**ALGORITHM ANALYSIS & DESIGN**

**ETCS 254**

Faculty: Student Name:

**Mr. Sandeep Tayal**

Roll No. :

Semester:

**IV (B.Tech CSE)**

Group:

**4C789**



**Maharaja Agrasen Institute Of Technology**

Sector – 22, Rohini, New Delhi – 110085

(Affiliated to Guru Gobind Singh Indraprastha University)

**Algorithm Analysis and Design Lab**

**PRACTICAL RECORD**

**PAPER CODE : ETCS-254**

Name of the student :

University Roll No. :

Branch : B.Tech CSE (IInd Shift)

Section/ Group : 4C789

**PRACTICAL DETAILS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program No. | Program | Date | Teacher Signature | Marks  (10) |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

1. **TO ANALYZE COMPLEXITY OF INSERTION SORT ALGORITHM :**

**ALGORITHM:**

Let A be a linear array of n numbers A [1], A [2], A [3], ...... ,A [n]......TEMP be a temporary

variable to interchange the two values. Pos is the control variable to hold the position of

each pass.

Step 1: Input an array A of *n* numbers

Step 2: Initialize *i* = 1 and repeat through steps 4 by incrementing *i* by one.

(*a*) If (*i* < = *n* – 1)

(*b*) TEMP = A [I],

(*c*) Pos = *i* – 1

Step 3: Repeat the step 3 if (TEMP < A[Pos] and (Pos >= 0))

(*a*) A [Pos+1] = A [Pos]

(*b*) Pos = Pos-1

Step 4: A [Pos +1] = TEMP

Step 5: Exit.

**ANALYSIS:**

The running time of the algorithm is the sum of running times for each statement executed. So, we have

T(n) = c1n + c2 (n − 1) + c4 (n − 1) + c5 ∑2≤j≤n (tj) + c6 ∑2≤j≤n (tj − 1) + c7 ∑2≤j≤n(tj − 1)

+ c8 (n − 1)

In the above equation we supposed that tj  be the number of times the while-loop is executed for that value of j.

**Worst-Case**

The worst-case occurs if the array is sorted in reverse order i.e., in decreasing order. In the reverse order, we always find that *A*[*i*] is greater than the key in the while-loop test. So, we must compare each element *A*[*j*] with each element in the entire sorted subarray *A*[1 .. *j* − 1] and so *tj* = *j* for *j* = 2, 3, ..., *n*. Equivalently, we can say that since the while-loop exits because *i* reaches to 0, there is one additional test after (*j* − 1) tests. Therefore, *tj* = *j* for *j* = 2, 3, ..., *n* and the worst-case running time can be computed using equation (1) as follows:

T(*n*) = *c*1*n* + *c*2 (*n* − 1) + *c*4  (*n* − 1) + *c*5 ∑2≤*j*≤*n* (*j* ) + *c*6 ∑2≤*j*≤*n*(*j* − 1) + *c*7 ∑2≤*j*≤*n*(*j* − 1)

+ *c*8 (*n* − 1)

And using the summations, we have

T(*n*) = *c*1*n* + *c*2 (*n* − 1) + *c*4  (*n* − 1) + *c*5 ∑2≤*j*≤*n*[*n*(*n*+1)/2 + 1] + *c*6 ∑2≤*j*≤*n* [*n*(*n* − 1)/2]

+ *c*7 ∑2≤*j*≤*n* [*n*(*n* − 1)/2] + *c*8 (*n* − 1)

T(*n*) = (*c*5/2 + *c*6/2 + *c*7/2) *n*2 + (*c*1 + *c*2 + *c*4 + *c*5/2 − *c*6/2 − *c*7/2 + *c*8) *n* − (*c*2 + *c*4 + *c*5 + *c*8)

This running time can be expressed as (*an*2 + *bn* + *c*) for constants *a*, *b*, and *c* that again depend on the statement costs *ci*. Therefore, T(*n*) is a quadratic function of n.

T(*n*) = *an*2 + *bn* + *c* = O(*n*2)

**Best Case:**

The best case occurs if the array is already sorted. For each *j* = 2, 3, ..., *n*, we find that *A*[*i*] less than or equal to the key when *i* has its initial value of (*j* − 1). In other words, when *i* = *j* −1, always find the key *A*[*i*] upon the first time the WHILE loop is run.

Therefore, *tj* = 1 for *j* = 2, 3, ..., *n* and the best-case running time can be computed using equation (1) as follows:

T(*n*) = *c*1*n* + *c*2 (*n* − 1) + *c*4 (*n* − 1) + *c*5 ∑2≤*j*≤*n* (1) + *c*6 ∑2≤*j*≤*n*(1 − 1) + *c*7 ∑2≤*j*≤*n*(1 − 1)

+ *c*8 (*n* − 1)

T(*n*) = *c*1*n* + *c*2 (*n* − 1) + *c*4 (*n* − 1) + *c*5 (*n* − 1) + *c*8 (*n* − 1)

T(*n*) =*(c*1 + *c*2 + *c*4  + *c*5  + *c*8 ) *n* + (*c*2  + *c*4  + *c*5  + *c*8)

This running time can be expressed as *an* + *b* for constants *a* and *b* that depend on the statement costs *ci*. Therefore, T(*n*) it is a linear function of *n*.

T(*n*) = *an* + *b* = O(*n*)

**IMPLEMENTATION:**

#include<iostream.h>

#include<conio.h>

void main()

{clrscr();

int a[20],t,n,temp,j;

cout<<"\nEnter the number of elements of Array: ";

cin>>n;

cout<<"\nEnter the elements of the array: "<<endl;

for(int i=0;i<n;i++)

{cin>>a[i];}

cout<<"\nThe array is: ";

for(i=0;i<n;i++)

{cout<<" "<<a[i];}

cout<<"\n";

for(i=0;i<n;i++)

{temp=a[i];

j=i-1;

while(temp<a[j] && j>=0)

{a[j+1]=a[j];

j=j-1;

}

a[j+1]=temp;

cout<<"\n The array after "<<i+1<<" iteration is : ";

for(int k=0;k<n;k++)

{cout<<" "<<a[k];}

}

cout<<"\nThe sorted array is\n "<<endl;

for(i=0;i<n;i++)

{cout<<"\t"<<a[i];}

getch();

}

**OUTPUT:**

Enter the number of elements of Array: 5

Enter the elements of the array:

2

5

3

7

1

The array is: 2 5 3 7 1

The array after 1 iteration is: 2 5 3 7 1

The array after 2 iteration is: 2 5 3 7 1

The array after 3 iteration is: 2 3 5 7 1

The array after 4 iteration is: 2 3 5 7 1

The array after 5 iteration is: 1 2 3 5 7

The sorted array is

1 2 3 5 7

1. **TO ANALYZE COMPLEXITY OF QUICK SORT ALGORITHM :**

**ALGORITHM:**

**function** *quicksort*(array)

**if** length(array) > 1

pivot **:=** *select any element of* array

left **:= first index of** array

right **:=** **last index of** array

**while** left ≤ right

**while** array[left] < pivot

left := left + 1

**while** array[right] > pivot

right := right - 1

**if** left ≤ right

**swap** array[left] **with** array[right]

left := left + 1

right := right - 1

quicksort(array **from first index to** right)

quicksort(array **from** left **to last index**)

### ANALYSIS:

As T(N) = T(i) + T(N - i -1) + cN

The time to sort the file is equal to the time to sort the left partition with i elements, plus

the time to sort the right partition with N-i-1 elements, plus the time to build the partitions

**Worst case**

The pivot is the smallest element

T(N) = T(N-1) + cN, N > 1

T(N-1) = T(N-2) + c(N-1)

T(N-2) = T(N-3) + c(N-2)

T(N-3) = T(N-4) + c(N-3)

T(2) = T(1) + 2c

Add all equations:

T(N) + T(N-1) + T(N-2) + … + T(2)

= T(N-1) + T(N-2) + … + T(2) + T(1) + c(N) + c(N-1) … + c.2

T(N) = T(1) + c(2 + 3 + … + N)

T(N) = 1 + c(N(N+1)/2 -1)

Therefore **T(N) = O(N2)**

**Best-case :**

The pivot is in the middle

T(N) = 2T(N/2) + cN

Divide by N:

T(N) / N = T(N/2) / (N/2) + c

Telescoping:

T(N/2) / (N/2) = T(N/4) / (N/4) + c

T(N/4) / (N/4) = T(N/8) / (N/8) + c

……

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

Add all equations and solving we get:

T(N)/N = T(1) + cLogN = 1 + cLogN

T(N) = N + cNLogN

Therefore **T(N) = O(NlogN)**

**Average case :**

The average value of T(i) is 1/N times the sum of T(0) through T(N-1)

1/N ∑ T(j), j = 0 thru N-1

T(N) = 2/N (∑T(j)) + cN

Multiply by N

NT(N) = 2(∑ T(j)) + cN\*N

To remove the summation, we rewrite the equation for N-1:

(N-1)T(N-1) = 2(∑ T(j)) + c(N-1)2, j = 0 thru N-2

and subtract:

NT(N) - (N-1)T(N-1) = 2T(N-1) + 2cN -c

Prepare for telescoping. Rearrange terms, drop the insignificant c:

NT(N) = (N+1)T(N-1) + 2cN

Divide by N(N+1):

T(N)/(N+1) = T(N-1)/N + 2c/(N+1)

Telescope:

T(N)/(N+1) = T(N-1)/N + 2c/(N+1)

T(N-1)/(N) = T(N-2)/(N-1)+ 2c/(N)

T(N-2)/(N-1) = T(N-3)/(N-2) + 2c/(N-1)

….

T(2)/3 = T(1)/2 + 2c /3

Add the equations and cross equal terms:

T(N)/(N+1) = T(1)/2 +2c ∑ (1/j), j = 3 to N+1

T(N) = (N+1)(1/2 + 2c ∑(1/j))

The sum ∑ (1/j), j =3 to N-1, is about LogN

Thus T(N) = O(NlogN)

**IMPLEMENTATION:**

#include<iostream.h>

#include<conio.h>

int partition(int\* ,int ,int);

void quick(int\* ,int ,int);

void quick(int arr[],int lower,int higher)

{int j;

if(higher>lower)

{j=partition(arr,lower,higher);

quick(arr,lower,j-1);

quick(arr,j+1,higher);

}

}

int partition(int arr[],int lower,int higher)

{

int j,lft,rgt,temp;

lft=lower+1;

rgt=higher;

j=arr[lower];

while(lft<=rgt)

{while(arr[lft]<j)

{lft++;}

while(arr[rgt]>j)

{rgt--;}

if(lft<rgt)

{temp=arr[lft];

arr[lft]=arr[rgt];

arr[rgt]=temp;

}

}

temp=arr[lower];

arr[lower]=arr[rgt];

arr[rgt]=temp;

return rgt;

}

main()

{

//clrscr();

int i,n,a[20],high,low;

cout<<"\nEnter no of elements of the array: ";

cin>>n;

low=0;

high=n-1;

cout<<"\nEnter the elements of the array: "<<endl;

for(i=0;i<n;i++)

{cin>>a[i];}

quick(a,low,high);

cout<<"\nThe sorted array is: ";

for(i=0;i<n;i++)

{cout<<" "<<a[i];}

getch();

return 0;

}

**OUTPUT:**

Enter no. of elements of the array: 5

Enter the elements of the array:

-1

7

3

0

-9

The sorted array is: -9 -1 0 3 7

1. **TO ANALYZE COMPLEXITY OF MERGE SORT ALGORITHM :**

**ALGORITHM:**

**function** merge\_sort(m)

**if** length(m) ≤ 1

**return** m

**var** *list* left, right, result

**var** *integer* middle = length(m) / 2

**for each** x **in** m **up to** middle

add x to left

**for each** x **in** m **after or equal** middle

add x to right

left = merge\_sort(left)

right = merge\_sort(right)

result = merge(left, right)

**return** result

**function** merge(left,right)

**var** *list* result

**while** length(left) > 0 **or** length(right) > 0

**if** length(left) > 0 **and** length(right) > 0

**if** first(left) ≤ first(right)

append first(left) to result

left = rest(left)

**else**

append first(right) to result

right = rest(right)

**else if** length(left) > 0

append first(left) to result

left = rest(left)

**else if** length(right) > 0

append first(right) to result

right = rest(right)

**end of while**

**return** result

**ANALYSIS**

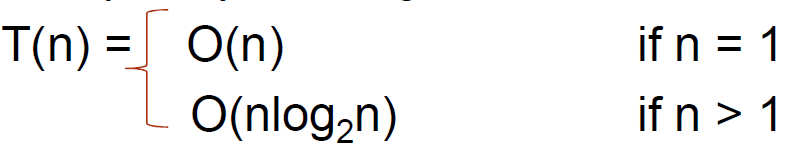
When n ≥ 2, time for merge sort steps:

* Divide : Just compute q as the average of p and r, which takes constant time i.e. Θ(1).
* Conquer: Recursively solve 2 sub-problems, each of size n/2, which is 2T(n/2).
* Combine: MERGE on an n-element sub-array takes Θ(n) time.

Summed together they give a function that is linear in n, which is Θ(n). Therefore, the recurrence for merge sort running time is

merge sort recurrence

Hence, complexity of merge sort is



A drawback of Merge sort is that it needs an additional space of Θ(n) for the temporary array *.*

**IMPLEMENTATION:**

#include <iostream.h>

#include <conio.h>

void MergeSort(int arrays[], int temp[], int size);

void m\_sort(int arrays[], int temp[], int left, int right);

void merge(int arrays[], int temp[], int left, int mid, int right);

int arrays[30];

int temp[30];

void MergeSort(int arrays[], int temp[], int size)

{m\_sort(arrays,temp,0,size-1);}

void m\_sort(int arrays[], int temp[], int left, int right)

{int mid;

if(right>left)

{mid=(right+left)/2;

m\_sort(arrays,temp,left,mid);

m\_sort(arrays,temp,mid+1,right);

merge(arrays,temp,left,mid+1,right);

}

}

void merge(int arrays[], int temp[], int left, int mid, int right)

{int i,left\_end,num\_ele,temp\_pos;

left\_end=mid-1;

temp\_pos=left;

num\_ele=right-left+1;

while((left<=left\_end)&&(mid<=right))

{if(arrays[left]<=arrays[mid])

{temp[temp\_pos]=arrays[left];

temp\_pos=temp\_pos+1;

left=left+1;

}

else

{temp[temp\_pos]=arrays[mid];

temp\_pos=temp\_pos+1;

mid=mid+1;

}

}

while(left<=left\_end)

{temp[temp\_pos]=arrays[left];

left=left+1;

temp\_pos=temp\_pos+1;

}

while(mid<=right)

{temp[temp\_pos]=arrays[mid];

mid=mid+1;

temp\_pos=temp\_pos+1;

}

for(i=0;i<=num\_ele;i++)

{arrays[right]=temp[right];

right=right-1;

}

}

void main()

{clrscr();

int i,n;

cout<<" \nEnter the number of elements: ";

cin>>n;

cout<<"\n\nEnter the elements: "<<endl;

for(i=0;i<n;i++)

{cin>>arrays[i];}

cout<<" \n\n\nThe Unsorted array is -> \n\n\t\t";

for(i=0;i<n;i++) // Before Merging

{cout<<arrays[i]<<" ";}

MergeSort(arrays,temp,n);

cout<<" \n\n\nThe Sorted array is -> \n\n\t\t";

for(i=0;i<n;i++) //After merging

{cout<<arrays[i]<<" ";}

getch();

}

**OUTPUT:**

Enter the number of elements: 6

Enter the elements:

3

2

1

4

6

5

The Unsorted array is ->

3 2 1 4 6 5

The Sorted array is ->

1 2 3 4 5 6

1. **PROGRAM TO IMPLEMENT LCS PROBLEM USING DYNAMIC PROGRAMMING**:

**ALGORITHM AND ANALYSIS:**

The longest common subsequence problem (LCS) is finding a longest sequence which is a subsequence of all sequences in a set of sequences (often just two).

Let two sequences be defined as follows: *X* = (*x*1, *x*2...*x*m) and *Y* = (*y*1, *y*2...*y*n). The prefixes of *X* are *X*1, 2,...m; the prefixes of *Y* are *Y*1, 2,...n. Let *LCS*(*Xi*, *Yj*) represent the set of longest common subsequence of prefixes *Xi* and *Yj*. This set of sequences is given by the following.


LCS\left(X_{i},Y_{j}\right) =
\begin{cases}
  \empty
& \mbox{ if }\ i = 0 \mbox{ or }  j = 0 \\
  \textrm{  } LCS\left(X_{i-1},Y_{j-1}\right) +  1
& \mbox{ if } x_i = y_j \\
  \mbox{longest}\left(LCS\left(X_{i},Y_{j-1}\right),LCS\left(X_{i-1},Y_{j}\right)\right)
& \mbox{ if } x_i \ne y_j \\
\end{cases}


COMPUTING THE LENGTH OF THE LCS

The below function takes as input sequences X[1..m] and Y[1..n] computes the LCS between X[1..i] and Y[1..j] for all 1 ≤ i ≤ m and 1 ≤ j ≤ n, and stores it in C[i,j]. C[m,n] will contain the length of the LCS of X and Y.

function LCS(X[1..m], Y[1..n])

C = array(0..m, 0..n)

for i := 1..m

for j := 1..n

if X[i] = Y[j]

C[i,j] := C[i-1,j-1] + 1

else:

C[i,j] := max(C[i,j-1], C[i-1,j])

return C

BACKTRACKING

The following function backtracks the choices taken when computing the C table. If the last characters in the prefixes are equal, they must be in an LCS. If not, check what gave the largest LCS of keeping xi and yj, and make the same choice. Just choose one if they were equally long.

Call the function with i=m and j=n.

function backTrack(C[0..m,0..n], X[1..m], Y[1..n], i, j)

if i = 0 or j = 0

return ""

else if X[i] = Y[j]

return backTrack(C, X, Y, i-1, j-1) + X[i]

else

if C[i,j-1] > C[i-1,j]

return backTrack(C, X, Y, i, j-1)

else

return backTrack(C, X, Y, i-1, j)

If choosing x[i] and y[j] would give an equally long result, both resulting subsequences should be shown. This is returned as a set by this function. Notice that this function is not polynominal, as it might branch in almost every step if the strings are similar.

function backTrackAll(C[0..m,0..n], X[1..m], Y[1..n], i, j)

if i = 0 or j = 0

return {}

else if X[i] = Y[j]:

return {Z + X[i-1] for all Z in backTrackAll(C, X, Y, i-1, j-1)}

else:

R := {}

if C[i,j-1] ≥ C[i-1,j]:

R := R ∪ backTrackAll(C, X, Y, i, j-1)

if C[i-1,j] ≥ C[i,j-1]:

R := R ∪ backTrackAll(C, X, Y, i-1, j)

return R

COMPLEXITY:

For the case of two sequences of n and m elements, the running time of the dynamic programming approach is [O](http://en.wikipedia.org/wiki/Big_O_notation)(n × m).

**IMPLEMENTATION:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

void print\_lcs(char b[][20],char x[],int i,int j)

{

if(i==0 || j==0)

return;

if(b[i][j]=='c')

{

print\_lcs(b,x,i-1,j-1);

printf(" %c",x[i-1]);

}

else if(b[i][j]=='u')

print\_lcs(b,x,i-1,j);

else

print\_lcs(b,x,i,j-1);

}

void lcs\_length(char x[],char y[])

{

int m,n,i,j,c[20][20];

char b[20][20];

m=strlen(x);

n=strlen(y);

for(i=0;i<=m;i++)

c[i][0]=0;

for(i=0;i<=n;i++)

c[0][i]=0;

for(i=1;i<=m;i++)

for(j=1;j<=n;j++)

{

if(x[i-1]==y[j-1])

{

c[i][j]=c[i-1][j-1]+1;

b[i][j]='c'; //c stands for left upright cross

}

else if(c[i-1][j]>=c[i][j-1])

{

c[i][j]=c[i-1][j];

b[i][j]='u'; //u stands for upright or above

}

else

{

c[i][j]=c[i][j-1];

b[i][j]='l'; //l stands for left

}

}

print\_lcs(b,x,m,n);

}

void lcs()

{

int i,j;

char x[20],y[20];

printf("1st sequence:");

gets(x);

printf("2nd sequence:");

gets(y);

printf("\nLCS are:\n");

lcs\_length(x,y);

printf("\n");

lcs\_length(y,x);

}

main()

{

char ch;

do

{

lcs();

printf("\nContinue(y/n):");

ch=getch();

}

while(ch=='y'||ch=='Y');

getch();

return 0;

}

**OUTPUT:**

1st sequence: AGTBAHATGHA

2nd sequence: AGBAHATAGAH

LCS are:

A G B A H A T G H

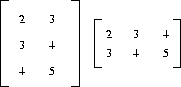
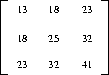
A G B A H A T G A

Continue(y/n): n

1. **PROGRAM TO IMPLEMENT MATRIX CHAIN MULTIPLICAYION PROBLEM USING DYNAMIC PROGRAMMING**

PROBLEM: Multiplying a Sequence of Matrices . Suppose a long sequence of matrices A x B x C x D...   has to be multiplied

Multiplying an X x Y matrix by a Y x Z matrix (using the common algorithm) takes X x Y x Z multiplications.

In matrix multiplication it is better to avoid big intermediate matrices, and since matrix multiplication is associative, we can parenthesize however we want.

Matrix multiplication is not commutative, so the order of the matrices can not be permuted without changing the result. Example:-

Consider A x B x C x D, where A is 30 x 1, B is 1 x 40, C is 40 x 10, and D is 10 x 25.

There are three possible parenthesizations:

Description: displaymath14860

Description: displaymath14861

Description: displaymath14862

The order makes a big difference in real computation. Let *M*(*i*,*j*) be the minimum number of multiplications necessary to compute Description: tex2html_wrap_inline14908.

The key observations are

* The outermost parentheses partition the chain of matricies (*i*,*j*) at some k.
* The optimal parenthesization order has optimal ordering on either side of k.

A recurrence for this is:

Description: eqnarray7163

If there are n matrices, there are n+1 dimensions.

A direct recursive implementation of this will be exponential, since there is a lot of duplicated work as in the Fibonacci recurrence.

Divide-and-conquer is seems efficient because there is no overlap, but ...

There are only Description: tex2html_wrap_inline14912substrings between 1 and n. Thus it requires only Description: tex2html_wrap_inline14914space to store the optimal cost for each of them.

All the possibilities can be represented in a triangle matrix. We can also store the value of k in another triangle matrix to reconstruct to order of the optimal parenthesisation.

The diagonal moves up to the right as the computation progresses. On each element of the kth diagonal |*j*-*i*| = *k*.

**ALGORITHM**:-

for *i*=1 to n do *M*[*i*, *j*]=0

for *diagonal*=1 to n-1

for *i*=1 to n-diagonal do

*j*=*i*+*diagonal*

Description: tex2html_wrap_inline14928

faster(*i*,*j*)=*k*

return [*m*(1, *n*)]

Pseudocode:

ShowOrder(*i*, *j*)

if (*i*=*j*) write ( Ai )

else

*k*=factor(*i*, *j*)

write “('”

ShowOrder(*i*, *k*)

write “\*”

ShowOrder (*k*+1, *j*)

write “)”

**IMPLEMENTAION:**

#include<iostream.h>

#include<conio.h>

void print(int s[10][10],int a,int b);

void main()

{

clrscr();

int x,i,j,n,l,k,a[10],b[10],m[10][10],q,p[10],s[10][10];

cout<<"\n Enter the number of matrices: ";

cin>>n;

cout<<"\n Enter the rows of matrices : \n";

for(i=1;i<=n;i++)

{ cout<<" "<<i<<" -: ";

cin>>a[i];}

cout<<"\n Enter the columns of matrices : \n";

for(j=1;j<=n;j++)

{ cout<<" "<<j<<" -: ";

cin>>b[j]; }

p[0]=a[1];

for(x=1;x<=n;x++)

p[x]=b[x];

cout<<"\n Matrices are -: ";

for(x=1;x<=n;x++)

cout<<"\nA["<<x<<"]"<< " = " <<a[x]<<" \* "<<b[x];

// display the size of matrices

for(i=1;i<=n;i++)

m[i][i]=0;

for(l=2;l<=n;l++)

{

for(i=1;i<=n-l+1;i++)

{

j=i+l-1;

m[i][j]=1000000;

for(k=i;k<=j-1;k++)

{

q=m[i][k]+m[k+1][j]+p[i-1]\*p[k]\*p[j];

if(q<m[i][j])

{

m[i][j]=q;

s[i][j]=k;

}

}

}

}

cout<<"\n\n Order of Multiplication -: ";

print(s,i-1,j);

getch();}

void print(int s[10][10],int a,int b)

{

if(a==b)

{

cout<<"A"<<a;

}

else

{

cout<<"(";

print(s,a,s[a][b]);

print(s,s[a][b]+1,b);

cout<<")";

}

}

**OUTPUT**

Enter the number of matrices: 3

Enter the rows of matrices:

1-:2

2-:3

3-:4

Enter the columns of matrices:

1-:3

2-:4

3-:2

Matrices are -:

2\*3 3\*4 4\*2

Order of Multiplication -:

(A1 (A2 A3 ))

1. **PROGRAM TO IMPLEMENT STRASSEN’S MATRIX MULTIPLICATION:**

In the mathematical discipline of linear algebra, **the Strassen algorithm**, named after Volker Strassen, is an algorithm used for matrix multiplication. It is asymptotically faster than the standard matrix multiplication algorithm, but slower than the fastest known algorithm.

**ALGORITHM*:***

Let *A*, *B* be two square matrices over a field *F*. We want to calculate the matrix product *C* as

\mathbf{C} = \mathbf{A} \mathbf{B} \qquad \mathbf{A},\mathbf{B},\mathbf{C} \in F^{2^n \times 2^n}

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

We partition *A*, *B* and *C* into equally sized block matrices

\mathbf{A} = \begin{bmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{bmatrix} \mbox { , } \mathbf{B} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{bmatrix} \mbox { , } \mathbf{C} = \begin{bmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{bmatrix}

with

\mathbf{A}_{i,j}, \mathbf{B}_{i,j}, \mathbf{C}_{i,j} \in F^{2^{n-1} \times 2^{n-1}}

then

\mathbf{C}_{1,1} = \mathbf{A}_{1,1} \mathbf{B}_{1,1} + \mathbf{A}_{1,2} \mathbf{B}_{2,1}

\mathbf{C}_{1,2} = \mathbf{A}_{1,1} \mathbf{B}_{1,2} + \mathbf{A}_{1,2} \mathbf{B}_{2,2}

\mathbf{C}_{2,1} = \mathbf{A}_{2,1} \mathbf{B}_{1,1} + \mathbf{A}_{2,2} \mathbf{B}_{2,1}

\mathbf{C}_{2,2} = \mathbf{A}_{2,1} \mathbf{B}_{1,2} + \mathbf{A}_{2,2} \mathbf{B}_{2,2}

With this construction we have not reduced the number of multiplications. We still need 8 multiplications to calculate the *Ci,j* matrices, the same number of multiplications we need when using standard matrix multiplication.

Now comes the important part. We define new matrices

\mathbf{M}_{1} := (\mathbf{A}_{1,1} + \mathbf{A}_{2,2}) (\mathbf{B}_{1,1} + \mathbf{B}_{2,2})

\mathbf{M}_{2} := (\mathbf{A}_{2,1} + \mathbf{A}_{2,2}) \mathbf{B}_{1,1}

\mathbf{M}_{3} := \mathbf{A}_{1,1} (\mathbf{B}_{1,2} - \mathbf{B}_{2,2})

\mathbf{M}_{4} := \mathbf{A}_{2,2} (\mathbf{B}_{2,1} - \mathbf{B}_{1,1})

\mathbf{M}_{5} := (\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \mathbf{B}_{2,2}

\mathbf{M}_{6} := (\mathbf{A}_{2,1} - \mathbf{A}_{1,1}) (\mathbf{B}_{1,1} + \mathbf{B}_{1,2})

\mathbf{M}_{7} := (\mathbf{A}_{1,2} - \mathbf{A}_{2,2}) (\mathbf{B}_{2,1} + \mathbf{B}_{2,2})

which are then used to express the *C*i,j in terms of *M*k. Because of our definition of the *M*k we can eliminate one matrix multiplication and reduce the number of multiplications to 7 (one multiplications for each *M*k) and express the *C*i,j as

\mathbf{C}_{1,1} = \mathbf{M}_{1} + \mathbf{M}_{4} - \mathbf{M}_{5} + \mathbf{M}_{7}

\mathbf{C}_{1,2} = \mathbf{M}_{3} + \mathbf{M}_{5}

\mathbf{C}_{2,1} = \mathbf{M}_{2} + \mathbf{M}_{4}

\mathbf{C}_{2,2} = \mathbf{M}_{1} - \mathbf{M}_{2} + \mathbf{M}_{3} + \mathbf{M}_{6}

We iterate this division process *n*-times until the submatrices degenerate into numbers.

Practical implementations of Strassen's algorithm switch to standard methods of matrix multiplication for small enough submatrices, for which they are more efficient; the overhead of Strassen's algorithm implies that these "small enough" submatrices are actually quite large, well into thousands of elements.

**ANALYSIS:**

The standard matrix multiplications takes n^3 = n^{\log_{2}8} multiplications of the elements in the field *F*. We ignore the additions needed because, depending on *F*, they can be much faster than the multiplications in computer implementations, especially if the sizes of the matrix entries exceed the word size of the machine.

With the Strassen algorithm we can reduce the number of multiplications ton^{\log_{2}7}\approx n^{2.807}.

The reduction in the number of multiplications however comes at the price at a somewhat reduced numeric stability.

**IMPLEMENTATION:**

#include<iostream.h>

#include<conio.h>

void main()

{int a[2][2],b[2][2],c[2][2],p,q,r,s,t,u,v,i,j;

clrscr();

cout<<"\nEnter the 1st matrix<2X2>:";

for(i=0;i<2;i++)

{for(j=0;j<2;j++)

cin>>a[i][j];

}

cout<<"\nEnter the 2nd matrix<2X2>:";

for(i=0;i<2;i++)

{for(j=0;j<2;j++)

cin>>b[i][j];

}

p=(a[0][0]+a[1][1])\*(b[0][0]+b[1][1]);

q=(a[1][0]+a[1][1])\*b[0][0];

r=a[0][0]\*(b[0][1]-b[1][1]);

s=a[1][1]\*(b[1][0]-b[0][0]);

t=(a[0][0]+a[0][1])\*b[1][1];

u=(a[1][0]-a[0][0])\*(b[0][0]+b[0][1]);

v=(a[0][1]-a[1][1])\*(b[1][0]+b[1][1]);

c[0][0]=p+s-t+v;

c[0][1]=r+t;

c[1][0]=q+s;

c[1][1]=p+r-q+u;

cout<<"\nProduct matrix:\n";

for(i=0;i<2;i++)

{for(j=0;j<2;j++)

cout<<c[i][j]<<" ";

cout<<endl;

}

getch();

}

**OUTPUT:**

Enter the 1st matrix<2X2>:

1 1

1 1

Enter the 2nd matrix<2X2>:

1 1

1 1

Product matrix:

2 2

2 2

1. **PROGRAM TO IMPLEMENT OBST USING DYNAMIC PROGRAMMING:**

Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible. The cost of a BST node is level of that node multiplied by its frequency.

Given:

*pi = prob. of access for Ai (i = 1, n)*

*qi = prob. of access for value between Ai and Ai +1 (i = 0, n)*

*[ p0 = pn+1 = 0]*

*root[i, j] = root of optimal tree on range qi-1 to qj*

*e[i, j] = cost of tree rooted at root[i, j]; this cost is the probability of looking for one of the*

*values (or gaps) in the range times the expected cost of doing in that tree*.

Algorithm computes root’s and e’s by increasing size, i.e. by increasing value of (j-i).

So

*root[i, i ] = i [ Initialization, i is the only key value in the range, so it must be the root]*

*e[i, i ] = qi-1 + pi + qi [the probability of searching in this tree with one internal node]*

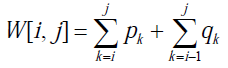
*If r = root; L = left subtree; R = right subtree;*

*W[tree] = probability of being in tree = probability of accessing root.*

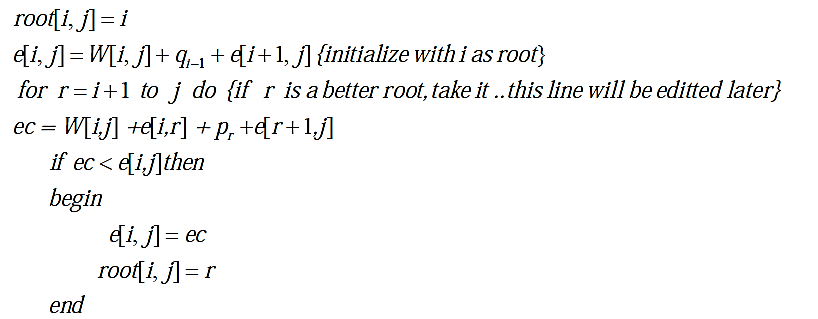
Then *C[tree rooted at r] = W[tree] + e[L] + e[R]*

It will clearly be handy to have W[i,j], the probability of accessing any node *qi−1 ,…, qj* or

*pi,…. , pj*.

So These are easy to compute in O(n2) time by computing

*W [i, j + 1] as W [i, j ] + pj+1 + qj+1.*



The approach of storing computed values and reusing them as required is known as MEMOIZATION, if a value of root[i, j ] or e[i, j ] has been computed ... use it; otherwise compute it and remember it. Hence an O(n3) algorithm as each of the O(n2 ) values takes O(n) time to compute given values on smaller ranges.

**IMPLEMENTATION:**

#include<iostream.h>

#include <stdio.h>

#include <limits.h>

#include<conio.h>

// A utility function to get sum of array elements freq[i] to freq[j]

int sum(int freq[], int i, int j);

/\* A Dynamic Programming based function that calculates minimum cost of

a Binary Search Tree. \*/

int optimalSearchTree(int keys[], int freq[], int n)

{

/\* Create an auxiliary 2D matrix to store results of subproblems \*/

int cost[n][n];

/\* cost[i][j] = Optimal cost of binary search tree that can be

formed from keys[i] to keys[j].

cost[0][n-1] will store the resultant cost \*/

// For a single key, cost is equal to frequency of the key

for (int i = 0; i < n; i++)

cost[i][i] = freq[i];

// Now we need to consider chains of length 2, 3, ... .

// L is chain length.

for (int L=2; L<=n; L++)

{

// i is row number in cost[][]

for (int i=0; i<=n-L+1; i++)

{

// Get column number j from row number i and chain length L

int j = i+L-1;

cost[i][j] = INT\_MAX;

// Try making all keys in interval keys[i..j] as root

for (int r=i; r<=j; r++)

{

// c = cost when keys[r] becomes root of this subtree

int c = ((r > i)? cost[i][r-1]:0) +

((r < j)? cost[r+1][j]:0) +

sum(freq, i, j);

if (c < cost[i][j])

cost[i][j] = c;

}

}

}

return cost[0][n-1];

}

// A utility function to get sum of array elements freq[i] to freq[j]

int sum(int freq[], int i, int j)

{

int s = 0;

for (int k = i; k <=j; k++)

s += freq[k];

return s;

}

// Driver program to test above functions

int main()

{

int i;

int keys[5];

int freq[5];

printf("Enter five keys with their corresponding frequency :\n");

printf("KEY FREQUENCY\n");

for(i=0;i<5;i++)

{

scanf("%d",&keys[i]);

scanf("%d",&freq[i]);

}

int n = sizeof(keys)/sizeof(keys[0]);

printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));

getch();

return 0;

}

**OUTPUT:**

Enter five keys with their corresponding frequency:

KEY FREQUENCY

1 87

2 30

3 56

4 32

5 70

Cost of Optimal BST is 556

1. **PROGRAM TO IMPLEMENT HUFFMAN CODING:**

Huffman coding is an [entropy encoding](http://en.wikipedia.org/wiki/Entropy_encoding) [algorithm](http://en.wikipedia.org/wiki/Algorithm) used for [lossless data compression](http://en.wikipedia.org/wiki/Lossless_data_compression). The term refers to the use of a [variable-length code](http://en.wikipedia.org/wiki/Variable-length_code) table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol. It was developed by [David A. Huffman](http://en.wikipedia.org/wiki/David_A._Huffman) while he was a [Ph.D.](http://en.wikipedia.org/wiki/Doctor_of_Philosophy) student at [MIT](http://en.wikipedia.org/wiki/Massachusetts_Institute_of_Technology).

**ALGORITHM & ANALYSIS:**

The simplest construction algorithm uses a [priority queue](http://en.wikipedia.org/wiki/Priority_queue) where the node with lowest probability is given highest priority:

* Create a leaf node for each symbol and add it to the priority queue.
* While there is more than one node in the queue:
  + Remove the two nodes of highest priority (lowest probability) from the queue
  + Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
  + Add the new node to the queue.
* The remaining node is the root node and the tree is complete.

Since efficient priority queue data structures require O(log n) time per insertion, and a tree with n leaves has 2n−1 nodes, this algorithm operates in O(n log n) time, where n is the number of symbols.

**IMPLEMENTATION:**

#include <stdio.h>

#include <stdlib.h>

// This constant can be avoided by explicitly calculating height of Huffman Tree

#define MAX\_TREE\_HT 100

// A Huffman tree node

struct MinHeapNode

{

    char data;  // One of the input characters

    unsigned freq;  // Frequency of the character

    struct MinHeapNode \*left, \*right; // Left and right child of this node

};

// A Min Heap:  Collection of min heap (or Hufmman tree) nodes

struct MinHeap

{

    unsigned size;    // Current size of min heap

    unsigned capacity;   // capacity of min heap

    struct MinHeapNode \*\*array;  // Attay of minheap node pointers

};

// A utility function allocate a new min heap node with given character

// and frequency of the character

struct MinHeapNode\* newNode(char data, unsigned freq)

{

    struct MinHeapNode\* temp =

          (struct MinHeapNode\*) malloc(sizeof(struct MinHeapNode));

    temp->left = temp->right = NULL;

    temp->data = data;

    temp->freq = freq;

    return temp;

}

// A utility function to create a min heap of given capacity

struct MinHeap\* createMinHeap(unsigned capacity)

{

    struct MinHeap\* minHeap =

         (struct MinHeap\*) malloc(sizeof(struct MinHeap));

    minHeap->size = 0;  // current size is 0

    minHeap->capacity = capacity;

    minHeap->array =

     (struct MinHeapNode\*\*)malloc(minHeap->capacity \* sizeof(struct MinHeapNode\*));

    return minHeap;

}

// A utility function to swap two min heap nodes

void swapMinHeapNode(struct MinHeapNode\*\* a, struct MinHeapNode\*\* b)

{

    struct MinHeapNode\* t = \*a;

    \*a = \*b;

    \*b = t;

}

// The standard minHeapify function.

void minHeapify(struct MinHeap\* minHeap, int idx)

{

    int smallest = idx;

    int left = 2 \* idx + 1;

    int right = 2 \* idx + 2;

    if (left < minHeap->size &&

        minHeap->array[left]->freq < minHeap->array[smallest]->freq)

      smallest = left;

    if (right < minHeap->size &&

        minHeap->array[right]->freq < minHeap->array[smallest]->freq)

      smallest = right;

    if (smallest != idx)

    {

        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);

    }

}

// A utility function to check if size of heap is 1 or not

int isSizeOne(struct MinHeap\* minHeap)

{

    return (minHeap->size == 1);

}

// A standard function to extract minimum value node from heap

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap)

{

    struct MinHeapNode\* temp = minHeap->array[0];

    minHeap->array[0] = minHeap->array[minHeap->size - 1];

    --minHeap->size;

    minHeapify(minHeap, 0);

    return temp;

}

// A utility function to insert a new node to Min Heap

void insertMinHeap(struct MinHeap\* minHeap, struct MinHeapNode\* minHeapNode)

{

    ++minHeap->size;

    int i = minHeap->size - 1;

    while (i && minHeapNode->freq < minHeap->array[(i - 1)/2]->freq)

    {

        minHeap->array[i] = minHeap->array[(i - 1)/2];

        i = (i - 1)/2;

    }

    minHeap->array[i] = minHeapNode;

}

// A standard funvtion to build min heap

void buildMinHeap(struct MinHeap\* minHeap)

{

    int n = minHeap->size - 1;

    int i;

    for (i = (n - 1) / 2; i >= 0; --i)

        minHeapify(minHeap, i);

}

// A utility function to print an array of size n

void printArr(int arr[], int n)

{

    int i;

    for (i = 0; i < n; ++i)

        printf("%d", arr[i]);

    printf("\n");

}

// Utility function to check if this node is leaf

int isLeaf(struct MinHeapNode\* root)

{

    return !(root->left) && !(root->right) ;

}

// Creates a min heap of capacity equal to size and inserts all character of

// data[] in min heap. Initially size of min heap is equal to capacity

struct MinHeap\* createAndBuildMinHeap(char data[], int freq[], int size)

{

    struct MinHeap\* minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)

        minHeap->array[i] = newNode(data[i], freq[i]);

    minHeap->size = size;

    buildMinHeap(minHeap);

    return minHeap;

}

// The main function that builds Huffman tree

struct MinHeapNode\* buildHuffmanTree(char data[], int freq[], int size)

{

    struct MinHeapNode \*left, \*right, \*top;

    // Step 1: Create a min heap of capacity equal to size.  Initially, there are

    // modes equal to size.

    struct MinHeap\* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1

    while (!isSizeOne(minHeap))

    {

        // Step 2: Extract the two minimum freq items from min heap

        left = extractMin(minHeap);

        right = extractMin(minHeap);

        // Step 3:  Create a new internal node with frequency equal to the

        // sum of the two nodes frequencies. Make the two extracted node as

        // left and right children of this new node. Add this node to the min heap

        // '$' is a special value for internal nodes, not used

        top = newNode('$', left->freq + right->freq);

        top->left = left;

        top->right = right;

        insertMinHeap(minHeap, top);

    }

    // Step 4: The remaining node is the root node and the tree is complete.

    return extractMin(minHeap);

}

// Prints huffman codes from the root of Huffman Tree.  It uses arr[] to

// store codes

void printCodes(struct MinHeapNode\* root, int arr[], int top)

{

    // Assign 0 to left edge and recur

    if (root->left)

    {

        arr[top] = 0;

        printCodes(root->left, arr, top + 1);

    }

    // Assign 1 to right edge and recur

    if (root->right)

    {

        arr[top] = 1;

        printCodes(root->right, arr, top + 1);

    }

    // If this is a leaf node, then it contains one of the input

    // characters, print the character and its code from arr[]

    if (isLeaf(root))

    {

        printf("%c: ", root->data);

        printArr(arr, top);

    }

}

// The main function that builds a Huffman Tree and print codes by traversing

// the built Huffman Tree

void HuffmanCodes(char data[], int freq[], int size)

{

   //  Construct Huffman Tree

   struct MinHeapNode\* root = buildHuffmanTree(data, freq, size);

   // Print Huffman codes using the Huffman tree built above

   int arr[MAX\_TREE\_HT], top = 0;

   printCodes(root, arr, top);

}

// Driver program to test above functions

int main()

{

    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};

    int freq[] = {5, 9, 12, 13, 16, 45};

    int size = sizeof(arr)/sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;

}

**OUTPUT:**

f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111

1. **PROGRAM TO IMPLEMENT ACTIVITY SELECTION PROBLEM:**

An activity-selection is the problem of scheduling a resource among several competing activity.  
**Problem Statement**

Given a set *S* of *n* activities with and start time, *Si*and *fi*, finish time of an ith activity. Find the maximum size set of mutually compatible activities.

**Compatible Activities**

Activities *i* and *j* are compatible if the half-open internal [*si, fi*) and [*sj, fj*) do not overlap, that is, *i* and *j* are compatible if *si* ≥ *fj*and *sj*≥ *fi*

GREEDY ALGORITHM FOR SELECTION PROBLEM

I.     Sort the input activities by increasing finishing time.  
*f1* ≤  *f2* ≤  . . . ≤*fn*

II.    Call **GREEDY-ACTIVITY-SELECTOR** (s, f)

*n* = length [*s*]

*A*={*i*}

*j* = 1

**for** *i* = 2 **to**  n

**do if**  *si* ≥ *fj*

**then**  A= AU{*i*}

*j* = *i*

**return** *set A*

**Analysis**

Part I requires O*(n lg n)* time (use merge of heap sort).  
Part II requires θ*(n)* time assuming that activities were already sorted in part I by their finish time.

**IMPLEMENTATION**

#include<iostream.h>

#include<conio.h>

main()

{

int s[10],f[10],A[10],n,i,j,k=1;

cout<<"\nEnter the no. of activities( max 10):";

cin>>n;

cout<<"\nEnter the start and finishing time of each activity";

cout<<"\n NOTE: In increasing order of finishing time\n";

for(i=0;i<n;i++)

{

cin>>s[i];

cin>>f[i];

}

cout<<"\n ACTIVITY TABLE is:\n ";

for(i=0;i<n;i++)

{

cout<<i+1<<"\t";

}

cout<<"\nStart :\t";

for(i=0;i<n;i++)

{

cout<<s[i]<<"\t";

}

cout<<"\nFinish:\t";

for(i=0;i<n;i++)

{

cout<<f[i]<<"\t";

}

A[0]=0; //Taking first activity by default

i=0;

for(j=1;j<n;j++)

{

if(s[j]>=f[i])

{

A[k]=j;

i=j;

k++;

}

}

cout<<"\nACTIVITY SELECTED ARE:\t";

for(j=0;j<k;j++)

{

cout<<(A[j]+1)<<"\t";

}

getch();

return 0;

}

**OUTPUT:**

Enter the no. of activities( max 10): 10

Enter the start and finishing time of each activity

NOTE: in increasing order of finishing time

2 3

3 4

1 5

4 6

5 7

6 8

7 11

9 12

10 13

12 14

ACTIVITY TABLE is:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Start : | 2 | 3 | 1 | 4 | 5 | 6 | 7 | 9 | 10 | 12 |
| Finish: | 3 | 4 | 5 | 6 | 7 | 8 | 11 | 12 | 13 | 14 |

ACTIVITY SELECTED are: 1 2 4 6 8 10

1. **PROGRAM TO IMPLEMENT KNAPSACK PROBLEM:**

The **knapsack problem** is a problem in combinatorial optimization. It derives its name from the maximization problem of choosing possible essentials that can fit into one bag (of maximum weight) to be carried on a trip. Given a set of items, each with a cost and a value, then determine the number of each item to include in a collection so that the total cost is less than some given cost and the total value is as large as possible.

GREEDY APPROXIMATION ALGORITHM

Martello and Toth (1990) proposed a greedy approximation algorithm to solve the knapsack problem. Their version sorts the essentials in decreasing order and then proceeds to insert them into the sack, starting from the first element (the greatest) until there is no longer space in the sack for more. If *k* is the maximum possible number of essentials that can fit into the sack, the greedy algorithm is guaranteed to insert at least *k*/2 of them.

DYNAMIC PROGRAMMING FOR 0-1 KNAPSACK PROBLEM

Mathematically the 0-1-knapsack problem can be formulated as:

Let there be n items, x_1 to x_n where x_i has a value v_i and weight w_i. The maximum weight that we can carry in the bag is *W*. It is common to assume that all values and weights are nonnegative. To simplify the representation, we also assume that the items are listed in increasing order of weight.

* *Maximize \qquad \sum_{i=1}^n v_ix_i subject to \qquad \sum_{i=1}^n w_ix_i \leqslant W, \quad \quad x_i \in \{0,1\}*

*Maximize the sum of the values of the items in the knapsack so that the sum of the weights must be less than the knapsack's capacity.*

*The****bounded knapsack problem****removes the restriction that there is only one of each item, but restricts the number x_i of copies of each kind of item to an integer value c_i.*

*Mathematically the bounded knapsack problem can be formulated as:*

* *maximize \qquad \sum_{i=1}^n v_ix_i subject to \qquad \sum_{i=1}^n w_ix_i \leqslant W, \quad \quad x_i \in \{0,1,\ldots,c_i\}*

*The****unbounded knapsack problem****(****UKP****) places no upper bound on the number of copies of each kind of item and can be formulated as above except for that the only restriction on x_i is that it is a non-negative integer. If the example with the colored bricks above is viewed as an unbounded knapsack problem, then the solution is to take three yellow boxes and three grey boxes.*

*Assume w_1,\,w_2,\,\ldots,\,w_n,\,W are strictly positive integers. Define m[i,w] to be the maximum value that can be attained with weight less than or equal to w using items up to i.*

*We can define m[i,w] recursively as follows:*

* *m[i,\,w]=m[i-1,\,w] if w_i > w\,\! (the new item is more than the current weight limit)*
* *m[i,\,w]=\max(m[i-1,\,w],\,m[i-1,w-w_i]+v_i) if w_i \leqslant w.*

The solution can then be found by calculating m[n,W]. To do this efficiently we can use a table to store previous computations.

The following is pseudo code for the dynamic program:

**Input:**  
Values (stored in array v)  
Weights (stored in array w)  
Number of distinct items (n)  
Knapsack capacity (W)

**for** w **from** 0 **to** W **do**

m[0, w] := 0

**end for**

**for** i **from** 1 **to** n **do**

**for** j **from** 0 **to** W **do**

**if** j >= w[i] **then**

m[i, j] := max(m[i-1, j], m[i-1, j-w[i]] + v[i])

**else**

m[i, j] := m[i-1, j]

**end if**

**end for**

**end for**

This solution will therefore run in O(nW) time and O(nW) space. Additionally, if we use only a 1-dimensional array m[w] to store the current optimal values and pass over this array i+1 times, rewriting from m[W] to m[1] every time, we get the same result for only O(W) space.

**IMPLEMENTATION:**

#include<stdio.h>

#include<conio.h>

// A utility function that returns maximum of two integers

int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

int i, w;

int K[n+1][W+1];

// Build table K[][] in bottom up manner

for (i = 0; i <= n; i++)

{

for (w = 0; w <= W; w++)

{

if (i==0 || w==0)

K[i][w] = 0;

else if (wt[i-1] <= w)

K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w]);

else

K[i][w] = K[i-1][w];

}

}

return K[n][W];

}

int main()

{

int val[10];

int wt[10];

int W = 50;

int n,i;

printf("\nEnter no. of items :");

scanf("%d",&n);

printf("\nEnter the capacity of KNAPSACK :");

scanf("%d",&W);

printf("\nEnter the weights of items :");

for(i=0;i<n;i++)

{

scanf("%d",&wt[i]);

}

printf("\nEnter the value of items :");

for(i=0;i<n;i++)

{

scanf("%d",&val[i]);

}

printf("\nOPTIMAL VALUE OF KNAPSACK :");

printf("%d", knapSack(W, wt, val, n));

getch();

return 0;

}

**OUTPUT:**

Enter no. of items : 3

Enter the capacity of KNAPSACK : 50

Enter the weights of items : 14 15 30

Enter the value of items : 19 25 40

OPTIMAL VALUE OF KNAPSACK : 65