 + T(1)/2$ $T(1)/2 = 2/2 + T(0)$

Analysis of Quick Sort

Adding both sides:

$T(n)/(n+1) + [T(n-1)/n + T(n-2)/(n-1) + \text{----------} + T(2)/3 + T(1)/2]$

$= [T(n-1)/n + T(n-2)/(n-1) + \text{----------} + T(2)/3 + T(1)/2] + T(0) + [2/(n+1) + 2/n + 2/(n-1) + \text{---------} + 2/4 + 2/3]$

Cancelling the common terms:

$T(n)/(n+1) = 2[1/2 + 1/3 + 1/4 + \text{----------} + 1/n + 1/(n+1)]$

Finally, We will get, $O(n \log n)$

MERGE SORT

MERGE SORT

```
Algorithm MERGESORT (low, high)
// a (low : high) is a global array to be sorted.
{
        if (low <high)
        {
            mid := (low +high)/2;//finds where to split the set
            MERGESORT(low, mid); //sort one subset
            MERGESORT(mid+1, high);  //sort the other subset
            MERGE(low, mid, high);   // combine the results
        }

}
```

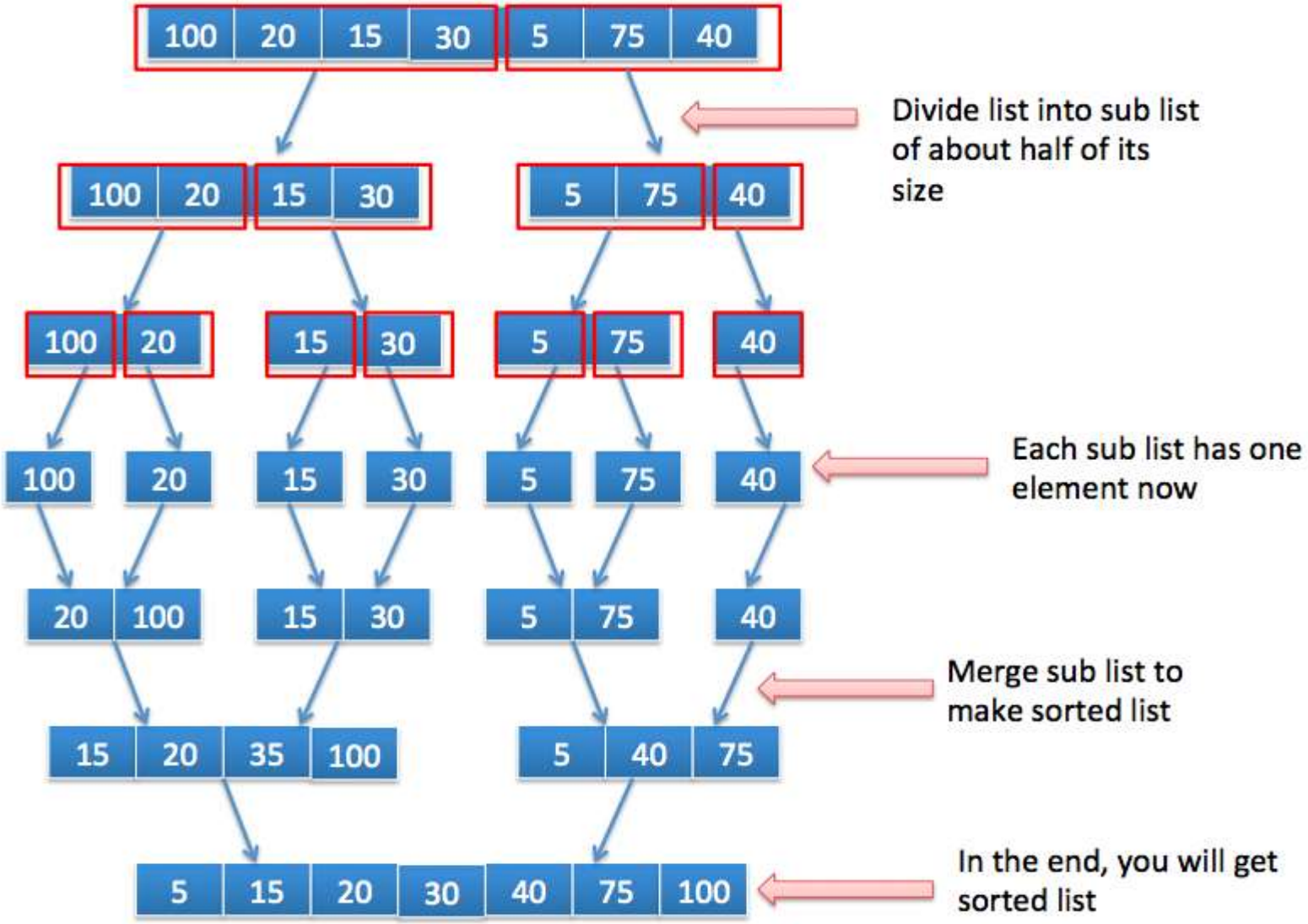
MERGE SORT

```
Algorithm MERGE (low, mid,high)
// a (low : high) is a global array containing two sortedsubsets
// in a (low : mid) and in a (mid + 1 :high).
// The objective is to merge these sorted sets into singlesorted
// set residing in a (low : high). An auxiliary array B isused.
{
        h :=low; i := low; j:= mid + 1;
        while ((h ≤mid) and (J ≤high))do
        {
          if (a[h] ≤a[j])then
         {
                b[i] :=a[h];    h:=h+1;
          }
          else
           {
```

MERGE SORT

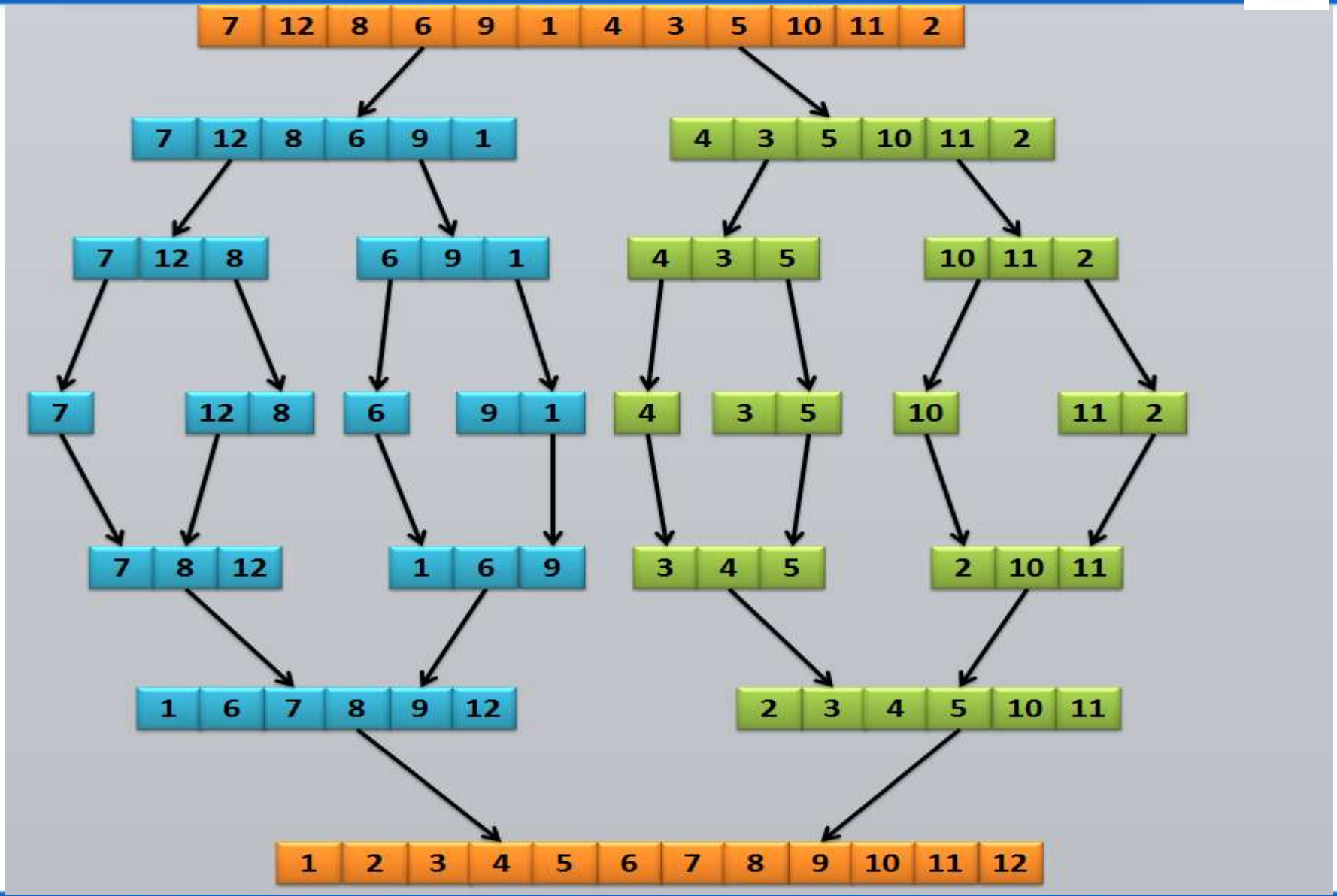
```
                b[i] :=a[j]; j := j +1;
              }
            i := i +1;
        }// while
 if (h > mid) then
            for k := j to high do
              {
b[i] := a[k]; i := i +1
              }
            for k := h to mid do
             {
b[i]  := a[K];  i := i +l;
              }
for k := low to high do
   a[k] :=b[k];   } //end MERGE
```

Example 1


100 20 15 30 5 75 40
Divide list into sub list of about half of its size
100 20 15 30 5 75 40
100 20 15 30 5 75 40
100 20 15 30 5 75 40
Each sub list has one element now
20 100 15 30 5 75 40
Merge sub list to make sorted list
15 20 35 100 5 40 75
5 15 20 30 40 75 100
In the end, you will get sorted list

Analysis of Merge Sort

Example 2


7 12 8 6 9 1 4 3 5 10 11 2
7 12 8 6 9 1
4 3 5 10 11 2
7 12 8
6 9 1
4 3 5
10 11 2
7
12 8
6
9 1
4
3 5
10
11 2
7 8 12
1 6 9
3 4 5
2 10 11
1 6 7 8 9 12
2 3 4 5 10 11
1 2 3 4 5 6 7 8 9 10 11 12

Analysis of Merge Sort

- We will assume that ‘n’ is a power of 2, so that we always split into even halves, so we solve for the case $n = 2^k$.

•For n = 1, the time to merge sort is constant, which we will be denote by 1.

•Otherwise, the time to merge sort ‘n’ numbers is equal to the time to do two recursive merge sorts of size n/2, plus the time to merge, which is linear.

The equation says this exactly:

T(1) =1

T(n) = 2 T(n/2) +n

T(1) = 1

T(n) = 2T(n/2) + n

next we will solve this recurrence relation.

First we divide (2) by n:

T(n) / n = T(n/2) / (n/2) + 1

n is a power of two, so we can write

T(n/2) / (n/2) = T(n/4) / (n/4) +1

T(n/4) / (n/4) = T(n/8) / (n/8) +1

T(n/8) / (n/8) = T(n/16) / (n/16) +1

……

……..

T(2) / 2 = T(1) / 1 + 1

$T(n) / n + T(n/2) / (n/2) + T(n/4) / (n/4) + \ldots + T(2)/2 =$
$T(n/2) / (n/2) + T(n/4) / (n/4) + \ldots.+ T(2) / 2 + T(1) / 1 + Logn$

$T(n)/n = T(1)/1 + Logn$

$T(1)$ is 1, hence we obtain

$T(n) = n + nlogn = O(nlogn)$

Hence the complexity of the MergeSort algorithm is **O(nlogn).**

Strassen's Matrix Multiplication

Strassen's Matrix Multiplication

- The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

- Let A and B be two n×n Matrices.

- The product matrix C=A*B is also a n×n matrix whose i, j^{th} element is formed by taking elements in the i^{th} row of A and j^{th} column of B and multiplying them to get result.

Matrix multiplication

Matrix multiplication

The problem:

Multiply two matrices *A* and *B*, each of size $[n \times n]$

$$\left[\begin{array}{c} A \end{array}\right]_{n\times n} \cdot \left[\begin{array}{c} B \end{array}\right]_{n\times n} = \left[\begin{array}{c} C \end{array}\right]_{n\times n}$$

The traditional way:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

use three for-loop

$$\therefore T(n) = O(n^3)$$

The Divide-and-Conquer way:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

transform the problem of multiplying *A* and *B*, each of size [n×n] into 8 sub problems, each of size $\left[\frac{n}{2} \times \frac{n}{2}\right]$

$$\therefore T(n) = 8 \cdot T(\frac{n}{2}) + an^2$$

$$= O(n^3)$$

which an^2 is for addition

so, it is no improvement compared with the traditional way

Example: use Divide-and-Conquer way to solve it as following:

$$\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}\cdot\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$$

$$\begin{bmatrix}1&1&1&0\\1&1&1&0\\1&1&1&0\\0&0&0&0\end{bmatrix}\cdot\begin{bmatrix}1&1&1&0\\1&1&1&0\\1&1&1&0\\0&0&0&0\end{bmatrix}=\begin{bmatrix}3&3&3&0\\3&3&3&0\\3&3&3&0\\0&0&0&0\end{bmatrix}$$

$$C_{11}=\begin{bmatrix}1&1\\1&1\end{bmatrix}\cdot\begin{bmatrix}1&1\\1&1\end{bmatrix}+\begin{bmatrix}1&0\\1&0\end{bmatrix}\cdot\begin{bmatrix}1&1\\0&0\end{bmatrix}=\begin{bmatrix}3&3\\3&3\end{bmatrix}$$

$$C_{12}=\begin{bmatrix}1&1\\1&1\end{bmatrix}\cdot\begin{bmatrix}1&0\\1&0\end{bmatrix}+\begin{bmatrix}1&0\\1&0\end{bmatrix}\cdot\begin{bmatrix}1&0\\0&0\end{bmatrix}=\begin{bmatrix}3&0\\3&0\end{bmatrix}$$

$$C_{21}=\begin{bmatrix}1&1\\0&0\end{bmatrix}\cdot\begin{bmatrix}1&1\\1&1\end{bmatrix}+\begin{bmatrix}1&0\\0&0\end{bmatrix}\begin{bmatrix}1&1\\0&0\end{bmatrix}=\begin{bmatrix}3&3\\0&0\end{bmatrix}$$

$$C_{22}=\begin{bmatrix}1&1\\0&0\end{bmatrix}\cdot\begin{bmatrix}1&0\\1&0\end{bmatrix}+\begin{bmatrix}1&0\\0&0\end{bmatrix}\cdot\begin{bmatrix}1&0\\0&0\end{bmatrix}=\begin{bmatrix}3&0\\0&0\end{bmatrix}$$

Strassen's Matrix Multiplication

Strassens introduces new way of computing the C_{ij}'s using 7 multiplications and 18 additions or subtractions

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

Analysis

$$T(n) = \begin{cases} 7 \cdot T(\frac{n}{2}) + an^2 & n > 2 \\ b & n \leq 2 \end{cases}$$

$$T(n) = 7 \cdot T(\frac{n}{2}) + an^2$$

$$= 7^2 \cdot T(\frac{n}{2^2}) + (\frac{7}{4})an^2 + an^2$$

$$= 7^3 \cdot T(\frac{n}{2^3}) + (\frac{7}{4})^2 \cdot an^2 + (\frac{7}{4}) \cdot an^2 + an^2$$

$$\bullet \bullet \bullet$$

Analysis

Assume $n = 2^k$ for some integer k

$$= 7^{k-1} \cdot T(\frac{n}{2^{k-1}}) + an^2 \cdot \left[(\frac{7}{4})^{k-2} + \cdots + 1 \right]$$

$$= 7^{k-1} \cdot b + an^2 \left[\frac{(\frac{7}{4})^{k-1} - 1}{\frac{7}{4} - 1} \right]$$

$$\leq b \cdot 7^k + c \cdot n^2 \cdot (\frac{7}{4})^k$$

$$= b \cdot 7^{Lgn} + cn^2 \cdot (\frac{7}{4})^{Lgn} = b \cdot 7^{Lgn} + cn^2 (n)^{Lg\frac{7}{4}}$$

$$= b \cdot n^{Lg7} + cn^{Lg7} = (b + c) \cdot n^{Lg7}$$

$$= O(n^{Lg7}) = O(n^{2.81})$$

Strassen's Matrix Multiplication

Strassen's Matrix Multiplication

- The matrix multiplication of algorithm due to Strassens is the most dramatic example of divide and conquer technique (1969).

- Let A and B be two n×n Matrices.

- The product matrix C=A*B is also a n×n matrix whose i, j^{th} element is formed by taking elements in the i^{th} row of A and j^{th} column of B and multiplying them to get result.

Matrix multiplication

Matrix multiplication

The problem:

Multiply two matrices *A* and *B*, each of size $[n \times n]$

$$\left[\begin{array}{c} A \end{array}\right]_{n\times n} \cdot \left[\begin{array}{c} B \end{array}\right]_{n\times n} = \left[\begin{array}{c} C \end{array}\right]_{n\times n}$$

The traditional way:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

use three for-loop

$$\therefore T(n) = O(n^3)$$

The Divide-and-Conquer way:

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$

$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$

$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$

$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$

transform the problem of multiplying *A* and *B*, each of size [n×n] into 8 sub problems, each of size $\left[\frac{n}{2} \times \frac{n}{2}\right]$

$$\therefore T(n) = 8 \cdot T(\frac{n}{2}) + an^2$$

$$= O(n^3)$$

which an^2 is for addition

so, it is no improvement compared with the traditional way

Example: use Divide-and-Conquer way to solve it as following:

$$\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}\cdot\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$$

$$\begin{bmatrix}1&1&1&0\\1&1&1&0\\1&1&1&0\\0&0&0&0\end{bmatrix}\cdot\begin{bmatrix}1&1&1&0\\1&1&1&0\\1&1&1&0\\0&0&0&0\end{bmatrix}=\begin{bmatrix}3&3&3&0\\3&3&3&0\\3&3&3&0\\0&0&0&0\end{bmatrix}$$

$$C_{11}=\begin{bmatrix}1&1\\1&1\end{bmatrix}\cdot\begin{bmatrix}1&1\\1&1\end{bmatrix}+\begin{bmatrix}1&0\\1&0\end{bmatrix}\cdot\begin{bmatrix}1&1\\0&0\end{bmatrix}=\begin{bmatrix}3&3\\3&3\end{bmatrix}$$

$$C_{12}=\begin{bmatrix}1&1\\1&1\end{bmatrix}\cdot\begin{bmatrix}1&0\\1&0\end{bmatrix}+\begin{bmatrix}1&0\\1&0\end{bmatrix}\cdot\begin{bmatrix}1&0\\0&0\end{bmatrix}=\begin{bmatrix}3&0\\3&0\end{bmatrix}$$

$$C_{21}=\begin{bmatrix}1&1\\0&0\end{bmatrix}\cdot\begin{bmatrix}1&1\\1&1\end{bmatrix}+\begin{bmatrix}1&0\\0&0\end{bmatrix}\begin{bmatrix}1&1\\0&0\end{bmatrix}=\begin{bmatrix}3&3\\0&0\end{bmatrix}$$

$$C_{22}=\begin{bmatrix}1&1\\0&0\end{bmatrix}\cdot\begin{bmatrix}1&0\\1&0\end{bmatrix}+\begin{bmatrix}1&0\\0&0\end{bmatrix}\cdot\begin{bmatrix}1&0\\0&0\end{bmatrix}=\begin{bmatrix}3&0\\0&0\end{bmatrix}$$

Strassen's Matrix Multiplication

Strassens introduces new way of computing the C_{ij}'s using 7 multiplications and 18 additions or subtractions

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

Analysis

$$T(n) = \begin{cases} 7 \cdot T(\frac{n}{2}) + an^2 & n > 2 \\ b & n \leq 2 \end{cases}$$

$$T(n) = 7 \cdot T(\frac{n}{2}) + an^2$$

$$= 7^2 \cdot T(\frac{n}{2^2}) + (\frac{7}{4})an^2 + an^2$$

$$= 7^3 \cdot T(\frac{n}{2^3}) + (\frac{7}{4})^2 \cdot an^2 + (\frac{7}{4}) \cdot an^2 + an^2$$

$$\bullet \bullet \bullet$$

Analysis

Assume $n = 2^k$ for some integer k

$$= 7^{k-1} \cdot T(\frac{n}{2^{k-1}}) + an^2 \cdot \left[(\frac{7}{4})^{k-2} + \cdots + 1 \right]$$

$$= 7^{k-1} \cdot b + an^2 \left[\frac{(\frac{7}{4})^{k-1} - 1}{\frac{7}{4} - 1} \right]$$

$$\leq b \cdot 7^k + c \cdot n^2 \cdot (\frac{7}{4})^k$$

$$= b \cdot 7^{Lgn} + cn^2 \cdot (\frac{7}{4})^{Lgn} = b \cdot 7^{Lgn} + cn^2 (n)^{Lg\frac{7}{4}}$$

$$= b \cdot n^{Lg\,7} + cn^{Lg\,7} = (b + c) \cdot n^{Lg\,7}$$

$$= O(n^{Lg\,7}) = O(n^{2.81})$$

UNIT 3
GREEDY METHOD AND DYNAMIC PROGRAMMING

GENERAL METHOD

- Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints.
- Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function.
- A feasible solution that does this is called an optimal solution.

- The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage.
- At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure.
- If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution.

- The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem.
- Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*.
- Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm*

Algorithm

CONTROLABSTRACTION Algorithm Greedy (a,n)

- // a(1 : n) contains the ‘n’ inputs
- {
- solution:=$^{\phi}$; // initialize the solution to be empty
- for i:=1 to ndo
- {
- x := select(a);
- if feasible (solution, x)then
- solution := Union (Solution,x);
- }
- return solution;
- }

JOB SEQUENCING WITH DEADLINES

- Given a set of ‘n’ jobs. Associated with each Job i, deadline di ≥ 0 and profit Pi ≥ 0. For any job ‘i’ the profit pi is earned iff the job is completed by its deadline.
- Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.
- Sort the jobs in ‘j’ ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], $1 \leq r \leq k$ are the jobs in ‘j’ and d (j [1]) ≤ d (j[2]) ≤ . . . ≤ d (j[k]).
- To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that d [J[r]] ≤ r, $1 \leq r \leq k+1$.

Example:

Let n=4,(P1,P2,P3,P4,)=(100,10,15,27)and(d1 d2 d3 d4)=(2,1,2,1).The
feasible solutions and their values are:

Sl.No	Feasible Solution	Procuring sequence	Value	Remarks
1	1,2	2,1	110	
2	1,3	1,3 or3,1	115	
3	1,4	4,1	127	**OPTIMA**
4	2,3	2,3	25	
5	3,4	4,3	42	
6	1	1	100	
7	2	2	10	
8	3	3	15	
9	4	4	27	

Algorithm

- The algorithm constructs an optimal set J of jobs that can be processed by their deadlines.
- Algorithm GreedyJob (d, J,n)
- // J is a set of jobs that can be completed by their deadlines.
- {
- J :={1};
- for i := 2 to ndo
- {
- if (all jobs in J U {i} can be completed by their deadlines) then J := J U{i};
- }
- }
- The greedy algorithm is used to obtain an optimal solution.

Algorithm

- Algorithm js(d, j, n)
- //d→ dead line, j→subset of jobs ,n→ total number of jobs
- // d[i]≥1 1 ≤ i ≤ n are the dead lines,
- // the jobs are ordered such that p[1]≥p[2]≥---≥p[n]
- //j[i] is the ith job in the optimal solution 1 ≤ i ≤ k, k→ subset range
- {
- d[0]=j[0]=0;
- j[1]=1;

- k=1;
- for i=2 to n do{
- r=k;
- while((d[j[r]]>d[i]) and [d[j[r]]≠r)) do
- r=r-1;
- if((d[j[r]]≤d[i]) and (d[i]> r)) then
- {
- for q:=k to (r+1) setp-1 do j[q+1]= j[q];
- j[r+1]=i;
- k=k+1;
- }
- }
- return k;
- }

KNAPSACK PROBLEM

- Let us apply the greedy method to solve the knapsack problem. We are given ‘n’ objects and a knapsack. The object ‘i’ has a weight wi and the knapsack has a capacity ‘m’. If a fraction xi, 0 < xi < 1 of object i is placed into the knapsack then a profit of pi xi is earned. The objective is to fill the knapsack that maximizes the total profit earned.
- Since the knapsack capacity is ‘m’, we require the total weight of all chosen objects to be at most ‘m’. The problem is stated as:

$$\text{Maximize} \sum_{i=1}^{n} v_i x_i$$

subject to

$$\sum_{i=1}^{n} w_i x_i \leqslant W, \qquad x_i \in \{0, 1\}$$

Algorithm

- Algorithm GreedyKnapsack (m,n)
- // P[1 : n] and w[1 : n] contain the profits and weights respectively of
- // Objects ordered so that p[i] / w[i]> p[i + 1] / w[i + 1].
- // m is the knapsack size and x[1: n] is the solution vector.
- {
- for i := 1 to n do
- x[i] :=0.0 ; //initialize the solution vector
- U :=m;

Algorithm

- for i := 1 to n do
- {
- if (w(i) > U) then break;
- x [i] := 1.0;
- U := U –w[i];
- }
- if (i ≤n) then x[i] := U /w[i];
- }

Example:

Consider the following instance of the knapsack problem: n = 3, m = 20, (p1, p2, p3) = (25, 24, 15) and (w1, w2, w3) = (18, 15,10).

1. First, we try to fill the knapsack by selecting the objects in some order:

x1	x2	x3	∑wi xi	∑pi xi
1/2	1/3	1/4	18 x 1/2 + 15 x 1/3 + 10 x1/4 =16.5	25 x 1/2 + 24 x 1/3 + 15 x 1/4= 24.25

2. Select the object with the maximum profit first (p = 25). So, x1 = 1 and profit earned is 25. Now, only 2 units of space is left, select the object with next largest profit (p = 24). So, x2 =2/15

x1	x2	x3	∑wi xi	∑pi xi
1	2/15	0	18 x 1 + 15 x 2/15 =20	25 x 1 + 24 x 2/15 =28.2

3. Considering the objects in the order of non-decreasing weightswj.

x1	x2	x3	∑ wi xi	∑ pi xi
0	2/3	1	15 x 2/3 + 10 x 1 =20	24 x 2/3 + 15 x 1 =31

4. Considered the objects in the order of the ratio pi / wi.

p1/w1	p2/w2	p3/w3
25/18	24/15	15/10
1.4	1.6	1.5

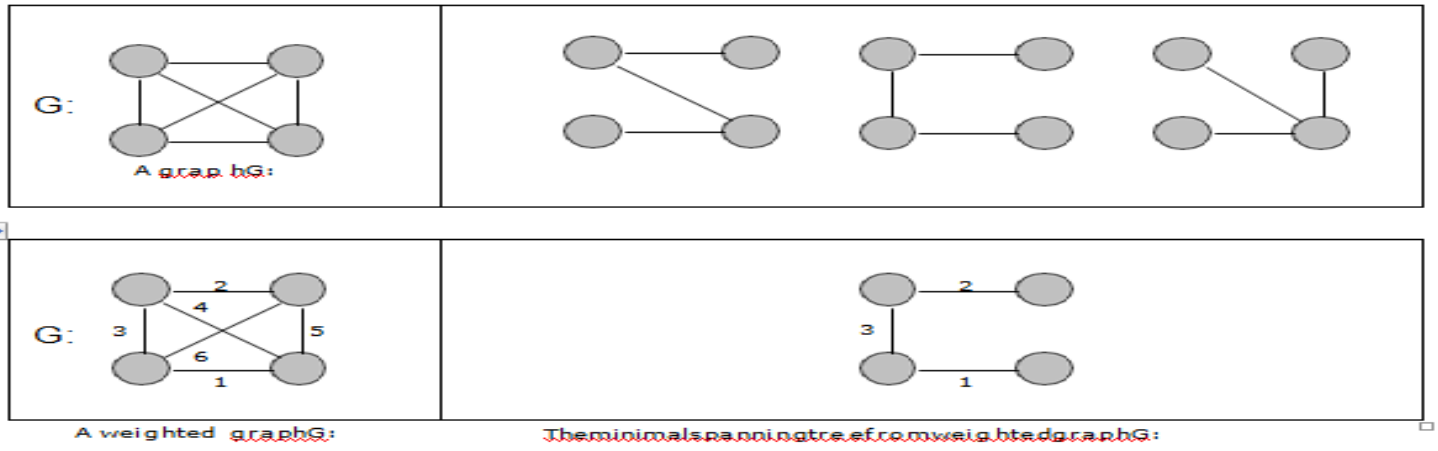
Sort the objects in order of the non-increasing order of the ratio pi / xi. Select the object with the maximumpi / xi ratio, so, x2 = 1 and profit earned is 24. Now, only 5 units of space is left, select the object with next largest pi / xi ratio, so x3 = ½ and the profit earned is7.5.

x1	x2	x3	∑wi xi	∑pi xi
0	1	1/2	15 x 1 + 10 x 1/2 =20	24 x 1 + 15 x 1/2 =31.5

This solution is the optimal solution.

Minimum Cost Spanning Trees(MST):

- A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph.
- i.e., any connected graph will have a spanning tree.
- Weight of a spanning tree w (T) is the sum of weights of all edges in T.
- The Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.



- To explain further upon the Minimum Spanning Tree, and what it applies to,let's consider a couple of real-world examples:
- 1. One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.
- 2. Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. Obviously, the further one has to travel, the more it will cost, so MST can be applied to optimize airline routes by finding the least costly paths with no cycles.
-
-

- To explain how to find a Minimum Spanning Tree, we will look at two algorithms:
- the Kruskal algorithm
- and the Prim algorithm.
- Both algorithms differ in their methodology, but both eventually end up with the MST. Kruskal's algorithm uses edges, and Prim’s algorithm uses vertex connections in determining theMST

Algorithm Kruskal

- Algorithm Kruskal (E, cost, n,t)
- // E is the set of edges in G. G has n vertices. cost [u, v] isthe
- // cost of edge (u, v). ‘t’ is the set of edges in the minimum-cost spanningtree.
- // The final cost isreturned.
- {
- Construct a heap out of the edge costs usingheapify; for i := 1 to n do parent [i] :=-1;
- // Each vertex is in a differentset.
- i := 0; mincost :=0.0;

Algorithm Kruskal

- while ((i < n -1) and (heap not empty))do
- {
- Delete a minimum cost edge (u, v) from the heapand re-heapify usingAdjust;
- j := Find (u); k := Find(v); if (j ¹k)then
- {
- i := i +1;
- t [i, 1] := u; t [i, 2] := v; mincost :=mincost + cost [u,v]; Union (j,k);
- }
- }
- if (i ¹n-1) then write ("no spanning tree"); else returnmincost;
- }

Algorithm Prims

Algorithm Prim (E, cost, n,t)

- // E is the set of edges in G. cost [1:n, 1:n] is thecost
- // adjacency matrix of an n vertex graph such that cost [i, j]is
- // either a positive real number or µif no edge (i, j)exists.
- // A minimum spanning tree is computed and stored as a setof
- // edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edgein
- // the minimum-cost spanning tree. The final cost isreturned.
- {
- Let (k, l) be an edge of minimum cost inE; mincost := cost [k,l];
- t [1, 1] := k; t [1, 2] :=l;

Algorithm Prims

- for i :=1 tondo //Initializenearif (cost [i, l] < cost [i, k]) then near [i] :=l;
- else near [i] := k; near [k] :=near [l] :=0;
- for i:=2 to n - 1do // Find n - 2 additional edges fort.
- {
- Let j be an index such that near [j] ^{1}0**and**
- cost [j, near [j]] isminimum;
- t [i, 1] := j; t [i, 2] := near [j]; mincost := mincost + cost [j, near [j]]; near [j] :=0
- for k:= 1 tondo // Updatenear[].
- if ((near [k] ^{1}0) **and** (cost [k, near [k]] > cost [k, j])) then near [k] :=j;
- }
- returnmincost;
- }

Single Source Shortest-Path Problem

- In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.
- Dijkstra’s algorithm is similar to prim's algorithm for finding minimal spanning trees.
- Dijkstra’s algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between then (or one of the shortest paths) if there is more than one.
- The principle of optimality is the basis for Dijkstra’s algorithms.Dijkstra’s algorithm does not work for negative edges at all.

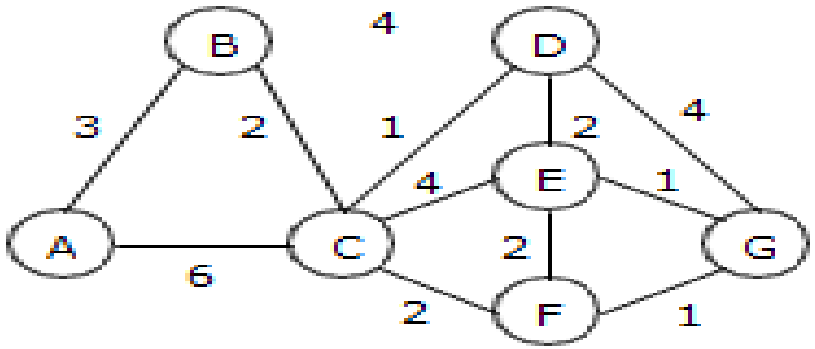
- **Algorithm Shortest-Paths (v, cost, dist,n)**
- // dist [j], 1 ≤j ≤n, is set to the length of the shortest path
- // from vertex v to vertex j in the digraph G with n vertices.
- // dist [v] is set to zero. G is represented by its
- // cost adjacency matrix cost [1:n,1:n].
- {
- for i :=1 to n do
- {
- S [i]:=false; //Initialize S. dist [i] :=cost [v,i];
- }

Algorithm

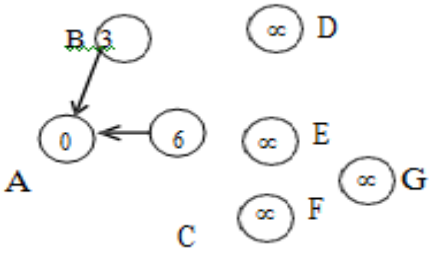
- S[v] := true; dist[v] :=0.0; // Put v in S. for num := 2 to n – 1do
- {
- Determine n - 1 paths from v.
- Choose u from among those vertices not in S such that dist[u] is minimum; S[u]:=true; // Put u is S.
- for (each w adjacent to u with S [w] = false)do
- if (dist [w] > (dist [u] + cost [u, w])then //Update distances dist [w] := dist [u] + cost [u,w];
- }
- }

Example1:

Use Dijkstras algorithm to find the shortest path from A to each of the other six vertices in the graph:

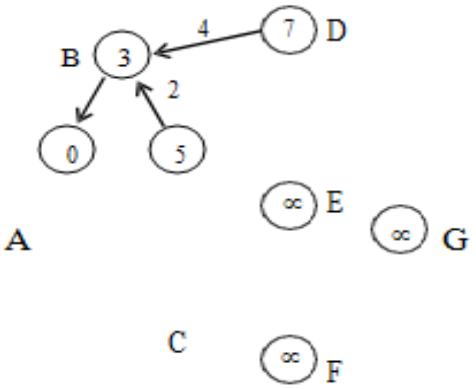


Step1:



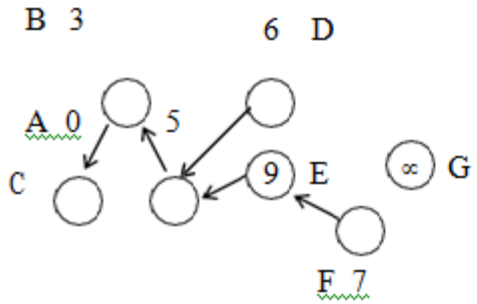
Vertex	A	B	C	D	E	F	G
Status	0	1	1	1	1	1	1
Dist.	0	3	6	∞	∞	∞	∞
Next	*	A	A	A	A	A	A

Step2:



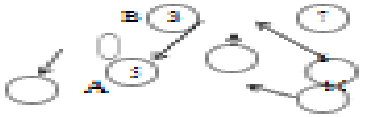
Vertex	A	B	C	D	E	F	G
Status	0	0	1	1	1	1	1
Dist.	0	3	5	7	∞	∞	∞
Next	*	A	B	B	A	A	A

Step3:



Vertex	A	B	C	D	E	F	G
Status	0	0	0	1	1	1	1
Dist.	0	3	5	6	9	7	∞
Next	*	A	B	C	C	C	A

Step4:



D

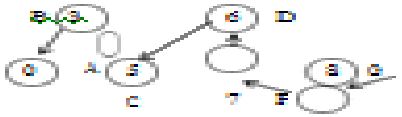
E

G

F

Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	1	1
Dist.	0	3	5	6	8	7	10
Next	*	A	B	C	D	C	D

Step5:

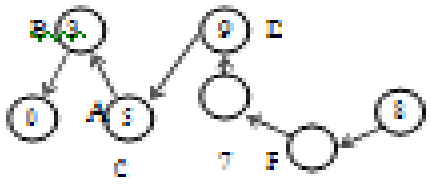


Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	1	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step6:

Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	1
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Step7:



G

Vertex	A	B	C	D	E	F	G
Status	0	0	0	0	0	0	0
Dist.	0	3	5	6	8	7	8
Next	*	A	B	C	D	C	F

Dynamic Programming:

- **General Method**
- Dynamic programming is a name, coined by Richard Bellman in 1955. Dynamic programming, as greedy method, is a powerful algorithm design technique that can be used when the solution to the problem may be viewed as the result of a sequence of decisions.
- In the greedy method we make irrevocable decisions one at a time, using a greedy criterion. However, in dynamic programming we examine the decision sequence to see whether an optimal decision sequence contains optimal decision subsequence.

- When optimal decision sequences contain optimal decision subsequences, we can establish recurrence equations, called *dynamic-programming recurrence equations that* enable us to solve the problem in an efficient way.
- Dynamic programming is based on the principle of optimality (also coined by Bellman). The principle of optimality states that no matter whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision.
- The principle implies that an optimal decision sequence is comprised of optimal decision subsequences. Since the principle of optimality may not hold for some formulations of some problems, it is necessary to verify that it does hold for the problem being solved. Dynamic programming cannot be applied when this principle does not hold.

- The steps in a dynamic programming solution are:
- Verify that the principle of optimality holds. Set up the dynamic-programming recurrence equations. Solve the dynamic-programming recurrence equations for the value of the optimal solution. Perform a trace back step in which the solution itself is constructed.
- Dynamic programming differs from the greedy method since the greedy method produces only one feasible solution, which may or may not be optimal, while dynamic programming produces all possible sub-problems at most once, one of which guaranteed to be optimal. Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered

- Two difficulties may arise in any application of dynamic programming:
- 1. It may not always be possible to combine the solutions of smaller problems to form the solution of a larger one.
- 2.The number of small problems to solve may be un-acceptably large.
- There is no characterized precisely which problems can be effectively solved with dynamic programming; there are many hard problems for which it does not seen to be applicable, as well as many easy problems for which it is less efficient than standard algorithms.

MULTI STAGE GRAPHS

- A multistage graph G = (V, E) is a directed graph in which the vertices are partitioned into k ≥2 disjoint sets Vi, 1 ≤i ≤k. In addition, if <u, v> is an edge in E, then u ∈Vi and v ∈Vi+1 for some i, 1 ≤i <k.
- Let the vertex ‘s’ is the source, and ‘t’ the sink. Let c (i, j) be the cost of edge <i, j>. The cost of a path from ‘s’ to ‘t’ is the sum of the costs of the edges on the path.
- The multistage graph problem is to find a minimum cost path from ‘s’ to ‘t’. Each set Vi defines a stage in the graph. Because of the constraints on E, every path from ‘s’ to ‘t’ starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k.
- A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every stop path is the result of a sequenceofk–2decisions.The ith

Decision involves determining which vertex in vi+1, 1 ≤i ≤k - 2, is to beon the path. Let c (i, j) be the cost of the path from source to destination. Then using the forward approach, we obtain:

$$\text{cost}(i, j) = \min_{l \text{ in } V_{i+1}} \{c(j, l) + \text{cost}(i+1, l)\}$$

$$\langle j, l \rangle \text{ in } E$$

.

Algorithm

- **Algorithm Fgraph**(G, k, n,p)
- // The input is a k-stage graph G = (V, E) with n vertices
- // indexed in order or stages. E is a set of edges and c [i,j]
- // is the cost of (i, j). p [1 : k] is a minimum cost path.
- {
- cost [n] :=0.0;
- for j:= n - 1 to 1 step – 1do
- { // compute cost[j]
- let r be a vertex such that (j, r) is an edge of G and c [j, r] + cost [r] is minimum; cost [j] := c [j, r] + cost[r];

```
d [j] :=r:
}
p [1] := 1; p [k] :=n;    // Find a minimum cost path.
for j := 2 to k - 1 do
p [j] := d [p [j -1]];
}
```

Algorithm

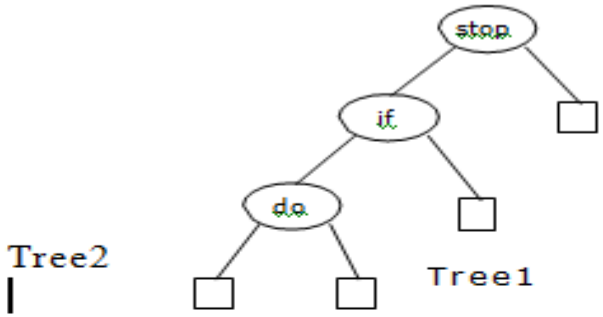
- **Algorithm Bgraph**(G, k, n,p)
- // Same function asFgraph
- {
- Bcost [1] :=0.0;
- for j := 2 to ndo
- { // Compute Bcost[j].
- Let r be such that (r, j) is an edge of G and Bcost [r] + c [r, j] is minimum; Bcost [j] := Bcost [r] + c [r,j];
- D [j] :=r;
- } //find a minimum costpath
- p [1] := 1; p [k] :=n;
- for j:= k - 1 to 2 do p [j] := d [p [j +1]];
- }

OPTIMAL BINARY SEARCH TREE(OBST)

- Let us assume that the given set of identifiers is {a1, . . . , an} with a1 < a2 < . . . . < an. Let p (i) be the probability with which we search for ai. Let q (i) be the probability that the identifier x being searched for is such that ai < x < ai+1, 0 ≤i ≤n (assume a0 = - ∞and an+1 = +∞). We have to arrange the identifiers in a binary search tree in a way that minimizes the expected total access time.
- In a binary search tree, the number of comparisons needed to access an element at depth 'd' is d + 1, so if 'ai' is placed at depth 'di', then we want to minimize:

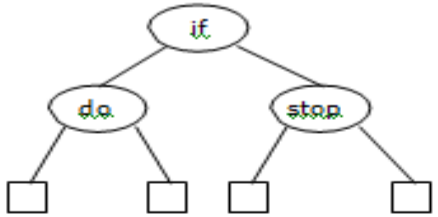
$$\begin{aligned}\text{Expected Cost of tree} &= \sum_{i=1}^{n} \text{cost}(k_i)p_i \\ &= \sum_{i=1}^{n} (\text{depth}(k_i) + 1)p_i \\ &= \sum_{i=1}^{n} \text{depth}(k_i)p_i + \sum_{i=1}^{n} p_i \\ &= \left(\sum_{i=1}^{n} \text{depth}(k_i)p_i\right) + 1\end{aligned}$$

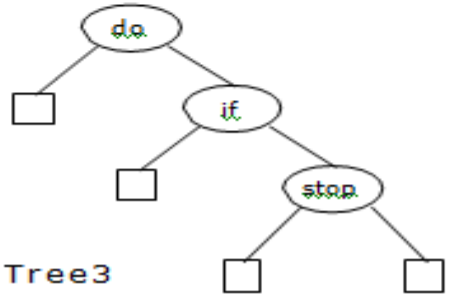
Example 1: The possible binary search trees for the identifier set (a1, a2, a3) = (do, if, stop) are as follows. Given the equal probabilities p (i) = Q (i) = 1/7 for all i, we have:



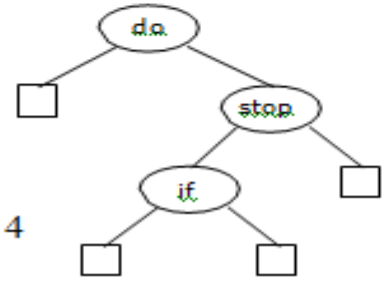
Tree2

Tree1





Tree3



Tree 4

- **Example1:**
- Let n = 4, and (a1, a2, a3, a4) = (do, if, need, while) Let P (1: 4) = (3, 3, 1, 1) and Q (0: 4) = (2, 3, 1, 1,1)

Solution:

Table for recording W (i, j), C (i, j) and R (i,j):

Column / Row	0	1	2	3	4
0	2, 0,0	3, 0,0	1, 0,0	1, 0,0,	1, 0,0
1	8, 8,1	7, 7,2	3, 3,3	3, 3,4	
2	12, 19,1	9, 12,2	5, 8,3		
3	14, 25,2	11, 19,2			
4	16, 32,2				

- **First**, computing all C (i, j) such that j - i = 1; j = i + 1 and as $0 \leq i < 4$; i = 0, 1, 2 and 3; $i < k \leq J$. Start with i = 0; so j = 1; as $i < k \leq j$, so the possible value for k =1
- W (0, 1) = P (1) + Q (1) + W (0, 0) = 3 + 3 + 2 =8
- C (0, 1) = W (0, 1) + min {C (0, 0) + C (1, 1)} =8
- R (0, 1) = 1 (value of 'K' that is minimum in the above equation). Next with i = 1; so j = 2; as $i < k \leq j$, so the possible value for k =2
- W (1, 2) = P (2) + Q (2) + W (1, 1) = 3 + 1 + 3 =7
- C (1, 2) = W (1, 2) + min {C (1, 1) + C (2, 2)} =7
- R (1, 2) =2
-

- Next with i = 2; so j = 3; as i < k ≤ j, so the possible value for k =3
- W (2, 3) = P (3) + Q (3) + W (2, 2) = 1 + 1 + 1 =3
- C (2, 3) = W (2, 3) + min {C (2, 2) + C (3, 3)} = 3 + [(0 + 0)] =3
- R (2, 3) =3
- Next with i = 3; so j = 4; as i < k ≤ j, so the possible value for k =4 W (3, 4) = P (4) + Q (4) + W (3, 3) = 1 + 1 + 1 =3
- C (3, 4) = W (3, 4) + min {[C (3, 3) + C (4, 4)]} = 3 + [(0 + 0)] =3
- R (3, 4) =4

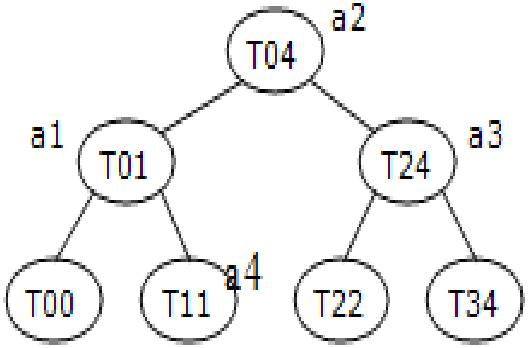
- **Second**, Computing all C (i, j) such that j - i = 2; j = i + 2 and as 0 ≤i < 3; i = 0, 1, 2; i < k ≤ J. Start with i = 0; so j = 2; as i < k ≤ J, so the possible values for k = 1 and2.
-
- W (0, 2) = P (2) + Q (2) + W (0, 1) = 3 + 1 + 8 =12
- C (0, 2) = W (0, 2) + min {(C (0, 0) + C (1, 2)), (C (0, 1) + C (2,2))}
- = 12 + min {(0 + 7, 8 + 0)} =19
- R (0, 2) =1
- Next, with i = 1; so j = 3; as i < k ≤ j, so the possible value for k = 2 and3.
-

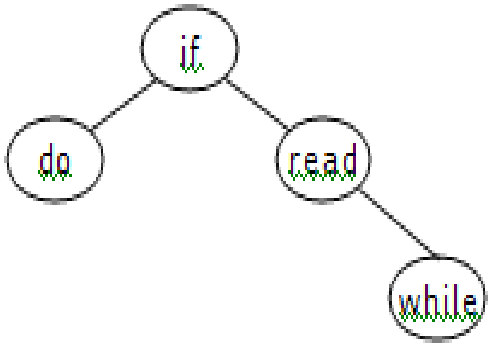
- W (1, 3) = P (3) + Q (3) + W (1, 2) = 1 + 1+ 7 =9
- C (1, 3) = W (1, 3) + min {[C (1, 1) + C (2, 3)], [C (1, 2) + C (3,3)]}
- = W (1, 3) + min {(0 + 3), (7 + 0)} = 9 + 3 =12
- R (1, 3) =2
- Next, with i = 2; so j = 4; as $i < k \leq j$, so the possible value for k = 3 and 4. W (2, 4) = P (4) + Q (4) + W (2, 3) = 1 + 1 + 3 =5
- C (2, 4) = W (2, 4) + min {[C (2, 2) + C (3, 4)], [C (2, 3) + C (4,4)]
- = 5 + min {(0 + 3), (3 + 0)} = 5 + 3 =8
- R (2, 4) =3

- **Third**, Computing all C (i, j) such that J - i = 3; j = i + 3 and as $0 \leq i < 2$; i = 0,1;
- $i < k \leq J$. Start with i = 0; so j = 3; as $i < k \leq j$, so the possible values for k = 1, 2 and3.
- W (0, 3) = P (3) + Q (3) + W (0, 2) = 1 + 1 + 12 =14
- C (0, 3) = W (0, 3) + min {[C (0, 0) + C (1, 3)], [C (0, 1) + C (2,3)],
- [C (0, 2) + C (3,3)]}
- = 14 + min {(0 + 12), (8 + 3), (19 + 0)} = 14 + 11 =25
- R (0, 3) =2
- Start with i = 1; so j = 4; as $i < k \leq j$, so the possible values for k = 2, 3 and4. W (1, 4) = P (4) + Q (4) + W (1, 3) = 1 + 1 + 9 =11
- C (1, 4) = W (1, 4) + min {[C (1, 1) + C (2, 4)], [C (1, 2) + C (3,4)],
- [C (1, 3) + C (4,4)]}
- = 11 + min {(0 + 8), (7 + 3), (12 + 0)} = 11 + 8 =19
- R (1, 4) =2

- **Fourth,** Computing all C (i, j) such that j - i = 4; j = i + 4 and as 0 ≤i < 1; i = 0; i < k ≤J.
- Start with i = 0; so j = 4; as i < k ≤ j, so the possible values for k = 1, 2, 3 and4.
- W (0, 4) = P (4) + Q (4) + W (0, 3) = 1 + 1 + 14 =16
- C (0, 4) = W (0, 4) + min {[C (0, 0) + C (1, 4)], [C (0, 1) + C (2,4)],
- [C (0, 2) + C (3, 4)], [C (0, 3) + C (4,4)]}
- = 16 + min [0 + 19, 8 + 8, 19+3, 25+0] = 16 + 16 =32
- R (0, 4) =2

- From the table we see that C (0, 4) = 32 is the minimum cost of a binary search tree for (a1, a2, a3, a4). The root of the tree 'T04' is'a2'.
- Hence the left sub tree is 'T01' and right sub tree is T24. The root of 'T01' is 'a1' and the root of 'T24' isa3.
- The left and right sub trees for 'T01' are 'T00' and 'T11' respectively. The root of T01 is 'a1'
-
- The left and right sub trees for T24 are T22 and T34 respectively. The root of T24 is'a3'.
- The root of T22 is null The root of T34 is a4.


T04
a2
a1
T01
T24
a3
T00
T11
a4
T22
T34


if
do
read
while

0/1 –KNAPSACK

- We are given n objects and a knapsack. Each object i has a positive weight wi and a positive value Vi. The knapsack can carry a weight not exceeding W. Fill the knapsack so that the value of objects in the knapsack is optimized.
- A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables x1, x2, ... , xn. A decision on variable xi involves determining which of the values 0or1 is to be assigned to it.Let us assume that decisions on the xi are made in the order xn, xn-1, ...x1. Following a decision on xn, we may be in one of two possible states: the capacity remaining in m – wn and a profit of pn has accrued.

It is clear that the remaining decisions xn-1, . . . , x1 must be optimal with respect to the problem state resulting from the decision on xn. Otherwise, xn,. . . . , x1 will not be optimal. Hence, the principal of optimality holds.

$$F_n(m) = \max\{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \quad -- \quad 1$$

For arbitrary $f_i(y)$, $i > 0$, this equation generalization:

$$F_i(y) = \max\{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \quad -- \quad 2$$

- Solution:
- Initially, fo (x) = 0, for all x and fi (x) = - ∞ if x < 0. Fn (M) = max {fn-1 (M), fn-1 (M - wn) +pn}
- F3 (6) = max (f2 (6), f2 (6 – 4) + 5} = max {f2 (6), f2 (2) +5}
- F2 (6) = max (f1 (6), f1 (6 – 3) + 2} = max {f1 (6), f1 (3) +2}
- F1 (6) = max (f0 (6), f0 (6 – 2) + 1} = max {0, 0 + 1} =1
- F1 (3) = max (f0 (3), f0 (3 – 2) + 1} = max {0, 0 + 1} =1
- Therefore, F2 (6) = max (1, 1 + 2} =3
- F2 (2) = max (f1 (2), f1 (2 – 3) + 2} = max {f1 (2), - 0+ 2}
- F1 (2) = max (f0 (2), f0 (2 – 2) + 1} = max {0, 0 + 1} =1
- F2 (2) = max {1, - 0+ 2} =1
- Finally, f3 (6) = max {3, 1 + 5} =6

Other Solution:

For the given data we have:

$S^0 = \{(0,0)\}$; $S^0 = \{(1,2)\}$

$S^1 = (S^0 \cup S^0{}_1) = \{(0, 0), (1,2)\}$

$x - 2 = 0 \Rightarrow x = 2$ $\quad$ $y - 3 = 0 \Rightarrow y = 3$

$x - 2 = 1 \Rightarrow x = 3$ $\quad$ $y - 3 = 1 \Rightarrow y = 5$

$S^1{}_1 = \{(2, 3), (3,5)\}$

$S^2 = (S^1 \cup S^1{}_1) = \{(0, 0), (1, 2), (2, 3), (3,5)\}$

$x - 5 = 0 \Rightarrow x = 5$ $\quad$ $y - 4 = 0 \Rightarrow y = 4$

$x - 5 = 1 \Rightarrow x = 6$ $\quad$ $y - 4 = 1 \Rightarrow y = 6$

$x - 5 = 2 \Rightarrow x = 7$ $\quad$ $y - 4 = 2 \Rightarrow y = 7$

$x - 5 = 3 \Rightarrow x = 8$ $\quad$ $y - 4 = 3 \Rightarrow y = 9$

$S^2{}_1 = \{(5, 4), (6, 6), (7, 7), (8,9)\}$

$S^3 = (S^2 \cup S^2{}_1) = \{(0, 0), (1, 2), (2, 3), (3, 5), (5, 4), (6, 6), (7, 7), (8,9)\}$

By applying Dominance rule,

$S^3 = (S^2 \cup S^2{}_1) = \{(0, 0), (1, 2), (2, 3), (5, 4), (6,6)\}$

From (6, 6) we can infer that the maximum Profit $\sum p_i x_i = 6$ and weight $\sum x_i w_i = 6$

All pairs shortest path Problem

- In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G. That is, for every pair of vertices (i, j), we are to find a shortest path from i to j as well as one from j to i. These two paths are the same when G is undirected.
- When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.
- The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires O (n2) time, the matrix A can be obtained in O (n3) time.

- In the all pairs shortest path problem, we are to find a shortest path between every pair of vertices in a directed graph G. That is, for every pair of vertices (i, j), we are to find a shortest path from i to j as well as one from j to i. These two paths are the same when G is undirected.
- When no edge has a negative length, the all-pairs shortest path problem may be solved by using Dijkstra's greedy single source algorithm n times, once with each of the n vertices as the source vertex.
- The all pairs shortest path problem is to determine a matrix A such that A (i, j) is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source problems using the algorithm shortest Paths. Since each application of this procedure requires O (n2) time, the matrix A can be obtained in O (n3)time.

- The shortest i to j path in G, i ≠ j originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively.
- Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. Let Ak (i, j) represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain:
- $A^k(i, j) = \{\min \{\min \{A^{k-1}(i, k) + A^{k-1}(k, j)\}, c(i,j)\}$
- $1 \leq k \leq n$

- **Algorithm All Paths** (Cost, A,n)
- // cost [1:n, 1:n] is the cost adjacency matrix of a graph which
- // n vertices; A [I, j] is the cost of a shortest path from vertex
- // i to vertex j. cost [i, i] = 0.0, for 1 ≤i ≤n.
- {
- for i := 1 to n do
- for j:= 1 to n do
- A [i, j] := cost [i,j]; // copy cost into A
- for k := 1 to n do
- for i := 1 to n do
- for j := 1 to n do
- A [i, j] := min (A [i, j], A [i, k] + A [k,j]);
- }

- General formula: min {Ak-1 (i, k) + Ak-1 (k, j)}, c (i,j)}
- $1 \leq k \leq n$
- Solve the problem for different values of k = 1, 2 and3
- **Step 1**: Solving the equation for, k =1;
- A1 (1, 1) = min {(Ao (1, 1) + Ao (1, 1)), c (1, 1)} = min {0 + 0, 0} =0
- A1 (1, 2) = min {(Ao (1, 1) + Ao (1, 2)), c (1, 2)} = min {(0 + 4), 4} =4
- A1 (1, 3) = min {(Ao (1, 1) + Ao (1, 3)), c (1, 3)} = min {(0 + 11), 11} =11
- A1 (2, 1) = min {(Ao (2, 1) + Ao (1, 1)), c (2, 1)} = min {(6 + 0), 6} =6
- A1 (2, 2) = min {(Ao (2, 1) + Ao (1, 2)), c (2, 2)} = min {(6 + 4), 0)} =0
- A1 (2, 3) = min {(Ao (2, 1) + Ao (1, 3)), c (2, 3)} = min {(6 + 11), 2} =2
- A1 (3, 1) = min {(Ao (3, 1) + Ao (1, 1)), c (3, 1)} = min {(3 + 0), 3} =3
- A1 (3, 2) = min {(Ao (3, 1) + Ao (1, 2)), c (3, 2)} = min {(3 + 4), 0} =7
- A1 (3, 3) = min {(Ao (3, 1) + Ao (1, 3)), c (3, 3)} = min {(3 + 11), 0} =0

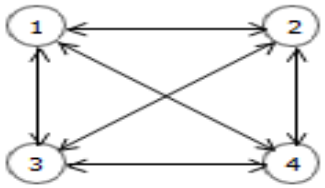
- **Step 2**: Solving the equation for, K =2;
- A2 (1, 1) = min {(A1 (1, 2) + A1 (2, 1), c (1, 1)} = min {(4 + 6), 0} = 0
- A2 (1, 2) = min {(A1 (1, 2) + A1 (2, 2), c (1, 2)} = min {(4 + 0), 4} = 4
- A2 (1, 3) = min {(A1 (1, 2) + A1 (2, 3), c (1, 3)} = min {(4 + 2), 11} =6
- A2 (2, 1) = min {(A (2, 2) + A (2, 1), c (2, 1)} = min {(0 + 6), 6} =6
- A2 (2, 2) = min {(A (2, 2) + A (2, 2), c (2, 2)} = min {(0 + 0), 0} =0
- A2 (2, 3) = min {(A (2, 2) + A (2, 3), c (2, 3)} = min {(0 + 2), 2} =2
- A2 (3, 1) = min {(A (3, 2) + A (2, 1), c (3, 1)} = min {(7 + 6), 3} =3
- A2 (3, 2) = min {(A (3, 2) + A (2, 2), c (3, 2)} = min {(7 + 0), 7} =7
- A2 (3, 3) = min {(A (3, 2) + A (2, 3), c (3, 3)} = min {(7 + 2), 0} =0

- **Step 3**: Solving the equation for, k =3;
- A3 (1, 1) = min {A2 (1, 3) + A2 (3, 1), c (1, 1)} = min {(6 + 3), 0} =0
- A3 (1, 2) = min {A2 (1, 3) + A2 (3, 2), c (1, 2)} = min {(6 + 7), 4} =4
- A3 (1, 3) = min {A2 (1, 3) + A2 (3, 3), c (1, 3)} = min {(6 + 0), 6} =6
- A3 (2, 1) = min {A2 (2, 3) + A2 (3, 1), c (2, 1)} = min {(2 + 3), 6} =5
- A3 (2, 2) = min {A2 (2, 3) + A2 (3, 2), c (2, 2)} = min {(2 + 7), 0} =0
- A3 (2, 3) = min {A2 (2, 3) + A2 (3, 3), c (2, 3)} = min {(2 + 0), 2} =2
- A3 (3, 1) = min {A2 (3, 3) + A2 (3, 1), c (3, 1)} = min {(0 + 3), 3} =3
- A3 (3, 2) = min {A2 (3, 3) + A2 (3, 2), c (3, 2)} = min {(0 + 7), 7} =7
- A3 (3, 3) = min {A2 (3, 3) + A2 (3, 3), c (3, 3)} = min {(0 + 0), 0} =0

TRAVELLING SALES PERSON PROBLEM

- Let G = (V, E) be a directed graph with edge costs Cij. The variable cijis defined such that cij> 0 for all I and j and cij= ?if < i, j>?E.
- Let |V| = n and assume n > 1. A tour of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour.
- The traveling sales person problem is to find a tour of minimum cost. The tour is to be a simple path that starts and ends at vertex1.
- Let g (i, S) be the length of shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function g (1, V – {1}) is the length of an optimal salesperson tour. From the principal of optimality it followsthat:
- C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

- **Example1:**
- For the following graph find minimum cost tour for the traveling sales person problem:



The cost adjacency matrix =

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

Let us start the tour from vertex1:

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c1k + g(k, V - \{1,K\})\} \quad - \quad (1)$$

More generally writing:

$$g(i, s) = \min \{c_{ij} + g(J, s - \{J\})\} \quad - \quad (2)$$

- Clearly, g (i, 0) = ci1 , 1 ≤ i ≤ n.
- g (2, 0) = C21 =5
- g (3, 0) = C31 = 6 g (4, 0) = C41 =8
- Using equation – (2) we obtain:
- g (1, {2, 3, 4}) = min {c12 + g (2, {3, 4}, c13 + g (3, {2, 4}), c14 + g (4, {2,3})}
- g (2, {3, 4}) = min {c23 + g (3, {4}), c24 + g (4,{3})}
- = min {9 + g (3, {4}), 10 + g (4,{3})}
- g (3, {4}) = min {c34 + g (4, 0)} = 12 + 8 =20
- g (4, {3}) = min {c43 + g (3, 0)} = 9 + 6 =15
-
-

- Therefore, g (2, {3, 4}) = min {9 + 20, 10 + 15} = min {29, 25} =25
- g (3, {2, 4}) = min {(c32 + g (2, {4}), (c34 + g (4,{2})}
- g (2, {4}) = min {c24 + g (4, 0)} = 10 + 8 =18
- g (4, {2}) = min {c42 + g (2, 0)} = 8 + 5 =13
- Therefore, g (3, {2, 4}) = min {13 + 18, 12 + 13} = min {41, 25} =25
- g (4, {2, 3}) = min {c42 + g (2, {3}), c43 + g (3,{2})}
- g (2, {3}) = min {c23 + g (3, 0} = 9 + 6 =15
- g (3, {2}) = min {c32 + g (2, 0} = 13 + 5 =18
- Therefore, g (4, {2, 3}) = min {8 + 15, 9 + 18} = min {23, 27} =23
- g (1, {2, 3, 4}) = min {c12 + g (2, {3, 4}), c13 + g (3, {2, 4}), c14 + g (4, {2,3})}= min {10 + 25, 15 + 25, 20 + 23} = min {35, 40, 43} =35
- The optimal tour for the graph has length = 35 The optimal tour is: 1, 2, 4, 3,1.

UNIT 4
BACKTRACKING AND BRANCH AND BOUND

BACKTRACKING: GENERAL METHOD

- Backtracking is used to solve problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criterion. The desired solution is expressed as an n-tuple (x1, ... , xn) where each xi Є S, S being a finite set.

- The solution is based on finding one or more vectors that maximize, minimize, or satisfy a criterion function P (x1, ... , xn). Form a solution and check at every step if this has any chance of success. If the solution at any point seems not promising, ignore it.

- All solutions requires a set of constraints divided into two categories: explicit and implicit constraints.

- **Definition 1:** Explicit constraints are rules that restrict each xi to take on values only from a given set. Explicit constraints depend on the particular instance I of problem being solved. All tuples that satisfy the explicit constraints define a possible solution space for I.

- **Definition 2:** Implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus, implicit constraints describe the way in which the xi‟s must relate to each other.

8-queens problem

Explicit constraints using 8-tuple formation, for this problem are S= {1, 2, 3, 4, 5, 6, 7, 8}.

- The implicit constraints for this problem are that no two queens can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

- Backtracking is a modified depth first search of a tree. Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a tree organization for the solution space.

- Backtracking is the procedure where by, after determining that a node can lead to nothing but dead end, we go back (backtrack) to the nodes parent and proceed with the search on the next child.

- A backtracking algorithm need not actually create a tree. Rather, it only needs to keep track of the values in the current branch being investigated. This is the way we implement backtracking algorithm. We say that the state space tree exists implicitly in the algorithm because it is not actually constructed.

Terminology:

- ***Problem state*** is each node in the depth first search tree.
- ***Solution states*** are the problem states „S‟ for which the path from the root node to „S‟ defines a tuple in the solution space.
- ***Answer states*** are those solution states for which the path from root node to s defines a tuple that is a member of the set of solutions.

- ***State space*** is the set of paths from root node to other nodes. *State space* tree is the tree organization of the solution space. The state space trees are called static trees. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instance.

- In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called dynamic trees.

- ***Live node*** is a node that has been generated but whose children have not yet been generated.

- ***E-node*** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

- ***Dead node*** is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

- ***Branch and Bound*** refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

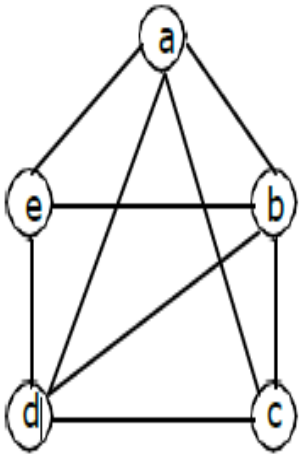
- Depth first node generation with bounding functions is called ***backtracking***. State generation methods in which the E-node remains the E-node until it is dead, lead to branch and bound methods.

Planar Graphs:

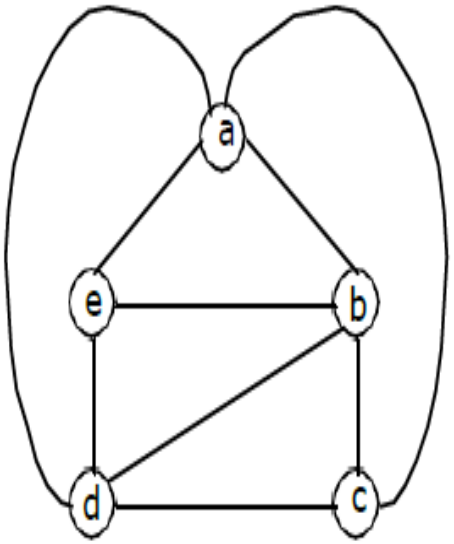
- When drawing a graph on a piece of a paper, we often find it convenient to permit edges to intersect at points other than at vertices of the graph. These points of interactions are called crossovers.

- A graph G is said to be planar if it can be drawn on a plane without any crossovers; otherwise G is said to be non-planar i.e., A graph is said to be planar iff it can be drawn in a plane in such a way that no two edges cross each other.

Example:



the following graph can be redrawn without crossovers as follows:

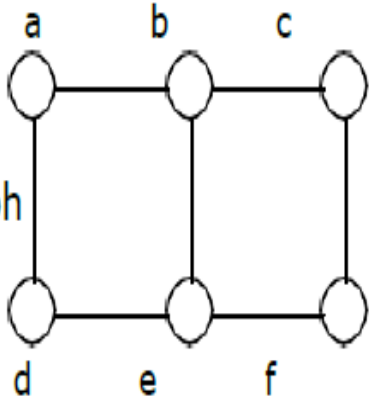


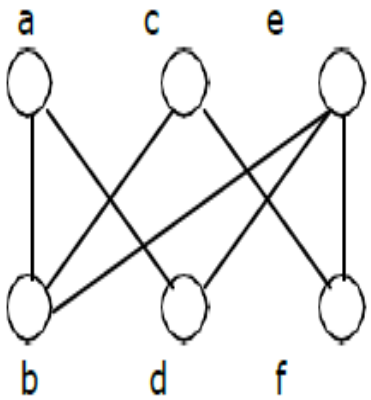
Bipartite Graph:

A bipartite graph is a non-directed graph whose set of vertices can be portioned into two sets V1 and V2 (i.e. V1 U V2 = V and V1 ∩ V2 = ø) so that every edge has one end in V1 and the other in V2. That is, vertices in V1 are only adjacent to those in V2 and vice- versa.

The vertex set V = {a, b, c, d, e, f} has been partitioned into V1 = {a, c, e} and V2 = {b, d, f}. The complete bipartite graph for which V1 = n and V2 = m is denoted Kn,m.

Example:

The graph 

is bipartite. We can redraw it as 

Applications-8-Queens Problem

Applications-8-Queens Problem

- Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two “attack”, that is, no two of them are on the same row, column, or diagonal.
- All solutions to the 8-queens problem can be represented as 8-tuples (x1, ... , x8), where xi is the column of the ith row where the ith queen is placed.
- The explicit constraints using this formulation are Si = {1, 2, 3, 4, 5, 6, 7, 8}, 1 < i < 8. Therefore the solution space consists of 8-tuples.

- The implicit constraints for this problem are that no two xi‟s can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

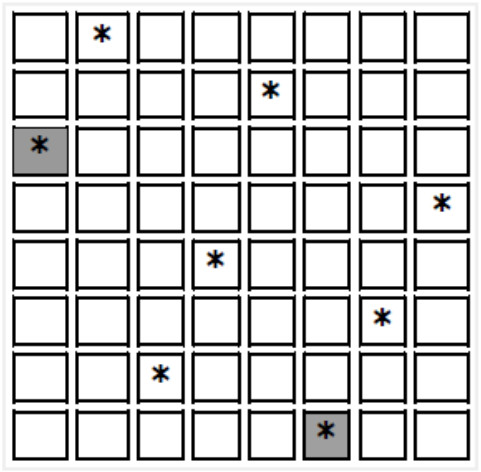
- Suppose two queens are placed at positions (i, j) and (k, l) Then:
- Column Conflicts: Two queens conflict if their xi values are identical.

- Diag 45 conflict: Two queens i and j are on the same 450 diagonal if:

- i – j = k – l.
- **This implies, j – l = i – k**

- Diag 135 conflict:
- i + j = k + l.
- **This implies, j – l = k – I**
- Where, j be the column of object in row i for the ith queen and l be the column of object in row „k" for the kth queen.

To check the diagonal clashes, let us take the following tile configuration:



In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

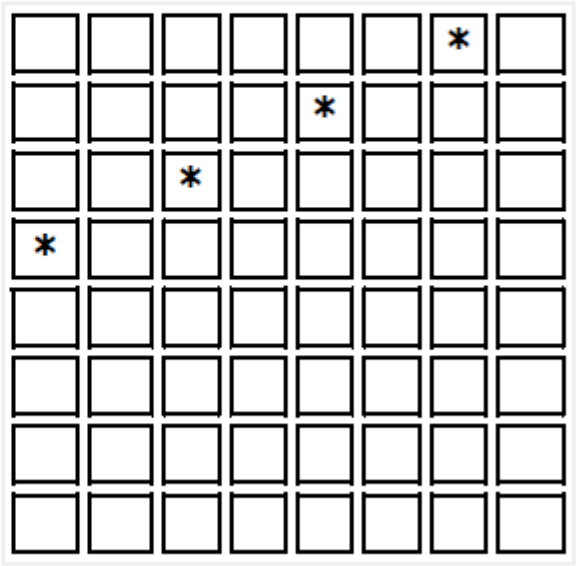
Let us consider for the 3rd row and 8th row case whether the queens on are conflicting or not. In this case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

$$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8|$$
$$\Rightarrow 5 = 5$$

In the above example we have, $|j - l| = |i - k|$, so the two queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.



Step 1:

- Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:

- If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less then 8, repeat Step 1.

Step 3:

- If the sequence is not feasible, then *backtrack through the sequence until we find the most recent place at which we can exchange a value. Go back to Step 1.*

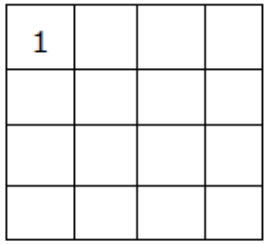
1	2	3	4	5	6	7	8	Remarks
7	5	3	1					
7	5	3	1*	2*				$\|j - l\| = \|1 - 2\| = 1$ $\|i - k\| = \|4 - 5\| = 1$
7	5	3	1	4				
7*	5	3	1	4	2*			$\|j - l\| = \|7 - 2\| = 5$ $\|i - k\| = \|1 - 6\| = 5$
7	5	3*	1	4	6*			$\|j - l\| = \|3 - 6\| = 3$ $\|i - k\| = \|3 - 6\| = 3$
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		$\|j - l\| = \|4 - 2\| = 2$ $\|i - k\| = \|5 - 7\| = 2$
7	5	3	1	4*	8	6*		$\|j - l\| = \|4 - 6\| = 2$ $\|i - k\| = \|5 - 7\| = 2$
7	5	3	1	4	8			*Backtrack*
7	5	3	1	4				*Backtrack*

7	5	3	1	6				
7*	5	3	1	6	2*			$\|j - l\| = \|1 - 2\| = 1$ $\|i - k\| = \|7 - 6\| = 1$
7	5	3	1	6	4			
7	5	3	1	6	4	2		
7	5	3*	1	6	4	2	8*	$\|j - l\| = \|3 - 8\| = 5$ $\|i - k\| = \|3 - 8\| = 5$
7	5	3	1	6	4	2		*Backtrack*
7	5	3	1	6	4			*Backtrack*
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	**SOLUTION**

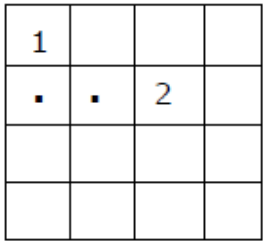
4-queens problem

4-queens problem

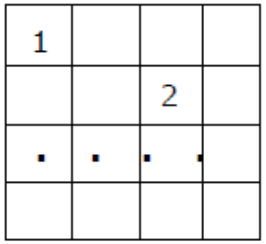
- Let us see how backtracking works on the 4-queens problem. We start with the root node as the only live node. This becomes the E-node. We generate one child.
- Let us assume that the children are generated in ascending order. Thus node number 2 of figure is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed.
- The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13.
- The path is now (1, 4). The board configurations as backtracking proceeds are as follows:



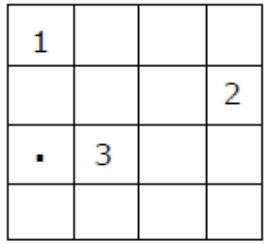
(a)



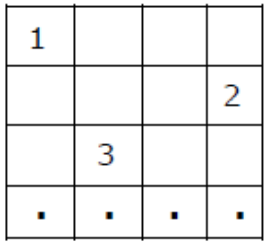
(b)



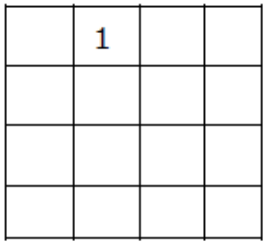
(c)



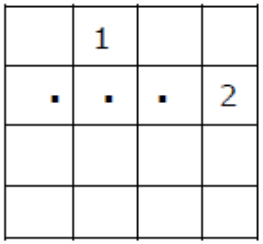
(d)



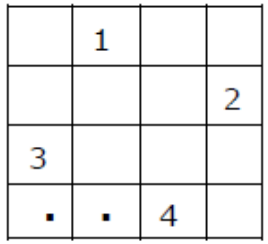
(e)



(f)

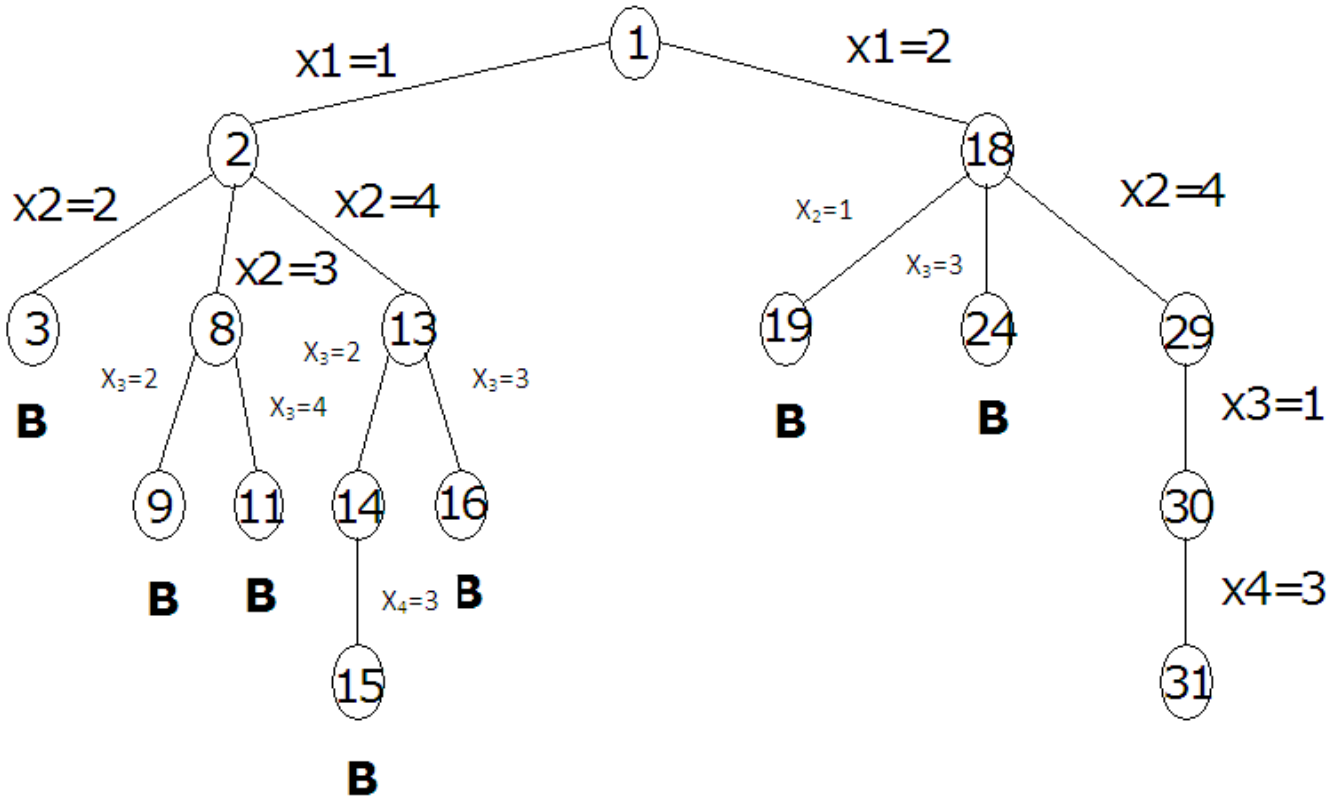


(g)



(h)

- The above figure shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen, which were tried and rejected because another queen was attacking.
- In Figure (b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In figure (c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In figure (d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.



Portion of the tree generated during backtracking

Complexity Analysis:

$$1 + n + n^2 + n^3 + \ldots\ldots\ldots\ldots + n^n = \frac{n^{n+1} - 1}{n - 1}$$

For the instance in which n = 8, the state space tree contains:

$$\frac{8^{8+1} - 1}{8 - 1} = 19,173,961 \text{ nodes}$$

Sum of Subsets

Sum of Subsets

- Given positive numbers wi, 1 ≤ i ≤ n, and m, this problem requires finding all subsets of wi whose sums are „m".
- All solutions are k-tuples, 1 ≤ k ≤ n. Explicit constraints:
- xi **Є {j | j is an integer and 1 ≤ j ≤ n}.**

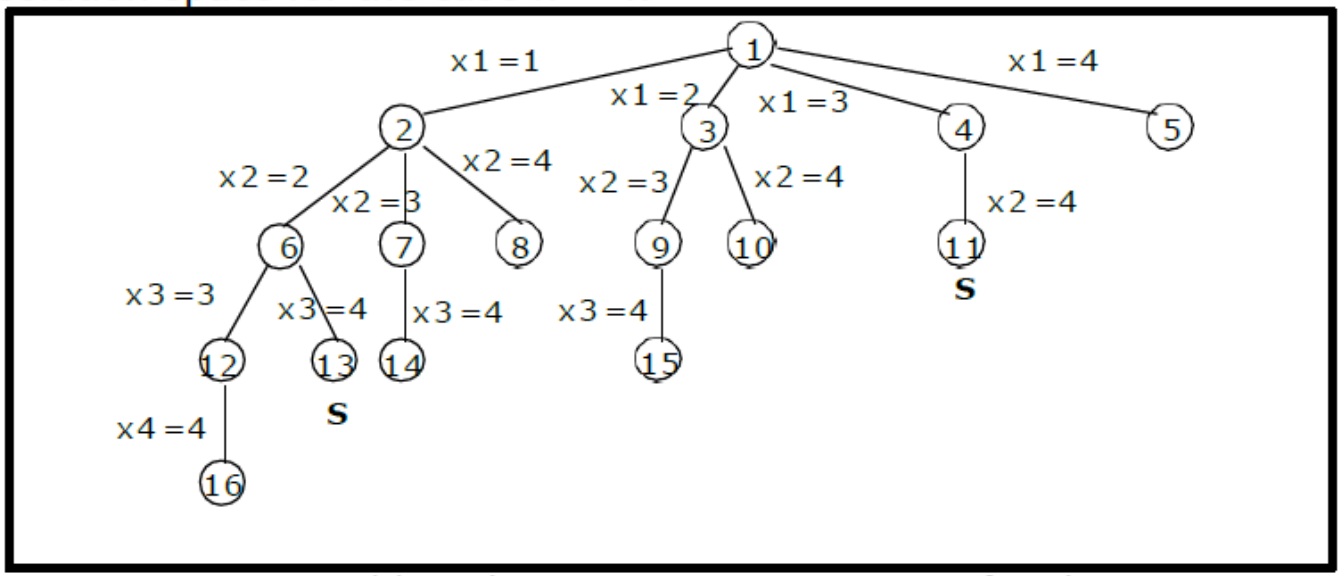
- Implicit constraints:
- No two xi can be the same.

- The sum of the corresponding wi"s be m.

- xi < xi+1 , 1 ≤ i < k (total order in indices) to avoid generating multiple instances of the same subset (for example, (1, 2, 4) and (1, 4, 2) represent the same subset).

- A better formulation of the problem is where the solution subset is represented by an n-tuple (x1, ... , xn) such that xi **Є {0, 1}.**
- The above solutions are then represented by (1, 1, 0, 1) and (0, 0, 1, 1). For both the above formulations, the solution space is 2n distinct tuples.
- For example, n = 4, w = (11, 13, 24, 7) and m = 31, the desired subsets are (11,
- 13, 7) and (24, 7).

The following figure shows a possible tree organization for two possible formulations of the solution space for the case n = 4.



A possible solution space organisation for the sum of the subsets problem.

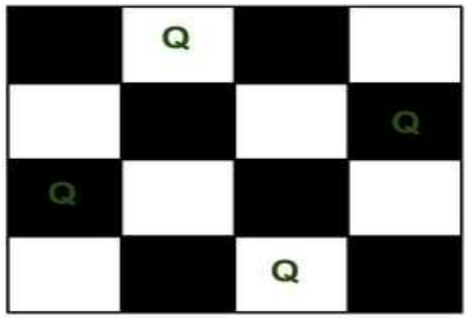
- The tree corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level i+1 node represents a value for xi. At each node, the solution space is partitioned into sub - solution spaces.
- All paths from the root node to any node in the tree define the solution space, since any such path corresponds to a subset satisfying the explicit constraints.

- The possible paths are (1), (1, 2), (1, 2, 3), (1, 2, 3, 4), (1, 2, 4), (1, 3, 4), (2), (2,
- 3), and so on. Thus, the left mot sub-tree defines all subsets containing w1, the next sub-tree defines all subsets containing w2 but not w1, and so on.

N-Queens Problem

N Queen Problem

- The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



- The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example following is the output matrix for above 4 queen solution.
- { 0, 1, 0, 0}
- { 0, 0, 0, 1}
- { 1, 0, 0, 0}
- { 0, 0, 1, 0}

Naive Algorithm

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

- while there are untried conflagrations{ generate the next configuration if queens don't attack in this configuration then
- { print this configuration;
- }
- }

Backtracking Algorithm

- The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes then we backtrack and return false.
- 1) Start in the leftmost column
- 2) If all queens are placed return true
- 3) Try all rows in the current column. Do following for every tried row.
- a) If the queen can be placed safely in this row then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
- b) If placing queen in [row, column] leads to a solution then return true.
- c) If placing queen doesn't lead to a solution then umark this [row, column] (Backtrack) and go to step (a) to try other rows.
- 3) If all rows have been tried and nothing worked, return false to trigger backtracking.

Implementation of Backtracking solution

```
# Python program to solve N Queen  using backtracking
 global N
N = 4
 def printSolution(board):
    for i in range(N):
        for j in range(N):
            print board[i][j],
        print

 # A utility function to check if a queen can # be placed on board[row][col]. Note that this # function is called when "col" queens are # already placed in columns from 0 to col -1.
# So we need to check only left side for # attacking queens
def isSafe(board, row, col):      # Check this row on left side
    for i in range(col):
        if board[row][i] == 1:
            return False     # Check upper diagonal on left side
    for i,j in zip(range(row,-1,-1), range(col,-1,-1)):
        if board[i][j] == 1:
```

```
    # Check lower diagonal on left side
    for i,j in zip(range(row,N,1), range(col,-1,-1)):
        if board[i][j] == 1:
            return False
     return True

def solveNQUtil(board, col):
    # base case: If all queens are placed     # then return true
    if col >= N:
        return True

    # Consider this column and try placing     # this queen in all rows one by one
    for i in range(N):
        if isSafe(board, i, col):
            # Place this queen in board[i][col]
            board[i][col] = 1
             # recur to place rest of the queens
            if solveNQUtil(board, col+1) == True:
                return True
             # If placing queen in board[i][col           # doesn't lead to a solution, then           # queen from board[i][col]
            board[i][col] = 0
```

```
# if queen can not be place in any row in     # this colum col  then return false
   return False
# This function solves the N Queen problem using # Backtracking. It mainly uses solveNQUtil() to
# solve the problem. It returns false if queens # cannot be placed, otherwise return true and
# placement of queens in the form of 1s. # note  that there may be more than one
# solutions, this function prints one  of the # feasible solutions.
def solveNQ():
   board = [ [0, 0, 0, 0],
            [0, 0, 0, 0],
            [0, 0, 0, 0],
            [0, 0, 0, 0]
           ]
    if solveNQUtil(board, 0) == False:
      print "Solution does not exist"
      return False
    printSolution(board)
   return True
# driver program to test above function
```

Applications-8-Queens Problem

- Let us consider, N = 8. Then 8-Queens Problem is to place eight queens on an 8 x 8 chessboard so that no two “attack”, that is, no two of them are on the same row, column, or diagonal.
- All solutions to the 8-queens problem can be represented as 8-tuples (x1, ... , x8), where xi is the column of the ith row where the ith queen is placed.
- The explicit constraints using this formulation are Si = {1, 2, 3, 4, 5, 6, 7, 8}, 1 < i < 8. Therefore the solution space consists of 88 8-tuples.

- The implicit constraints for this problem are that no two xi‟s can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal.

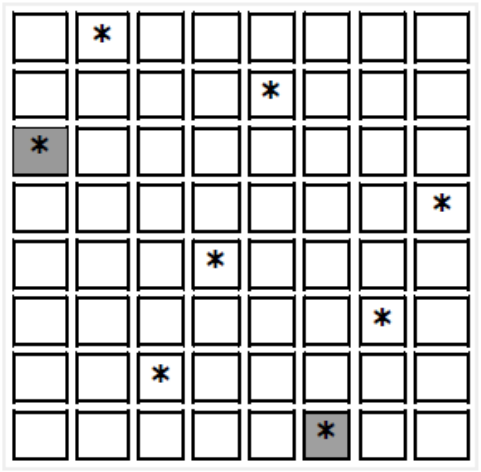
- Suppose two queens are placed at positions (i, j) and (k, l) Then:
- Column Conflicts: Two queens conflict if their xi values are identical.

- Diag 45 conflict: Two queens i and j are on the same 450 diagonal if:

- i – j = k – l.
- **This implies, j – l = i – k**

- Diag 135 conflict:
- i + j = k + l.
- **This implies, j – l = k – l**
- Therefore, two queens lie on the same diagonal if and only if:
- |j - l| = |i – k|
-
- Where, j be the column of object in row i for the ith queen and l be the column of object in row „k" for the kth queen.

To check the diagonal clashes, let us take the following tile configuration:



In this example, we have:

i	1	2	3	4	5	6	7	8
x_i	2	5	1	8	4	7	3	6

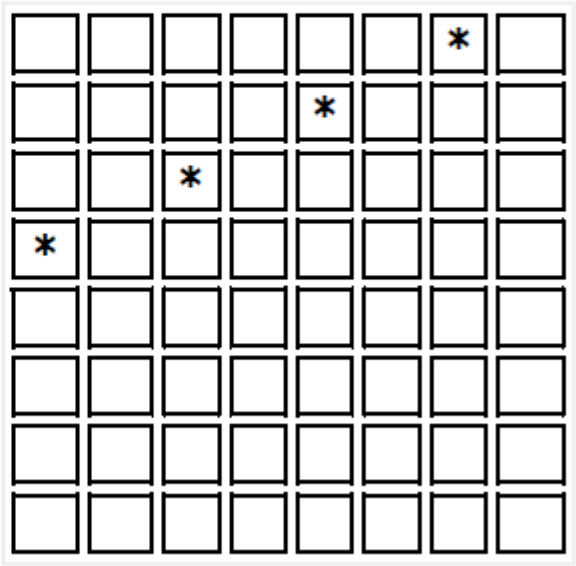
Let us consider for the 3rd row and 8th row case whether the queens on are conflicting or not. In this case (i, j) = (3, 1) and (k, l) = (8, 6). Therefore:

$$|j - l| = |i - k| \Rightarrow |1 - 6| = |3 - 8|$$
$$\Rightarrow 5 = 5$$

In the above example we have, $|j - l| = |i - k|$, so the two queens are attacking. This is not a solution.

Example:

Suppose we start with the feasible sequence 7, 5, 3, 1.



Step 1:

- Add to the sequence the next number in the sequence 1, 2, . . . , 8 not yet used.

Step 2:

- If this new sequence is feasible and has length 8 then STOP with a solution. If the new sequence is feasible and has length less then 8, repeat Step 1.

Step 3:

- If the sequence is not feasible, then *backtrack through the sequence until we find the most recent place at which we can exchange a value. Go back to Step 1.*

1	2	3	4	5	6	7	8	Remarks
7	5	3	1					
7	5	3	1*	2*				$\|j - l\| = \|1 - 2\| = 1$ $\|i - k\| = \|4 - 5\| = 1$
7	5	3	1	4				
7*	5	3	1	4	2*			$\|j - l\| = \|7 - 2\| = 5$ $\|i - k\| = \|1 - 6\| = 5$
7	5	3*	1	4	6*			$\|j - l\| = \|3 - 6\| = 3$ $\|i - k\| = \|3 - 6\| = 3$
7	5	3	1	4	8			
7	5	3	1	4*	8	2*		$\|j - l\| = \|4 - 2\| = 2$ $\|i - k\| = \|5 - 7\| = 2$
7	5	3	1	4*	8	6*		$\|j - l\| = \|4 - 6\| = 2$ $\|i - k\| = \|5 - 7\| = 2$
7	5	3	1	4	8			*Backtrack*
7	5	3	1	4				*Backtrack*

7	5	3	1	6				
7*	5	3	1	6	2*			$\|j - l\| = \|1 - 2\| = 1$ $\|i - k\| = \|7 - 6\| = 1$
7	5	3	1	6	4			
7	5	3	1	6	4	2		
7	5	3*	1	6	4	2	8*	$\|j - l\| = \|3 - 8\| = 5$ $\|i - k\| = \|3 - 8\| = 5$
7	5	3	1	6	4	2		*Backtrack*
7	5	3	1	6	4			*Backtrack*
7	5	3	1	6	8			
7	5	3	1	6	8	2		
7	5	3	1	6	8	2	4	**SOLUTION**

Graph Coloring

Graph Coloring (for planar graphs)

- Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color, yet only m colors are used. This is termed the m-colorabiltiy decision problem.
- The m-colorability optimization problem asks for the smallest integer m for which the graph G can be colored.
- Given any map, if the regions are to be colored in such a way that no two adjacent regions have the same color, only four colors are needed.
- For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient for planar graphs.

- The function m-coloring will begin by first assigning the graph to its adjacency matrix, setting the array x [] to zero. The colors are represented by the integers 1, 2, . . . , m and the solutions are given by the n-tuple (x1, x2, . . ., xn), where xi is the color of node i.
- A recursive backtracking algorithm for graph coloring is carried out by invoking the statement mcoloring(1);

Algorithm mcoloring (k)

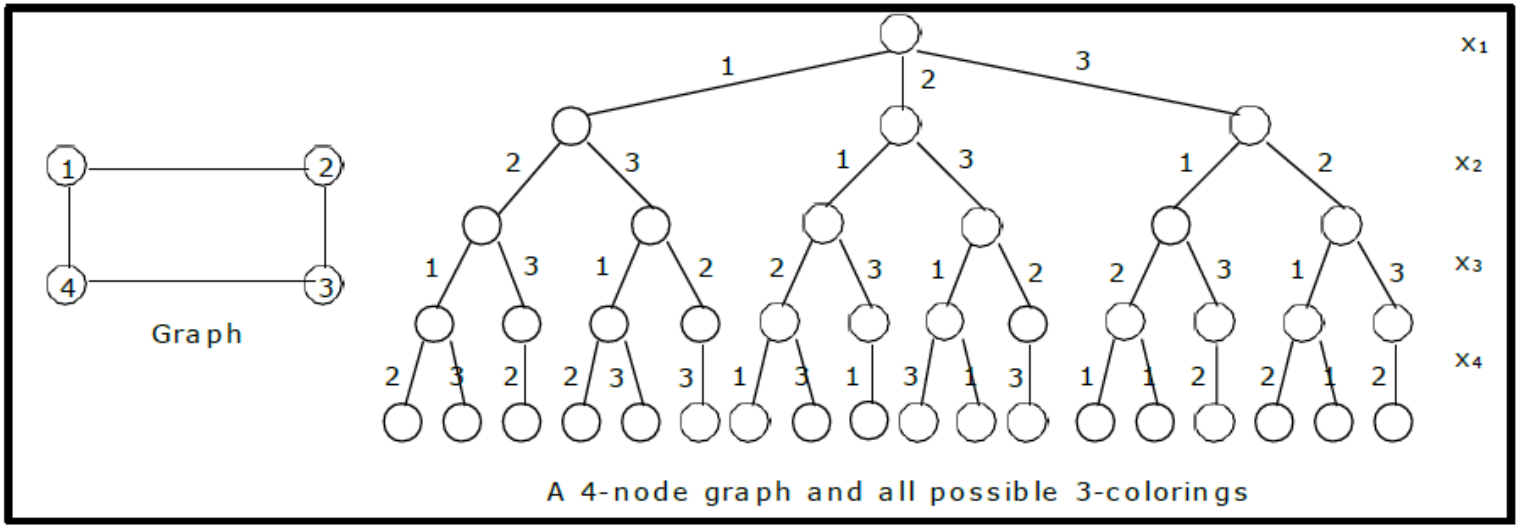
- // This algorithm was formed using the recursive backtracking schema. The graph is
- // represented by its Boolean adjacency matrix G [1: n, 1: n]. All assignments of
- // 1, 2, ... , m to the vertices of the graph such that adjacent vertices are assigned
- // distinct integers are printed. k is the index of the next vertex to color.
- {
- repeat
- { // Generate all legal assignments for x[k].
- NextValue (k); // Assign to x [k] a legal color. If (x [k] = 0) then return; // No new color possible If (k = n) then // at most m colors have been
- // used to color the n vertices.
- write (x [1: n]);
- else mcoloring (k+1);
- } until (false);
- }

Algorithm NextValue (k)

- // x [1] , ... x [k-1] have been assigned integer values in the range [1, m] such that
- // adjacent vertices have distinct integers. A value for x [k] is determined in the range
- // [0, m].x[k] is assigned the next highest numbered color while maintaining distinctness
- // from the adjacent vertices of vertex k. If no such color exists, then x [k] is 0.
- {
- repeat
- {
- x [k]: = (x [k] +1) mod (m+1) // Next highest color.
- If (x [k] = 0) then return; // All colors have been used for j := 1 to n do
- { // check if this color is distinct from adjacent colors if ((G [k, j] ≠ 0) and (x [k] = x [j]))
- // If (k, j) is and edge and if adj. vertices have the same color. then break;
- }
- if (j = n+1) then return; // New color found
- } until (false); // Otherwise try to find another color.
- }

Example:

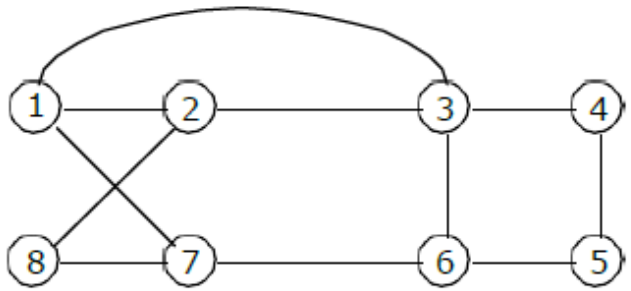
Color the graph given below with minimum number of colors by backtracking using state space tree.



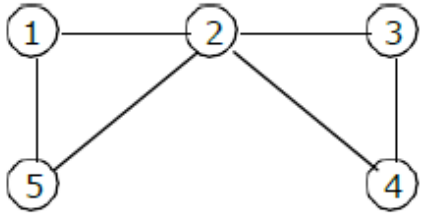
Hamiltonian Cycles

Hamiltonian Cycles

- Let G = (V, E) be a connected graph with n vertices. A Hamiltonian cycle (suggested by William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position.
- In other vertices of G are visited in the order v1, v2, ... , vn+1, then the edges (vi, vi+1) are in E, 1 < i < n, and the vi are distinct expect for v1 and vn+1, which are equal.
- The graph G1 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph G2 contains no Hamiltonian cycle.



Graph G1



Graph G2

Two graphs to illustrate Hamiltonian cycle

- The backtracking solution vector (x1, ... xn) is defined so that xi represents the ith visited vertex of the proposed cycle. If k = 1, then x1 can be any of the n vertices.
- To avoid printing the same cycle n times, we require that x1 = 1. If 1 < k < n, then xk can be any vertex v that is distinct from x1, x2, . . . , xk–1 and v is connected by an edge to kx-1. The vertex xn can only be one remaining vertex and it must be connected to both xn-1 and x1.

- Using NextValue algorithm we can particularize the recursive backtracking schema to find all Hamiltonian cycles. This algorithm is started by first initializing the adjacency matrix G[1: n, 1: n], then setting x[2: n] to zero and x[1] to 1, and then executing Hamiltonian(2).

- The traveling salesperson problem using dynamic programming asked for a tour that has minimum cost. This tour is a Hamiltonian cycles. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists.

Algorithm NextValue (k)

```
// x [1: k-1] is a path of k – 1 distinct vertices . If x[k] = 0, then no vertex has as yet been
// assigned to x [k]. After execution, x[k] is assigned to the next highest numbered vertex
// which does not already appear in x [1 : k – 1] and is connected by an edge to x [k – 1].
// Otherwise x [k] = 0. If k = n, then in addition x [k] is connected to x [1].
{
repeat
{
x [k] := (x [k] +1) mod (n+1); // Next vertex. If (x [k] = 0) then return;
If (G [x [k – 1], x [k]] ≠ 0) then
{ // Is there an edge?
for j := 1 to k – 1 do if (x [j] = x [k]) then break;
// check for distinctness.
If (j = k) then // If true, then the vertex is distinct. If ((k < n) or ((k = n) and G [x [n], x [1]] ≠ 0))
then return;
}
} until (false);
}
```

Hamiltonian Cycles

- **Algorithm NextValue** (k)
- // x [1: k-1] is a path of k – 1 distinct vertices . If x[k] = 0, then no vertex has as yet been
- // assigned to x [k]. After execution, x[k] is assigned to the next highest numbered vertex
- // which does not already appear in x [1 : k – 1] and is connected by an edge to x [k – 1].
- // Otherwise x [k] = 0. If k = n, then in addition x [k] is connected to x [1].
- {
- repeat
- {
- x [k] := (x [k] +1) mod (n+1); // Next vertex. If (x [k] = 0) then return;
- If (G [x [k – 1], x [k]] □ 0) then

- { // Is there an edge?
- for j := 1 to k – 1 do if (x [j] = x [k]) then break;
- // check for distinctness.
- If (j = k) then // If true, then the vertex is distinct.
  If ((k < n) or ((k = n) and G [x [n], x [1]] □ 0))
- then return;
- }
- } until (false);
- }

- **Algorithm Hamiltonian (k)**
- // This algorithm uses the recursive formulation of backtracking to find all the Hamiltonian
- // cycles of a graph. The graph is stored as an adjacency matrix G [1: n, 1: n]. All cycles begin
- // at node 1.
- {

- repeat
- { // Generate values for x [k].
- NextValue (k); //Assign a legal Next value to x [k]. if (x [k] = 0) then return;
- if (k = n) then write (x [1: n]); else Hamiltonian (k + 1)
- } until (false);
- }

Branch and Bound

Branch and Bound: General method

- Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

- 1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.
- 2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

- Branch and Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node
- Branch and Bound is the generalization of both graph search strategies, BFS and D- search.

- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).

- A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

- *Definition 1:* Live node is a node that has been generated but whose children have not yet been generated.

- *Definition 2:* E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

- *Definition 3:* Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

- *Definition 4:* Branch-an-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

- *Definition 5:* The adjective "heuristic", means" related to improving problem solving performance".

- As a noun it is also used in regard to "any method or trick used to improve the efficiency of a problem solving problem". But imperfect methods are not necessarily heuristic or vice versa. "A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy, trick simplification or any other kind of device which drastically limits search for solutions in large problem spaces.
- Heuristics do not guarantee optimal solutions, they do not guarantee any solution at all. A useful heuristic offers solutions which are good enough most of the time.

Least Cost (LC) search:

- In both LIFO and FIFO Branch and Bound the selection rule for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.
- The search for an answer node can be speeded by using an “intelligent” ranking Function *c*(.) for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

- $c(x) = f(h(x)) + g(x)$

- where, $c(x)$ is the cost of x.

- h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

- $g(x)$ is an estimate of the additional effort needed to reach an answer node from x.
- A search strategy that uses a cost function $c(x) = f(h(x)) + g(x)$ to select the next E-node would always choose for its next E-node a live node with least LC–search (Least Cost search)
- $c(.)$ is called a BFS and D-search are special cases of LC-search.
- If $g(x) = 0$ and $f(h(x))$ = level of node x, then an LC search generates nodes by levels. This is eventually the same as a BFS. If $f(h(x)) = 0$ and essentially a D-search.
- $g(x) > g(y)$ whenever y is a child of x, then the search is An LC-search coupled with bounding functions is called an LC-branch and bound search
- We associate a cost c(x) with each node x in the state space tree. It is not possible to easily compute the function c(x). So we compute a estimate $c(x)$ of c(x).

Control Abstraction for LC-Search:

- Let t be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the subtree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.
- A heuristic *c* (.) is used to estimate c(). This heuristic should be easy to compute and
- generally has the property that if x is either an answer node or a leaf node, then c(x) =*c*(*x*).

- LC-search uses *c* to find an answer node. The algorithm uses two functions Least() and Add() to delete and add a live node from or to the list of live nodes, respectively.

- Least() finds a live node with least c(). This node is deleted from the list of live nodes and returned.

- Add(x) adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.
- Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When the answer node g is found, the path from g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

- Listnode = **record**
- {
- Listnode * next, *parent; float cost;
- }

- Algorithm **LCSearch**(t)
- { //Search t for an answer node
- if *t is an answer node then output *t and return; E := t; //E-node.
- initialize the list of live nodes to be empty; repeat
- {
- for each child x of E do
- {
- if x is an answer node then output the path from x to t and return; Add (x); //x is a new live node.
- (x à parent) := E; // pointer for path to root
- }
- if there are no more live nodes then
- {
- write (“No answer node”); return;
- }
- E := Least();
- } until (false);
- }

- The root node is the first, E-node. During the execution of LC search, this list contains all live nodes except the E-node. Initially this list should be empty. Examine all the children of the E-node, if one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If the child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes and its parent field set to E.
- When all the children of E have been generated, E becomes a dead node. This happens only if none of E‟s children is an answer node. Continue the search further until no live nodes found. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

- LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.

Bounding

Bounding

- A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. The three search methods differ only in the selection rule used to obtain the next E-node.
-
- A good bounding helps to prune efficiently the tree, leading to a faster exploration of the solution space.
-
- A cost function $c(.)$ such that $c(x) < c(x)$ is used to provide lower bounds on solutions obtainable from any node x. If upper is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $c(x) > c(x) >$ upper. The starting value for upper can be obtained by some heuristic or can be set to ∞

- As long as the initial value for upper is not less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of upper can be updated.
-
- Branch-and-bound algorithms are used for optimization problems where, we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.
-
- To formulate the search for an optimal solution for a least-cost answer node in a state space tree, it is necessary to define the cost function c(.), such that c(x) is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for c(.).

- For nodes representing feasible solutions, c(x) is the value of the objective function for that feasible solution.
- For nodes representing infeasible solutions
- For nodes representi $c(x) = \infty$. ons, c(x) is the cost of the minimum-cost node i ith root x.
- Since, c(x) is generally hard to compute, the branch-and-bound algorithm will use an estimate $c(x)$ such that $c(x) < c(x)$ for all x.

- **The 15 – Puzzle Problem:**

- The 15 puzzle is to search the state space for the goal state and use the path from the initial state to the goal state as the answer. There are 16! (16! ≈ 20.9 x 1012) different arrangements of the tiles on the frame.

- As the state space for the problem is very large it would be worthwhile to determine whether the goal state is reachable from the initial state. Number the frame positions 1 to 16.
-
- Position i is the frame position containing title numbered i in the goal arrangement of Figure 8.1(b). Position 16 is the empty spot. Let position(i) be the position number in the initial state of the title number i. Then position(16) will denote the position of the empty spot.

- For any state let: *less(i) be the number of tiles j such that j < i and position(j) > position(i).*
- The goal state is reachable from the initial state iff

Here, x = 1 if in the initial state the empty spot is at one of the shaded positions of figure 8.1(c) and x = 0 if it is at one of the remaining positions.

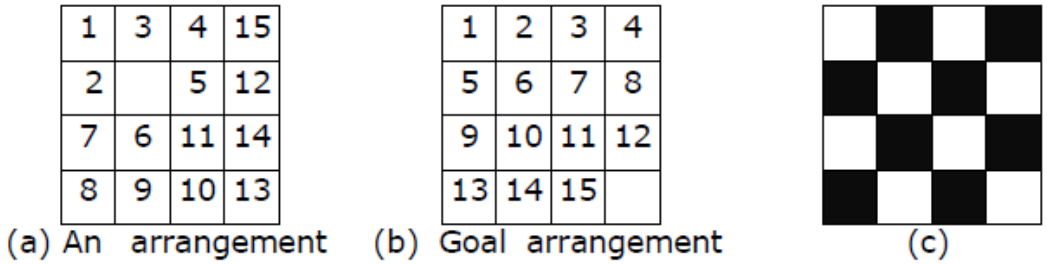


Figure 8.1. 15-puzzle arrangement

Example 1:

For the state of Figure 8.1(a) we have less(i) values as follows:

less(1) = 0	less(2) = 0	less(3) = 1	less(4) = 1
less(5) = 0	less(6) = 0	less(7) = 1	less(8) = 0
less(9) = 0	less(10) = 0	less(11) = 3	less(12) = 6
less(13) = 0	less(14) = 4	less(15) = 11	less(16) = 10

Therefore, $\sum_{i=1}^{16} less(i) + x = (0 + 0 + 1 + 1 + 0 + 0 + 1 + 0 + 0 + 0 + 3 + 6 + 0 + 4 + 11 + 10) + 0 = 37 + 0 = 37.$

Hence, goal is *not reachable*.

Example 2:

For the root state of Figure 8.2 we have less(i) values are as follows:

less(1) = 0	less(2) = 0	less(3) = 0	less(4) = 0
less(5) = 0	less(6) = 0	less(7) = 0	less(8) = 1
less(9) = 1	less(10) = 1	less(11) = 0	less(12) = 0
less(13) = 1	less(14) = 1	less(15) = 1	less(16) = 9

Therefore, $\sum_{i=1}^{16} less(i) + x = (0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 + 1 + 1 + 0 + 0 + 1 +$

$1 + 1 + 9) + 1 = 15 + 1 = 16.$

Hence, goal is *reachable*.

- **LC Search for 15 Puzzle Problem:**
- A depth first state space tree generation will result in the subtree of Figure 8.3 when the next moves are attempted in the order: move the empty space up, right, down and left. The search of the state space tree is blind. It will take the leftmost path from the root regardless of the starting configuration. As a result, the answer node may never be found.

- A breadth first search will always find a goal node nearest to the root. However, such a search is also blind in the sense that no matter what the initial configuration, the algorithm attempts to make the same sequence of moves.

- We need a more intelligent search method. We associate a cost c(x) with each node x in the state space tree. The cost c(x) is the length of a path from the root to a nearest goal node in the subtree with root x. The easy to compute estimate *c* (x) of c(x) is as follows:

- $c(x) = f(x) + g(x)$
- where, f(x) is the length of the path from the root to node x and
- $g(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x. Here, $g(x)$ is the number of nonblank tiles not in their goal position.
-
- An LC-search of Figure 8.2, begin with the root as the E-node and generate all child nodes 2, 3, 4 and 5. The next node to become the E-node is a live node with least
- $c(x)$.
- $c(2) = 1 + 4 = 5$
- $c(3) = 1 + 4 = 5$
- $c(4) = 1 + 2 = 3$ and
- $c(5) = 1 + 4 = 5$.
- Node 4 becomes the E-node and its children are generated. The live nodes at this time are 2, 3, 5, 10, 11 and 12. So:
- $c(10) = 2 + 1 = 3$
- $c(11) = 2 + 3 = 5$ and
- $c(12) = 2 + 3 = 5$.

- The live node with least $\hat{c}$ is node 10. This becomes the next E-node. Nodes 22 and 23 are generated next. Node 23 is the goal node, so search terminates.

- LC-search was almost as efficient as using the exact function c(), with a suitable choice for $\hat{c}$ (), LC-search will be far more selective than any of the other search methods.

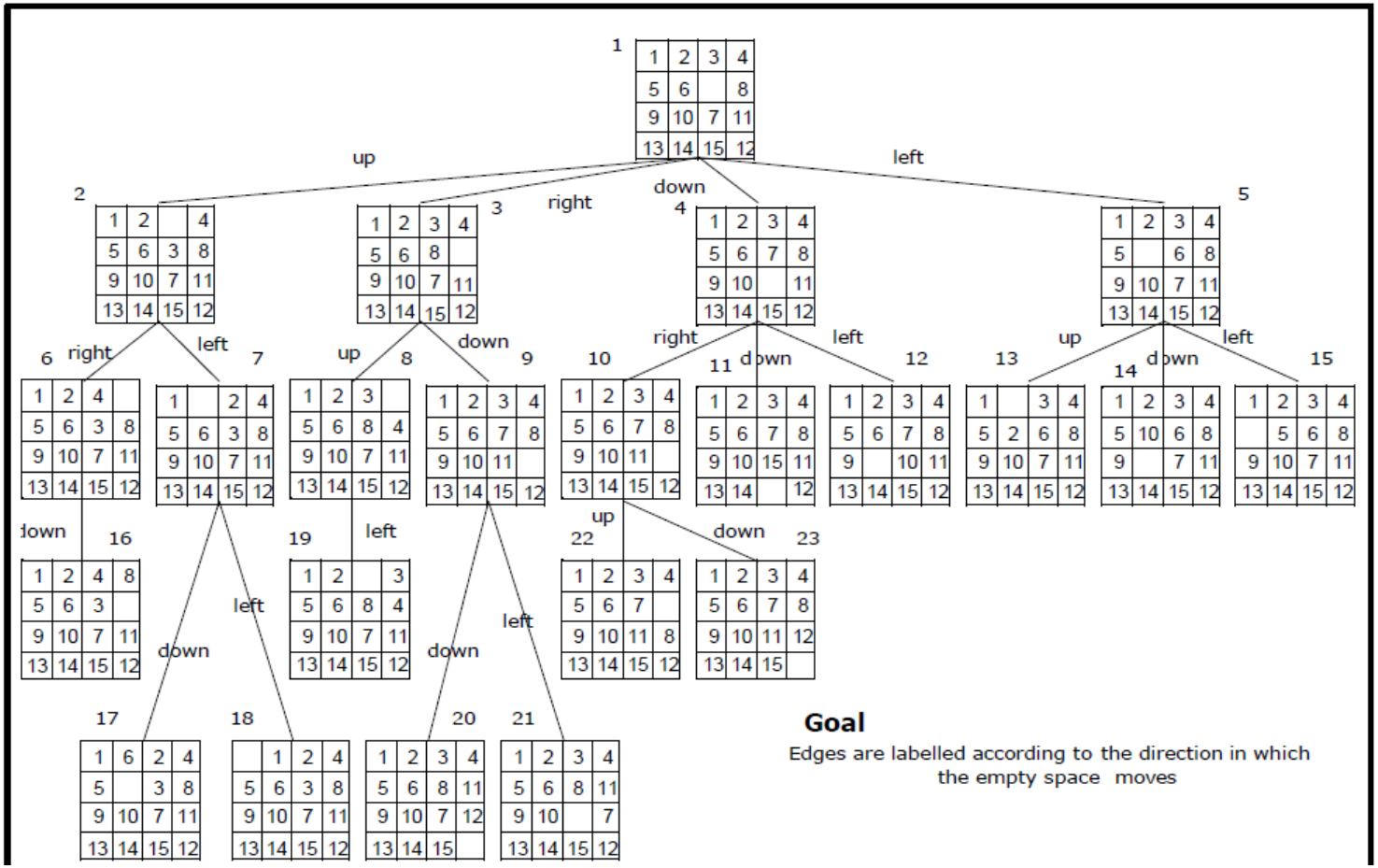


Figure 8.2. Part of the state space tree for 15-puzzle problem

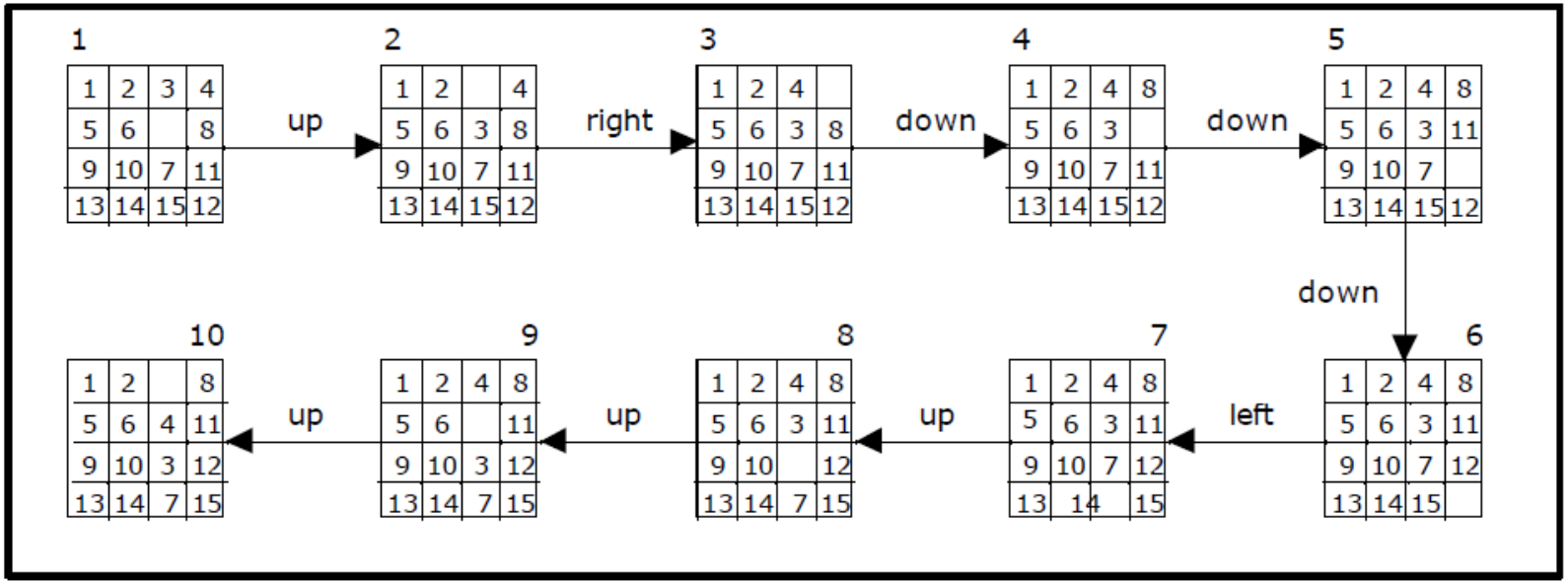


Figure 8. 3. First ten steps in a depth first search

0/1 Knapsack

0/1 Knapsack:

- Given n positive weights wi, n positive profits pi, and a positive number m that is the knapsack capacity, the problem calls for choosing a subset of the weights such that:

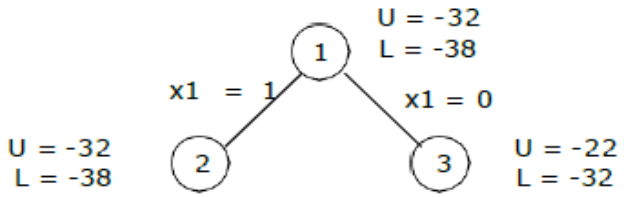
$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i \text{ is maximized.}$$

- The xi‟s constitute a zero–one-valued vector.
- The solution space for this problem consists of the 2n distinct ways to assign zero or one values to the xi‟s.
- Bounding functions are needed to kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than the value of the best solution determined so far, than that live node can be killed.

- We continue the discussion using the fixed tuple size formulation. If at node Z the values of xi, 1 < *i < k, have already been determined, then an upper bound for Z can be obtained by relaxing the requirements xi = 0 or 1.*
- *(Knapsack problem using backtracking is solved in branch and bound chapter)*

- **0/1 Knapsack Problem**
- Consider the instance: M = 15, n = 4, (P1, P2, P3, P4) = (10, 10, 12, 18) and
- (w1, w2, w3, w4) = (2, 4, 6, 9).
- 0/1 knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.
- Place first item in knapsack. Remaining weight of knapsack is 15 – 2 = 13. Place next item w2 in knapsack and the remaining weight of knapsack is 13 – 4 = 9. Place next item w3 in knapsack then the remaining weight of knapsack is 9 – 6 = 3. No fractions are allowed in calculation of upper bound so w4 cannot be placed in knapsack.
- Profit = P1 + P2 + P3 = 10 + 10 + 12
- So, Upper bound = 32
- To calculate lower bound we can place w4 in knapsack since fractions are allowed in calculation of lower bound.

- Lower bound = 10 + 10 + 12 + (3 *X* 18) = 32 + 6 = 38
-
- Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.
- Therefore, Upper bound (U) = -32
- Lower bound (L) = -38
- We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.
-



Now we will calculate upper bound and lower bound for nodes 2, 3.

For node 2, $x_1 = 1$, means we should place first item in the knapsack.

U = 10 + 10 + 12 = 32, make it as -32

$$L = 10 + 10 + 12 + \frac{3}{9} \times 18 = 32 + 6 = 38, \text{ make it as -38}$$

For node 3, $x_1 = 0$, means we should not place first item in the knapsack.

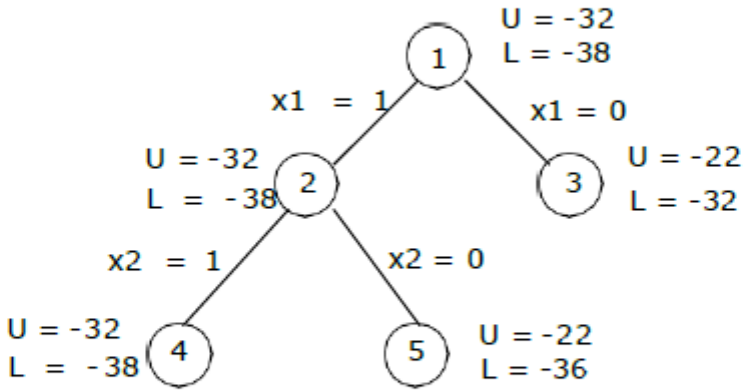
U = 10 + 12 = 22, make it as -22

$$L = 10 + 12 + \frac{5}{9} \times 18 = 10 + 12 + 10 = 32, \text{ make it as -32}$$

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3

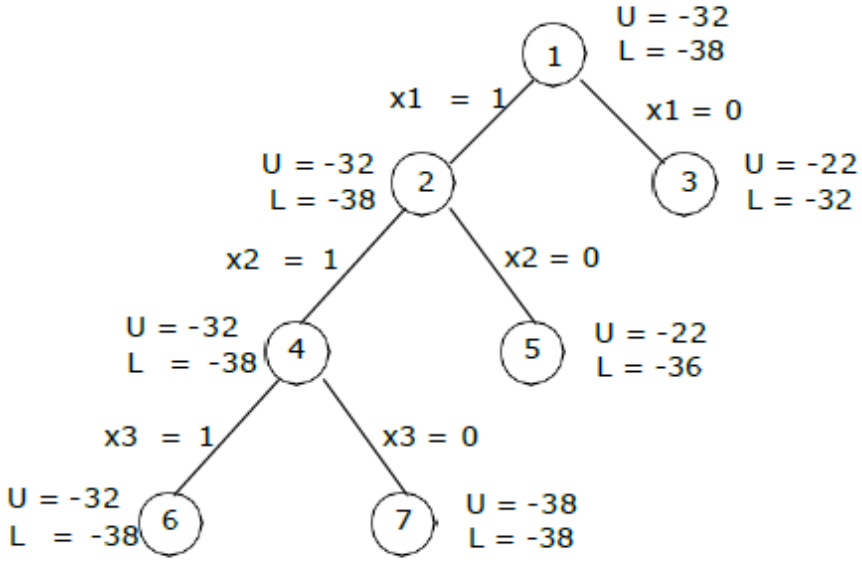
For node 2, U – L = -32 + 38 = 6
For node 3, U – L = -22 + 32 = 10

Choose node 2, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

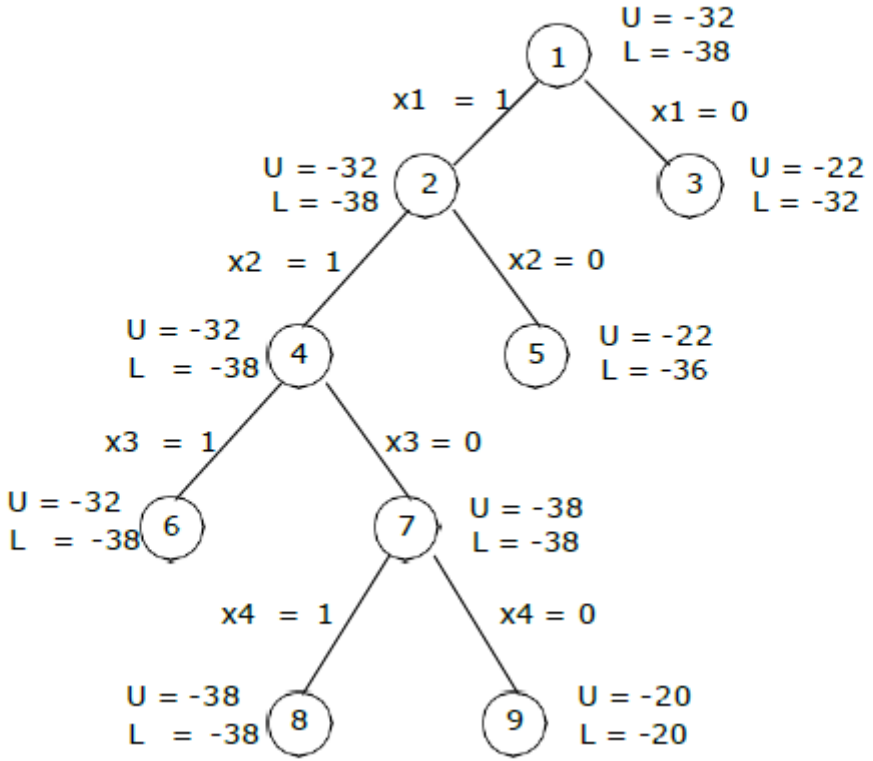
For node 4, U – L = -32 + 38 = 6
For node 5, U – L = -22 + 36 = 14



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 6, U – L = -32 + 38 = 6
For node 7, U – L = -38 + 38 = 0

Choose node 7, since it is minimum difference value of 0.

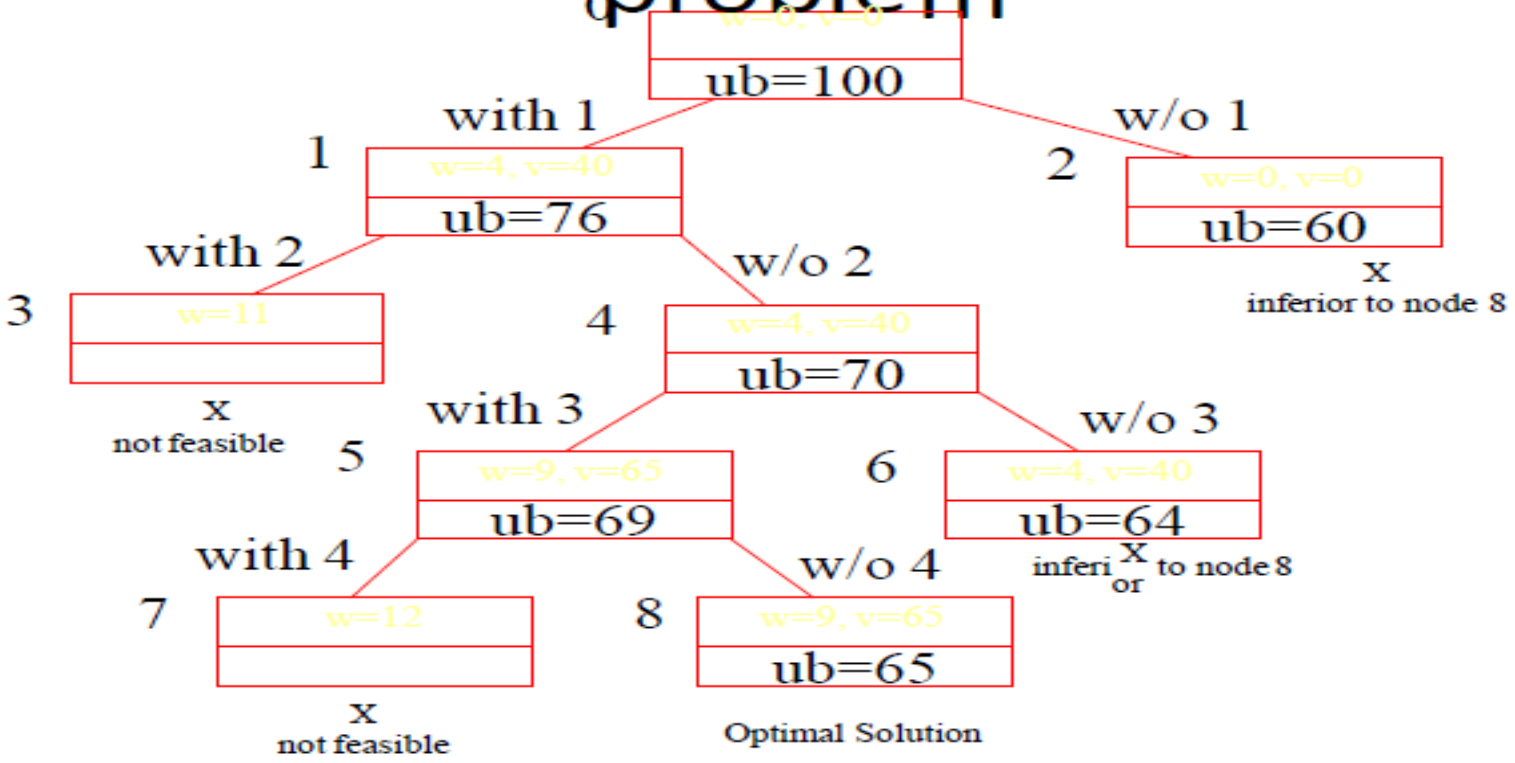


- Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.
- For node 8, U – L = -38 + 38 = 0
- For node 9, U – L = -20 + 20 = 0
- Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.
- Consider the path from 1 -> 2 -> 4 ->7 -> 8
- X1 = 1
- X2 = 1
- X3 = 0
- Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.
- For node 8, U – L = -38 + 38 = 0
- For node 9, U – L = -20 + 20 = 0
-

- Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.
- Consider the path from 1 -> 2 -> 4 ->7 -> 8
- X1 = 1
- X2 = 1
- X3 = 0
-
- X4 = 1
- The solution for 0/1 Knapsack problem is (x1, x2, x3, x4) = (1, 1, 0, 1) Maximum profit is:
- $\sum Pi\ xi$ = 10 x 1 + 10 x 1 + 12 x 0 + 18 x 1
- = 10 + 10 + 18 = 38.

- KNAPSACK PROBLEM
- •N items of known weights wi and values vi, i=1,2,….n
- •Knapsack capacity W =10 •
- Item Weight Value Value/Weight
- 1 4 $40 10
- 2 7 $42 6
- 3 5 $25 5
- 4 3 $12 4

State space tree of knapsack problem



Traveling Sale Person (TSP) using Backtracking

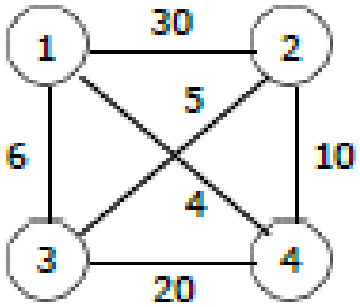
Traveling Sale Person (TSP) using Backtracking

- We have solved TSP problem using dynamic programming. In this section we shall solve the same problem using backtracking..

Traveling Salesman Problem

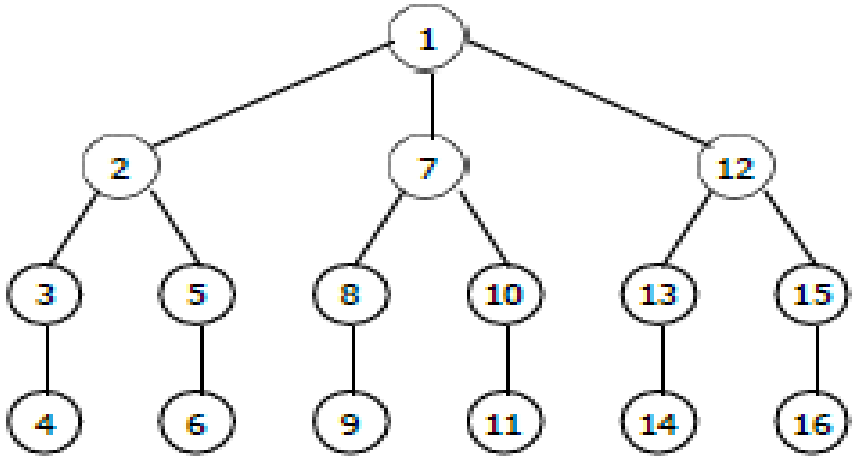
- For each city i, 1 <= i <=n, find the sum of the distances from city i to the two nearest cities.
- Compute the sum s of these n numbers
- Divide the result by 2
- If all the distances are integers, round up the result to the nearest integer • lb = [s/2]

Consider the graph shown below with 4 vertices.



A graph for TSP

The solution space tree, similar to the n-queens problem is as follows



- We will assume that the starting node is 1 and the ending node is obviously 1. Then 1, {2, … ,4}, 1 forms a tour with some cost which should be minimum. The vertices shown as {2, 3, …. ,4} forms a permutation of vertices which constitutes a tour. We can also start from any vertex, but the tour should end with the same vertex.
- Since, the starting vertex is 1, the tree has a root node R and the remaining nodes are numbered as depth-first order. As per the tree, from node 1, which is the live node, we generate 3 braches node 2, 7 and 12.
- We simply come down to the left most leaf node 4, which is a valid tour {1, 2, 3, 4, 1} with cost 30 + 5 + 20 + 4 = 59. Currently this is the best tour found so far and we backtrack to node 3 and to 2, because we do not have any children from node 3.

- When node 2 becomes the E- node, we generate node 5 and then node 6. This forms the tour {1, 2, 4, 3, 1} with cost 30 + 10 + 20 + 6 = 66 and is discarded, as the best tour so far is 59.
- Similarly, all the paths from node 1 to every leaf node in the tree is searched in a depth first manner and the best tour is saved. In our example, the tour costs are shown adjacent to each leaf nodes. The optimal tour cost is therefore 25.

Traveling Sale Person (TSP) using Backtracking

Traveling Sale Person

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Let G = (V, E) is a connected graph. Let C(i, J) be the cost of edge <i, j>. $c_{ij} = \infty$ if <i, j> $\notin$ E and let |V| = n, the number of vertices. Every tour starts at vertex 1 and ends at the same vertex. So, the solution space is given by S = {1, π, 1 | π is a

permutation of (2, 3, . . . , n)} and |S| = (n – 1)!. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots . i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n - 1$ and $i_0 = i_n = 1$.

Procedure for solving traveling sale person problem:

- 1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:

- a) *Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.
- b) Find the sum of elements, which were subtracted from rows.

- c) Apply column reductions for the matrix obtained after row reduction.

- *Column reduction:* Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.
- d) Find the sum of elements, which were subtracted from columns.

- e) Obtain the cumulative sum of row wise reduction and column wise reduction.

- Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.
- Associate the cumulative reduced sum to the starting state as lower bound and ∞ as upper bound.

2. Calculate the reduced cost matrix for every node R. Let A is the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge <i, j> in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:

a) Change all entries in row i and column j of A to ∞.

b) Set A (j, 1) to ∞.

c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∞. Let r is the total amount subtracted to reduce the matrix.

c) Find $\hat{c}(S) = \hat{c}(R) + A(i, j) + r$, where 'r' is the total amount subtracted to reduce the matrix, $\hat{c}(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $\hat{c}(S)$ is called the cost function.

- 3. Repeat step 2 until all nodes are visited.

FIFO Branch and Bound

FIFO Branch and Bound

- FIFO branch-and-bound algorithm for the job sequencing problem can begin with upper = α as an upper bound on the cost of a minimum-cost answer node.
-
- Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.4, nodes 2, 3, 4, and 5 are generated. Then u(2) = 19, u(3) = 14, u(4) =18, and u(5) = 21.

- The variable upper is updated to 14 when node 3 is generated. Since c (4) and c(5) are greater than upper, nodes 4 and 5 get killed. Only nodes 2 and 3 remain alive.

- Node 2 becomes the next E-node. Its children, nodes 6, 7 and 8 are generated.
- Then u(6) = 9 and so upper is updated to 9. The cost gets killed. Node 8 is infeasible and so it is killed.
- $c(7) = 10 >$ upper and node 7

- Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then u(9) =8 and so upper becomes 8.
- The cost $c(10) = 11 >$ upper, and this node is killed.
- The next E-node is node 6. Both its children are infeasible. Node 9‟s only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.
-

- When implementing a FIFO branch-and-bound algorithm, it is not economical to kill live nodes with $\hat{c}(x)$ > upper each time upper is updated. This is so because live nodes are in the queue in the order in which they were generated.
- Hence, nodes with $\hat{c}(x)$ with > upper are distributed in some random way in the queue. Instead, live nodes $\hat{c}(x)$ > upper can be killed when they are about to become E-nodes.

LC Branch and Bound

- An LC Branch-and-Bound search of the tree of Figure 8.4 will begin with upper = ∞ and node 1 as the first E-node.
- When node 1 is expanded, nodes 2, 3, 4 and 5 are generated in that order.
- As in the case of FIFOBB, upper is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $c(4) >$ upper and $c(5) >$ upper.
- Node 2 is the next E-node as $c(2) = 0$ and $c(3) = 5$. Nodes 6, 7 and 8 are generated and upper is updated to 9

- when node 6 is generated. So, node 7 is killed as $c(7) = 10 > upper$. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and 6.
- Node 6 is the next E-node as $c(6) = 0 < c(3)$. Both its children are infeasible.

- Node 3 becomes the next E-node. When node 9 is generated, upper is updated to 8 as u(9) = 8. So, node 10 with
- $c(10) = 11$ is killed on generation. Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answer node. 2 3
- **The path = 1 -> 3 -> 9 = 5 + 3 = 8**

UNIT 5
NP-Hard and NP-Complete Problems

Basic concepts

In Computer Science, many problems are solved where the objective is to maximize or minimize some values, where as in other problems we try to find whether there is a solution or not. Hence, the problems can be categorized as follows.

Basic concepts

Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,
Finding the minimum number of colors needed to color a given graph.Finding the shortest path between two vertices in a graph.

Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,Whether a given graph can be colored by only 4-colors.
Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

Basic concepts

P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time $O(n^k)$ in worst-case, where **k** is constant.

These problems are called **tractable**, while others are called **intractable or superpolynomial**.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial ***p(n)*** such that the algorithm can solve any instance of size **n** in a time ***O(p(n))***.

Basic concepts

NP-Class

The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren't asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in *NP* - *P* are intractable.

•

Non-deterministic algorithms

Non-deterministic algorithms

As we have seen, a state space is a useful abstraction in analyzing problems. The value of the abstraction is that it provides a particular kind of modularization: One can consider separately the space to be searched and the algorithm used to search it.

However, as algorithms become complex, the language of states and operators quickly becomes too inexpressive and awkward. What is needed is a representation that combines the abstraction of a state space with the expressivity of a procedural programming language. This is achieved in the notion of a *non-deterministic algorithm.*

Non-deterministic algorithms

The language of non-deterministic algorithms consists of six reserved words: **choose, pick, fail, succeed, either/or** . These are defined as follows:
choose X satisfying P(X). Consider alternatively all possible values of X that satisfy P(X), and proceed in the code. One can imagine the code as forking at this point, with a separate thread for each possible value of X. If any of the threads succeed, then the choice succeeds. If a choose operator in thread T generates subthreads T1 ... Tk, then T succeeds just if at least one of T1 ... Tk succeeds.
If thread T reaches the statement "**choose** X satisfying P(X)" and there is no X that satisfied P(X), then T fails.
pick X satisfying P(X). Find any value V that satisfies P(V) and assign X := V. This does not create a branching threads.
fail The current thread fails.
succeed The current thread succeeds and terminates.
either S1 **or** S2 **or** S3 ... **or** Sk. Analogous to choose. Create k threads T1 ... Tk where thread Ti executes statement Si and continues.

Non-deterministic algorithms

Some examples

General comment on pseudo-code: I use a combination of the notations I like in Pascal, C, and Ada, together with English. I hope it's clear enough. In particular, the labelling of procedure parameters as "in", "out", and "in/out" is taken from Ada. Following Pascal, assignment is notated := and equality is notated =. I declare variables only when I feel like it.

N Queens problem

Place N queens on an NxN board so that no two queens can take one another.

```
function attacked(in I,J,B) : returns boolean;{ attack = false;  for K := 1 to I-1 do
if ((B[K] = J) or  ((B[K]-J)=(K-I)) or ((B[K]-J)=(I-K))       then attack = true;
return(attack);}   N-QUEENS1(in N : integer; out B : array[1..N] of integer)  /*
B[I] = the row of the queen in the Ith column. -1 initially */   {  B := -1;   for I := 1
to N do {     choose J in 1..N such that not attacked(I,J,B)     B[I]=J;    }
```

Non-deterministic algorithms

} N-QUEENS1 above fills in the board left to right. Using "pick" we can generalize that to fill in columns in arbitrary order, to be chosen by the implementor.

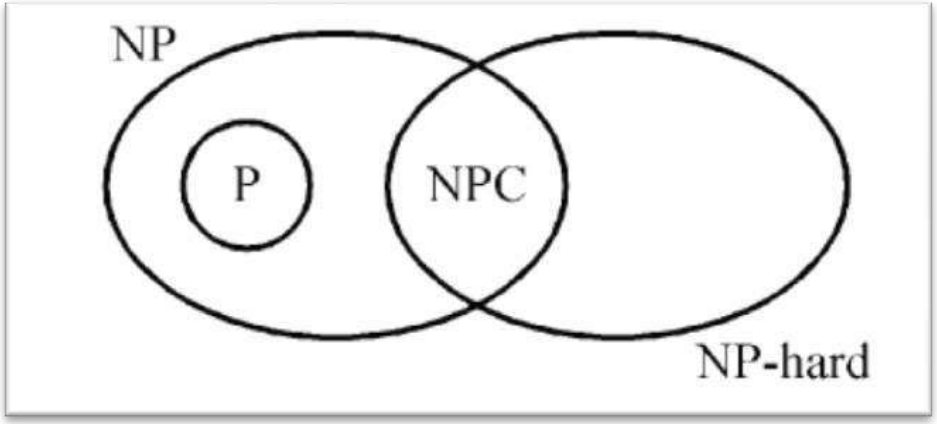
function attacked(in I,J,B) : returns boolean;{ attack = false; for K := 1 to N do if ((B[K] != -1) and ((B[K] = J) or ((B[K]-J)=(K-I)) or ((B[K]-J)=(I-K))) then attack = true; return(attack);} N-QUEENS2(in N : integer; out B : array[1..N] of integer) /* B[I] = the row of the queen in the Ith column. -1 initially */ { B := -1; for K := 1 to N do { pick I in 1..N such that B[I]=-1; choose J such that not attacked(I,J,B) B[I]=J; }}

The classes NP - Hard and NP complete

The classes NP - Hard and NP complete

If an NP-hard problem can be solved in polynomial time, then all NP-complete problems can be solved in polynomial time.
All NP-complete problems are NP-hard, but all NP- hard problems are not NP-complete.

The class of NP-hard problems is very rich in the sense that it contains many problems from a wide variety of disciplines.

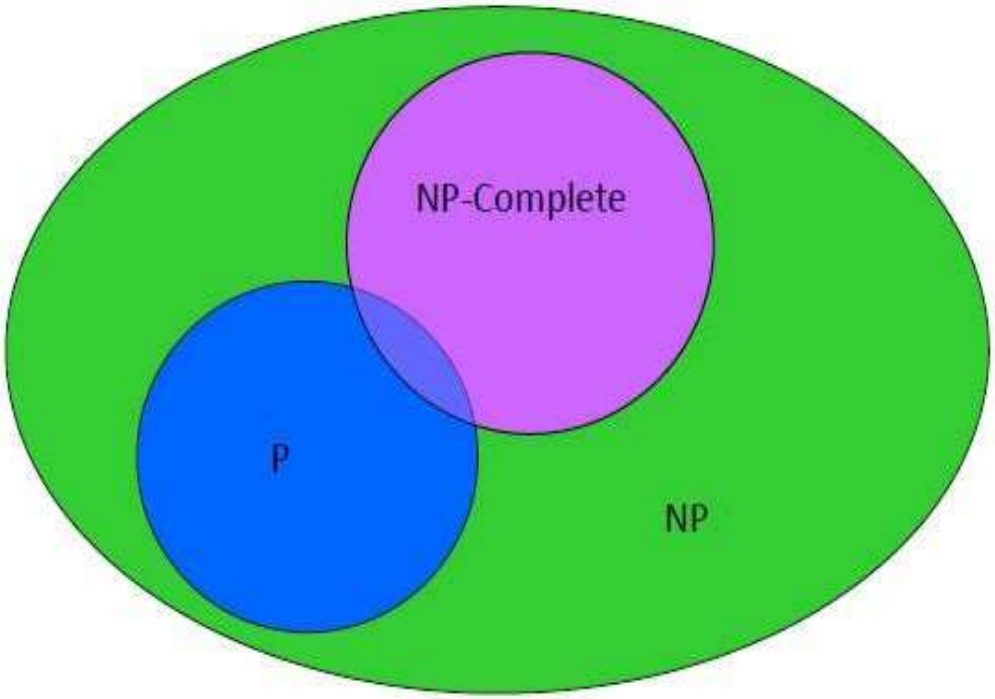


The classes NP - Hard and NP complete

P: The class of problems which can be solved by a deterministic polynomial algorithm.
NP: The class of decision problem which can be solved by a non-deterministic polynomial algorithm.

NP-hard: The class of problems to which every NP problem reduces
NP-complete (NPC): the class of problems which are NP-hard and belong to NP. NP-Competence
How we would you define NP-Complete
They are the “hardest” problems in NP

The classes NP - Hard and NP complete



The classes NP - Hard and NP complete

Nondeterministic algorithms:

A non deterministic algorithm consists of Phase 1: Guessing

Phase 2: Checking

The classes NP - Hard and NP complete

If the checking stage of a non deterministic algorithm is of polynomial time- complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm.

NP problems : (must be decision problems)

–e.g. searching, MST Sorting

Satisfy ability problem (SAT) travelling salesperson problem (TSP)

Example of a non deterministic algorithm

// The problem is to search for an element x //

// Output j such that A(j) =x; or j=0 if x is not in A // j choice (1 :n)

if A(j) =x then print(j) ; success endif print (‘0’) ; failure

complexity 0(1);

Non-deterministic decision algorithms generate a zero or one as their output.

Deterministic search algorithm complexity is n.

The classes NP - Hard and NP complete

The classes NP - Hard and NP complete

Satisfiability:

Letx1, x2, x3…. xn denotes Boolean variables.
Let xi denotes the relation of xi.
A literal is either a variable or its negation.
A formula in the prepositional calculus is an expression that can be constructed using literals and the operators and ^ or v.
A clause is a formula with at least one positive literal.
The satisfy ability problem is to determine if a formula is true for some assignment of truth values to the variables.
It is easy to obtain a polynomial time non determination algorithm that terminates s successfully if and only if a given prepositional formula E(x1, x2……xn) is satiable.
Such an algorithm could proceed by simply choosing (non deterministically) one of the 2n possible assignment so f truth values to (x1, x2…xn) and verify that E(x1,x2…xn) is true for that assignment.

The classes NP - Hard and NP complete

- **The satisfy ability problem:**

The logical formula:

x1v x2 v x3 & -x1 & -x2

the assignment : x1 ← F , x2 ← F , x3 ← T will make the above formula true . (-x1, -x2, x3) represents x1 ← F, x2 ← F, x3 ← T

If there is at least one assignment which satisfies a formula, then we say that this formula is satisfiable; otherwise, it is un satisfiable.

An un satisfiable formula:

x1vx2 &x1v-x2 &-x1vx2 &-x1v-x2

The classes NP - Hard and NP complete

Some NP-hard Graph Problems

The strategy to show that a problem L2 is NP-hard is

- Pick a problem L1 already known to be NP-hard.
- Show how to obtain an instance I1 of L2m from any instance I of L1 such that from the solution of I1 We can determine (in polynomial deterministic time) thesolutiontoinstanceIofL1
- Conclude from (ii) that L1L2.
- Conclude from (1),(2), and the transitivity of that Satisfiability L1 L1L2 Satisfiability L2 L2is NP-hard

The classes NP - Hard and NP complete

Examples:

CNF-Satisfy ability with at most three literals per clause is NP-hard. If each clause is restricted to have at most two literals then CNF-satisfy ability is polynomial solvable. Generating optimal code for a parallel assignment statement is NP-hard, However if the expressions are restricted to be simple variables, then optimal code can be generated in polynomial time.

Generating optimal code for level one directed a-cyclic graphs is NP-hard but optimal code for trees can be generated in polynomial time.

Determining if a planner graph is three color able is NP-Hard

NP Hard Problems

NP Hard Problems

Some NP-hard Graph Problems

ThestrategytoshowthataproblemL2isNP-hardis

- •Pick a problem L1 already known to be NP-hard.
- •Show how to obtain an instance I1 of L2m from any instance I of L1 such that from the solution of I1 We can determine (in polynomial deterministic time) thesolutiontoinstanceIofL1
- •Conclude from (ii) that L1L2.
- •Conclude from (1),(2), and the transitivity of that Satisfiability L1 L1L2

Satisfiability L2 L2is NP-hard

NP Hard Problems

Chromatic Number Decision Problem (CNP)

A coloring of a graph G = (V,E) is a function f : V □ { 1,2, …, k} i V

If (U, V) E then f(u) f(v).

The CNP is to determine if G has a coloring for a given K.

Satisfiability with at most three literals per clause chromatic number problem. CNP is NP-hard.

NP Hard Problems

Directed Hamiltonian Cycle(DHC)

Let G=(V,E) be a directed graph and length n=1V1

TheDHCisacyclethatgoesthrougheveryvertexexactlyonceandthenreturns tothestartingv ertex.

The DHC problem is to determine if G has a directed Hamiltonian Cycle.

Theorem: CNF (Conjunctive Normal Form) satisfiability DHC DHC is NP-hard.

NP Hard Problems

Travelling Salesperson Decision Problem (TSP) :

The problem is to determine if a complete directed graph G = (V,E) with edge costs C(u,v) has a tour of cost at most M.

Theorem: Directed Hamiltonian Cycle (DHC) TSP

But from problem (2) satisfiability DHC Satisfiability TSP TSP is NP-hard.

NP Hard Problems

NP Hard Problems

NP-Hard Problems:

'Efficient' Problems

A long time ago1, theoretical computer scientists like Steve Cook and Dick Karp decided that a minimum requirement of any efficient algorithm is that it runs in polynomial time: $O(n^c)$ for some constant c. People recognized early on that not all problems can be solved this quickly, but we had a hard time figuring out exactly which ones could and which ones couldn't. So Cook, Karp, and others, defined the class of *NP-hard* problems, which most people believe *cannot* be solved in polynomial time, even though nobody can prove a super-polynomial lower bound.

NP Hard Problems

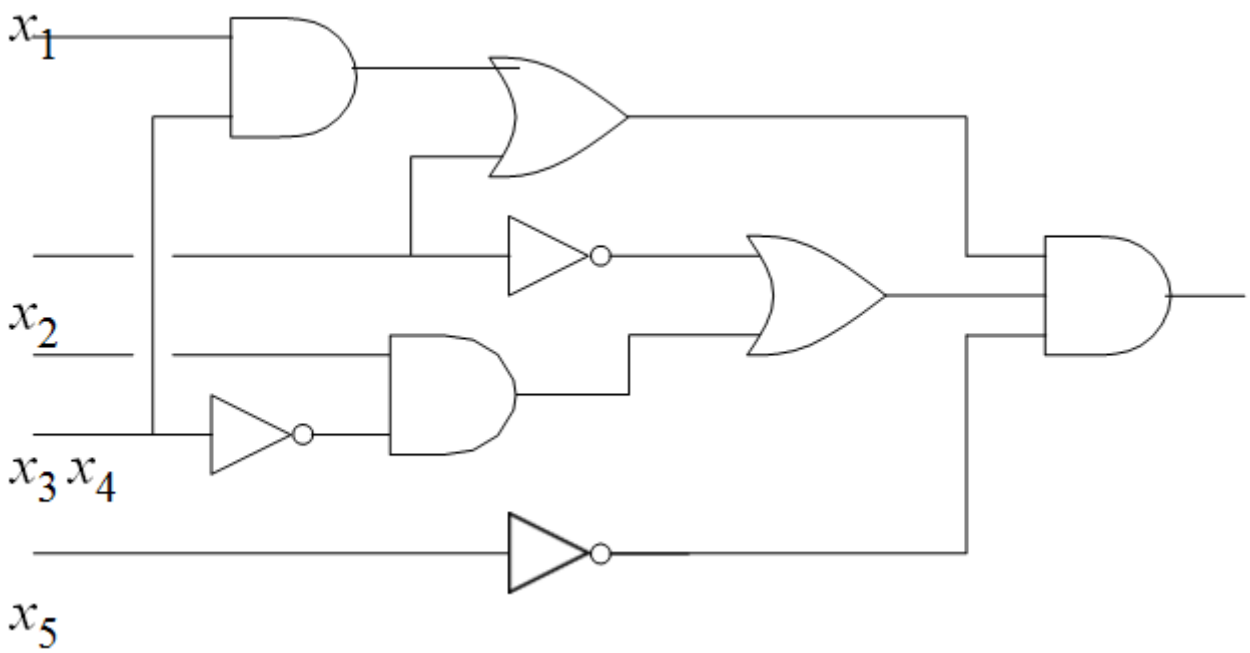
Circuit satisfiability is a good example of a problem that we don't know how to solve in poly- nomial time. In this problem, the input is a *boolean circuit* : a collection of and, or, and not gates connected by wires. We will assume that there are no loops in the circuit (so no delay lines or flip-flops). The input to the *circuit* is a set of *m* boolean (true/false) values $x_1, \ldots, x_m$. The output is a single boolean value. Given specific input values, we can calculate the output in polynomial (actually, *linear*) time using depth-first-search and evaluating the output of each gate in constant time.

NP Hard Problems

The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output True, or conversely, whether the circuit *always* outputs False. Nobody knows how to solve this problem faster than just trying all 2*m* possible inputs to the circuit, but this requires exponential time. On the other hand, nobody has ever proved that this is the best we can do; maybe there's a clever algorithm that nobody has discovered yet!

NP Hard Problems

An and gate, an or gate, and a not gate.



A boolean circuit. Inputs enter from the left, and the output leaves to the right.

Clique decision problem

Clique Decision Problem

The Clique Problem:

Cliques:

Suppose that *G* is an undirected graph. Say that a set *S* of vertices of *G* form a *clique* if each vertex in *S* is adjacent to each other vertex in *S*.

Clique Decision Problem

The Clique Problem

The clique problem is as follows.

Input. An undirected graph *G* and a positive integer *K*.

Question. Does *G* have a clique of size at least *K*?

Let's look at the same example graph that was used earlier for the Vertex Cover and Independent Set problems, where G_1 has vertices

{1, 2, 3, 4, 5, 6, 7, 8, 9}

and its edges are

{1, 2},{1, 4},{1, 6},{1, 8},{2, 3},{3, 4},{4, 5},
{5, 6},{6, 7},{7, 8},{8, 9},{2, 9}.

Clique Decision Problem

We saw that {2, 4, 6, 8} is an independent set in G_1, which means that {2, 4, 6, 8} is a clique in G_1. But what about a clique in G_1?
A largest clique in G_1 is {1,2}, having just two vertices.
But look at graph G_2 with vertices {1, 2, 3, 4, 5, 6, 7, 8} and edges
{1, 2},{1, 6},
{1, 7},{1, 8},
{2, 3},{2, 5},
{2, 6},{3, 5},
{3, 6},{4, 5},
{4, 6},{5, 6},
{7, 8}.
Does G_2 have a clique of size 3? Yes: {1, 7, 8}. But what about a clique of size 4?

Clique Decision Problem

- **An idea for finding large cliques**

Here is an idea for finding a large clique.
The *degree* of a vertex *v* is the number of edges that are connected to *v*.
To find a clique of *G*:
Suppose that *G* has *n* vertices.
Find a vertex *v* of the smallest possible degree in *G*.
If the degree of *v* is *n* − 1, stop; *G* is a clique, so the largest clique in *G* has size *n*.
Otherwise, remove *v* and all of its edges from *G*. Find the largest clique in the smaller graph. Report that as the largest clique in *G*.

Clique Decision Problem

- **How to test a program**

  It consists of two phases.

  1. Debugging is detection and correction of errors.
  2. Profiling or performance measurement is the actual amount of time required by the program to compute the result.

Chromatic number decision problem

Chromatic number decision problem

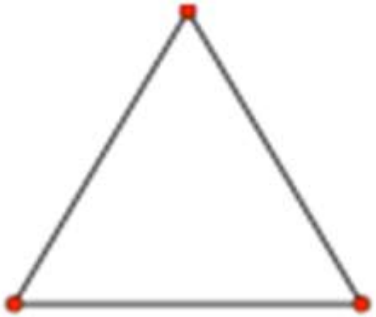
The chromatic number decision problem is defined as follows:

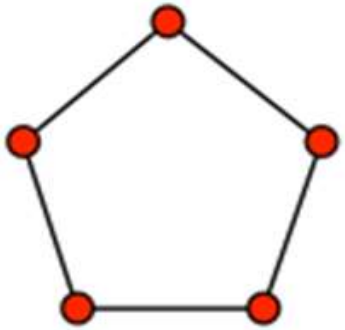
We are given a graph G = (V, E). For each vertex, relate it with a color in such a way that if two vertices are connected by an edge, then these two vertices must be related with different colors. The chromatic number decision problem is to find the least possible number of colors to color the vertices of the given graph.

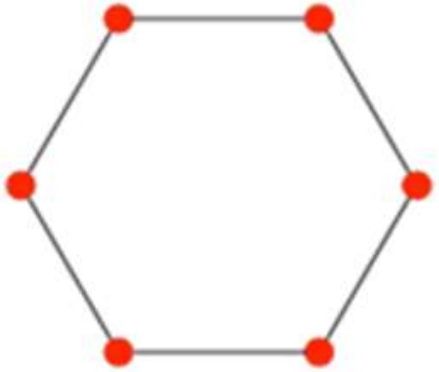
Chromatic number decision problem

- **Types of Graphs with their respective chromatic numbers:**

Cycle Graph: Examples







Cycle Graph Chromatic Number: 2 colors if number of vertices is even

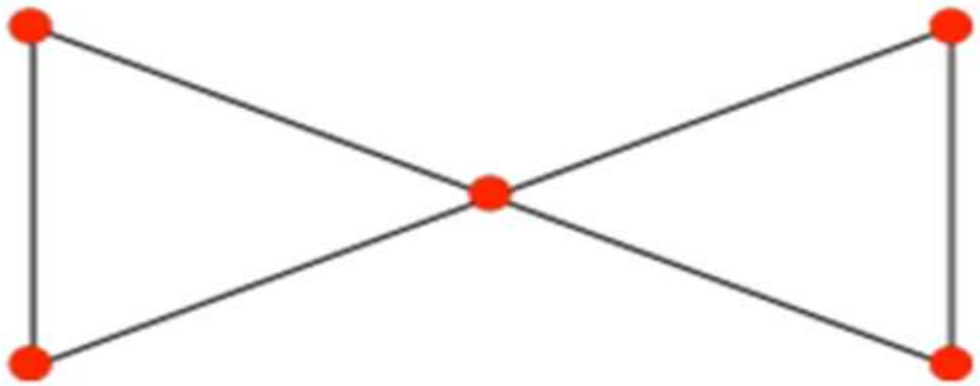
3 colors if number of vertices is odd

Chromatic number decision problem

- **Planar Graph:**

In planar graphs, the edges do not cross (except at a vertex). Planar graphs are widely used to represent maps.

Examples:



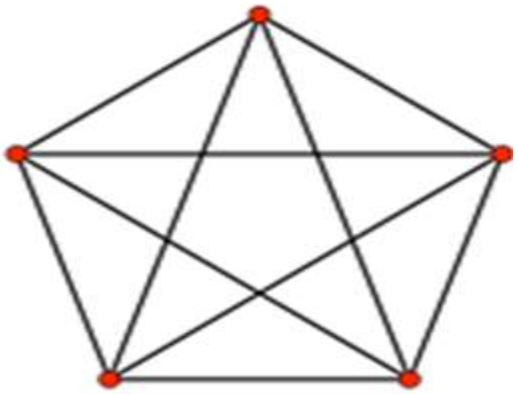
- **Planar Graph Chromatic Number:** Less than or equal to 4

Chromatic number decision Problem

- **Complete Graph:**

In a complete graph, each vertex is connected to every other vertex by an edge.

Examples:



- **Complete Graph Chromatic Number:** Equals to the number of vertices of the given complete graph.

Cook's theorem

Cook's theorem

Stephen Cook presented four theorems in his paper “The Complexity of Theorem Proving Procedures”. These theorems are stated below. We do understand that many unknown terms are being used in this chapter, but we don’t have any scope to discuss everything in detail.

Following are the four theorems by Stephen Cook –

Theorem-1

If a set **S** of strings is accepted by some non-deterministic Turing machine within polynomial time, then **S** is P-reducible to {DNF tautologies}.

Cook's theorem

Theorem-2

The following sets are P-reducible to each other in pairs (and hence each has the same polynomial degree of difficulty): {tautologies}, {DNF tautologies}, D3, {sub-graph pairs}.

Theorem-3

For any $T_Q(k)$ of type **Q**, TQ(k)kV(logk)2TQ(k)k(logk)2 is unbounded

There is a $T_Q(k)$ of type **Q** such that TQ(k)⩽2k(logk)2TQ(k)⩽2k(logk)2

Cook's theorem

Theorem-4

If the set S of strings is accepted by a non-deterministic machine within time $T(n) = 2^n$, and if $T_Q(k)$ is an honest (i.e. real-time countable) function of type **Q**, then there is a constant **K**, so **S** can be recognized by a deterministic machine within time $T_Q(K8^n)$.

Cook's theorem

Cook's theorem

Cook's Theorem

Cook's Theorem states that *Any NP problem can be converted to SAT in polynomial time.*

In order to prove this, we require a uniform way of representing NP problems. Remember that what makes a problem NP is the existence of a polynomial-time algorithm—more specifically, a Turing machine—for checking candidate certificates. What Cook did was somewhat analogous to what Turing did when he showed that the *Entscheidungs problem* was equivalent to the Halting Problem. He showed how to encode as Propositional Calculus clauses both the relevant facts about the problem instance and the Turing machine which does the certificate-checking, in such a way that the resulting set of clauses is satisfiable if and only if the original problem instance is positive. Thus the problem of determining the latter is reduced to the problem of determining the former.

Cook's theorem

Let us assume that M has q states numbered 0, 1, 2, . . . , q 1, and a tape alphabet a_1, a_2, . . . , a_s. We shall assume that the operation of the machine is governed by the functions T , U , and D as described in the chapter on the *Entscheidungsproblem*. We shall further assume that the initial tape is inscribed with the problem instance on the squares 1, 2, 3, . . . , n, and the putative certificate on the squares m, . . . , 2, 1. Square zero can be assumed to contain a designated separator symbol. We shall also assume that the machine halts scanning square 0, and that the symbol in this square at that stage will be a_1 if and only if the candidate certificate is a true certificate. Note that we must have m P (n). This is because with a problem instance of length n the computation is completed in at most P (n) steps; during this process, the Turing machine head cannot move more than P (n) steps to the left of its starting point.

Cook's theorem

We define some atomic propositions with their intended interpretations as follows:

–

For $i = 0, 1, \ldots, P(n)$ and $j = 0, 1, \ldots, q\ 1$, the proposition Q_{ij} says that after i computation steps, M is in state j.

For $i = 0, 1, \ldots, P(n)$, $j = -P(n), \ldots, P(n)$, and $k = 1, 2, \ldots, s$, the proposition S_{ijk} says that after i computation steps, square j of the tape contains the symbol a_k.

–

$i = 0, 1, \ldots, P(n)$ and $j =$ $P(n), \ldots, P(n)$, the proposition T_{ij} says that after i computation steps, the machine M is scanning square j of the tape.