Design Analysis and Algorithms

INTRODUCTION

Algorithm: An algorithm is a step by step procedure to represent the solution for a given problem. It consists of a sequence of steps representing a procedure defined in a simple language to solve the problem.

It was proposed by Persian mathematician, ABU JAFAR MOHAMMED IBN MUSA AL KHOWARIZMI IN 825 A.D.

Characteristics of an Algorithm:

1) An algorithm must have finite number of steps.
2) An algorithm must be simple and must not be ambiguous.
3) It must have zero or more inputs.
4) It must produce one or more outputs.
5) Each step must be clearly defined and must perform a specific function.
6) There must be relationship between every two steps.
7) It must terminate after a finite no. of steps.

How to develop an algorithm

1. The problem for which an algorithm is being devised has taken precisely and clearly defined.
2. Develop a mathematical model for the problem. In modeling mathematical structures that are best suited are selected.

3. Data structures and program structures used to develop a solution are planned.

4. The most common method of designing an algorithm is step wise refinement. Step wise refinement breaks the logic into series of steps. The process starts from converting the specifications of the module into an abstract description of an algorithm containing a few abstract statements.

5. Once algorithm is designed, its correctness should be verified. The most common procedure to correct an algorithm is to run the algorithm on varies of test cases. An ideal algorithm is characterized by its run time and space occupied.

How to analyze an algorithm:

Analysis of algorithm means estimating the efficiency of algorithm in terms of time and storage an algorithm requires.

1. First step is to determine what operations must be done and what are their relative cost. Basic operations such as addition, subtraction, comparison have constant time.

2. Second task is to determine number of data sets which cause algorithm to exhibit all possible patterns.

It requires to understand the best and worst behavior of algorithm for different data configurations.

The analysis of algorithm two phases,

1) Priori Analysis: In this analysis we find some functions which bounds algorithm time and space complexities. By the help of these functions it is possible to compare the efficiency of algorithms.

2) Posterior Analysis: Estimating actual time and space when an algorithm is executing is called posterior analysis.

Priori analysis depends only on number of inputs and operations done, where as posterior analysis depends on both machine and language used, which are not constant. Hence analysis of most of the algorithms is made through priori analysis.

How to validate an algorithm:

The purpose of the validation of an algorithm is to assure that this algorithm will work correctly independently of the issues such as machine, language, platform etc. If an algorithm is according to given specifications and is producing the desired outputs for a given input then we can say it is valid.

TIME & SPACE COMPLEXITIES

Space complexity:

It is defined as the amount of memory need to run to completion by an algorithm.

The memory space needed by an algorithm has two parts

1. One is fixed part, such as memory needed for local variables, constants, instructions etc.
2. Second one is variable part, such as space needed for reference variables, stack space etc. They depend upon the problem instance.

The space requirement S(P) of an algorithm P can be written as $S(P) = c + Sp$, where c is a constant(representing fixed part). Thus while analyzing an algorithm Sp (variable part) is estimated.

Time Complexity:

The time T(P) taken by a program is the sum of compile time and run time. The compile dos not depend upon the instance characteristics, hence it is fixed. Where as run time depends on the characteristics and it is variable. Thus while analyzing an algorithm run time Tp is estimated.

Order of magnitude:

It refers to the frequency of execution of a statement. The order of magnitude of an algorithm is sum of all frequencies of all statements. The running time of an algorithm increases with size of input and number of operations.

Ex1: In linear search, if the element to be found is in the first position, the basic operation done is one. If it is some where in array, basic operation (comparison) is done n times.

If in an algorithm, the basic operations are done same number of times every time then it is called 'Every time complexity analysis" T(n).

Ex1:

Addition of n numbers

Input : n

Alg: *for 1 to n*

Sum = sum+n;

T(n) = n;

Ex:2:

Matrix multiplication

Input : n

Alg: *for i = 1 to m*

for j = 1 to n

*multiplying a[i] * a[j]*

T(n) = n^2;

Worst case: w(n)

It is defined as maximum number of times that an algorithm will ever do.

Ex: Sequential search

Inputs: n

Alg: comparison

If x is last element, basic operations are done n times.

w(n) = n

Best case: B(n)

It is defined as minimum numer of times algorithm will do its basic operation.

Ex: Sequential search

If x is first element B(n) = 1

Average case: A(n)

It is average number of times algorithm does basic operations for n.

Ex: Sequential search

Inputs: n

<u>Case 1:</u>

If x is in array

Let probability of x to be in Kth slot = 1/n

If x is in Kth slot, no. of times basic operation done is K.

$$A(n) = \sum_{k=1}^{n}\left(KX\frac{1}{n}\right) = \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2}$$

<u>Case 2:</u>

If x is not in array

Probability that x is in Kth slot = p/n

Probability that x is not in array = 1-P

$$A(n) = \sum_{k=1}^{n}\left(KX\frac{p}{n}\right) = n(1\text{-}P) = \frac{P}{n}\left(\frac{n(n+1)}{2}\right) + n(1\text{-}P) = n\left(1-\frac{p}{2}\right)+\frac{p}{2}$$

Order

It is sum of all frequencies of execution of statements in a program.

1. Algorithm with time complexities of O(n) and 100n are called linear time algorithms since time complexity is linear with input size.
2. Algorithm such as $O(n^2)$, $0.01n^2$ are called quadratic time algorithms.
3. Any linear time algorithm is efficient than quadratic time algorithm.

Set of all complexity functions that can be classified with pure quadratic functions is called $\theta(n^2)$.

θ (n^2) is called quadratic time algorithm.

In priori analysis, all factors regarding machine and long are ignored and only inputs and no. of operations are taken into account. For this purpose O-notation can be used.

Big O:

O(f(n)): For a function f(n), O(f(n)) is the set of complexity functions g(n) for which there exist a constant C and N such that for all $n \geq N$

$G(n) \leq C*f(n)$

Where g(n) is computing time

n-number of inputs

O(f(n)) – algorithm complex time

If $g(n) \in O(f(n))$ then g(n) is Big O(f(n)). It puts an asymptotic upper bound on a function.

Ex: Let $g(n) = n^2 + 10n$

$F(n) = n^2$

$G(n) \in O(f(n))$

When C=2 and N=10

For c=2

$G(n) = n^2+10n$ *and* $c\,f(n) = 2n^2$

Time complexity of $g(n) \geq c\,f(n)$

For $n \leq 10$

If $n>10$ *then* $g(n) \leq f(n)$

Thus N = 10 keeps an upper bound complexity of a 40 function.

If algorithm is run on same complexity on same type of data, but with higher magnitude of n, the resulting time is less than some constant time f(n).

Thus

$O(1) \leq O(\log n) \leq O(n) \leq O(n\log n) \leq O(n^2) \leq O(2^n)$ *for a given N.*

O(f(n)) – stands for a quantity that is not explicitly known.

Every appearance of O(f(n)) means, there are positive constants C and N such that there occurs a number x_n represented by O(f(x)) such that for all n≥N

$X_n \leq cf(n)$

Ex: $1^2+2^2+3^2+\text{----------}n^2 = \frac{1}{3}n(n+\frac{1}{2})(n+1)$

$= \frac{1}{3}n^3+\frac{1}{2}n^2+\frac{1}{6}n$

so

$1^2+2^2+3^2+\text{----------}n^2 = O(n^4)$ *may be*

$1^2+2^2+3^2+\text{----------}n^2 = O(n^3)$ *strong*

$1^2+2^2+3^2+\text{----------}n^3 = \frac{1}{3}n^3+O(n^2)$ *strong*

Theorem:

If $P(n) = a_0+a_1n+\text{----------}a_mn^m$ show that $P(n) = O(n^m)$

Proof:

$P(n) \leq |a_0| +|a_1|n+\text{----------}|a_m|n^m$

$\leq \left(|a_0|/n+|a_1|/n^{m-1}+|a_m|\right) n^m$

$\leq |a_0|+|a_1|n+\text{----------}|a_m|n^m$

when n≥1

let $C= |a_0|+|a_1|+\text{----------}|a_m|$

$P(n) \leq Cn^m$

$P(n) = O(n^m)$

The symbol O(f(n)) stands for set of all functions g of integers such that

$g(n) \leq cf(n)$

Contest of O-notation identifies the variable that is involved and the range of variable.

Ex: Show that $n^2 + 10n \in O(n^2)$

C = 11 for N=1

$n^2+10n \in O(n^2)$

Big Ω

Ωf(n)): It is the set of complexity functions g(n) for which there exists C and N for all $n \geq N$ so that $g(n) \geq cf(n)$. Always it represents the lower boundary.

Big θ

$\theta(f(n)) = O(f(n)) \cap \Omega (f(x))$

It is the set of complexity functions g(n) for which there exists some +ve real constants c, d and N such that

$c(f(x) \leq g(n) \leq d\ f(x)$

Ex: $T(n) = \frac{n(n-1)}{2} = O(n^2) = \Omega (n^2)$

$= \theta(n^2)$

If $g(n) = \theta(f(n))$ then $f(n)$ is both upper and lower bounds.

Small O: O(f(n))

$g(n) \sim O(f(n))$

if exact computing time of g(n) is known we can find f(n), such that g is asymptotic to f.

Ex: $f(n) = a_k n^k$+ --------a_0

$= O(n^k)$

and $f(n) \sim O(a_k n^k)$

Divide and Conquer

Introduction:

It is similar to top down approach but, TD approach is a design methodology (procedure) where as DAC is a plan or policy to solve the given problem.

TD approach deals with only dividing the program into modules where as DAC divides the size of input.

It splits 'n' inputs into K distinct sets, yielding k sub problems. These sub problems must be solved and sub solutions are combined to form solution.

Control Abstraction:

```
Procedure DAC(p,q)
{
    If RANGE (p,q) is small
    Then return G(p,q)
Else
{ m=DIVIDE(p,q)
return (COMBINE (DAC(p,m), DAC(m+1,q))
}}
```

Step1: If input size (range) (q-p+1) is small, find solution G(p,q)

Step2: If input size is large split range into two data sets (p,m) and (m+1, q) and repeat.

Step3: Repeat the process until sub problem is small enough to solve.

Time Complexity:

$$T(n) = \begin{cases} g(n) & \text{n is small} \\ 2T(n/2) + f(n) & \text{Otherwise} \end{cases}$$

f(n) is time taken to divide and combine

g(n) is time taken for conquer

let $n = 2^m$

$$T(n) = 2T\left(\frac{2^m}{2}\right) + f(2^m)$$

$$\frac{T(2^m)}{2^m} = \frac{T(2^{m-1})}{2^m} + \frac{f(2^m)}{2^m}$$

let k=1

$$= \frac{T(2^{m-1})}{2^{m-1}} + \frac{k(2^m)}{2^m}$$

$$\frac{T(2^m)}{2^m} = \frac{T(2^{m-1})}{2^{m-1}} + 1$$

$$= \frac{T(2^{m-1})}{2^{m-1}} + 1 + 1$$

$$= \frac{T(2^{m-m})}{2^{m-m}} + m$$

$$= T(1) + m$$

$$= m+1$$

$$T(2^m) = 2^m (m+1)$$

$$T(n) = n(\log n + 1)$$

$$= O(n \log n)$$

Ex: Find time complexity for DAC

a) *If g(n) = O(1) and f(n) = O(n)*

b) *If g(n) = O(1) and f(n) = O(1)*

a) *It is the same as above case*

i.e., $O(n) = k.2^m = f(n)$

$T(n) = n\log n$

b) $T(n) = 2T(n/2) + f(n)$

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

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

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

$= 2^kT(n/2^k)+2^{k-1}+\text{---------}+2+1$

$= nT(1)+\sum_{i=0}^{k}2^{i-1}$

$= n+\sum_{i=0}^{k}2^{i-1}-2^{-1}$

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

$=2n-3/2$

$T(n) = O(n)$

Binary Search:

Problem:

To search whether an element x is present in the given list. If x is present, the value j must be displayed such that $a_j=x$

Algorithm:

Step1: DAC suggests break the input 'n' elements I = (a_1, a_2, ---------a_n) into

I_1 = (a_1, a_2, ---------a_{k-1}), I_2= a_k and

I_3 = (a_{k+1}, a_{k+2}, ---------a_n) into data sets.

Step 2: If x=a_k no need of searching I_1, I_3.

Step 3: If x<a_k only I_1 is searched

Step 4: If x<a_k only I_3 is searched

Step 5: Repeat the steps

K is chosen such that a_k is middle of 'n' elements k=n+1/2.

Control Abstraction:

```
Procedure BINSRCH(1,n)
    {     m = n+1/2
          if A(m) = x then
                {
                      loc = mid;
                      return;       }
    else
          if A(mid)>x then
                BINSRCH(1, m-1)
          Else
                BINSRCH(m+1, n)
          }
```

Time Complexity:

$$T(n) = \begin{cases} g(n) & n \text{ is } \le 1 \\ T(n/2) + g(n) & n > 1 \end{cases}$$

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

$= T\left(\frac{n/2}{2}\right) + 2g(n)$

$= T(n/2^2)+2g(n)$

$= T(n/2^k)+k\ g(n)$

if $n=2^k$

$= T(1)+k$

$= k+1 = O(\log n)$

it is valid if

n is $2^{k-1} \le n \le 2^k$

fig.

square nodes are at levels k and k+1, for unsuccessful search

$n=2^k$

$T(n) = O(\log n)$

Number of comparisons for successful search is k i.e., O(logn) and for unsuccessful k or k-1 O(logn)

Ex: Take instances *12 14 18 22 24 38*

1 2 3 4 5 6

key=24

mid=3 *low* *high* *mid*

1 *6* *3*

4 *6* *5*

thus number of comparisons needed are two.

Theorem:

Prove that E=I+2n, for a binary search tree.

E = Length of external nodes

I = Length internal nodes

n = Internal nodes

for level 2 for n=3

$E = 2^2.2=8$ E = I+2n

$I = 2^1.1=2$ 8=2+6=8

By induction if we add these branches to form a tree it satisfies E=I+2n.

Theorem:

Prove Average successful search time is equal to average unsuccessful search time U, if n $\square \propto$

$$lt_{n\to\infty} S_{avg} = lt_{n\to\infty} U_{avg}$$

$$U_{avg} = \frac{E}{n+1} \qquad S_{avg} = \frac{I}{n} + 1$$

$$= \frac{I+2n}{n+1} \cong \frac{I+2n}{n}$$

$$= \frac{I}{n} + 2 = \frac{I+2n}{n} = E_{avg}$$

$$U = \frac{I+2n}{n+1}$$

$$I = U(n+1)-2n$$

$$S = \frac{U(n+1)-2n}{n} + 1$$

$$lt_{n\to\infty} S = U-2+1 = U-1 \cong U$$

$$S_{avg} = \frac{Total\ internal\ path\ lengths}{n} + 1$$

$$U_{avg} = \frac{Total\ external\ path\ lengths}{n+1}$$

MAX-MIN

Problem:

To find out maximum and minimum elements in a given list of elements.

Iterative algorithm:

Procedure:

```
Max-Min(1 ------n)
for i = 2 to n do
if a[i] > max
        then max=A[i]
end if
if A[i]<min
        then min = A[i]
endif
endfor
```

Algorithm requires (n-1) comparisons for finding largest number and (n-1) comparisons for smallest number.

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

If same algorithm is modified as

```
If A[i]>max
        Then max=A[i]
Else
A[i]<min
```

Then min = A[i]

And if elements are in ascending order

No. of comparisons = (n-1)

If elements are in descending order

Number of comparisons = 2(n-1)

Average no of comp's = $\frac{(n-1)+2(n-1)}{2} = \frac{3n}{2} - 1$

Algorithm based on DAC

Step 1: I = (a[1], a[2]--------a[n]) is divided into

I_1 = (a[1], a[2]--------a[n/2])

I_1 = (a[n/2+1], --------a[n])

Step 2: Max(I) = large of (Max(I_1); Max(I_2))

Min(I) = large of (Min(I_1); Min(I_2))

```
Procedure MAX-MIN(i,j)
        {
    if I=j then
    max = min = A[I]
        else
        if j-I=1
        if A[I]>A[j]
        max = A[j]
        min = A[j]
        else
    max = A[j]
        min = A[I]
        else
        p= (i+j)/2
        {
MAX-MIN((I,p),h_max, h_min)

MAX-MIN((p+1,j),l_max, l_min)
        Max = large(h_max, h_min)
        Min = small(l_max, l_min)
```

}

Time complexity:

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 2T\left(\frac{n}{2}\right) + 2 & n > 2 \end{cases}$$

F(n)=2 since there are two operations

1. merging for largest
2. merging for smallest

$T(n) = 2T\left(\frac{n}{2}\right) + 2$

$= 2\{ 2T\left(\frac{n/2}{2}\right) + 2\} + 2$

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

$= 2^{k-1}T(n/2^{k-1}) + 2^{k-1} + \text{-------} + \text{-----} + 2$

let $n=2^k$

$= 2^{k-1}T(2) + \sum_{i=1}^{k-1} 2^i$

but T(2) = 1

$= 2^{k-1} + \sum_{i=0}^{k-1} 2^i - 1$

$= 2^{k-1} + 2^k - 1 - 1$

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

$= 3/2\ 2^k\text{-}2 = 3/2\ n - 2$

$T(n) = O(3/2\ n)$

Assignment Problems

1. Write the algorithm for trinary search and derive the expression for its time complexity?

Trinary Search

```
Procedure: TRISRCH(p,q)
{
        a=pt((q-p)/3)
        b=p+((2*(q-p))/3)
if A[a]=x then
        return(a);
else
if A[b]=x then
return(b);
else
if A[b]=x then
return(b);
else
if A[a]>x then
        TRISRCH(p,(a-1))
else
if A[b]>x then
        TRISRCH((a+1),(b-1))
else
        TRISRCH((b+1),q)
}
```

Time complexity:

$$T(n) = \begin{cases} g(n) & n \le 2 \\ T\left(\frac{n}{3}\right) + f(n) & n > 2 \end{cases}$$

$T(n) = T(n/3)+f(n)$

Let f(n)=1

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

$= T(n/3^2)+2$

$= T(n/3^k)+k=k+1$

$n=3^k$

$= \log_3 n$

$T(n) = O(\log_3 n)$

2. Give an algorithm for binary search so that the two halfs are unequal. Derive the expression for its time complexity?

Binary Search:

(Split the input size into 2 sets such that 1/3rd and 2/3rds)

```
procedure BINSRCH(p,q)
        {
        mid = p+(q-p)/3
        if A(mid)=x then
                return (mid);
        else
        if A(mid)>x then
        BINSRCH (p, mid-1);
        else
        BINSRCH(mid+1,q);
        }
        }
```

Time Complexity:

$$T(n) = \begin{cases} g(n) & \text{if } n = 1 \\ T\left(\frac{2n}{3}\right) + g(n) & / \quad T\left(\frac{n}{3}\right) + g(n) \end{cases}$$

Worst case best case

$T(n) = T(cn/3) + g(n)$

$= T(c^k n/3^k) + kg(n)$

let $n=3^k$

$T(n) = T(c^k)+\log_3 n$

If c=2

$T(n) = O(2^k)+\log n = O(2^k)$

C=1

$T(n) = O(1) + \log n = \log n$

3. Give the algorithm for finding min-max elements when the given list is divided into 3 sub sets. Derive expression for its time complexity?

Max-Min

```
If each leaf node has 3 elements
        If I=j then
                Max = min = a[i]
        Else
        If j-I=1 then
                If a[i]>a[j]
                        Max = a[i];
                        Min = a[j];
                Else
                        Max = a[j]
                        Min = a[i];
                }
                if j-i = 2 then
                        if a[i]>a[j]
                        {
                        if      a[i]>a[i+1]
then
                        max = a[i];
                        {
                        if      a[I]>a[I+1)
then
                        min = a(I+1];
                        else
                                min=a[j];
                        else
                                max     =
a[j+1];
                                min = a[j];
                        }
                                else
                                        if
a[j]>a[I+1}then
                                        max=a[j];
                                        if
a[I]>a[I+1] then
                                        min     =
a[I+1]
                                else
                                        min = a[I]
                                else
                                        max     =
a[I+1]
                                        min = a[I]
                                }
                                }
                                }
                        else
                                p = I+j/2

        MAX-MIN((I,p),h_max, h_min)

        MAX-MIN((p+1,j),l_max, l_min)
                                f_Max   =
        large(h_max, h_min)
                                f_Min   =
        small(l_max, l_min)
                                }
```

Time Complexity:

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ 3 & n = 3 \\ 2T\left(\frac{n}{2}\right) + 2 & n > 3 \end{cases}$$

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

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

$= 2^k(T(n/2^k))+2^k+\text{------}2$

$2^k=n/3$

kth level there are 3 elements for each node.

$$T(n) = 2^k.\ T(2)+ \sum_{i=1}^{2^k} 2^i$$

$= 3.\ 2^k+2^{k+1}-2$

$= n+2n/3-2$

$= 5n/3 -2$

$O(n) = O(5n/3)$

Theorem:

Prove that $T(n) = O(n^{\log m})$

if $T(n) = m(Tn/2) + an^2$

$T(n) = nT(n/2)+an^2$

$= m(mT(n/2/2)+an^2)an^2$

$= m^2T(n./2^2)+(m+1)an^2$

$= m^kT(n/2^k)+an^2 \sum_{i=1}^{k} m^{i-1}$

$n=2^k$

$= m^k+an^2(\sum_{i=1}^{k} m^{i-1} -1/m)$

$= m^k+an^2\left(\frac{m^{k+1-1}}{m-1}-\frac{1}{m}\right)$

$= m^k+an^2\left(\frac{m^{k}-1}{m-1}-\frac{1}{m}\right)$

$\cong m^k(1+an^2/m)$

if $m>>n$

$= m^k= m^{\log n}$

but $O(m^{\log n}) = O(n^{\log m})$

$(x^{\log y} = y^{\log x})$

Merge Sort:

Problem: Sorting the given list of elements in ascending or decreasing order.

Algorithm:

Step 1: Given list of elements

I = {A(1), A(2) --------- A(n)} is divided into two sets

I_1 = {A(1), A(2) -------A(n/2)}

I_2 = {A(n/2 + 1-------A(n)}

Step 2: Sort I_1 and Sort I_2 separately

Step 3: Merge them

Step 4: Repeat the process

```
Procedure Merge sort (p,q)
        {
                If p<q
                mid = p+q / 2:
                mergesort (p,mid);
                merge  sort  (mid+1,
q);
                merge (p, mid, q);
                endif
                }
procedure Merge(p, mid, q)
        {
        let h=p; I=p; j=mid+1
        while (h≤mid and j≤q) do
                {
                        if  A(h)  ≤  A(j)
then
                        B(i) = A(h);
                        h = h+1;
                        else
                        B(i) = A(j);
                        j = j+1;
                        end if
                        I=I+1;
                }
        If h>mid
                For k=j to q
                B(i) = A(k);
                I = I+1;
                Repeat
        Else if j>q
                For k=h to mid
                B(i) = A(k);
                I = I+1;
                Repeat
        Endif
                For k=p to q
                A(k) = B(k);
                Repeat
```

Time complexity:

$$T(n) = \begin{cases} a & n = 1 \\ 2T(\frac{n}{2}) + cn & n > 1 \end{cases}$$

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

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

$= 2^kT(n/2^k) + kcn$

$n = 2^k$

$= n.a + kcn$

$= na + cn\ logn$

$= O(n\ logn)$

Ex : Let us consider the instances 10, 285, 179, 652, 351,423,861,254,450,520

310, 285, 179 652, 351 423,861, 254, 450,520

1 2 3 4 5 6 7 8 9 10

For first recursive call left half will be sorted

(310) and (285) will be merged, then

(285,310) and 179 will be merged

and at last (179,285,310) and (652,351) are merged to give 179,285,310,351,652

Similarly, for second recursive call right half will be sorted to give
254,423,450,520,861

Finally merging of these two lists produces
179,254,285,310,351,423,450,520,652,820

Quick sort:

Problem: Arranging all the elements in a list in either ascending or descending order.

File A(1---n) was divided into subfiles, so that sorted subfiles does not need merging. This can be achieved if

A[I] $\leq$A(j) for all I<j<mid

mid+1<j<n

thus A(1----m) and A(m+1, -------n) are independently sorted.

Algorithm:

Step 1: Select first element as pivot

Step 2: Two variables are initialized

Lower = a+1 upper = b

Lower scans from left to right

Upper scans from right to left

Step 3: if lower ≤ right

Corresponding elements are swapped. When lower and upper cross each other, process is stepped.

Step 4: Pivot is swapped with A(upper) which becomes new pivot and

repeat the process.

```
Procedure Quick sort(a, b)
    {
        if a<b then
        k = partition (a,b);
        quick sort (a, k-1);
        quick sort (k+1, b);
        }
```

```
Procedure partition (m,p)
    {
        let x=A[m], I = m;
        while (I<p)
        {
            do
            I = I+1;
            Until (A[I]≥x)
            Do
            P = p-1;
            Until (A[p] ≤ x)
            If I<p
            Swap(A[I], A[P])
            }
        A[m]=A[p];
        A[p] = x
        }
```

Time Complexity:

Let us assume file size n is 2^m , $m= \log_2 n$

Also assume position of pivot is exactly in the middle of array.

In first pass (n-1) comparisons are made.

In second pass (n/2-1) comparisons are made.

Total no. of comparisons = (n-1)+2*(n/2-1)+4(n/4-1)+-------n*(n/2^m-1) $\propto$ mn

T(n) = O (nlogn)

If the original file is already sorted, the file is split into sub files of size into 0 and n=1.

Total comparisons = (n-1)+(n-2)+(n-3)---------(n-n) $\propto$ nn = 0(n^2)

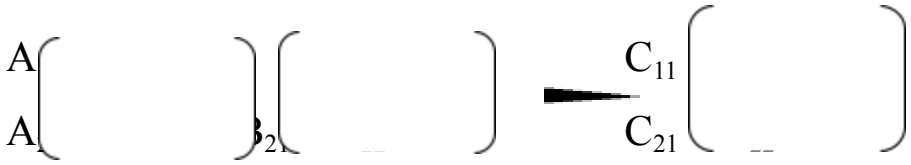
STRASSEN'S MATRIX MULTIPLICAITON

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

$$C(I,j) = \sum A(i,k)B(k,j)$$

For all I and j between 1 to n. To compute C(i,j) using this formula we need n multiplications. As the matrix has n^2 elements, the time for the resulting matrix multiplication algorithm has a time complexity of $\theta(n^3)$.

According to divided and conquer the given matrices A and B are divided into four square sub matrices each of dimension n/2 and n/2. The product AB can be computed using the product of 2 x 2 matrices.



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

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

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

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

To find AB we need 8 multiplications and 4 additions. Since two matrices can be added in time cn^2 for a constant c.

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

where b and c are constants

This recurrence can be solved in the same way as earlier recurrences to obtain $T(n)=O(n^3)$. Hence no improvement over the conventional method has been made. Since matrix multiplications are more expensive than matrix additions O(n3) versus $O(n^2)$, we can attempt to reformulate the equations for C_{ij} so as to have fewer multiplications and possibly more additions. Volker Strassen has discovered a way to compute the C_{ij}'s of using only 7 multiplications and 18 additions or subtractions. This method involves first computing the seven n/2Xn/2 matrices P, Q, R, S, T, U, and V. then the C_{ij}'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 additions or subtractions. The C_{ij} 's require an additional 8 additions or subtractions.

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

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

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

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

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

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

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

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

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

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

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

The resulting recurrence relation for T(n) is

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

where a and b are constants. Working with this formula, we get

$$T(n) = an^2[1+7/4+(7/4)^2+-------+(7/4)^{k-1}]+7^kT(1)$$

$$\le cn^2 (7/4)^{\log n}+7^{\log n}, \text{ c a constant}$$

$$= 0(n^{\log 7}) \approx O(n^{2.81}).$$

Greedy Method:

According to greedy method, to find a solution for a given problem, two parameters must be considered. They are

1) Feasibility: Any subset of solutions, that satisfies some specified constraints then it is said to be feasible.
2) Objective function: This feasible solution must either maximize or minimize a given function to achieve the given objective.

A solution which satisfies these two conditions, is called optimal solution. Greedy method works in stages, considering one input at a time. At each stage,

Step 1: A decision is made regarding whether or not a particular input is in an optimal solution. Select an input.

Step 2: If inclusion of input results infeasible solution, then delete form solution.

Step 3: Repeat until optimization is achieved.

Control Abstract:

```
Procedure GREEDY(A[],n)
{
    solution = φ
    for I = 1 to n do
    {
    x = select (a)
    if FEASIBLE (solution, x)
    solution = UNION (solution, x)
    }
    return(solution);
    }
```

Optimal storage:

Problem 1:

To store n programs on a tape of length L such that mean retrieval time is minimum.

Solution:

Let each program is of length L_i, ($1 \le i \le n$), such that the length of all tapes must be $\le L$. If the programs are in the order $I = i_1, i_2$---------i_n, then the time t_j to retrieve a program is proportional to $\sum_{1 \le k \le j} l_{ik}$.

If all the programs are retrieved with equal probabilities, mean retrieval time MRT = $\frac{1}{n} \sum_{1 \le j \le n} t_j$. To minimize MRT, we must store the programs such that their lengths are in non decreasing order $l_1 < l_2 < l_n$

MRT = $\frac{1}{n} \sum_{1 \le j \le n} \sum_{1 \le k \le j} l_{ik}$

Ex: There are 3 programs of length I=(5,10,3).

1,2,3 5+15+18 = 38

1,3,2 5+5+18 = 31

3,1,2 3+8+18 = 29 is the best case where retrieval time is minimized, if the next program of least length is added.

Time complexity:

If programs are assumed to be in order, then there is only one

loop in algorithm

Time complexity = O(n)

If we consider sorting also, sorting requires O(nlogn)

$$T(n) = O(n\log n)+O(n)$$
$$= O(n\log n)$$

Problem 3:

To store different programs of lengths l_1, l_2, l_3----------l_n with different frequencies f_1, f_2----------f_n on a single tape so that MRT is less.

Solution:

If frequencies are taken into account then MRT, $t_j = \sum_{1\le k\le j} \frac{f_{ik}}{l_{ik}}$. The programs must be arranged such that ratio of frequency and length must be in non increasing order.

$l_1 \le l_2 \le l_3$ ---------- l_n

$f_1 \ge f_2 \ge f_3$----------f_n

$$\frac{f_1}{l_1} \ge \frac{f_2}{l_2} \ge \text{----------} \frac{f_n}{l_n}$$

Knapsack problem:

Given n objects and a knapsack, object i has a weight w_i and knapsack has capacity of M. If a fraction x_i, $0 \le x_i \le 1$ of object i is placed into knapsack then a profit $p_i x_i$ is earned.

Problem: To fill the knapsack so that total profit is maximum and satisfies two conditions.

1) objective $\sum_{1\le i\le n} p_i x_i$ =maximum

2) <u>feasibility</u> $\sum_{1\le i\le n} w_i x_i \le m$

Types of knapsack:

1. Normal knapsack: In normal knapsack , a fraction of the last object is included to maximize the profit.

i.e., $0\le x\le 1$ so that $\sum w_i x_i = M$

2.O/I knapsack: In 0/1 knapsack, if the last object is able to insert completely, then only it is selected otherwise it not selected.

i.e., x= 1/0 so that $\sum w_i x_i \le M$

<u>Control Abstract:</u>

P(1:n), w(1:n) contains profits & weights of n objects are ordered so that $\frac{p_i}{w_i} \ge \frac{p_{i+1}}{w_{i+1}}$ and x(1:n) is the solution vector.

```
Normal knapsack (p,w,m,x,n)
{
      x=0;
      c=M;
      for I=1 to n do
      if w[I] ≤c then
      {
      x[I] = 1;
      c=c-w[I];
      }
      else
      {
            x[I]=c/w[I]
            return;
            }
            }
```

```
O/1 knapsack:
      {
            x=0
```

```
c=M                                  x[I] = 1
for I=1 to n do                      c=c-w[I]
if w[I] ≥c then return               }
else                                 }
```

Ex: Find an optimal solution to the knapsack instance n=7 objects and the capacity of knapsack m=15. The profits and weights of the objects are given below.

$P_{1,}$----------P_7= 10 5 15 7 6 18 3

W_1, -------w_7= 2 3 5 7 1 4 1

Sol:

P1/w1----p7/w7 = 5 1.6 3 1 6 4.5 3

Arrange in non increasing order of p_i/w_i

Order = 5 1 6 3 7 2 4

P/w = 6 5 4.5 3 3 1.6 1

C=15 x=0

1) *x[1] = 1* *x=0*
2) *x[2] = 1* *c=15-1=14*
3) *x[3] = 1* *c=14-2=12*
4) *x[4] = 1* *c=12-4=8*
5) *x[5] = 1* *c=8-5=3*
6) *x[6] = 2/3* *c=3-1=2*
7) *x[7] = 0* *c=2-2/3.3=0*

X= 1 1 1 1 1 2/3 0

Rearranging into original form

X=1 2/3 1 0 1 1 1

Solution vector for O/I knapsack is X=1010111

Total profit $\sum p_i x_i$ *= 10+15+0+6+18+3=52*

Theorem:

If $p_1/w_1 \geq p_2/w_2 \geq$----$p_n/w_n$ then greedy knapsack generate optimal solution.

Proof: Let $X=(x_1= (x_1,$ -------$x_2)$ be solution vector, let j be the least index such that $x_j \# 1$.

$X_i=1$ for $1 \leq i \leq j$

$X_i=0$ for $1 \leq i \leq n$

$X_i=0$ for $1 \leq x_j \leq 1$

Let $y = (y_1$-----$y_n)$ be optimal solution

$$\sum w_i x_i = M$$

let k be least index such that $y_k \# x_k$ and $y_k < x_k$

1) if $k<j$ then $x_k=1$, but $y_k<x_k$
2) if k=j then $\sum w_i x_i = M$, $y_i=x_i$ for $1<i<j$ $y_k<x_k$
3) if k>j then $\sum w_i x_i > M$ which is not possible.

If we increase y_k to x_k and decrease as many of $(y_{k+1}$----------$y_n)$. this results a new solution.

$Z= (z_1,$--------$z_n)$ with $z_i=x_i$, $1 \leq i \leq k$ and $\Sigma w_i(y_i-z_i) = w_k(z_k-y_k)$

$$\Sigma p_i z_i = \Sigma_{1<i<n} p_i y_i + (z_k - y_k) w_k p_k / w_k - \Sigma_{k<I<n} (y_i - z_i) w_i p_i / w_i$$

$$= \Sigma_{1<i<n} p_i y_i + [(z_k - y_k) w_k - \Sigma_{1<i<} n(y_i - z_i) w_i] p_k / w_k$$

$$= \Sigma_{1<i<n} p_i y_i$$

Theorem:

Why O/I Knapsack does not necessarily yield optimal solution.

Solution:

If w[i] >c for a certain i then

w[i] ≤ c-w(i-1)

In other words remaining capacity at (i-1) stage may have been better utilized such that by including w[i], c becomes 0 or minimum. Hence it is optimal.

Job Sequencing:

Let there are n jobs, for any job i, profit P_i is earned if it is completed in dead line d_i.

Object: Obtain a feasible solution j=(j1,j2,…) so that the profits i.e., $\sum_{i \in j} p_i$ is maximum. The set of jobs J is such that each job completes within deadline.

Control Abstract:

```
Procedure JS(D, J, N, K)
{
D(0)=J(0)=0;
K=1, J(1)=1
For I=2 to n do
{
r=k
while D(J(r))>max(D(i),r)
{
r=r-1;
}
if D(i)>r then
for (L=k; l=r+1,l=l-1)
J(l+1)=J(l)
}
J(r+1)=i;
k=k+1;
}
```

}

}

Ex: Given 5 jobs, such that they have the profits and should be complete within dead lines as given below. Find and optimal sequence to achieve maximum prodit.

P_1-------P_5 = 20 15 10 5 1

d_1------d_5 = 2 2 1 3 3

	feasible solution	*processing sequence*	*value*
1)	*1,2*	*2,1 or 1,2*	*35*
2)	*1,3*	*3,1*	*30*
3)	*1, 4*	*1,4*	*25*
4)	*1,2,3*	*-*	*-*
5)	*1,2,4*	*1,2,4*	*40*

optimal solution is 1,2,4

Try all possible permutations and if J can be produced in any of these permutations without violating dead line.

If J is a set of k jobs and $\sigma=i_1, i_2,$ ------i_k, such that $d_{i1}\leq d_{i2}\leq$------d_{ik} then j is a feasible solution.

Optimal Merge pattern:

Merging of an ‘n’ record file and ‘m’ record file requires n+m records

to move. At each step merge smallest size files. Two way merge pattern is

represented by a merge tree. Leaf nodes (squares) represent the given files. File obtained by merging the files is a parent file (circles). Number in each node is length of file or number of records.

If d_i is the distance from root to external node for file f_i, q_i is the length

of f_i, then total number of record moves in a tree = $\sum_{i=1}^{n} d_i q_i$, Known as weighted external path length.

Algorithm:

```
Procedure Tree (l,n)
For I =1 to n
{
Call GETNODE (T)
L CHILD(T)=LEAST (L);
R CHILD(T)=LEAST (L);
WEIGHT(T) =
WEIGHT(LCHILD(T))+WEIGHT(RCHILD(T));
Call INSERT(L,T);
}
return(LEAST(L))
}
```

ANALYSIS:

Main loop is executing (n-1) times. If L is in non decreasing order

LEAST(L) requires only O(1) and INSERT(L,T) can be done in O(n) times.

Total time taken = $O(n^2)$

Minimum spanning Tree:

Let G=(V,E) is an undirected connected graph. A sub graph T=(V,E) of G is a spanning tree of F iff T is a tree.0

Any connected graph with n vertices must have at least n-1 edges and all connected graphs with n-1 edge are trees. Each edge in the graph is associated with a weight. Such a weighted graph is used for construction of a set of communication links at a minimum cost. Removal of any one of the links in the graph if it has cycles will result a spanning tree with minimum cost. The cost of minimum spanning tree is sum of costs of edges in that tree.

A greedy method is to obtain a minimum cost spanning tree. The next edge to include is chosen according to optimization criteria in the sum of costs of edges so far included.

1) If A is the set of edges in a spanning tree.
2) The next edge (u,v) to be included in A is a minimum cost edge and

   A U (u,v) is also a tree.

PRIMS Algorithm:

The edge (i,j) to be is such that i is a vertex already included in the tree, j is a vertex not included and cost of (i,j)must be minimum among all edges (k,l) such that k is in the tree and l is not in the tree

Ex:

Edge	*Cost*	*ST*
(1,2)	*10*	*(1) ------2*
(1,4)	*30*	
(4,5)	*20*	
(5,3)	*35*	

Algorithm:

```
Procedure PRIM(E, COST, n, mincost, int NEAR(n), T[n][2], I, j, k, l)
    {
    (k,l)   = edge with mini cost
    min cost = cost (k,l)
    (T(1,1), T(1,2)) = (k,l)
    for i=1 to n
    {
    if COST (i,l)<cost(i,k)
    then NEAR(i) = l
    else
    NEAR(i) = k
    }
    }
    NEAR(k) = NEAR(l)=0
    For i =2 to n-1
    {
    If NEAR(j)=0 find cost(j,NEAR(j))which is mini
    T(i,1), T(i,2) = (j, NEAR(j))
    Mincost = mincost+cost(j, NEAR(j))
    NEAR(j)=0
    For k=1to n
    {
    If NEAR(k) !=0 and cost(k,NEAR(k))>COST(k,j)
    Then NEAR(k) =j;
    }
    }
```

Time complexity:

The total time required for the algorithm is $\theta(n^2)$.

Krushkal's algorithm:

If E is the set of all edges of G, determine an edge with minimum

cost(v,w) and delete from E. If the edge(v,w) does not create any cycle in the tree T, add(v,w) to T.

Algorithm:

```
Procedure KRUSKAL(E, COST, n, T)
        {
        construct a heap with edge costs
        I=0
        Min cost=0
        Parent (1,n) = -1,
        While I<n-1 and heap not empty
        Delete minimum cost edge (u,v)
        from heap;
        ADJUST heap;
        J = FIND(u)
        K = FIND(v)
        If j != k
        {
        I = I+1;
        T(I,1) = u;
        T(I,2) = v;
        Mincost = mincost+cost(u,v)
        UNION(j,k)
        }
        }
        }
Procedure FIND(i)
        {
        J=I
        While PARENT(j)>0
        {
        J=PARENT(j)
        }
        K=I;
        While K!= j
        {
        T = PARENT(k);
        PARENT(k) = j;
        K = t
        }
        Return(j)
        }
Procedure UNION(i,j)
{
x=PARENTT(i) + PARENT(j)
        If PARENT(i)>PARENT(j)
        {
        PARENT(i)=j;
        PARENT(j) = x
        else
        PARENT(j)=i;
        PARENT(i) = x
        }
        }
```

Time Complexity:

To develop a heap the time taken is O(n). To adjust the heap it takes a

time complexity of O(logn). The set of edges to be included in minimum cost spanning tree is n, hence it is O(n).

Total time complexity is in the order of O(nlogn)