**UNIT – I (PART-A)**

# 1.1 Algorithm

**Informal Definition:**

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the i/p into the o/p.

**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.

**Issues or study of Algorithm:**

1. How to device or design an algorithm 🡪 creating and algorithm.
2. How to express an algorithm 🡪 definiteness.
3. How to analysis an algorithm 🡪 time and space complexity.
4. How to validate an algorithm 🡪 fitness.
5. Testing the algorithm 🡪 checking for error.

# 

# 1.2Algorithm Specification:

Algorithm can be described in three ways.

1. Natural language like English:

When this way is chooses care should be taken, we should ensure that each & every statement is definite.

2. Graphic representation called flowchart:

This method will work well when the algorithm is small& simple.

1. Pseudo-code Method:

In this method, we should typically describe algorithms as program, which resembles language like Pascal & algol.

**Pseudo-Code Conventions:**

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.

1. Assignment of values to variables is done using the assignment statement.

<Variable>:= <expression>;

1. There are two Boolean values TRUE and FALSE.

🡪 Logical Operators AND, OR, NOT

🡪Relational Operators <, <=,>,>=, =, !=

1. The following looping statements are employed.

For, while and repeat-until

While Loop:

While < condition > do

{

<statement-1>

**.**

**.**

**.**

<statement-n>

}

**For Loop:**

For variable: = value-1 to value-2 step step do

{

<statement-1>

**.**

**.**

**.**

<statement-n>

}

**repeat-until:**

repeat

<statement-1>

**.**

**.**

**.**

<statement-n>

until<condition>

1. A conditional statement has the following forms.

🡪 If <condition> then <statement>

🡪 If <condition> then <statement-1>

Else <statement-1>

**Case statement:**

Case

{

**:** <condition-1> **:** <statement-1>

**.**

**.**

**.**

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

**:** else **:** <statement-n+1>

}

1. Input and output are done using the instructions read & write.
2. There is only one type of procedure:

Algorithm, the heading takes the form,

Algorithm Name (Parameter lists)

🡪 As an example, the following algorithm fields & returns the maximum of ‘n’ given numbers:

1. algorithm Max(A,n)
2. // A is an array of size n
3. {
4. Result := A[1];
5. for I:= 2 to n do
6. if A[I] > Result then
7. Result :=A[I];
8. return Result;
9. }

In this algorithm (named Max), A & n are procedure parameters. Result & I are Local variables.

🡪 Next we present 2 examples to illustrate the process of translation problem into an algorithm.

# 1.3) Performance Analysis:

**Space Complexity:**

The space complexity of an algorithm is the amount of money it needs to run to compilation.

**Time Complexity:**

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

## 

## 1.3.1)Space Complexity:

Space Complexity Example:

Algorithm abc(a,b,c)

{

return a+b++\*c+(a+b-c)/(a+b) +4.0;

}

🡪 The Space needed by each of these algorithms is seen to be the sum of the following component.

1.A fixed part that is independent of the characteristics (eg:number,size)of the inputs and outputs.

The part typically includes the instruction space (ie. Space for the code), space for simple variable and fixed-size component variables (also called aggregate) space for constants, and so on.

* A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that is depends on instance characteristics), and the recursion stack space.
  + The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics)

Where ‘c’ is a constant.

**Example 2:**

Algorithm sum(a,n)

{

s=0.0;

for I=1 to n do

s= s+a[I];

return s;

}

1. The problem instances for this algorithm are characterized by n,the number of elements to be summed. The space needed d by ‘n’ is one word, since it is of type integer.
2. The space needed by ‘a’a is the space needed by variables of tyepe array of floating point numbers.
3. This is atleast ‘n’ words, since ‘a’ must be large enough to hold the ‘n’ elements to be summed.
4. So,we obtain Ssum(n)>=(n+s)

[ n for a[],one each for n,I a& s]

## 1.3.2 Time Complexity:

The time T(p) taken by a program P is the sum of the compile time and the run time(execution time)

🡪The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation .This rum time is denoted by tp(instance characteristics).

🡪 The number of steps any problem statemn t is assigned depends on the kind of statement.

For example, comments 🡪 0 steps.

Assignment statements 🡪 1 steps.

[Which does not involve any calls to other algorithms]

Interactive statement such as for, while & repeat-until🡪 Control part of the statement.

* + - * We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.

This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

**Algorithm:**

Algorithm sum(a,n)

{

s= 0.0;

count = count+1;

for I=1 to n do

{

count =count+1;

s=s+a[I];

count=count+1;

}

count=count+1;

count=count+1;

return s;

}

🡪 If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

🡪First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

🡪By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

|  |  |  |  |
| --- | --- | --- | --- |
| Statement | S/e | Frequency | Total |
| 1. Algorithm Sum(a,n)  2.{  3. S=0.0;  4. for I=1 to n do  5. s=s+a[I];  6. return s;  7. } | 0  0  1  1  1  1  0 | -  -  1  n+1  n  1  - | 0  0  1  n+1  n  1  0 |
| Total |  |  | 2n+3 |

**Average –Case Analysis**

* Most of the time, average-case analysis are performed under the more or less realistic assumption that all instances of any given size are equally likely.
* For sorting problems, it is simple to assume also that all the elements to be sorted are distinct.
* Suppose we have ‘n’ distinct elements to sort by insertion and all n! permutation of these elements are equally likely.
* To determine the time taken on a average by the algorithm ,we could add the times required to sort each of the possible permutations ,and then divide by n! the answer thus obtained.
* An alternative approach, easier in this case is to analyze directly the time required by the algorithm, reasoning probabilistically as we proceed.
* For any I,2 In, consider the sub array, T[1….i].
* The partial rank of T[I] is defined as the position it would occupy if the sub array were sorted.
* For Example, the partial rank of T[4] in [3,6,2,5,1,7,4] in 3 because T[1….4] once sorted is [2,3,5,6].
* Clearly the partial rank of T[I] does not depend on the order of the element in
* Sub array T[1…I-1].

**Analysis**

**Best case**:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

**Worst case**:

This analysis constrains on the input, other than size. Resulting in the fasters possible run time

**Average case:**

This type of analysis results in average running time over every type of input.

**Complexity:**

Complexity refers to the rate at which the storage time grows as a function of the problem size

**Asymptotic analysis:**

Expressing the complexity in term of its relationship to know function. This type analysis is called asymptotic analysis.

**Asymptotic notation:**

**Big ‘oh’:** the function f(n)=O(g(n)) iff there exist positive constants c and no such that f(n)≤c\*g(n) for all n, n ≥ no.

**Omega:** the function f(n)=Ω(g(n)) iff there exist positive constants c and no such that f(n) ≥ c\*g(n) for all n, n ≥ no.

**Theta:** the function f(n)=ө(g(n)) iff there exist positive constants c1,c2 and no such that c1 g(n) ≤ f(n) ≤ c2 g(n) for all n, n ≥ no.

**Recursion:**

Recursion may have the following definitions:

-The nested repetition of identical algorithm is recursion.

-It is a technique of defining an object/process by itself.

-Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied.

**When to use recursion:**

Recursion can be used for repetitive computations in which each action is stated in terms of previous result. There are two conditions that must be satisfied by any recursive procedure.

* 1. Each time a function calls itself it should get nearer to the solution.
  2. There must be a decision criterion for stopping the process.

In making the decision about whether to write an algorithm in recursive or non-recursive form, it is always advisable to consider a tree structure for the problem. If the structure is simple then use non-recursive form. If the tree appears quite bushy, with little duplication of tasks, then recursion is suitable.

The recursion algorithm for finding the factorial of a number is given below,

**Algorithm** : factorial-recursion

**Input** : n, the number whose factorial is to be found.

**Output :** f, the factorial of n

**Method** : if(n=0)

f=1

else

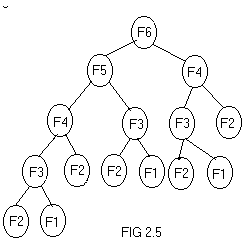
f=factorial(n-1) \* n

if end

algorithm ends.

The general procedure for any recursive algorithm is as follows,

1. Save the parameters, local variables and return addresses.
2. If the termination criterion is reached perform final computation and goto step 3 otherwise perform final computations and goto step 1



1. Restore the most recently saved parameters, local variable and return address and goto the latest return address.

**Iteration v/s Recursion:**

**Demerits of recursive algorithms**:

1. Many programming languages do not support recursion; hence, recursive mathematical function is implemented using iterative methods.
2. Even though mathematical functions can be easily implemented using recursion it is always at the cost of execution time and memory space. For example, the recursion tree for generating 6 numbers in a Fibonacci series generation is given in fig 2.5. A Fibonacci series is of the form 0,1,1,2,3,5,8,13,…etc, where the third number is the sum of preceding two numbers and so on. It can be noticed from the fig 2.5 that, f(n-2) is computed twice, f(n-3) is computed thrice, f(n-4) is computed 5 times.
3. A recursive procedure can be called from within or outside itself and to ensure its proper functioning it has to save in some order the return addresses so that, a return to the proper location will result when the return to a calling statement is made.
4. The recursive programs needs considerably more storage and will take more time.

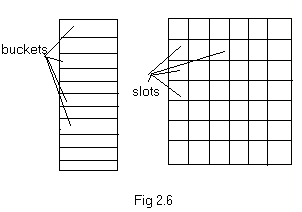
**Demerits of iterative methods :**

1. Mathematical functions such as factorial and Fibonacci series generation can be easily implemented using recursion than iteration.
2. In iterative techniques looping of statement is very much necessary.

Recursion is a top down approach to problem solving. It divides the problem into pieces or selects out one key step, postponing the rest.

Iteration is more of a bottom up approach. It begins with what is known and from this constructs the solution step by step. The iterative function obviously uses time that is O(n) where as recursive function has an exponential time complexity.

It is always true that recursion can be replaced by iteration and stacks. It is also true that stack can be replaced by a recursive program with no stack.



**SOLVING RECURRENCES :-**( Happen again (or) repeatedly)

1. The indispensable last step when analyzing an algorithm is often to solve a recurrence equation.
2. With a little experience and intention, most recurrence can be solved by intelligent guesswork.
3. However, there exists a powerful technique that can be used to solve certain classes of recurrence almost automatically.
4. This is a main topic of this section the technique of the characteristic equation.

* **Intelligent guess work:**

This approach generally proceeds in 4 stages.

* Calculate the first few values of the recurrence
* Look for regularity.
* Guess a suitable general form.
* And finally prove by mathematical induction(perhaps constructive induction).

Then this form is correct.

Consider the following recurrence,

0 if n=0

T(n) = 3T(n ÷ 2)+n otherwise

1. First step is to replace n ÷ 2 by n/2
2. It is tempting to restrict ‘n’ to being ever since in that case n÷2 = n/2, but recursively dividing an even no. by 2, may produce an odd no. larger than 1.
3. Therefore, it is a better idea to restrict ‘n’ to being an exact power of 2.
4. First, we tabulate the value of the recurrence on the first few powers of 2.

**n** 1 2 4 8 16 32

**T(n)** 1 5 19 65 211 665

\* For instance, T(16) = 3 \* T(8) +16

= 3 \* 65 +16

= 211.

\* Instead of writing T(2) = 5, it is more

useful to write T(2) = 3 \* 1 +2.

Then,

T(A) = 3 \* T(2) +4

= 3 \* (3 \* 1 +2) +4

= (32 \* 1) + (3 \* 2) +4

\* We continue in this way, writing ‘n’ as an explicit power of 2.

**n T(n)**

* + 1
  + 3 \* 1 +2

22 32 \* 1 + 3 \* 2 + 22

23 33 \* 1 + 32 \* 2 + 3 \* 22 + 23

24 34 \* 1 + 33 \* 2 + 32 \* 22 + 3 \* 23 + 24

25 35 \* 1 + 34 \* 2 + 33 \* 22 + 32 \* 23 + 3 \* 24 + 25

1. The pattern is now obvious.

T(2k ) = 3k20 + 3k-121 + 3k-222+…+312k-1 + 302k.

= ∑ 3k-i 2i

= 3k ∑ (2/3)i

= 3k \* [(1 – (2/3)k + 1) / (1 – (2/3)]

= 3k+1 – 2k+1

**Proposition: (Geometric Series)**

Let Sn be the sum of the first n terms of the geometric series a, ar, ar2….Then

Sn = a(1-rn)/(1-r), except in the special case when r = 1; when Sn = an.

= 3k \* [ (1 – (2/3) k+1) / (1 – (2/3))]

= 3k \* [((3 k+1 – 2 k+1)/ 3 k+1) / ((3 – 2) / 3)]

3 k+1 – 2k+1 3

= 3k \* ----------------- \* ----

3 k+1 1

3 k+1 – 2k+1

= 3k \* -----------------

3k+1-1

= 3k+1 – 2k+1

\* It is easy to check this formula against our earlier tabulation

# Eg : 2

0 n=0

tn = 5 n=1

3tn-1 + 4tn-2, otherwise

tn = 3tn-1 – 4tn-2 = 0 🡪 General function

Characteristics Polynomial, x2 – 3x – 4 = 0

(x – 4)(x + 1) = 0

Roots r1 = 4, r2 = -1

General Solution, fn = C1r1n + C2 r2n 🡪 (A)

n=0 🡺 C1 + C2 = 0 🡪 (1)

n=1 🡺 C1r1 + C2r2 = 5 🡪 (2)

Eqn 1 🡺 C1 = -C2

sub C1 value in Eqn (2)

-C2r1 + C2r2 = 5

C2(r2 – r1) = 5

5

C2 = -------

r2 – r1

5

= ------

-1 + 4

= 5 / (-5) = -1

C2 = -1 , C1 = 1

Sub C1, C2, r1 & r2 value in equation 🡪 (A)

fn = 1. 4n + (-1) . (-1)n

**fn = 4n + 1n**

**PART-II**

# 1.4 DIVIDE-AND-CONQUER

# 1.4 GENERAL METHOD:

Given a function to compute on n inputs the divide-and-conquer strategy suggests,

**Divide :** Splitting the inputs into k distinct subsets, 1< k ≤n , yielding k

sub problems.

**Conquer :** These Sub-problems must be solved, and then

**Combine :** a method must be found to combine sub-solutions into a

solution of the whole.

***Note : If the sub-problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.***

We can write a control abstraction that mirrors the way an algorithm based on divide-and-conquer will look.

**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 input size is small enough that the answer can be computed without splitting.
* If this is so, the function S is invoked. Otherwise the problem P is divided into smaller sub-problems.
* These sub-problems P1,P2,………,Pk are solved by recursive applications of DAndC.
* 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.

**Algorithm: control abstraction for divide-and conquer**

1 Algorithm DAndC(P)

2 {

3 If small(P) then return S(P);

4 else

5 {

6 divide P into smaller instances P1,P2,………,Pk, k>=1;

7 Apply DAndC to each of these subproblems;

8 return Combine(DAndC(P1),DAndC(P2),…… DAndC(Pk));

9 }

10 }

The complexity of many divide-and-conquer algorithms is given by recurrences of the form(**General method**)

{

T(n)= T(1) n=1

aT(n/b)+f(n) n>1

**n** : *size of original problem*.

**T(i)** : *time to solve problem of size* ***i****.*

**a** : *number of subproblems*.

**b** : *size of each subproblem.*

**f(n)** : *time to divide and combine subproblems*.

where ***a*** and ***b*** are known constants. We assume that ***T(1)*** is known and ***n*** is a power of ***b***

***(i.e., n=bk)***.

Example:

Consider the case in which a = 2 and b = 2. Let T(1) = 2 and f(n) = n.

We have

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

= 2[2T(n/4) + n/2] + n

= 4T(n/4) + 2n

= 4[2T(n/8) + n/4] + 2n

= 8T(n/8) + 3n

Is in the form 23T(n/23)+3n

= 2kT(n/2k)+kn

As we know that ***n*** is a power of ***b*** that is***,***

***n=bk***. [:.where b = 2;]

apply **log** on both sides, then,

***n=bk***. [:.where b = 2;]

=> log n = log2k

=> k = log2 n. [substitute the k value in the above k-th

term.]

T(n) = **2log2nT(n/ 2log2n) + nlog2n**, corresponding to the choice of k= log2n.

= nT(n/n)+n log2n

= nT(1)+ n log2n

= 2n+ n log2n

*\* as in the rule of* ***Big-Oh****, if f(n) is a polynomial of degree k, then f(n) is O(nk), ie.,* ***drop lower-order terms and constant factors.***

= ~~2~~ ~~n~~ + n log2n

= O(n logn).

Analysis of QuickSort

**Best Case**: Each (recursive) Partition call will split the array in halves. We would get the familiar recurrence for the running time:

T(n) = 2T(n/2) + O(n) = O(n log n):

**Worst Case**: Partition always causes an unbalanced split (all elements going to one of the subarrays). The recurrence we get in this case is

T(n) = T(n -1) + O(n);

which solves to T(n) = O(n2).

# 1.5 BINARY SEARCH:

A binary search algorithm is a technique for finding a particular value in a sorted list. The binary search consists of the following steps:

* Let ***ai*, 1 < i < n** be a list of elements that are sorted in non decreasing order.
* Search a sorted array by repeatedly dividing the search interval in half.
* Begin with an interval covering the whole array.
* If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half.
* Repeatedly check until the value is found or the interval is empty.

The most straight farward implementation is recursion.

**Algorithm :**

**Algorithm BSearch(a,l,u,x)**

**{**

**if(l = u) then**

**{**

if(x = a[u]) then return u;

else return 0;

**}**

**else**

**{ // reduce P into a smaller subproblems.**

mid := (l+u)/2;

if(x < a[mid]) then

return BSearch(a,l,mid-1,x);

else if (x>a[mid]) then

return BSearch(a,mid+1,u,x);

else return mid;

**}**

**}**

**Example :**

Given a list of elements **1, 4, 5, 9, 13, 35, 37 41.** now we need to search for an element **x = 35** in the given list. To do that,

**1**

**4**

**5**

**9**

**13**

**35**

**37**

**41**

1

2

3

4

5

6

7

8

L

mid

U

Find the middle element by the step

**mid = (l+u)/2 = (1+8)/2 = 4.** now the value with index ***4*** is the middle element.

x > a[mid] [:. ( 35 > 9 ) ]

**1**

**4**

**5**

**9**

**13**

**35**

**37**

**41**

1

2

3

4

5

6

7

8

***L***

***mid***

***U***

l = mid + 1;

Check the relation between the search element x = 35 and the middle element which we calculated above. The search element is greater than the middle element. So, the search element may exist in the right sub-list to the middle element.

Now the *lower bound* of the sub-list is **5** which will be pointed with ***mid+1*** and the *upper bound* is ***u***.

Then solve the right sub-list as in the above recursively.

In this sub-list the middle element is

**mid = (l + u) / 2 = (5 + 8) / 2 = 6.**

X = a[mid] [:. (35 = 35)]

**1**

**4**

**5**

**9**

**13**

**35**

**37**

**41**

1

2

3

4

5

6

7

8

***L***

***mid***

***U***

The element located in the mid position is 35. now check the search element x=35 with the middle element. In this case both are equal, means the element found at the position mid, and return the position ***6***. So, the element found at index position 6 in the given list.

# 

# 1.6 Maximum and Minimum:

* Let us consider another simple problem that can be solved by the divide-and-conquer technique.
* The problem is to find the maximum and minimum items in a set of ‘n’ elements.
* In analyzing the time complexity of this algorithm, we once again concentrate on the no. of element comparisons.
* More importantly, when the elements in a[1:n] are polynomials, vectors, very large numbers, or strings of character, the cost of an element comparison is much higher than the cost of the other operations.
* Hence, the time is determined mainly by the total cost of the element comparison.
* Algorithm straight MaxMin(a,n,max,min)
* // set max to the maximum & min to the minimum of a[1:n]
* {
* max:=min:=a[1];
* for I:=2 to n do
* {
* if(a[I]>max) then max:=a[I];
* if(a[I]<min) then min:=a[I];
* }
* }

**Algorithm:** Straight forward Maximum & Minimum

* Straight MaxMin requires 2(n-1) element comparison in the best, average & worst cases.
* An immediate improvement is possible by realizing that the comparison a[I]<min is necessary only when a[I]>max is false.
* Hence we can replace the contents of the for loop by,

If(a[I]>max) then max:=a[I];

Else if (a[I]<min) then min:=a[I];

* Now the best case occurs when the elements are in increasing order.

🡪 The no. of element comparison is (n-1).

* The worst case occurs when the elements are in decreasing order.

🡪 The no. of elements comparison is 2(n-1)

* The average no. of element comparison is < than 2(n-1)
* On the average a[I] is > than max half the time, and so, the avg. no. of comparison is 3n/2-1.
* A divide- and conquer algorithm for this problem would proceed as follows:

🡪 Let P=(n, a[I] ,……,a[j]) denote an arbitrary instance of the problem.

🡪 Here ‘n’ is the no. of elements in the list (a[I],….,a[j]) and we are interested in finding the maximum and minimum of the list.

* If the list has more than 2 elements, P has to be divided into smaller instances.
* For example , we might divide ‘P’ into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2],a[[n/2]+1],…..,a[n])
* After having divided ‘P’ into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

**Algorithm:** Recursively Finding the Maximum & Minimum

1. Algorithm MaxMin (I,j,max,min)
2. //a[1:n] is a global array, parameters I & j
3. //are integers, 1<=I<=j<=n.The effect is to
4. //set max & min to the largest & smallest value
5. //in a[I:j], respectively.
6. {
7. if(I=j) then max:= min:= a[I];
8. else if (I=j-1) then // Another case of small(p)
9. {
10. if (a[I]<a[j]) then
11. {
12. max:=a[j];
13. min:=a[I];
14. }
15. else
16. {
17. max:=a[I];
18. min:=a[j];
19. }
20. }
21. else
22. {
23. // if P is not small, divide P into subproblems.
24. // find where to split the set [**mid:=[(I+j)/2**](mid:=[(I+j)/2)**];**
25. //solve the subproblems
26. MaxMin(I,mid,max.min);
27. MaxMin(mid+1,j,max1,min1);
28. //combine the solution
29. if (max<max1) then max=max1;
30. if(min>min1) then min = min1;
31. }
32. }

* The procedure is initially invoked by the statement,

MaxMin(1,n,x,y)

* Suppose we simulate MaxMin on the following 9 elements

A: [1] [2] [3] [4] [5] [6] [7] [8] [9]

22 13 -5 -8 15 60 17 31 47

* A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made.
* For this Algorithm, each node has 4 items of information: I, j, max & imin.
* Examining fig: we see that the root node contains 1 & 9 as the values of I &j corresponding to the initial call to MaxMin.
* This execution produces 2 new calls to MaxMin, where I & j have the values 1, 5 & 6, 9 respectively & thus split the set into 2 subsets of approximately the same size.
* From the tree, we can immediately see the maximum depth of recursion is 4. (including the 1st call)
* The include no.s in the upper left corner of each node represent the order in which max & min are assigned values.

No. of element Comparison:

* If T(n) represents this no., then the resulting recurrence relations is

T(n)={ T([n/2]+T[n/2]+2 n>2

* 1. n=2
     + n=1

🡪 When ‘n’ is a power of 2, n=2^k for some +ve integer ‘k’, then

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

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

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

\*

\*

= 2^k-1T(2)+

= 2^k-1+2^k-2

= 2^k/2+2^k-2

= n/2+n-2

= (n+2n)/2)-2

#### T(n)=(3n/2)-2

\*Note that (3n/3)-3 is the best-average, and worst-case no. of comparisons when ‘n’ is a power of 2.

# 1.7 MERGE SORT:

we assume throughout that the elements are to be sorted in non-decreasing order. Given a sequence of n elements (also called keys) a[1],…,a[n].

The general idea is,

* Divide the list into two sets a[1],…,a[n/2] and a[n/2] +1],….,a[n].
* Each set is recursively sorted, and then,
* Sorted sequences are merged to produce a single sorted sequence of n elements (final result).

**FIRST PART**

**SECOND PART**

**Sorted *Array***

**FIRST PART**

**SECOND PART**

**Merge**

*Divide array into two halves*

*Recursively sort*

**Algorithm : Merge sort**

1 **Algorithm MergeSort(low, high)**

2 //a[low : high] is a global array to be sorted.

3 //Small(P) is true if there is only one element

4 //to sort. In this case the list is already sorted.

5 **{**

6 **If(low<high)** then // if there are more than one element.

7  **{**

8 // Divide P into subproblems.

9 // Find where to split the set.

10 **Mid:=(low + high)/2;**

11 // Solve the subproblems.

12 **MergeSort(low,mid);**

13 **MergeSort(mid + 1, high);**

14 // Combine the solutions.

15  **Merge(low, mid, high );**

16  **}**

17  **}**

1 **Algorithm Merge(low, mid, high)**

2 **//** a[low : high] is a global array containing two sorted

3 **//** subsets in a[low : mid] and in a[mid + 1: high] The goal

4 **//** is to merge these two sets into a single set residing

5 **//** in a[low : high]. B[ ] is an auxiliary global array.

6 **{**

7 h:= low; i:= low; j:= mid + 1;

8 **While ((h ≤ mid) and (j ≤ high)) do**

9 **{**

10  **If(a[h] ≤ a[j]) then**

11  **{**

12 b[i]:= a[h];h:= h + 1;

13 **}**

14  **else**

15 **{**

16 b[i]:= a[j]; j:= j + 1;

17 **}**

18 i:= i + 1;

19 **}**

20  **If(h>mid) then**

21 **for k:= j to high do**

22 **{**

23 b[i]:= a[k]; i:= i + 1;

24  **}**

25  **else**

26  **for k:= h to mid do**

27 **{**

28 b[i]:= a[k]; i:= i + 1;

29  **}**

30 **for k:= low to high do a[k]:= b[k];**

31  **}**

**Example :**

The operations of two way merge sort algorithm is illustrated as follows.

We assume to sort the given array a[n] into ascending order. We split it into two sub arrays: a[1],…a[n/2] and a[[n/2]+1],…a[n]. each sub array is individually sorted, and the resulting sorted sub arrays are merged to produce a single sorted array of n elements.

Consider an array of 9 elements : { 75, 40, 10, 90, 50, 95, 55, 15, 65 }. The merge sort algorithm divides the array into sub arrays and merges them into sorted sub arrays by the above ***Merge()*** algorithm.

6

5

4

3

2

1

**15**

**55**

**95**

**50**

**90**

**10**

**40**

**75**

9

8

7

**65**

**15**

**55**

**95**

9

8

7

6

**65**

5

**50**

2

1

**90**

**10**

**40**

**75**

4

3

9

8

**65**

**15**

7

6

**55**

**95**

4

2

1

5

3

**90**

**40**

**75**

**50**

**10**

9

8

7

**55**

6

5

4

3

2

**75**

1

**65**

**15**

**95**

**50**

**90**

**10**

**40**

5

**90**

4

**50**

2

1

**75**

9

8

**65**

**15**

7

6

**95**

**55**

**40**

2

1

**75**

**40**

9

8

7

6

**95**

**65**

**55**

**15**

3

2

1

**75**

**40**

**10**

5

**90**

4

3

2

1

**75**

**50**

**40**

**10**

8

7

6

5

4

3

2

1

**95**

**90**

**75**

**65**

**55**

**50**

**40**

**15**

**10**

9

**Note**:- ***Dashed-lines*** indicate the process of splitting and the ***regular arrows*** the merging process.

If the time for the merging operation is proportional to n, then the computing time for merge sort is described by the recurrence relation

{

a n=1, a is a constant

T(n)=

2T(n/2)+cn n>1, c is a constant

When n is a power of 2, n = 2k, we can solve this equation by successive substitutions:

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

= 4T(n/4) + 2cn

= 4(2T(n/8) + cn/4) + 2cn

.

.

.

.

= 2kT(1) + kcn

= an + cnlogn

It is easy to see that if 2k < n ≤ 2k+1, then T(n) ≤ T(2k+1).

:. T(n) =O(nlogn).

# 1.8 QUICKSORT:

In quick sort, the division into two sub arrays is made so that the sorted subarrays do not need to be merged later. This is accomplished by rearranging the elements in a[1 : n] such that a[i] ≤ a[j] for all i between 1 and m and all j between m + 1 and n for some m, 1≤m≤n. Thus , the elements in a[1 : m] and a[m+1 : n] can be independently sorted. No merge is needed.

**Algorithm: partition the array a[m:p-1] about a[m]**

// Within a[m],a[m+1],..,a[p-1] the elements are

// rearranged in such a manner that if initially t = a[m],

// then after completion a[q] = t for some q berween m

// and p – 1, a[k] ≤ t for m ≤ k < q,and a[k] ≥ t

// for q<k<p. q is returned.

**Algorithm Partition(a,l,u)**

**{**

**repeat**

**{**

**repeat**

p:= p + 1;

**until (a[p] ≥ pivot element);**

**repeat**

q:= q –1;

**until (a[q] ≤ pivot element);**

**If(p < q)** then **interchange(a,p,q);**

**} until (p ≥ q);**

*Interchange* the **a[q]** and **pivot element**; return q;

**}**

**Algorithm interchange(a,i,j)**

// Exchange a[i] with a[j].

**{**

temp:= a[i];

a[i] := a[j]; a[j] :=temp;

**}**

**Algorithm: Sorting by partitioning.**

**Algorithm QuickSort(p,q)**

// sorts the elements a[p],…a[q] which reside in the global

// array a[1:n] into ascending order; a[n+1] is considered to

// be defined and must be >= all the elements in a[1:n].

**{**

**If(p<q) then** //if there are more than one element

**{** // divide P into two subproblems.

**j:= partition(a,p,q+1);** // j is the position of the partitioning element.

// solve the subproblems.

**QuickSort(p,j-1);**

**QuickSort(j+1,q);**

// There is no need for combining solutions.

**}**

**}**

**Example :** Given list of elements

10 7 15 12 4 9 13 3

**Quick Sort** :

Select the first element/object as a pivot element/object, which plays the main role in the division of given problem into sub-problems. Mark ***p*** and ***q*** pointers that points the first & last elements in the remaining list.

10 7 15 12 4 9 13 3

***q***

***p***

***pivot***

Move the pointer **p** towards **right** until it reaches the element which is bigger than the pivot element and **q** towards **left** until it reaches the element which is smaller than the pivot element.

10 7 **15** 12 4 9 13 **3**

***p***

***q***

***pivot***

Now exchange the **p** and **q** values. Then the list as follows:

10 7 3 **12** 4 **9** 13 15

***q***

***p***

***pivot***

Againexchange the **p** and **q** values. Then the list as follows:

10 7 3 9 **4** **12** 13 15

***p***

***q***

***pivot***

Here we find that ***q*** crosses ***p*.** “when ever ***p/q*** crosses ***q/p”*** or **“**even if the process stops at the element which is pointed by both p & q” then perform the exchange operation of ***q*** and ***pivot*** element.

Now the pivot element is in its original position. That means, after performing the exchange operation the elements appearing to the left of pivot element are smaller than the pivot element and the elements appearing to the right of pivot element are greater than the pivot element .

4 7 3 9 **10** 12 13 15

Based on the position of pivot element we divide the list into two halves as above.

Apply the same Quick sort technique to each of the sub-lists recursively. Finally we get all the elements in the sorted order.

### *Time Complexity Analysis*

The analysis of the procedure QUICK\_SORT is given by

T(N) = P(N) + T(J-LB) + T(UB - J)

The worst case time analysis, assuming J = LB, then becomes

Tw = P(N) + Tw(0) + Tw(N-1)

= c\*N + Tw(N-1)

= c\*N + c\*(N-1) + Tw(N-2)

= c\*N + c\*(N-1) + c\*(N-2) + Tw(N-3)

.

.

.

= c \* {(N+1)(N)}/2 = **O(N2)**

The best case analysis occurs when the table is always partitioned in half, that is, J = [(LB+UB)/2]. The analysis becomes:

Tb = P(N) + 2Tb(N/2)

= c\*N + 2Tb(N/2)

= c\*N + 2c(N/2) + 4Tb(N/4)

= c\*N + 2c(N/2) + 4c(N/4) + 8Tb(N/8)

.

.

.

= (log2N)\*c\*N + 2log2N\*Tb(1)

= **O(Nlog2N)**

The average case analysis (albeit difficult to evaluate) also comes out to be **O(Nlog2N)**.

# 1.9 Selection Sort:

1. Suppose we Must devise an algorithm that sorts a collection of n>=1 elements of arbitrary type.
2. A Simple solution given by the following.
3. ( From those elements that are currently unsorted ,find the smallest & place it next in the sorted list.)

Algorithm:

1. For i:= 1 to n do

2. {

3. Examine a[I] to a[n] and suppose the smallest element is at a[j];

4. Interchange a[I] and a[j];

5. }

🡪 Finding the smallest element (sat a[j]) and interchanging it with a[ i ]

1. We can solve the latter problem using the code,

t := a[i];

a[i]:=a[j];

a[j]:=t;

1. The first subtask can be solved by assuming the minimum is a[ I ];checking a[I] with a[I+1],a[I+2]…….,and whenever a smaller element is found, regarding it as the new minimum. a[n] is compared with the current minimum.
2. Putting all these observations together, we get the algorithm Selection sort.

**Theorem:**

Algorithm selection sort(a,n) correctly sorts a set of n>=1 elements .The result remains is a a[1:n] such that a[1] <= a[2] ….<=a[n].

**Selection Sort:**

Selection Sort begins by finding the least element in the list. This element is moved to the front. Then the least element among the remaining element is found out and put into second position. This procedure is repeated till the entire list has been studied.

**Example:**

## List L = 3,5,4,1,2

1 is selected , 🡪 1,5,4,3,2

2 is selected, 🡪1,2,4,3,5

3 is selected, 🡪1,2,3,4,5

4 is selected, 🡪1,2,3,4,5

**Proof:**

* We first note that any I, say I=q, following the execution of lines 6 to 9,it is the case that a[q] Þ a[r],q<r<=n.
* Also observe that when ‘i’ becomes greater than q, a[1:q] is unchanged. Hence, following the last execution of these lines (i.e. I=n).We have a[1] <= a[2] <=……a[n].
* We observe this point that the upper limit of the for loop in the line 4 can be changed to n-1 without damaging the correctness of the algorithm.

**Algorithm:**

1. Algorithm selection sort (a,n)

2. // Sort the array a[1:n] into non-decreasing order.

3.{

4. for I:=1 to n do

5. {

6. j:=I;

7. for k:=i+1 to n do

8. if (a[k]<a[j])

9. t:=a[I];

10. a[I]:=a[j];

11. a[j]:=t;

12. }

13. }

**Recursive Algorithms:**

* A Recursive function is a function that is defined in terms of itself.
* Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
* An algorithm that calls itself is Direct Recursive.
* Algorithm ‘A’ is said to be Indirect Recursive if it calls another algorithm which in turns calls ‘A’.
* The Recursive mechanism, are externally powerful, but even more importantly, many times they can express an otherwise complex process very clearly. Or these reasons we introduce recursion here.
* The following 2 examples show how to develop a recursive algorithms.

🡪 In the first, we consider the Towers of Hanoi problem, and in the second, we generate all possible permutations of a list of characters.

* **Towers of Hanoi:**

**.**

**.**

**.**

Tower A Tower B Tower C

* It is Fashioned after the ancient tower of Brahma ritual.
* According to legend, at the time the world was created, there was a diamond tower (labeled A) with 64 golden disks.
* The disks were of decreasing size and were stacked on the tower in decreasing order of size bottom to top.
* Besides these tower there were two other diamond towers(labeled B & C)
* Since the time of creation, Brehman priests have been attempting to move the disks from tower A to tower B using tower C, for intermediate storage.
  + As the disks are very heavy, they can be moved only one at a time.
  + In addition, at no time can a disk be on top of a smaller disk.
  + According to legend, the world will come to an end when the priest have

completed this task.

* + A very elegant solution results from the use of recursion.
  + Assume that the number of disks is ‘n’.
  + To get the largest disk to the bottom of tower B, we move the remaining ‘n-1’

disks to tower C and then move the largest to tower B.

* + Now we are left with the tasks of moving the disks from tower C to B.
  + To do this, we have tower A and B available.
  + The fact, that towers B has a disk on it can be ignored as the disks larger than the

disks being moved from tower C and so any disk scan be placed on top of it.

**Algorithm:**

1. Algorithm TowersofHanoi(n,x,y,z)

2. //Move the top ‘n’ disks from tower x to tower y.

3. {

**.**

**.**

**.**

4.if(n>=1) then

5. {

6. TowersofHanoi(n-1,x,z,y);

7. Write(“move top disk from tower “ X ,”to top of tower “ ,Y);

* Towersofhanoi(n-1,z,y,x);
* }
* }

**Permutation Generator:**

Given a set of n>=1elements, the problem is to print all possible permutations of this set.

For example, if the set is {a,b,c} ,then the set of permutation is,

{ (a,b,c),(a,c,b),(b,a,c),(b,c,a),(c,a,b),(c,b,a)}

1.It is easy to see that given ‘n’ elements there are n! different permutations.

2.A simple algorithm can be obtained by looking at the case of 4 statement(a,b,c,d)

3.The Answer can be constructed by writing

* + a followed by all the permutations of (b,c,d)
  + b followed by all the permutations of(a,c,d)
  + c followed by all the permutations of (a,b,d)
  + d followed by all the permutations of (a,b,c)

**Algorithm:**

Algorithm perm(a,k,n)

{

if(k=n) then write (a[1:n]); // output permutation

else //a[k:n] ahs more than one permutation

// Generate this recursively.

for I:=k to n do

{

t:=a[k];

a[k]:=a[I];

a[I]:=t;

perm(a,k+1,n);

//all permutation of a[k+1:n]

t:=a[k];

a[k]:=a[I];

a[I]:=t;

}

}

# 1.10 STRASSEN’S MATRIX MULTIPLICATION:

Let A and B be two n x n matrices. The product matrix C = AB is also an n x n matrix whose i, jth element is formed by taking the elements in the ith row of A and the jth column of B and multiplying them to get

**C(i,j) = ∑ A(i,k)B(k,j)**

**1≤k≤n**

for all i and j between 1 and n. To compute C(i,j) using this formula, we need n multiplications. As the matrix C has n2 elements, the time for the resulting matrix multiplication algorithm, which we refer to as the conventional is O(n3).

Strassen’s matrix multiplication algorithm is able to perform this calculation in time O(n2.81). Thus one must be able to speed up the code by using Strassen’s algorithm instead of the traditional algorithm.

Given nxn matrices A and B we wish to calculate C=AB based on recursive Divide and Conquer scheme. To do this, first we divide the matrices as follows(decomposing C=AB into four blocks).

[

[

[

]

]

]

C11 C12 A11 A12 B11 B12

= X

C21 C22 A21 A22 B21 B22

*If the matrices A and B are not of type 2n x 2n  , we fill the missing rows and columns with zeros.*

The elements of C are given by:

C11 = A11 B11 + A12 B21

C12 = A11 B12 + A12 B22

C21 = A21 B11 + A22 B21

C22 = A21 B12 + A22 B22

This algorithm will continue applying itself to smaller-sized submatrices until n becomes suitably small (n=2) so that the product is computed directly.

To compute AB using the traditional method, we need to perform eight multiplications of n/2 x n/2 matrices and four additions of n/2 x n/2 matrices. Since two n/2 x n/2 matrices can be added in time cn2 for some constant c, the overall computing time T(n) of the resulting divide-and-conquer algorithm is given by the recurrence

{

b n≤2

T(n) = 8T(n/2) + cn2 n>2 *Where*  ***b*** *and* ***c*** *are constants*.

This recurrence can be solved in the same way as earlier recurrences to obtain T(n) = O(n3). Hence no improvement over the conventional method has been made.

Since matrix multiplications are more expensive than matrix additions (O(n3) versus O(n2)), we can attempt to reformulate the equations for Cij so as to have fewer multiplications and possibly more additions.

Volker Strassen has discovered a way to compute the Cij’s of using only 7 multiplications and 18 additions or substractions. His method involves first computing the seven n/2 x n/2 matrices P,Q,R,S,T,U and V as in. Then the Cij’s are computed using the formulas. As can be seen, P,Q,R,S,T,U and V can be computed using 7 matrix multiplications and 10 matrix addition or subtractions. The Cij’s require an additional 8 additions or subtractions.

P = (A11 + A22) (B11 + B22)

Q = (A21 + A22) B11

R = A11(B12 - B22)

S = A22(B21 - B11)

T = (A11 + A12) B22

U = (A21 - A11) (B11 + B12)

V = (A12 - A22) (B21 + B22)

C11 =P+S-T+V

C12 = R+T

C21 =Q+S

C22 = P+R-Q+U

The resulting recurrence relation for T(n) is

b n≤2

{

T(n) = 7T(n/2) + an2 n>2

Where a and b are constants.

**Example**, to perform the multiplication of A and B

B11

A11

[

[

]

]

B12

A12

**2 4 2 7**

B21

A22

A21

B22

AB =

**5 6 8 2**

The given matrices are small enough to perform the matrix multiplication. So, we can directly compute the multiplication using Strassen’s multiplication algorithm as follows:

P = ( 2 + 6 ) X ( 2 + 2 ) = 32

Q = ( 5 + 6 ) X 2 = 22

R = 2 X ( 7 – 2 ) = 10

S = 6 X ( 8 – 2 ) = 36

T = ( 2 + 4 ) X 2 = 12

U = ( 5 – 2 ) X ( 2 + 7 ) = 27

V = ( 4 – 6 ) X ( 8 + 2 ) = -20

C11 =P+S-T+V = 32 + 36 – 12 + (-20) = **36**

C12 = R+T = 10 + 12 = **22**

C21 =Q+S = 22 + 36 = **58**

C22 = P+R-Q+U = 32 + 10 – 22 + 27 = **47**

[

]

[

[

]

]

**2 4 2 7 36 22**

=

**5 6 8 2 58 47**

**Another Example**,

]

]

[

[

**1 2 3 4 1 4 2 7**

**0 6 0 3 3 1 3 5**

**AB = 4 1 1 2 2 0 1 3**

**0 3 5 0 1 4 5 1**

We define the following eight **n/2** by **n/2** matrices.

B12

B11

A12

A11

[

]

[

]

[

]

[

]

**1 2 3 4 1 4 2 7**

**0 6 0 3 3 1 3 5**

B22

B21

A22

A21

[

]

[

]

[

]

[

]

**4 1 1 2 2 0 1 3**

**0 3 5 0 1 4 5 1**

Substitute all the A’s and B’s in the strassen’s Matrix multiplication algorithm as in the previous example.

* The above algorithm gives the strassen’s matrix multiplication after dividing matrices into sub-matrices and recursively multiply sub-matrices.
* We perform ***seven n/2 and n/2 matrix multiplications*** and ***eighteen n/2 and n/2 matrix additions***. The matrix additions take O(n2) time. If the matrix multiplications are done recursively, then the running time satisfies

T(n) = 7T(n/2) + O(n2)

The solution of this recursion is T(n) = O(nlog27) = O(n2.81)**­­­**

# UNIT-II (PART-A)

# 2.1 GREEDY METHOD:

Many of the problems that computer algorithms are designed to solve have involve optimization or finding a best solution. That is some operation can be performed in a number of different ways, all meeting some basic constraint, and the problem is to find the way that is optimal. It means that it maximizes or minimizes the value of some desirable quantity.

In many optimization problems, there are two types of solutions: ***feasible solutions***, and ***optimal solutions***.

The greedy method suggests that one can devise an algorithm that works in stages, considering one input at a time. At each stage, a decision is made regarding whether a particular input is in an optimal solution.

Greedy algorithms are easy to invent, easy to implement and most of the time quite efficient. However, there are many problems that cannot be solved correctly by the greedy approach.

***Advantage :***

The principle advantage of greedy algorithm is that they are usually straightforward, easy to understand and easy to code.

***Disadvantage :***

Their main disadvantage is that for many problems there is no greedy algorithm.

“The one with maximum benefit from multiple choices is selected” is the basic idea of greedy method. A greedy method arrives at a solution by making a sequence of choices, each of which simply looks the best at the moment.

**Control abstraction** :

* The function ***Select*** selects an input from a[] and removes it. The selected inputs value is assigned to **x**.
* ***Feasible*** is a Boolean-valued function that determines whether **x** can be included into the solution vector.
* The function ***Union*** combines **x** with the solution and updates the objective function.
* The function ***Greedy*** describes the essential way that a greedy algorithm will look, once a particular problem is chosen and the functions Select, Feasible, and Union are properly implemented.

***Algorithm*** :

***Algorithm*** **Greedy**(a,n)

**{**

solution := 0; // initialize the solution

for i:=1 to n do

**{**

x := **Select**(a);

if **Feasible**(solution, x) then

solution := **Union**(solution, x);

**}**

return solution;

**}**

# 2.1KNAPSACK PROBLEM

we are given n objects and knapsack or bag with capacity M object I has a weight Wi where I varies from 1 to N.

1. The problem is we have to fill the bag with the help of N objects and the resulting profit has to be maximum.
2. Formally the problem can be stated as

Maximize xipi subject to XiWi<=M

Where Xi is the fraction of object and it lies between 0 to 1.

1. There are so many ways to solve this problem, which will give many feasible solution for which we have to find the optimal solution.
2. But in this algorithm, it will generate only one solution which is going to be feasible as well as optimal.
3. First, we find the profit & weight rates of each and every object and sort it according to the descending order of the ratios.
4. Select an object with highest p/w ratio and check whether its height is lesser than the capacity of the bag.
5. If so place 1 unit of the first object and decrement .the capacity of the bag by the weight of the object you have placed.
6. Repeat the above steps until the capacity of the bag becomes less than the weight of the object you have selected .in this case place a fraction of the object and come out of the loop.
7. Whenever you selected.

***ALGORITHM:***

1.Algorityhm Greedy knapsack (m,n)

2//P[1:n] and the w[1:n]contain the profit

3.// & weight res’.of the n object ordered.

4.//such that p[i]/w[i] >=p[i+1]/W[i+1]

5.//n is the Knapsack size and x[1:n] is the solution vertex.

6.{

7.for I=1 to n do a[I]=0.0;

8.U=n;

9.For I=1 to n do

10.{

11.if (w[i]>u)then break;

13.x[i]=1.0;U=U-w[i]

14.}

15.if(i<=n)then x[i]=U/w[i];

16.}

**Example:**

Capacity=20

N=3 ,M=20

Wi=18,15,10

Pi=25,24,15

Pi/Wi=25/18=1.36,24/15=1.6,15/10=1.5

Descending Order 🡺 Pi/Wi🡺1.6 1.5 1.36

Pi = 24 15 25

Wi = 15 10 18

Xi = 1 5/10 0

PiXi=1\*24+0.5\*15🡺31.5

The optimal solution is 🡺31.5

***X1 X2 X3*** ***WiXi*** ***PiXi***

½ 1/3 ¼ 16.6 24.25

1 2/5 0 20 18.2

0 2/3 1 20 31

0 1 ½ 20 31.5

Of these feasible solution Solution 4 yield the Max profit .As we shall soon see this solution is optimal for the given problem instance.

# 2.3 JOB SCHEDULING WITH DEAD LINES:

The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of job, which will be completed within its deadlines, and it should yield a maximum profit.

**Points To remember:**

* To complete a job, one has to process the job or a action for one unit of time.
* Only one machine is available for processing jobs.
* A feasible solution for this problem is a subset of j of jobs such that each job in this subject can be completed by this deadline.
* If we select a job at that time ,

🡪Since one job can be processed in a single m/c. The other job has to be in its waiting state until the job is completed and the machine becomes free.

🡪So the waiting time and the processing time should be less than or equal to the dead line of the job.

**ALGORITHM:**

Algorithm JS(d,j,n)

//The job are ordered such that p[1]>p[2]…>p[n]

//j[i] is the ith job in the optimal solution

// Also at terminal d [ J[ i]<=d[ J {i+1],1<i<k

{

d[0]= J[0]=0;

J[1]=1;

K=1;

For I =1 to n do

{ // consider jobs in non increasing order of P[I];find the position for I and check feasibility insertion

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) step –1 do J [q+1]=j[q]

J[r+1]=i;

K=k+1;

}

}

return k;

}

**Example :**

1. n=5 (P1,P2,…P5)=(20,15,10,5,1)

(d1,d2….d3)=(2,2,1,3,3)

***Feasible solution*** ***Processing Sequence*** ***Value***

(1) (1) 20

(2) (2) 15

(3) (3) 10

(4) (4) 5

(5) (5) 1

(1,2) (2,1) 35

(1,3) (3,1) 30

(1,4) (1,4) 25

(1,5) (1,5) 21

(2,3) (3,2) 25

(2,4) (2,4) 20

(2,5) (2,5) 16

(1,2,3) (3,2,1) 45

(1,2,4) (1,2,4) 40

The Solution 13 is optimal

n=4 (P1,P2,…P4)=(100,10,15,27)

(d1,d2….d4)=(2,1,2,1)

***Feasible solution*** ***Processing Sequence*** ***Value***

(1,2) (2,1) 110

(1,3) (1,3) 115

(1,4) (4,1) 127

(2,3) (9,3) 25

(2,4) (4,2) 37

(3,4) (4,3) 42

(1) (1) 100

(2) (2) 10

(3) (3) 15

(4) (4) 27

The solution 3 is optimal.

# 2.4 Minimum Spanning trees:

A *spanning tree* of a graph is just a subgraph that contains all the vertices and is a tree. A graph may have many spanning trees; for instance the complete graph on four vertices

Now suppose the edges of the graph have weights or lengths. The weight of a tree is just the sum of weights of its edges. Obviously, different trees have different lengths. The problem: how to find the minimum length spanning tree?

has sixteen spanning trees:

## *Applications of Minimum-Cost Spanning Trees*

## Minimum-cost spanning trees have many applications. Some are:

## Building cable networks that join n locations with minimum cost.

## Building a road network that joins n cities with minimum cost.

## Obtaining an independent set of circuit equations for an electrical network.

## In pattern recognition minimal spanning trees can be used to find noisy pixels.

This problem can be solved by many different algorithms. Two of them include :

* + - * ***“Prims”*** Algorithm
      * ***“Kruskal’s”***  Algorithm.

## 

## 2.4.1 PRIM’S ALGORITHM

Prim and Dijkstra discovered independently for creating a minimum-cost spanning tree for a weighted graph.

The minimum spanning tree grows in successive stages.

* At any stage, we can make out that we have a set of nodes that have already been included in the tree, the remainder of the nodes have not.
* The Prim’s algorithm then finds a new node to be included in the tree by choosing the edge ( vi , vj ) has the minimum weight (cost) among all edges, where vi  is the in the tree and vj  is yet to be added to the tree.

The algorithm starts by selecting a node arbitrarily, and then in each stage, we include an edge (by adding an associated node) in the tree.

* We can represent a weighted graph by an adjacency matrix to store the set of edges.
* An entry(i,j) in an adjacency matrix contains information on the edge that goes from the vertex i to the vertex j.
* Each matrix entry constrains the weight of the corresponding edge.
* A specially chosen weight value is used to indicate edges that the missing from the graph.
* A large value “INF” is used to denoted an edge that is missing from the graph(“INF” may be a constant equal to 99999). The adjacency matrix for the weighted undirected graph is shown as follows:

**28**

**10**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | ∞ | **28** | ∞ | ∞ | ∞ | **10** | ∞ |
| 2 | **28** | ∞ | **16** | ∞ | ∞ | ∞ | **14** |
| 3 | ∞ | **16** | ∞ | **12** | ∞ | ∞ | ∞ |
| 4 | ∞ | ∞ | **12** | ∞ | **22** | ∞ | **18** |
| 5 | ∞ | ∞ | ∞ | **22** | ∞ | **25** | **24** |
| 6 | **10** | ∞ | ∞ | ∞ | **25** | ∞ | ∞ |
| 7 | ∞ | **14** | ∞ | **18** | **24** | ∞ | ∞ |

**14**

**16**

**18**

**24**

**25**

**12**

**22**

**10**

**25**

**10**

**25**

**10**

**22**

**25**

**10**

**22**

**12**

**25**

**10**

**22**

**12**

**16**

**14**

**25**

**10**

**22**

**12**

**16**

**(a)**

**(b)**

**(c)**

**(d)**

**(e)**

**(f)**

* The Prim’s algorithm is implemented using the adjacency matrix of a graph.
* This matrix is denoted by adjMatrix [i,j] where I and j operate from 0 to n-1 for n-node weighted undirected graph.
* We assume that the [i,j] element of the adjacency matrix is infinity (INF), if there is no edge between the nodes i and j.
* we can use a large value for infinity, which is not nearer to the weights of the given graph.
* The selected is an integer array.
* The elements of selected contain either 0’s or 1’s. (0 represents not included in the minimum spanning tree and 1 included). The variables used in this algorithm are as follows:

**Size** = Number of nodes in the graph.

**INF** denoted a constant of having a large positive value of 99999.

**nEdges** = Current number of edges included in the MST.

**Min** = Minimum weight (cost) of edge.

**Row**, **col** indicate the row and column of the edge included in the MST.

**Sum** = Total minimum weight (cost) of the edges included in the MST.

***Algorithm***: Minimum cost spanning tree using Prim’s algorithm

Algorithm Prims( a[][] , int n)

{

size := n;

for i := 0 to size step 1 do

for j := 0 to size step 1 do

adjMatrix[i][j] := a[i][j];

selected[0] := 1;

nEdges := 1;

while(nEdges < size)

{

min := 99999;

for i:=0 to size step 1 do

if(selected[i] = 1 )

for j=0 to size step 1 do

if(selected[j] = 0)

if(min>adjMatrix[i][j])

{

min := adjMatrix[i][j];

row := i; col := j;

}

selected[col] := 1;

nEdges ++;

sum := sum + adjMatrix[row][col];

//selected edge is row - col

}

}

## 2.4.2 Kruskal's algorithm:

* Kruskals algorithm constructs the minimum spanning tree of a graph by adding edges to the spanning tree one-by-one.
* At all points during its execution, the set of edges selected by prim’s algorithm forms exactly one tree.
* On the other hand, The set of edges selected by kruskal’s algorithm forms a forest of trees.
* Kruskals algorithm is conceptually quite simple.

***Kruskal's algorithm****:*

***sort the edges of G in increasing order by length***

***keep a subgraph S of G, initially empty***

***for each edge e in sorted order***

***if the endpoints of e are disconnected in S***

***add e to S***

***return S***

Note that, whenever you add an edge (u,v), it's always the smallest connecting the part of S reachable from u with the rest of G, so by the lemma it must be part of the MST.

This algorithm is known as a *greedy algorithm*, because it chooses at each step the cheapest edge to add to S.

The kruskals algorithm by considering the graph G shown in figure 16.14(a).

* The edges of the graph are arranged in increasing order of weights
* initially the spanning tree, T is empty.
* We select the edge with smallest weight and include it in T.
* If the selected edge creates a cycle, then it will be removed from T.
* Repeat these two steps until the tree T contains n-1 edges(where n is the number of vertices in the graph G).
* If the tree T contains less than n-1 edges and the edge list is empty, then no spanning tree T is possible for the graph G. these steps are illustrated as follows:

Consider the given graph G:

**12**

**9**

**8**

**7**

**4**

**5**

**8**

**6**

**4**

The edges of the graph are arranged in increasing order of weights

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Edge | 1-6 | 2-3 | 1-2 | 3-6 | 3-4 | 4-5 | 4-6 | 0-1 | 0-5 |
| Weight | 4 | 4 | 5 | 6 | 7 | 8 | 8 | 9 | 12 |
| Include | yes | yes | yes | no | yes | yes | no | yes | no |

1. Initial forest of ***n*** trees (b) Edge 1-6 is added

**4**

**4**

**4**

**4**

**4**

**5**

(c) Edge 2-3 is added (d) Edge 1-2 is added

If we include edge 3-6 in the tree T, then a cycle (1-2-3-6-1) will be formed. So, reject the edge 3-6. Next edge 3-4 with weight 7 is added.

(e) Edge 3-4 is added (f) Edge 4-5 is added

**4**

**4**

**5**

**7**

**4**

**4**

**5**

**7**

**8**

Addition of edge 4-6 also creates a cycle. Finally, edge 0-1 is added to obtain the minimum spanning tree.

**9**

**8**

**7**

**4**

**5**

**4**

# 2.5 SINGLE SOURCE SHORTEST PATH:

A minimum spanning tree gives no indication about the shortest path between two vertices. In real-life, we are required to find the shortest path between two cities.

***For example***, one would be interested in finding most economical route between any two cities in a given railway network.

We are given a directed graph G in which every edge has a weight, and our problem is to find a path from one vertex v to another vertex w such that the sum of the weights on the path is as minimal as possible. We shall call such a path a shortest path, even though the weights may represent *costs*, *time*, or some other *quantity* other than *distance*.

**10**

**20**

**9**

**2**

**10**

**6**

**2**

**4**

**12**

It turns out that it is just easy to solve the more general problem of starting at one node called the source, and finding the shortest path to every other node, instead of to just one destination node. For simplicity, we take the source to be node 1, and our problem then consists of finding the shortest path from node 1 to every other node in the graph.

The solution we will show for the shortest-path problem is called Dijkstra’s Algorithm.

***Dijkstra’s Algorithm***

* This algorithm is based on the *adjacency matrix* representation of a graph.
* It finds not only the shortest path from one specified vertex to another, but the shortest paths from the specified vertex to all the other vertices.
* ***Dijkstra’s Algorithm*** is called as greedy technique.

This algorithm works by maintaining a set ***S*** of vertices whose shortest distance from the source is already known.

* Initially, ***S*** contains only the *source* vertex.
* At each step, we added to ***S*** a remaining vertex ***v*** whose distance from the source is as short as possible.
* Then we can find a shortest path from the source to ***v*** that passes only through vertices in ***S***. at each step of the algorithm, we use an array ***dist*** to record the length of the shortest path to each vertex.
* Once ***S*** includes all vertices, ***dist*** will hold the shortest distance from the source to each vertex.

***Greedy Algorithm to generate Shortest paths:***

**Algorithm *ShortestPaths*(v,cost,dist,n)**

**{**

for i:=1 to n do

{ *// Initialize S.*

S[i] := false; dist[i] := cost[v,i];

}

S[v] := true; dist[v] := 0.0;

for num = 2 to n-1 do

{

Choose u from among those vertices not in S such that dist[u] is minimum;

S[u] := true; *// put u in S*

for (each w adjacent to u with S[w] = false) do

*//update distances*

if(dist[w]>dist[u]+cost(u,w)) then

dist[w] := dist[u] + cost[u,w];

}

**}**

Let us apply ***Dijkstra*** to the given digraph

**10**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | ∞ | **2** | ∞ | **6** | **20** | **10** |
| 2 | ∞ | ∞ | **10** | ∞ | ∞ | ∞ |
| 3 | ∞ | ∞ | ∞ | ∞ | **2** | ∞ |
| 4 | ∞ | ∞ | **4** | ∞ | **12** | ∞ |
| 5 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| 6 | ∞ | ∞ | ∞ | ∞ | **9** | ∞ |

**2**

**9**

**20**

**6**

**12**

**2**

**10**

**4**

From the above adjacency matrix of a given graph Initially, S = {1},d[2] = 2, d[3] = ∞, d[4] = 6, d[5] = 20, d[6] = 10.

**10**

**20**

**2**

**6**

**d=10**

**d=12**

**d=6**

**d=10**

**d=2**

*S = {1,2,4,3}*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **iteration** | **S** | **d[2]** | **d[3]** | **d[4]** | **d[5]** | **d[6]** |
| **(initial)** | **{1}** | **2** | **∞** | **6** | **20** | **10** |
| **(1)** | **{1,2}** | **2** | **12** | **6** | **20** | **10** |
| **(2)** | **{1,2,4}** | **2** | **10** | **6** | **18** | **10** |
| **(3)** | **{1,2,4,3}** | **2** | **10** | **6** | **12** | **10** |

**4**

**12**

**2**

**10**

**20**

**2**

**6**

**d=10**

**d=18**

**d=6**

**d=10**

**d=2**

*S = {1,2,4}*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **iteration** | **S** | **d[2]** | **d[3]** | **d[4]** | **d[5]** | **d[6]** |
| **(initial)** | **{1}** | **2** | **∞** | **6** | **20** | **10** |
| **(1)** | **{1,2}** | **2** | **12** | **6** | **20** | **10** |
| **(2)** | **{1,2,4}** | **2** | **10** | **6** | **18** | **10** |

**4**

**12**

**10**

**10**

**20**

**2**

**6**

**d=10**

**d=20**

**d=6**

**d=12**

**d=2**

*S = {1,2}*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **iteration** | **S** | **d[2]** | **d[3]** | **d[4]** | **d[5]** | **d[6]** |
| **(initial)** | **{1}** | **2** | **∞** | **6** | **20** | **10** |
| **(1)** | **{1,2}** | **2** | **12** | **6** | **20** | **10** |

**10**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **iteration** | **S** | **d[2]** | **d[3]** | **d[4]** | **d[5]** | **d[6]** |
| **(initial)** | **{1}** | **2** | **∞** | **6** | **20** | **10** |

**10**

**20**

**2**

**6**

**d=10**

**d=20**

**d=6**

**d=∞**

**d=2**

*Initial S = {1}*

**10**

**2**

**6**

**d=10**

**d=12**

**d=6**

**d=10**

**d=2**

*S = {1,2,4,3,6}*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **iteration** | **S** | **d[2]** | **d[3]** | **d[4]** | **d[5]** | **d[6]** |
| **(initial)** | **{1}** | **2** | **∞** | **6** | **20** | **10** |
| **(1)** | **{1,2}** | **2** | **12** | **6** | **20** | **10** |
| **(2)** | **{1,2,4}** | **2** | **10** | **6** | **18** | **10** |
| **(3)** | **{1,2,4,3}** | **2** | **10** | **6** | **12** | **10** |
| **(4)** | **{1,2,4,3,6}** | **2** | **10** | **6** | **12** | **10** |

**4**

**12**

**2**

**9**

**10**

**2**

**6**

**d=10**

**d=12**

**d=6**

**d=10**

**d=2**

*S = {1,2,4,3,6,5}*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **iteration** | **S** | **d[2]** | **d[3]** | **d[4]** | **d[5]** | **d[6]** |
| **(initial)** | **{1}** | **2** | **∞** | **6** | **20** | **10** |
| **(1)** | **{1,2}** | **2** | **12** | **6** | **20** | **10** |
| **(2)** | **{1,2,4}** | **2** | **10** | **6** | **18** | **10** |
| **(3)** | **{1,2,4,3}** | **2** | **10** | **6** | **12** | **10** |
| **(4)** | **{1,2,4,3,6}** | **2** | **10** | **6** | **12** | **10** |
| **(5)** | **{1,2,4,3,6,5}** | **2** | **10** | **6** | **12** | **10** |

**4**

**12**

**2**

At last the d[i] contains the shortest path from source vertex 1 to the vertex i.

# DYNAMIC PROGRAMMING (PART II)

# 2.6 GENERAL METHOD

* Algorithms designed using *dynamic programming* are similar to those developed using *divide-and-conquer*.
* Both solve a problem by *breaking* it down into several sub-problems that can be solved recursively.
* The *difference between the two* is that in the dynamic programming approach, the results obtained from solving smaller sub-problems are reused in the calculation of large sub-problems.
* Thus dynamic programming is a *bottom-up technique* that usually begins by solving the smaller sub-problems, saving these results, and then reusing them to solve larger sub-problems until the solution to the original problem is obtained.
* This is in contrast to the divide-and-Conquer approach, which solves problems in a top-down method. **­­­**

# 2.7 MULTISTAGE GRAPH

* A multistage graph G = (V,E) is a directed graph in which the vertices are portioned 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.
* If there will be only one vertex, then the sets Vi and Vk are such that [Vi]=[Vk] = 1.
* Let ‘s’ and ‘t’ be the source and destination respectively.
* The cost of a path from source (s) to destination (t) is the sum of the costs of the edger 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. 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 terminates in stage-k.
* This *MULISTAGE GRAPH* problem can be solved in 2 ways.

## Forward Method.

Backward Method

## 2.7.1 FORWARD METHOD

Assume that there are ‘k’ stages in a graph.

In this *FORWARD* approach, we will find out the cost of each and every node starling from the ‘k’ th stage to the 1st stage.

We will find out the path (i.e.) minimum cost path from source to the destination (ie) [ Stage-1 to Stage-k ].

**PROCEDURE:**

V1 V2 V3 V4 V5



4 6





2 2

5 4

9 1

4



7 3 2





7 t



12

s

3



11 5 5

2



11 6

11



8

1. Maintain a cost matrix cost (n) which stores the distance from any vertex to the destination.
2. If a vertex is having more than one path, then we have to choose the minimum distance path and the intermediate vertex, which gives the minimum distance path, will be stored in the distance array ‘D’.
3. In this way we will find out the minimum cost path from each and every vertex.
4. Finally cost(1) will give the shortest distance from source to destination.
5. For finding the path, start from vertex-1 then the distance array D(1) will give the minimum cost neighbour vertex which in turn give the next nearest vertex and proceed in this way till we reach the Destination.
6. For a ‘k’ stage graph, there will be ‘k’ vertex in the path.
7. In the above graph V1…V5 represent the stages. This 5 stage graph can be solved by using forward approach as follows,

**STEPS: - DESTINATION, D**

Cost (12)=0 D (12)=0

Cost (11)=5 D (11)=12

Cost (10)=2 D (10)=12

Cost ( 9)=4 D ( 9)=12

1. For forward approach,

|  |
| --- |
| Cost (i,j) = min {C (j,l) + Cost (i+1,l) }  l ∈ Vi + 1  (j,l) ∈E |

Cost(8) = min {C (8,10) + Cost (10), C (8,11) + Cost (11) }

= min (5 + 2, 6 + 5)

= min (7,11)

= 7

cost(8) =7 =>D(8)=10

cost(7) = min(c (7,9)+ cost(9),c (7,10)+ cost(10))

(4+4,3+2)

= min(8,5)

= 5

cost(7) = 5 =>D(7) = 10

cost(6) = min (c (6,9) + cost(9),c (6,10) +cost(10))

= min(6+4 , 5 +2)

= min(10,7)

= 7

cost(6) = 7 =>D(6) = 10

cost(5) = min (c (5,7) + cost(7),c (5,8) +cost(8))

= min(11+5 , 8 +7)

= min(16,15)

= 15

cost(5) = 15 =>D(5) = 18

cost(4) = min (c (4,8) + cost(8))

= min(11+7)

= 18

cost(4) = 18 =>D(4) = 8

cost(3) = min (c (3,6) + cost(6),c (3,7) +cost(7))

= min(2+7 , 7 +5)

= min(9,12)

= 9

cost(3) = 9 =>D(3) = 6

cost(2) = min (c (2,6) + cost(6),c (2,7) +cost(7) ,c (2,8) +cost(8))

= min(4+7 , 2+5 , 1+7 )

= min(11,7,8)

= 7

cost(2) = 7 =>D(2) = 7

cost(1) = min (c (1,2)+cost(2) ,c (1,3)+cost(3) ,c (1,4)+cost(4) ,c(1,5)+cost(5))

= min(9+7 , 7 +9 , 3+18 , 2+15)

= min(16,16,21,17)

= 16

cost(1) = 16 =>D(1) = 2

The path through which you have to find the shortest distance.









(i.e.)

12

Start from vertex - 2

D ( 1) = 2

D ( 2) = 7

D ( 7) = 10

D (10) = 12

So, the minimum –cost path is,

9 2 3 2

12









**∴** The cost is 9+2+3+2+=16

ALGORITHM: FORWARD METHOD

**Algorithm FGraph (G,k,n,p)**

// The I/p is a k-stage graph G=(V,E) with ‘n’ vertex.

// Indexed in order of 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-1 do

{

//compute cost[j],

// let ‘r’ be the vertex such that <j,r> is an edge of ‘G’ &

// c[j,r]+cost[r] is minimum.

cost[j] = c[j+r] + cost[r];

d[j] =r;

}

// find a minimum cost path.

P[1]=1;

P[k]=n;

For j=2 to k-1 do

P[j]=d[p[j-1]];

}

**ANALYSIS:**

The time complexity of this forward method is O( V + E )

## 2.7.2 BACKWARD METHOD:

* if there one ‘K’ stages in a graph using back ward approach. we will find out the cost of each & every vertex starting from 1st

stage to the kth stage.

* We will find out the minimum cost path from destination to source (ie)[from stage k to stage 1]

**PROCEDURE:**

* It is similar to forward approach, but differs only in two or three ways.
* Maintain a cost matrix to store the cost of every vertices and a distance matrix to store the minimum distance vertex.
* Find out the cost of each and every vertex starting from vertex 1 up to vertex k.
* To find out the path star from vertex ‘k’, then the distance array D (k) will give the minimum cost neighbor vertex which in turn gives the next nearest neighbor vertex and proceed till we reach the destination.

**STEP:**

Cost(1) = 0 => D(1)=0

Cost(2) = 9 => D(2)=1

Cost(3) = 7 => D(3)=1

Cost(4) = 3 => D(4)=1

Cost(5) = 2 => D(5)=1

Cost(6) =min(c (2,6) + cost(2),c (3,6) + cost(3))

=min(13,9)

cost(6) = 9 =>D(6)=3

Cost(7) =min(c (3,7) + cost(3),c (5,7) + cost(5) ,c (2,7) + cost(2))

=min(14,13,11)

cost(7) = 11 =>D(7)=2

Cost(8) =min(c (2,8) + cost(2),c (4,8) + cost(4) ,c (5,8) +cost(5))

=min(10,14,10)

cost(8) = 10 =>D(8)=2

Cost(9) =min(c (6,9) + cost(6),c (7,9) + cost(7))

=min(15,15)

cost(9) = 15 =>D(9)=6

Cost(10)=min(c(6,10)+cost(6),c(7,10)+cost(7)),c (8,10)+cost(8)) =min(14,14,15)

cost(10)= 14 =>D(10)=6

Cost(11) =min(c (8,11) + cost(8))

cost(11) = 16 =>D(11)=8

cost(12)=min(c(9,12)+cost(9),c(10,12)+cost(10),c(11,12)+cost(11)) =min(19,16,21)

cost(12) = 16 =>D(12)=10

**PATH:**

Start from vertex-12

D(12) = 10

D(10) = 6

D(6) = 3

D(3) = 1

So the minimum cost path is,

1 7 3 2 6 5 10 2 12

The cost is 16.

**ALGORITHM :** **BACKWARD METHOD**

**Algorithm BGraph (G,k,n,p)**

// The I/p is a k-stage graph G=(V,E) with ‘n’ vertex.

// Indexed in order of 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.

{

bcost[1]=0.0;

for j=2 to n do

{

//compute bcost[j],

// let ‘r’ be the vertex such that <r,j> is an edge of ‘G’ &

// bcost[r]+c[r,j] is minimum.

bcost[j] = bcost[r] + c[r,j];

d[j] =r;

}

// find a minimum cost path.

P[1]=1;

P[k]=n;

For j= k-1 to 2 do

P[j]=d[p[j+1]];

}

# 2.8 ALL - PAIRS SHORTEST PATHS:

Let G = (V, E) be a direction graph with n vertices. Let cost be a cost adjacency matrix for G such that cost ( i , i ) = 0 , 1 < i < n . Then cost ( i, j ) is the length ( or cost) of edge ( i , j ) if ( i j ) € E(G) and cost( i , j ) = ∞ if i ≠ j and ( i , j ) ¢ E(G). The all-pairs shortest-path problem is to determine 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 problem using the algorithm Shortest Path.

A( i ,j ) = min{ { Ak-1 ( i , k ) + Ak-1 ( k, j )}, cost( i, j )}

Cleary, A0 ( i , j ) = cost( i , j ), 1 ≤ i ≤ n. We can obtain a recurrence for Ak (i, j) using an argument similar to that used before. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, Ak ( i, j ) = Ak-1 ( i , k ) + Ak-1 ( k, j ). If it does not then no intermediate vertex has index greater than k-1. Hence Ak (i, j) = Ak-1 (i, j). Combining, we get

Ak( i ,j ) = min{ Ak-1 ( i , j ) , Ak-1 ( i , k ) + Ak-1 ( k, j )} , k≥1

### Algorithm:function to compute lengths of shortest paths

**Algorithm AllPaths (cost, A, n)**

**//** cost [1: n, 1: n] is the cost adjacency matrix of a graph with n vertices; A [i, j]

// is the cost of a shortest path from vertex i to vertex j, cost [i, j] = 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]);

**}**

**Example :**

**6**

**4**

**2**

**3**

**11**

|  |  |  |  |
| --- | --- | --- | --- |
| **A0** | 1 | 2 | 3 |
| 1 | **0** | **4** | **11** |
| 2 | **6** | **0** | **2** |
| 3 | **3** | **∞** | **0** |

The given graph and has the cost matrix

The initial A matrix, A(0) , plus its values after 3 iterations A(1), A(2), A(3) are calculated as,

For **A(1)** matrix, where k=1,

Ak( i ,j ) = min{ Ak-1 ( i , j ) , Ak-1 ( i , k ) + Ak-1 ( k, j )} // From the above matrix **A0**

* A1( 1 ,2 ) = min{ A0 ( 1 , 2 ) , A0 ( 1 , 1 ) + A0 ( 1, 2 )}

= min{ 4, 0+4 } = **4**

* A1( 1 ,3 ) = min{ A0 ( 1 , 3 ) , A0 ( 1 , 1 ) + A0 ( 1, 3 )}

|  |  |  |  |
| --- | --- | --- | --- |
| **A(1)** | 1 | 2 | 3 |
| 1 | **0** | **4** | **11** |
| 2 | **6** | **0** | **2** |
| 3 | **3** | **7** | **0** |

= min{ 11, 0+11 } = **11**

* A1( 2 ,1 ) = min{ A0 ( 2 , 1 ) , A0 ( 2 , 1 ) + A0 ( 1, 1 )}

= min{ 6, 6+0 } = **6**

* A1( 2 ,3 ) = min{ A0 ( 2 , 3 ) , A0 ( 2 , 1 ) + A0 ( 1, 3 )}

= min{ 2, 6+11 } = **2**

* A1( 3 ,1 ) = min{ A0 ( 3 , 1 ) , A0 ( 3 , 1 ) + A0 ( 1, 1 )}

= min{ 3, 3+0 } = **3**

* A1( 3 ,2 ) = min{ A0 ( 3 , 2 ) , A0 ( 3 , 1 ) + A0 ( 1, 2 )}

= min{ **∞**, 3+4 } = **7**

For **A(2)** matrix, where k=2,

Ak( i ,j ) = min{ Ak-1 ( i , j ) , Ak-1 ( i , k ) + Ak-1 ( k, j )} // From the above matrix **A1**

* A2( 1 ,2 ) = min{ A1 ( 1 , 2 ) , A1 ( 1 , 2 ) + A1 ( 2, 2 )}

= min{ 4, 4+0 } = **4**

|  |  |  |  |
| --- | --- | --- | --- |
| **A(2)** | 1 | 2 | 3 |
| 1 | **0** | **4** | **6** |
| 2 | **6** | **0** | **2** |
| 3 | **3** | **7** | **0** |

* A2( 1 ,3 ) = min{ A1 ( 1 , 3 ) , A1 ( 1 , 2 ) + A1 ( 2, 3 )}

= min{ 11, 4+2 } = **6**

* A2( 2 ,1 ) = min{ A1 ( 2 , 1 ) , A1 ( 2 , 2 ) + A1 ( 2, 1 )}

= min{ 6, 6+0 } = **6**

* A2( 2 ,3 ) = min{ A1 ( 2 , 3 ) , A1 ( 2 , 2 ) + A1 ( 2, 3 )}

= min{ 2, 0+2 } = **2**

* A2( 3 ,1 ) = min{ A1 ( 3 , 1 ) , A1 ( 3 , 2 ) + A1 ( 2, 1 )}

= min{ 3, 7+6 } = **3**

* A2( 3 ,2 ) = min{ A1 ( 3 , 2 ) , A1 ( 3 , 2 ) + A1 ( 2, 2 )}

= min{ 7, 7+0 } = **7**

For **A(3)** matrix, where k=3,

Ak( i ,j ) = min{ Ak-1 ( i , j ) , Ak-1 ( i , k ) + Ak-1 ( k, j )} // From the above matrix **A2**

* A3( 1 ,2 ) = min{ A2 ( 1 , 2 ) , A2 ( 1 , 3 ) + A2 ( 3, 2 )}

= min{ 4, 6+7 } = **4**

* A3( 1 ,3 ) = min{ A2 ( 1 , 3 ) , A2 ( 1 , 3 ) + A2 ( 3, 3 )}

= min{ 6, 6+0 } = **6**

* A3( 2 ,1 ) = min{ A2 ( 2 , 1 ) , A2 ( 2 , 3 ) + A2 ( 3, 1 )}

= min{ 6, 2+3 } = **5**

|  |  |  |  |
| --- | --- | --- | --- |
| **A(3)** | 1 | 2 | 3 |
| 1 | **0** | **4** | **6** |
| 2 | **5** | **0** | **2** |
| 3 | **3** | **7** | **0** |

* A3( 2 ,3 ) = min{ A2 ( 2 , 3 ) , A2 ( 2 , 3 ) + A2 ( 3, 3 )}

= min{ 2, 2+0 } = **2**

* A3( 3 ,1 ) = min{ A2 ( 3 , 1 ) , A2 ( 3 , 3 ) + A2 ( 3, 1 )}

= min{ 3, 0+3 } = **3**

* A3( 3 ,2 ) = min{ A2 ( 3 , 2 ) , A2 ( 3 , 3 ) + A2 ( 3, 2 )}

= min{ 7, 0+7 } = **7**

Finally the values of A(3) table are the shortest paths between any pair of vertices in the given graph.

# 2.9 OPTIMAL BINARY SEARCH TREES:

* Get a fixed set of identifiers.
* we wish to create a binary search tree organization.
* We may expect different binary search trees for the same identifier set to have different performance characteristics.
* The following tree***(a)***, in the worst case, requires four comparisons to find an identifier,

**(a)**

**(b)**

* whereas the above tree***(b)***, which requires only three comparisons to find an identifier.
* On the average the two trees need 12/5 and 11/5 comparisons, respectively.

***For example***, in the case of tree***(a)***, it takes 1,2,3, and 4 comparisons, respectively, to find the identifiers **for ,do, while , int** and **if**. Thus the average number of comparisons is (1+2+3+4)/5 = 12/5. This calculation assumes that each identifier is searched for with equal probability and that no unsuccessful searches ( i.e., searches for identifiers not in the tree) are made.

In general situations, we can expect different identifiers to be searched for with different frequencies (or probabilities). In addition, we can expect unsuccessful searches also to be made.

* Let us assume that the given set of identifiers is {a1, a2… an} with a1 < a2 < ….<an .
* Let ***p( i )*** be the probability with which we search for ai.
* Let ***q(i)*** is probability of an unsuccessful search.
* Given this data, we wish to construct an optimal binary search tree for {a1, a2,…, an}.

**Algorithm** :

**Algorithm OBST(p,q,n)**

**{**

For i:=0 to n-1 do

{

w[i,i] := q[i]; r[i,i] = 0; c[i,i] =0;

// Optimal tree with one node

w[i,i+1] := q[i] + q[i+1] + p[i+1];

r[i,i+1] := i+1

c[i,i+1] := q[i] + q[i+1] + p[i+1];

}

For m:=2 to n do // find optimal trees with m nodes

For i:=0 to n-m do

{

j := i+m;

w[i,j] := w[i,j-1] + p[j] +q[j];

k := find(c,r,i,j); // it returns the value to the k that minimizes

//the ***c***

c[i,j] := w[i,j] + c[i,k-1]+c[k,j];

r[i,j] := k;

}

**}**

**Example**:

***Given problem*,**

Let n = 4 and (a1, a2, a3, a4 ) = ( do, if, int, while ).

Let p(1: 4) = (3, 3, 1, 1) and

q(0: 4) = (2, 3, 1, 1, 1).

Initially, we have the given constraints,

**(1) w(i, i ) = q(i),**

**(2) c(i, i) = 0,**

**(3) r(i, i) = 0, 0≤ i ≤ 4.**

**(4) w(i, j ) = p(j) + q(j) + w(i,j-1),**

**(5) c(i, j) = w(i,j) + min {c[i,k-1]+c[k,j]},**

**i<k<j**

**(6) r(i, j) = is the value of k that minimizes c(i,j).**

*from the above observations*, we get

**Solution :**

*Initially,*

w(0,0) = 2 w(1,1) = 3 w(2,2) = 1 w(3,3) = 1 w(4,4) = 1

c(0,0) = 0 c(1,1) = 0 c(2,2) = 0 c(3,3) = 0 c(4,4) = 0

r(0,0) = 0 r(1,1) = 0 r(2,2) = 0 r(3,3) = 0 r(4,4) = 0

*by using the above equation c(i,j) and the observation w(i,j), we get*

=> w (0, 1) = p(1) + q (1) + w (0, 0) = 8

c (0, 1) = w (0, 1) + min**{**c(0, 0) + c(1, 1)**}** = 8

0<k<1 (for k=1)

r (0, 1) = 1

=> w (1, 2) = p (2) + q(2) + w(1, 1) = 7

c (1, 2) = w(1, 2) + min{c(1, 1) + c(2, 2)} = 7

1<k<2 (for k=2)

r (1, 2) = 2

=> w (2, 3) = p(3) + q(3) + w(2, 2) = 3

c (2, 3) = w(2, 3) + min{c(2, 2) + c(3, 3)} = 3

2<k<3 (for k=3)

r (2, 3) = 3

=> w (3, 4) = p(4) + q(4) + w(3, 3) = 3

c (3, 4) = w(3, 4) + min{c(3, 3) + c(4, 4)} = 3

3<k<4 (for k=4)

r (3, 4) = 4

*knowing w(i,i+1) and c(i,i+1), 0< i < 4, we can again use the same equation c(i,j) and the observation w(i,j) to compute w(i,i+2), c(i, i+2) and r(i, i+2) , 0≤ i ≤ 3.*

=> w (0, 2) = p(2) + q (2) + w (0, 1) = 12

c (0, 2) = w (0, 2) + min**{**c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)**}** = 8

0<k<2 (for k=1) (for k=2)

= 12 + min**{ 0 + 7 , 8 + 0 }**

**=** 12 + 7 **(*7*** *is minimum for k=1***)**

= 19

r (0, 2) = 1

=> 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) } = 12

1<k<3 (for k=2) (for k=3)

r (1, 3) = 2

=> w (2, 4) = p(4) + q(4) + w(2, 3) = 5

c (2, 4) = w(2, 4) + min{c(2, 2) + c(3, 4) , c(2, 3) + c(4, 4)} = 8

2<k<4 (for k=3) (for k=4)

r (2, 4) = 3

*This process can be repeat until w(0, 4), c(0,4), and r(0, 4) are obtained.*

The table of Figure shows in below that shows the results of this computation.

The box in row i and column j shows the values of w(j, j + i), c(j, j + i) and r(j, j + i) respectively.

The computation is carried out by row from row 0 to row 4.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **j** | **0** | **1** | **2** | **3** | **4** |
| **i**  **0** | w00 = 2  c00 = 0  r00 = 0 | w11 = 3  c11 = 0  r11 = 0 | w22 = 1  c22 = 0  r22 = 0 | w33 = 1  c33 = 0  r33 = 0 | w44 = 1  c44 = 0  r44 = 0 |
| **1** | w01 = 8  c01 = 8  r01 = 1 | w12 = 7  c12 = 7  r12 = 2 | w23 = 3  c23 = 3  r23 = 3 | w34 = 3  c34 = 3  r34 = 4 |  |
| **2** | w02 = 12  c02 = 19  r02 = 1 | w13 = 9  c13 = 12  r13 = 2 | w24 = 5  c24 = 8  r24 = 3 |  |  |
| **3** | w03 = 14  c03 = 25  r03 = 2 | w14 = 11  c14 = 19  r14 = 2 |  |  |  |
| **4** | w04 = 19  c04 = 32  r04 = 2 |  |  |  |  |

From the table we see that c(0, 4) = 32 is the minimum cost of s binary search tree for (a1, a2, a3, a4 ). The root of tree t04 is a2. Hence, the left sub tree is t01 and the right sub tree t24. Tree t01 has root a1 and sub tree t00 and t11. Tree t24 has root a3; its left sub tree is t22 and its right sub tree t34. Thus, with the data in the table it is possible to reconstruct t04. the following figure shows t04 .

# 2.9 The 0/1 Knapsack problem:

### We have a Knapsack (bag) with capacity, W = 5. the weights and values of five items are given below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **item(i)** | **1** | **2** | **3** | **4** |
| **wi** | 2 | 3 | 4 | 5 |
| **vi** | 4 | 8 | 9 | 11 |

The objective of knapsack problem is to fill the knapsack with items to maximize the total value subject to its capacity. It is known as 0/1 knapsack problem. Because we put one item into the knapsack or not (may not include a fraction of an item).

To formulate the problem in terms of dynamic programming, we define a sub-problem *S[k,u]*, which will be the optimal solution for the first k terms, involving up to *u* weight. *S[k,u]* is a recursive formula, which will specify how the sub-problems can be solved. The real insight of dynamic programming is that we will take this recursive solution and convert it into an iterative solution.

### We will formulate this in terms of the solution that includes any of the items *1…k-1.* if we have the optimal solution for a given weight *u* for these *k-1* items, then the optimal solution for the first k items will be described as follows: there are two main cases:

### case: wk>u. as the weight of item k is greater than the weight constraint that we are working with, item k cannot be included in the knapsack.

### Case: wk<u. if item k is included, the value will be equal to the value of item k plus the maximum value of k-1 previous items, subject to a total weight of u-wk , since we have to include wk without exceeding the weight limit u. this is equivalent to S[k-1, u-wk] + vk. this leave us two sub-cases.

### if this total is less than the maximum for the first *k-1* values, we will not included item *k*. the new value will be *S[k-1,u]*.

### if it is larger, then we will include item *k*, and *S[k-1,u-wk] + vk* will be the new value.

### Thus, the recursive definition for the optimum is given as follows:

*S[k-1,u]* if *wk > u*

*S[k,u]* =

*Max{S[k-1,u],S[k-1,u-wk]+vk}* otherwise

{

### 

### We can store the sub-problems in a two-dimensional array, with first subscript being the item that we choose from 1 to *n*, and the second being the weight that we fill from 0 to *W*. for any given *k*, we can find all of the values of *S[k,u]* for *u*=0 to *W*.

### Algorithm :

**Algorithm Knapsack(w[1…n],v[1…n],W)**

**{**

***For u=0 to W***

S[0,u] = 0

***End for***

***For i=0 to n***

S[i,0] = 0

***End for***

***For i=0 to n***

***For u=0 to W***

If (w[i] < u) then

If(v[i]+S[i-1,u-w[i]]>s[i-1,u]) then

S[i,u] = v[i]+S[i-1,u-w[i]]

Else

S[i,u] = s[i-1,u]

Else

S[i,u] = s[i-1,u]

***End for***

***End for***

Print S[n][w]

i=n,k=W

***while*** (i>0 and k>0)

if(S[i][k] != S[i-1][k])

print ***i***

k = k – w[i]

end if

i = i - 1

***end while***

**}**

**Example :** Let us trace this algorithm through the following example. Capacity of knapsack is 5.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **item(i)** | **1** | **2** | **3** | **4** |
| **wi** | 2 | 3 | 4 | 5 |
| **vi** | 4 | 8 | 9 | 11 |

After the initialization step of the first row and first column, the matrix S will be completed as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Weight(*u*) | | | | | |
| Item(i) | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** |
| 1 | ***0*** |  |  |  |  |  |
| 2 | ***0*** |  |  |  |  |  |
| 3 | ***0*** |  |  |  |  |  |
| 4 | ***0*** |  |  |  |  |  |

*For* ***i=1***, the computations for inner loop (***u = 0,1,2,3,4,5***) are as follows:

w[1] = 2, v[1] = 4.

**u=0,**  *w[1]>u, S[1,0] = S[0,0] = 0.*

**u=1,** *w[1]>u, S[1,1] = S[0,1] = 0.*

**u=2,** *w[1]=u* ***and*** *(v[1]+s[0,0])>S[0,2], S[1,2] = v[1] + S[0,0] = 4+0 = 4*

**u=3,** *w[1]=u* ***and*** *(v[1]+s[0,1])>S[0,3], S[1,3] = v[1] + S[0,1] = 4+0 = 4*

**u=4,** *w[1]<u* ***and*** *(v[1]+s[0,2])>S[0,4], S[1,4] = v[1] + S[0,2] = 4+0 = 4*

**u=5,** *w[1]<u* ***and*** *(v[1]+s[0,3])>S[0,5], S[1,5] = v[1] + S[0,3] = 4+0 = 4*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Weight(*u*) | | | | | |
| Item(i) | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** |
| 1 | ***0*** | 0 | 4 | 4 | 4 | 4 |
| 2 | ***0*** |  |  |  |  |  |
| 3 | ***0*** |  |  |  |  |  |
| 4 | ***0*** |  |  |  |  |  |

*For* ***i=2***, the computations for inner loop (***u = 0,1,2,3,4,5***) are as follows:

w[2] = 3, v[2] = 8.

**u=0,**  *w[2]>u, S[2,0] = S[1,0] = 0.*

**u=1,** *w[2]>u, S[2,1] = S[1,1] = 0.*

**u=2,** *w[2]>u, S[2,2] = S[1,2] = 4.*

**u=3,** *w[2]=u* ***and*** *(v[2]+s[1,0])>S[1,3], S[2,3] = v[2] + S[1,0] = 8+0 = 8*

**u=4,** *w[2]<u* ***and*** *(v[2]+s[1,1])>S[1,4], S[2,4] = v[2] + S[1,1] = 8+0 = 8*

**u=5,** *w[2]<u* ***and*** *(v[2]+s[1,2])>S[1,5], S[2,5] = v[2] + S[1,2] = 8+4 = 12*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Weight(*u*) | | | | | |
| Item(i) | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** |
| 1 | ***0*** | 0 | 4 | 4 | 4 | 4 |
| 2 | ***0*** | 0 | 4 | 8 | 8 | 12 |
| 3 | ***0*** |  |  |  |  |  |
| 4 | ***0*** |  |  |  |  |  |

*For* ***i=3***, the computations for inner loop (***u = 0,1,2,3,4,5***) are as follows:

w[3] = 4, v[3] = 9.

**u=0,**  *w[3]>u, S[3,0] = S[2,0] = 0.*

**u=1,** *w[3]>u, S[3,1] = S[2,1] = 0.*

**u=2,** *w[3]>u, S[3,2] = S[2,2] = 4.*

**u=3,** *w[3]>u, S[3,3] = S[2,3] = 8.*

**u=4,** *w[3]=u* ***and*** *(v[3]+S[2,0])>S[2,4], S[3,4] = v[3] + S[2,0] = 9+0 = 9*

**u=5,** *w[3]<u* ***and*** *(v[3]+S[2,1])<S[2,5], S[3,5] = S[2,5] = 12*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Weight(*u*) | | | | | |
| Item(i) | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** |
| 1 | ***0*** | 0 | 4 | 4 | 4 | 4 |
| 2 | ***0*** | 0 | 4 | 8 | 8 | 12 |
| 3 | ***0*** | 0 | 4 | 8 | 9 | 12 |
| 4 | ***0*** |  |  |  |  |  |

*For* ***i=4***, the computations for inner loop (***u = 0,1,2,3,4,5***) are as follows:

w[4] = 5, v[4] = 11.

**u=0,**  *w[4]>u, S[4,0] = S[3,0] = 0.*

**u=1,** *w[4]>u, S[4,1] = S[3,1] = 0.*

**u=2,** *w[4]>u, S[4,2] = S[3,2] = 4.*

**u=3,** *w[4]>u, S[4,3] = S[3,3] = 8.*

**u=4,** *w[4]>u, S[4,4] = S[3,4] = 9.*

**u=5,** *w[4]=u* ***and*** *(v[4]+S[3,0])<S[3,5], S[4,5] = S[3,5] = 12*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Weight(*u*) | | | | | |
| Item(i) | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** | ***0*** |
| 1 | ***0*** | 0 | 4 | 4 | 4 | 4 |
| 2 | ***0*** | 0 | 4 | 8 | 8 | 12 |
| 3 | ***0*** | 0 | 4 | 8 | 9 | 12 |
| 4 | ***0*** | 0 | 4 | 8 | 9 | 12 |

We have essentially found the maximum value that can be places in the knapsack for any weight *u*, using all of the items up to *i*. to find the solution for the whole problem, we look at *S[n,W]* . in this case, the maximum value, *S[4,5]* from the above table is *12*.

To reconstruct the solution, we note that if **S[i,k] ≠ S[i-1,k]**, this happened because we added item I to the knapsack to increase the value. Thus, we can backtrack for i=n, k=W, adding an item to the list every time we see that **S[i,k] ≠ S[i-1,k]**

For our example, we start with i=4 and k = 5. since S[4,5] = S[3,5] = S[2,5], we know that items 4 and 3 are not in the knapsack. How ever, when i=2, S[2,5] ≠ S[1,5], so item 2 is in the knapsack, and we update k to be 3. when i=1,S[1,3] ≠ S[0,3], so we add item 1, and we are done. The final knapsack contains items 1 and 2, for total weight of 5 and a value of **12**.

# 2.10 TRAVELING SALESPERSON PROBLEM:

*Let,*

* G = (V,E) be a directed graph with edge costs cij.
* The variable cij is 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 traveling sales person problem finds application in a variety of situations.

*Suppose*,

* we have to route a postal van to pick up mail from mail boxes located at n different sites.
* An n+1 vertex graph can be used to represent the situation.
* One vertex represents the post office from which the postal van starts and to which it must return.
* Edge<i,j> is assigned to cost equal to the distance from site i to site j.
* the route taken by the postal van is a tour, and we are interested in finding a tour of minimum length.
* For simplicity we consider that the tour starts and ends at vertex 1.
* Every tour consists of an edge <1, k> for some k € V-{1} and a path from vertex k to vertex 1.
* The path from vertex k to vertex 1 goes through each vertex in V-{1, k} exactly once.
* It is easy to see that if the tour is optimal, then the path from k to 1 must be shortest k to 1 path going through all vertices in V-{1, k}.
* hence, the principle of optimality holds.
* Let g(i,S) be the length of a 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 sales person tour.

j € S

**g(i,S) = min {Cij+g(j,.S-{j})}**

* From the principal of optimality it follows in general.

The g values can be obtained by using the above formula.

***Clearly***,

***g(i,φ) = Ci1****, hencewe can use the above given formula to obtain* ***g(i,S) for all S of size 1****. Then we can obtain* ***g(i,S) for S with |S| = 2****, and* ***so on****.*

*When* ***|S| < n-1****, the values of* ***i*** *and* ***S*** *for which* ***g(i,S)*** *is needed are such that* ***i ≠ 1, 1¢S, and i¢S****.*

**Example:**

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

Consider the directed graph and the edge lengths are given by matrix are as follows:

*Thus,*

g(2, *φ*) = C21 = 5

g(3, *φ*) = C31 = 6

g(4, *φ*) = C41 = 8

*using the given equation we compute g(i,S) with |S| = 2, i ≠ 1, 1¢S, and i¢S,*

g(2,{3}) = C23 + g(3, *φ*) = 15

g(2,{4}) = C24 + g(4, *φ*) = 18

g(3,{2}) = C32 + g(2, *φ*) = 18

g(3,{4}) = C34 + g(4, *φ*) = 20

g(4,{2}) = C42 + g(2, *φ*) = 13

g(4,{3}) = C43 + g(3, *φ*) = 15

*next we compute g(i,S) with |S| = 2, i ≠ 1, 1¢S, and i¢S.*

g(2,{3,4}) = min{ C23 + g(3,{4}) , C24 + g(4,{3}) } = 25

g(3,{2,4}) = min{ C32 + g(2,{4}) , C34 + g(4,{2}) } = 25

g(4,{2,3}) = min{ C42 + g(2,{3}) , C43 + g(3,{2}) } = 23

***finally****, we ontain,*

g(1,{2,3,4}) = min{C12+g(2,{3,4}), C13+g(3,{2,4}), C14+g(4,{2,3})}

= min{ 35, 40, 43 }

= 35.

***An optimal tour of the given graph has length “*35*”.***

* A tour of this length can be constructed if we retain with each g(i,S) the value of j that minimizes the right-hand side of the given equation.
* Let j(i,S) be this value. Then, j(1,{2,3,4}) = 2. thus the tour starts from 1 and goes to 2.
* The remaining tour can be obtained from g(2,{3,4}). So j(2,{3,4}) = 4. thus the next edge is <2,4>.
* The remaining tour is for g(4,{3}). So, j(4,{3}) = 3.
* *The optimal tour is* **1, 2, 4, 3, 1**.

**Matrix Chain Multiplication :**

***Informal problem statement***

Given n matrices M1,M2….Mn. what is the fastest way to compute the product M1.M2….Mn

***Issues***

* What algorithm to use for multiplying just 2 matrices?
* Matrix multiplication is associative; how do we parenthesize the product?

***Lets compute the first issue*** ***Multiplying 2 matrices***,

How to compute C=AB where,

A:pxq matrix (given)

B:qxr (given)

C:pxr (to be computed)

***The second issue concerns Associativity***

Let me explain this with specific example involving only 3 matrices.

So For example say n=3.

Now we need to compute the product M1.M2.M3. (M1 times M2 times M3).

There are two ways to do this:

* One possibility is to multiply M2 and M3 first and then multiply that with M1.

M1.(M2.M3)

* Another possibility is to multiply M1 and M2 first and then multiply it with M3

(M1.M2).M3

Which is better M1.(M2.M3) or (M1.M2).M3?

***Let us take a specific example***.

Let us assume that M1 is the 3x2 matrix, M2 is the 2 x 4 matrix and M3 is the 4x1 matrix.

As in the first case,

M1.(M2.M3)

Time[M4 = M2.M3] = 2x4x1 = 8

Time[Ans = M1.M4] = 3x2x1 = 6

Time[M1.(M2.M3)] = 8 + 6 = 14

As in the second case

(M1. M2).M3

Time[M5 = M1.M2] = 3x2x4 = 24

Time[Ans = M5.M3] = 3x4x1 = 12

Time[(M1.M2).M3] = 24 + 12 = 36.

*So clearly this first way is better(faster).*

The problem now is that we can take our product that we can parenthesize in M1.(M2.M3) this way or we can parenthasise in (M1. M2).M3 this way.

***Another Example***:

• Suppose I want to compute A1.A2.A3.A4 .

• Matrix Multiplication is ***associative***, so we can do the multiplication in several different orders.

• A1 is 10 by 100 matrix

• A2 is 100 by 5 matrix

• A3 is 5 by 50 matrix

• A4 is 50 by 1 matrix

• A1A2A3A4 is a 10 by 1 matrix

5 different orderings = 5 different parenthesizations

• (A1(A2(A3A4)))

• ((A1A2)(A3A4))

• (((A1A2)A3)A4)

• ((A1(A2A3))A4)

• (A1((A2A3)A4))

Each parenthesization is a different number of mults

Let Aij = Ai · · ·Aj

• A1 is 10 by 100 matrix, A2 is 100 by 5 matrix, A3 is 5 by 50 matrix, A4 is 50 by 1 matrix, A1A2A3A4 is a 10 by 1 matrix.

• (A1(A2(A3A4)))

– A34 = A3A4 , 250 mults, result is 5 by 1

– A24 = A2A34 , 500 mults, result is 100 by 1

– A14 = A1A24 , 1000 mults, result is 10 by 1

– Total is 1750

• ((A1A2)(A3A4))

– A12 = A1A2 , 5000 mults, result is 10 by 5

– A34 = A3A4 , 250 mults, result is 5 by 1

– A14 = A12A34) , 50 mults, result is 10 by 1

– Total is 5300

• (((A1A2)A3)A4)

– A12 = A1A2 , 5000 mults, result is 10 by 5

– A13 = A12A3 , 2500 mults, result is 10 by 50

– A14 = A13A4 , 500 mults, results is 10 by 1

* Total is 8000

• ((A1(A2A3))A4)

– A23 = A2A3 , 25000 mults, result is 100 by 50

– A13 = A1A23 , 50000 mults, result is 10 by 50

– A14 = A13A4 , 500 mults, results is 10 by

– Total is 75500

• (A1((A2A3)A4))

– A23 = A2A3 , 25000 mults, result is 100 by 50

– A24 = A23A4 , 5000 mults, result is 100 by 1

– A14 = A1A24 , 1000 mults, result is 10 by 1

– Total is 31000

Conclusion Order of operations makes a huge difference. How do we compute the minimum?

***One Approach is :***

Each parenthesization defines a set of n-1 matrix multiplications. We justneed to pick the parenthesization that corresponds to the best ordering.

Now the question is how many parenthesizations are there?

Let **P(n)** ne the number of ways to parenthesize n matrices.

{

**∑ P(k) P(n-k)** if n>2

K=1

n-1

**1** if n = 1

P(n) =

This recurrence is related to the Catalan numbers, and solves to **P(n) =** **Ω(4n/n3/2)**

**The final conclusion is*” trying all possible parenthesizations is a bad idea”.* Inorder to overcome that we use *Dynamic Programming.* The steps include are*,***

1. Characterize the structure of an optimal solution

2. Recursively define the value of an optimal solution

3. Compute the value of an optimal solution bottom-up

4. Construct an optimal solution from the computed information

Structure of an optimal solution If the outermost parenthesization is

((A1A2 · · ·Ai)(Ai+1 · · ·An))

then the optimal solution consists of solving A1i and Ai+1,n optimally and then combining the solutions.



***Algorithm :***

Matrix-Chain-Order(p)

{

n = length[p] -1

for i=1 to n

do m[i,i] = 0

for l=2 to n

do for i=1 to n-l+1

do j = i+l-1

m[i,j] = ∞

for k=i to j-1

do q = m[i,k] + m[k+1,j] + pi-1pkpj

if q< m[i,j]

then m[i,j] = q

s[i,j] = k

return ***m*** and ***s***

}

# 

# UNIT-3

# BASIC SEARCH AND TRAVERSAL TECHNIQUE

# 3.1 GRAPH

# DEFINING GRAPH:

A graphs g consists of a set V of vertices (nodes) and a set E of edges (arcs). We write G=(V,E). V is a finite and non-empty set of vertices. E is a set of pair of vertices; these pairs are called as edges . Therefore,

V(G).read as V of G, is a set of vertices and E(G),read as E of G is a set of edges.

An edge e=(v, w) is a pair of vertices v and w, and to be incident with v and w.

**A graph can be pictorially represented as follows,**

FIG: Graph G

We have numbered the graph as 1,2,3,4. Therefore, V(G)=(1,2,3,4) and

E(G) = {(1,2),(1,3),(1,4),(2,3),(2,4)}.

## 3.1.1 BASIC TERMINOLGIES OF GRAPH:

### 3.1.1.UNDIRECTED GRAPH:

An undirected graph is that in which, the pair of vertices representing the edges is unordered.

### 3.1.1.2DIRECTED GRAPH:

An directed graph is that in which, each edge is an ordered pair of vertices, (i.e.) each edge is represented by a directed pair. It is also referred to as digraph.

**DIRECTED GRAPH**

**3.1.1.3COMPLETE GRAPH:**

An n vertex undirected graph with exactly n(n-1)/2 edges is said to be complete graph. The graph G is said to be complete graph .

# 3.2 TECHNIQUES FOR GRAPHS:

* The fundamental problem concerning graphs is the reach-ability problem.
* In it simplest from it requires us to determine whether there exist a path in the given graph, G +(V,E) such that this path starts at vertex ‘v’ and ends at vertex ‘u’.
* A more general form is to determine for a given starting vertex v6 V all vertex ‘u’ such that there is a path from if it u.
* This problem can be solved by starting at vertex ‘v’ and systematically searching the graph ‘G’ for vertex that can be reached from ‘v’.
* We describe 2 search methods for this.
  + Breadth first Search and Traversal.
  + Depth first Search and Traversal.

## 3.3 BREADTH FIRST SEARCH AND TRAVERSAL:

**Breadth first search:**

In Breadth first search we start at vertex v and mark it as having been reached. The vertex v at this time is said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next. There are new unexplored vertices. Vertex v has now been explored. The newly visited vertices have not been explored and are put onto the end of the list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left. The list of unexplored vertices acts as a queue and can be represented using any of the standard queue representations.

* In Breadth First Search we start at a vertex ‘v’ and mark it as having been reached (visited).
* The vertex ‘v’ is at this time said to be unexplored.
* A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjust from it.
* All unvisited vertices adjust from ‘v’ are visited next. These are new unexplored vertices.
* Vertex ‘v’ has now been explored. The newly visit vertices have not been explored and are put on the end of a list of unexplored vertices.
* The first vertex on this list in the next to be explored. Exploration continues until no unexplored vertex is left.
* The list of unexplored vertices operates as a queue and can be represented using any of the start queue representation.

**ALGORITHM:**

Algorithm BPS (v)

// A breadth first search of ‘G’ is carried out.

// beginning at vertex-v; For any node i, visit.

// if ‘i’ has already been visited. The graph ‘v’

// and array visited [] are global; visited []

// initialized to zero.

{ y=v; // q is a queue of unexplored 1visited (v)= 1

repeat

{ for all vertices ‘w’ adjacent from u do

{ if (visited[w]=0) then

{Add w to q;

visited[w]=1

}

}

if q is empty then return;// No delete u from q;

} until (false)

}

algrothim : breadth first traversal

algorithm BFT(G,n)

{

for i= 1 to n do

visited[i] =0;

for i =1 to n do

if (visited[i]=0)then BFS(i)

}

here the time and space required by BFT on an n-vertex e-edge graph one O(n+e) and O(n) resp if adjacency list is used.if adjancey matrix is used then the bounds are O(n2) and O(n) resp

## 3.4 Depth first search

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex u begins. When this new vertex has been explored, the exploration of u continues. The search terminates when all reached vertices have been fully explored. This search process is best-described recursively.

Algorithm DFS(v)

{

visited[v]=1

for each vertex w adjacent from v do

{

If (visited[w]=0)then

DFS(w);

}

}

# BACKTRACKING(PART-B)

# 3.5 GENERAL METHOD:

* It is one of the most general algorithm design techniques.
* Many problems which deal with searching for a set of solutions or for a optimal solution satisfying some constraints can be solved using the backtracking formulation.
* To apply backtracking method, tne desired solution must be expressible as an n-tuple (x1…xn) where xi is chosen from some finite set Si.
* The problem is to find a vector, which maximizes or minimizes a criterion function P(x1….xn).
* The major advantage of this method is, once we know that a partial vector (x1,…xi) will not lead to an optimal solution that (mi+1………..mn) possible test vectors may be ignored entirely.
* Many problems solved using backtracking require that all the solutions satisfy a complex set of constraints.
* These constraints are classified as:

i) Explicit constraints.

ii) Implicit constraints.

* **Explicit constraints:**

Explicit constraints are rules that restrict each Xi to take values only from a given set.

Some examples are,

Xi0 or Si = {all non-negative real nos.}

Xi =0 or 1 or Si={0,1}.

LiXiUi or Si= {a: LiaUi}

* All tupules that satisfy the explicit constraint define a possible solution space for I.
* **Implicit constraints:**

The implicit constraint determines which of the tuples in the solution space I can actually satisfy the criterion functions.

**Algorithm:**

Algorithm IBacktracking (n)

// This schema describes the backtracking procedure .All solutions are generated in X[1:n]

//and printed as soon as they are determined.

{

k=1;

While (k0) do

{

if (there remains all untried

X[k]  T (X[1],[2],…..X[k-1]) and Bk (X[1],…..X[k])) is true ) then

{

if(X[1],……X[k] )is the path to the answer node)

Then write(X[1:k]);

k=k+1; //consider the next step.

}

else k=k-1; //consider backtracking to the previous set.

}

}

* All solutions are generated in X[1:n] and printed as soon as they are determined.
* T(X[1]…..X[k-1]) is all possible values of X[k] gives that X[1],……..X[k-1] have already been chosen.
* Bk(X[1]………X[k]) is a boundary function which determines the elements of X[k] which satisfies the implicit constraint.

# 3.6 8(OR) N-QUEENS PROBLEM

The n-queens problem consists of placing n queens on an n-by-n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. This is a classical combinatorial problem. Figure 6.3 illustrates n-queens problem for n=1,2,3.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | 1 | | 1 | Q | | *n = 1* | | | |  |  |  | | --- | --- | --- | |  | 1 | 2 | | 1 |  |  | | 2 |  |  | | *n=2* | | | | |  |  |  |  | | --- | --- | --- | --- | |  | 1 | 2 | 3 | | 1 |  |  |  | | 2 |  |  |  | | 3 |  |  |  | | n=3 | | | | |
| **Trivial Solution** | **No Solution** | **No Solution** |

Let us consider the 4-queens problem and solve it by the backtracking technique. Each queen has to be placed in its own row; all we need to do assign a column for each queen on the following boarD

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 |  |
| 1 |  |  |  |  | <- Queen 1 |
| 2 |  |  |  |  | <- Queen 2 |
| 3 |  |  |  |  | <- Queen 3 |
| 4 |  |  |  |  | <- Queen 4 |

The 4 by 4 board(the following state space tree) shows a solution for n=4.

All nodes(boards) are numbered from 0 to 16. The numbers within parentheses indicate columns of the board. For example in board # 4, three queens are placed in columns 1,4 and 2. Let us place 4th queen in column 3. If we place 4th queen in wither row 1, 2, or 3, this will attack other three queens(already placed queens) row-wise. Also, 4th row does not permit, because it arracks diagonally queen at (3,1). Therefore, there is no solution in this branch of tree (nodes: 1, 3, 4); it backtracks to node 5 and search for alternate solution. Now, we have two solutions: board#8 and 12.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Q** |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Q** |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **Q** |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **Q** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Q** |  |  |
|  |  |  | **Q** |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **Q** |  |
| **Q** |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Q** |  |  |  |
|  |  | **Q** |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Q** |  |  |  |
|  |  |  | **Q** |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **Q** |
| **Q** |  |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **Q** |
|  | **Q** |  |  |
|  |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Q** |  |  |
|  |  |  | **Q** |
| **Q** |  |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **Q** |  |
| **Q** |  |  |  |
|  |  |  | **Q** |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Q** |  |  |
|  |  |  | **Q** |
| **Q** |  |  |  |
|  |  | **Q** |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **Q** |  |
| **Q** |  |  |  |
|  |  |  | **Q** |
|  | **Q** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Q** |  |  |  |
|  |  |  | **Q** |
|  | **Q** |  |  |
|  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **Q** |
| **Q** |  |  |  |
|  |  | **Q** |  |
|  |  |  |  |

**(Φ)**

**(1)**

**(2)**

**(3)**

**(4)**

**(1,3)**

**(1,4)**

**(2,4)**

**(3,1)**

**(4,1)**

**(4,2)**

**(1,4,2)**

**(2,4,1)**

**(3,1,4)**

**(4,1,3)**

**(2,4,1,3)**

**(3,1,4,2)**

*A complete backtrack search tree for 4-Queens problem*

We will introduce backtracking algorithm which can be used to construct all solutions for a given n.

* We could continue with the 4-queens example to obtain all possible solutions for n=4.
* Our goal now is to convert this approach into an explicit algorithm.
* We track the position of the queens by using an array row.
* Row[k] is the row in column k containing a queen.
* The main algorithm is the recursive algorithm rn\_queens().
* When rn\_queens(k,n) is called, queens have been successfully placed in columns 1 to k-1.
* The algorithm then attempts to place a queen in column k.
* If it is successful, then

- if k=n, it prints the solution.

- if k<n, it makes the call rn\_queens(k+1,n).

- it then returns to the call rn\_queens(k-1,n).

* To test for a valid position for a queen in column k, we use an algorithm position\_ok(k,n) which returns true if and only if the queen tentatively placed in column k does not conflict with the queens in positions 1 to k-1.
* The queens in columns I and k conflict if

- they are in the same row: row[i]=row[k].

- or in the same diagonal: absolute(row[k]-row[i])=k-i.

* The algorithm is invoked with rn\_queens(1,n).

***Algorithm 6.3(a):rn\_queens(k,n)***

1. for row[k]=1 to n
2. if position\_ok(k,n)=true, then
3. if k=n, then
4. print solution
5. else
6. rn\_queens(k+1,n)
7. endif
8. endif
9. endfor.

***Algorithm 6.3(b):position\_ok(k,n)***

1. for i=1 to k-1
2. if(row[k]=row[i] or abs(row[k]-row[i])=k-i), then
3. return false
4. else
5. return true
6. endif
7. endfor.

***Time Complexity***:

We will obtain an upper bound for the running time of the algorithm by bounding the number of times rn\_queens(k,n) is called for each k<n. There are(n-1)…(n-k+2) ways to place queens in the first k-1 columns in distinct rows. Ignoring recursive calls, rn\_queens(k,n) executes in Ө(n) time for k<n, there is at most one placement possible for the queen. Also, the loop in rn\_queens executes in Ө(n) time. There are n(n-1)…2 ways for the queens jto have been placed in the first n-1 columns, so the worst-case running time for rn\_queens(n,m) is:

n[n(n-1)…2]=nxn!

The algorithm runs in O(nxn!).

# 3.7 SUM OF SUBSETS PROBLEM:

The ***sum of subsets*** problem consists of finding a subset of a given set X = {x1,x2…xn} of *n* distinct positive integers and a positive integer ***S***. find all subsets of {x1,x2…xn} that sum to ***S.***

***For Example****,*

If ***X*** = {1,2,3,4,5,6} and ***S=*** 12, there will be more than one subset whose sum is 12. the subsets are {1,2,3,6}, {1,2,4,5}, {1,5,6}, {2,4,6}, {3,4,5}.

* We will assume a binary state space tree.
* The nodes at depth 1 are for including (yes = 1, no = 0) item 1, the nodes at depth 2 are for item 2, etc. The left branch includes xi, and the right branch excludes xi. The nodes contain the sum of the numbers included so far.
* Backtracking consists of doing a DFS of the state space tree, checking whether each node is promising and if the node is non-promising backtracking to the node’s parent.
* We call a node non-promising if it cannot lead to a feasible (or optimal) solution, otherwise it is promising.
* The state space tree consisting of expanded nodes only is called the pruned state space tree. The next coming Example shows the pruned state space tree for the sum of subsets problem.

Consider a node at depth I,

sumSoFar = sum of node –that is, sum of numbers included in partial solution node.

sumLeft = sum of the remaining items i+1 to n(for a node at depth i).

A node at depth I is non-promising:

If ((sumSoFar + sumLeft < S) or (sumSoFar + x[i+1] > S))

To be able to use this “promising function” the xi must be sorted in increasing order.

The *include[1..n]* is a Boolean array. If its value is true, then the node is included in the partial tree, else not included.

The following Algorithm: *sumOfSubsets(i, sumSoFar, sumLeft)* is invoked with:

i = 0, sumSoFar = 0, and sumLeft = x1+x2+…..+xn.

The algorithm is called with *sumOfSubsets(0, 0, 21)* for our next coming Example.

*Algorithm: sumOfSubsets(i, sumSoFar, sumLeft)*

1. If promising(i, sumSoFar, sumLeft), then // may lead to solution
2. If(sumSoFar, = S), then
3. print solution
4. Else //expand the node when sumSoFar < S
5. include[i+1] = true //include x[i+1]
6. sumOfSubsets(i+1, sumSoFar + x[i+1], sumLeft – x[i-1])
7. include[i+1] = false //exclude x[i+1]
8. sumOfSubsets(i+1, sumSoFar , sumLeft – x[i+1])
9. Endif
10. Endif.

*Algorithm: promising (i, sumSoFar, sumLeft)*

1. If(sumSoFar + sumLeft ≥ S) and
2. (sumSoFar = S or sumSoFar + x[i+1] ≤ S), then
3. Return true
4. Else
5. Return false
6. Endif.

*Example:* X = {3,5,6,7} and S = 15. There are only 15 nodes in the pruned state space tree

In the following fig. The full state space tree has 31 nodes (24+1 – 1). Trace of the Algorithm is given in ***Table* ,** next to the state space tree. Nodes of the tree are shown by circled numbers. Notice that the x1 is included at level 1 of the tree, x2 at level 2, and son on. We have only one solution with subset {3,5,7}. This occurs at node 6 in the following Diagram.

**0**

**0**

**3**

**8**

**3**

**5**

**0**

**5**

**11**

**8**

**14**

**3**

**9**

**8**

**15**

***3***

***5***

***6***

***0***

***7***

***0***

***0***

***0***

***6***

***0***

***6***

***0***

***5***

***0***

*Pruned State Space Tree - Example*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***Trace of sumOfSubsets – Example*** | | | | |
| *i* | *node* | *sumSoFar* | *sumLeft* | *Comment* |
| **0** | **1** | **0** | **21** | **Start** |
| **1** | **2** | **3** | **18** | *Include 3* |
| **2** | **3** | **8** | **13** | *Include 5* |
| **3** | **4** | **14** | **7** | *Include 6* |
| **3** | **5** | **8** | **7** | *Exclude 6* |
| **4** | **6** | **15** | **0** | *Include 7* |
|  |  |  | **3 5 7** | **<- Solution** |
| **4** | **7** | **8** | **0** | *Exclude 7* |
| **2** | **8** | **3** | **13** | *Exclude 5* |
| **3** | **9** | **9** | **7** | *Include 6* |
| **3** | **10** | **3** | **7** | *Exclude 6* |
| **1** | **11** | **0** | **18** | *Exclude 3* |
| **2** | **12** | **5** | **13** | *Include 5* |
| **3** | **13** | **11** | **7** | *Include 6* |
| **3** | **14** | **5** | **7** | *Exclude 6* |
| **2** | **15** | **0** | **13** | *Exclude 5* |

# 3.8 GRAPH COLORING:

* Graph coloring is a way of coloring the vertices of a graph such that no two adjacent vertices share the same color.
* The convention of using colors originates from coloring the countries of a map, where each face is literally colored.
* This was generalized to coloring the faces of a graph embedded in the plane. By planar duality(x and y coordinates) it became coloring the vertices, and in this form it generalizes to all graphs.

*Consider* an undirected graph G=(V,E) and an integer k. Assign one of k colors to each node, such that adjacent nodes get different colors.

For example, let G=(V,E) where V={1,2,3,4} and E={(1,2),(1,3),(2,3),(2,4),(3,4)} and suppose that k=3 (Red,Green,Blue). A valid coloring c of G is: c(1) = R, c(2) = G, c(3) =B, C(4) = R.

**Graph Coloring**

**R**

**R**

**G**

**B**

Suppose that |V|=n. Then,

* (c1,c2,……cn) is a possible coloring of graph where ci is the color of i-node in the graph.
* Note that there are kn possible colorings.
* A coloring is feasible or valid if no two adjacent nodes are given the same color, that is, if(i,j) ε E then ci ≠ cj.

Consider a graph shown above with k=3. There are six valid colorings of graph given in the table as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 6 valid colorings | | | | | | |
| *mode* | *(i)* | *(ii)* | *(iii)* | *(iv)* | *(v)* | *(vi)* |
| 1 | **R** | **R** | **G** | **G** | **B** | **B** |
| 2 | **G** | **B** | **B** | **R** | **R** | **G** |
| 3 | **B** | **G** | **R** | **B** | **G** | **R** |
| 4 | **R** | **R** | **G** | **G** | **B** | **B** |

Note that all these colorings are sort of equivalent. They all share the following structure:

* The same color is used for both node 1 and node 4. For colorings (i) and (ii), it is R, for colorings (iii) and (iv), it is G, and for colorings (v) and (vi), it is B.
* Nodes 2 and 3 must have distinct colors different from each other and from the color used for nodes 1 and 4.

By this observation, it follows that two colorings are equivalent if one can be transformed into another by permuting the k colors.

We will use the following strategy to find all valid colorings of a graph:

* Order nodes arbitrarily
* Assign the first node a color.
* Given a partial assignment of colors (c1,c2,……ci-1) to the first i-1 nodes, try to find a color for the i-th node in the graph.
* If there is no possible color for the i-th node given the previous choices, backtrack to a previous solution.

Consider a graph shown in figure 6.7 with k=3.

*Step 1:* Choose a color for node 1. It can be one of: R, B or G. Say we choose R.

*Step 2:* Given partial coloring (R), we choose a color for node 2. It can be one of : G or B. Say we choose G.

*Step 3:* Given partial coloring (R , G), we choose a color for node 3. It cannot be either R or G, so it must be B since k=3.

*Step 4:* Given partial coloring (R ,G, B), we choose a color for node 4. It cannot be B or G, so it must be this gives the coloring (R,G,B,R) which is coloring (i).

We have no more choices of colors in step 4, and in step 3. We have one choice in step 2.

*Step 2:* Given partial coloring (R), we choose a different color for node 2. we choose B for node 2.

*Step 3:* Given partial coloring (R,B), we choose a color for node 3. It cannot be R or B, so it must be G.

*Step 4:* Given partial coloring (R,G,B), we choose a color for node 4. It cannot be B or G, so it must be R. This gives the coloring (R,G,B,R) which is coloring (ii).

We have no more choices of colors in steps 4,3 and 2. We go back to step 1.

*Step 1:* We choose a different color for node 1, say B. This will produce a branch in the tree equivalent to the first branch where R and B are switched. Thus, we will get the colorings (v) and (vi).

If we choose G for node 1 then, we will again get a branch equivalent to the first one with R and G swapped. This will produce the colorings (iii) and (iv).

**R**

**R**

**R**

**R**

**R**

**R**

**R**

**R**

**R**

**R**

**G**

**G**

**G**

**G**

**G**

**G**

**G**

**G**

**G**

**G**

**B**

**B**

**B**

**B**

**B**

**B**

**B**

**B**

**B**

**B**

***Graph Coloring State Space Tree***

The following Algorithm solves the graph coloring problem. It generates one solution (only).

C [1....j-1]: a partial coloring for the first j-1 nodes of the graph.

adj [1..n][1..n]: adjacency matrix for n-node graph.

k: Number of colors(labels).

We invoke the algorithm: Coloring(C,j,k,n), where j is the first node(j = 1).

***Algorithm: Coloring (C[1..n],j,k,n)***

1. If j=n+1, then
2. print C
3. Endif
4. For i=1 to k
5. C[j] = i
6. If Valid(C,j,n) is true, then
7. Coloring(C,j+1,k,n)
8. Endif
9. Endfor.

***Algorithm 6.5(b): Valid(C[1..n], j, n)***

1. For i=1 to n //for all adjacent nodes v of j with v<j

2. If adj[j][i] ≠0, then

3. v = i

4. If C[v] = C[j], then

5. Return false

6. Endif

7. Endif

8. Endfor

9. Return true.

We now trace the Coloring algorithm by considering 4 node graph in the following fig.

**Graph Coloring**

**R**

**R**

**G**

**B**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** |
| **1** | 0 | 1 | 1 | 0 |
| **2** | 1 | 0 | 1 | 1 |
| **3** | 1 | 1 | 0 | 1 |
| **4** | 0 | 1 | 1 | 0 |

Initially, the algorithm: (C,j,k,n) is called with c = {0,0,0,0},j=1,k=3, and n=4. After first recursive call, C={1,0,0,0} – that is, the node 1 is labeled with 1(red). After second call, C={1,2,0,0} – the node 2 is labeled with 2 (Green). After third call, C={1,2,3,0} – the node 3 is labeled with 3(Blue). After fourth call, C={1,2,3,1} – the node 4 is labeled with 1(Red). It prints the labels 1,2,3,1.

After this, the algorithm backtracks, checks two more branches and terminates. The valid solution is : node 1(label 1), node 2(label 2), node 3(label 3), node 4(label 1). This is one of the six valid colorings shown in Column(i) of Colorings table above.

The number of internal nodes in the state space tree is given by: ∑ ki (where 0<i<n-1). Valid routine takes O(*kn*) time at each internal node to find the children corresponding to valid colorings. Hence,

**∑ ki n = n(k n+1 - 2) / (k-1) = O(nkn).** (*where 1 < i < n*)

# 3.8 HAMILTONIAN CYCLES:

A Hamiltonian cycle in a connected graph is a cycle that passes through every vertex exactly once. The graph in Figure 6.10(a) contains the Hamiltonian cycle A,B,C,E,F,D,A. The graph in the following Figure contains no Hamiltonian cycle.

***UnDirected Graphs***

There is no known easy way to determine whether a given graph contains a Hamiltonian cycle. We now look at a backtracking algorithm that finds the Hamiltonian cycle in a graph. The graph may be directed or undirected. Let us apply the backtracking technique to find the Hamiltonian cycle in the following graph (Figure 6.11(a)), and obtain a Hamiltonian cycle as shown in Figure 6.11(b).

***Illustration of Hamiltonian Cycles***

Example :

Find a Hamiltonian cycle for the directed graph.

***Directed Graph and its Adjacency List***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **N** |  | **1** |  | **2** |  | **3** |
| **1** | **→** | **2** |  |  |  |  |
| **2** | **→** | **3** | **→** | **4** | **→** | **6** |
| **3** | **→** | **4** |  |  |  |  |
| **4** | **→** | **1** | **→** | **7** |  |  |
| **5** | **→** | **3** |  |  |  |  |
| **6** | **→** | **1** | **→** | **4** | **→** | **7** |
| **7** | **→** | **5** |  |  |  |  |

Given a directed graph G = (V,E). We store the graph as an adjacency list (for each vertex v ε {1,2,…,n}, and store a list of the vertices w such that (v,w) ε E). We store a Hamiltonian cycle as A[1..n], where the cycle is:

A[n]→ A[n-1] → … → A[2] → A[1] → A[n]

We also set up an array D, the out-degree of each vertex. The values are tabulated as:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| D | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|  | 1 | 3 | 1 | 2 | 1 | 3 | 1 |

Pruning: We keep a Boolean array mark[1..n], where mark[i] is 1 if vertex i is visited and 0 if vertex is unvisited (prune here).

To solve the Hamiltonian cycle problem, we use arrays N (adjacency lists) and D (degree) for the input graph, and execute the Algorithm 6.6 (given in three parts). Assume that n ≥ 2.

*Algorithm : Initialization()*

1. For i = 1 to n-1
2. mark [i] =0
3. Endfor
4. mark [n] = 1; A[n] = n
5. Hamilton(n-1)

*Algorithm : Hamilton(k)*

1. If k = 0 , then
2. process(A)
3. Else
4. For j = 1 to D[A[k+1]]
5. w = N[A[k+1]][j]
6. If mark[w] = 0, then
7. mark[w] = 1; A[k] = w
8. Endif
9. Endfor
10. Hamilton(k-1)
11. mark[w]=0
12. Endif.

*Algorithm : Process(A)*

1. ok = 0
2. For j = 1 to D[A[1]]
3. If N[A[1],j] = A[n], then ok = 1
4. Endfor
5. If ok = 1, then Print(A).