1) Binary Search

Algorithm

BinarySearch\_Right(A[0..N-1], value, low, high) {

      // invariants: value >= A[i] for all i < low

                     value < A[i] for all i > high

      if (high < low)

          return low

      mid = low +((high – low) / 2)

      if (A[mid] > value)

          return BinarySearch\_Right(A, value, low, mid-1)

      else

          return BinarySearch\_Right(A, value, mid+1, high)

  }

Complexity

The maximum number of comparisons is logarithmic with respect to the number of items in the list. Therefore, the binary search is *O*(log*n*).

2) QuickSort

Algorithm

quickSort(alist,first,last):

if first<last:

splitpoint = partition(alist,first,last)

quickSort(alist,first,splitpoint-1)

quickSort(alist,splitpoint+1,last)

partition(alist,first,last):

pivot = alist[first]

leftmark = first+1

rightmark = last

done=false

while(!done)

while leftmark <= rightmark

leftmark = leftmark + 1

while alist[rightmark] >= pivot

rightmark = rightmark -1

if leftmark <= rightmark:

swap(alist[leftmark], alist[rightmark])

done=true

swap(alist[first], alist[rightmark])

Complexity

Average running time O(NlogN)

Worst-case running time O(N2)

3) MergeSort

Algorithm

MergeSort(arr[], l, r)

If r > l

middle m = (l+r)/2

Call mergeSort(arr, l, m)

Call mergeSort(arr, m+1, r)

Call merge(arr, l, m, r)

MERGE (A, p, q, r )

      n1 ← q − p + 1  
      n2 ← r − q  
      Create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]  
      **FOR** i ← 1 **TO** n1  
            L[i] ← A[p + i − 1]  
      **FOR** j ← 1 **TO** n2  
            R[j] ← A[q + j ]  
      L[n1 + 1] ← ∞  
      R[n2 + 1] ← ∞  
    i ← 1  
    j ← 1  
    **FOR** k ← p **TO** r  
         **IF** L[i ] ≤ R[ j]  
                **THEN** A[k] ← L[i]  
                        i ← i + 1  
                **ELSE** A[k] ← R[j]  
                        j ← j + 1

Complexity

Best case – When the array is already sorted O(nlogn).

Worst case – When the array is sorted in reverse order O(nlogn).

4) MinMax

Algorithm

Algorithm MaxMin(i, j, max, min)

if (i=j) then

max=min=a(i)

else

if (i=j -1) then

if (a(i)<a(j)) then

max= a(j)

min= a(i)

else

max= a(i)

min= a(j)

else

mid=(i+j)/2

maxmin(i, mid, max, min)

maxmin(mid+1, j, max1, min1)

if (max<max1) then max = max1

if (min>min1) then min = min1

end

Complexity

The worst time complexity will be T(n)= O(n) and best case time complexity will be O(1) when you have only one element in array, which will be candidate for both max and min.

5) Knapsack Greedy

Algorithm

Greedy-fractional-knapsack (*w, v, W*)

FOR *i* =1 to *n*  
    do *x*[*i*] =0  
weight = 0  
while weight < *W*  
    do *i* = best remaining item  
        IF weight + *w*[*i*] ≤ *W*  
            then *x*[*i*] = 1  
                weight = weight + *w*[*i*]  
            else  
*x*[*i*] = (*w* - weight) / *w*[*i*]  
                weight = *W*  
return *x*

 Complexity

If the items are already sorted into decreasing order of *vi / wi,* then the while-loop takes a time in *O(n)*; Therefore, the total time including the sort is in *O(n log n)*.

6) Knapsack 0/1

Algorithm

knapSack(W , wt , val , n)

{

K[n+1][W+1]

For I=0 to n

For w=0 to w

   if (i==0 || w==0)

        C[i][w] = 0

   else if (wt[i-1] <= w)

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

   else

        C[i][w] = C[i-1][w]

return K[n][W];

}

KnapSack Find()

C[n,W] is the maximal value of items that can be placed in the Knapsack.

Let i=n and k=W

if C[i,k] ≠C[i−1,k] then

mark the ith item as in the knapsack

i = i−1,

k= k-w

else

i = i−1

}

Complexity

This dynamic-0-1-kanpsack algorithm takes θ(*nw*) times, broken up as follows: θ(*nw*) times to fill the *c*-table, which has (*n* +*1*).(*w* +1) entries, each requiring θ(1) time to compute. *O*(*n*) time to trace the solution, because the tracing process starts in row *n* of the table and moves up 1 row at each step.

6) Huffman Encoding

Algorithm

Huffman (C)

n = the size of C

insert all the elements of C into T,using the value of the node as the priority

for i in 1..n-1 do

z = a new tree node

x = Extract-Minimum value (T)

y = Extract-Minimum value(T)

left node of z = x

right node of z = y

f[z] = f[x] + f[y]

Insert z in T

end for

return the complete tree

Complexity

For sorted cases the complexity is O(nlog(n))

7) Kuruskals

Algorithm

Kruskal(G)

{ Sort E in non decreasing order of the edge weights w(ei1)<=…….>= w(ei |E|))

ET ←φ ;

ecounter ← 0 k ← 0

while ecounter <|V|-1

do

{ k ← k+1

if ETU {eik} is acyclic

ET ← ET U {eik};

ecounter ← ecounter+1

}

return ET }

Complexity: With an efficient sorting algorithm, the time efficiency of kruskal‟s algorithm will be in O(|E| log |E)

8) Multi-forward

Algorithm

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]];

}

Complexity

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

9) Multi-Back

Algorithm

Algorithm BGraph(G,k,n,p)

. // Same function as FGraph

{

bcost[1] :=0.0; <r,j>

for j :=2 to n do

{ / / Compute bcost[j].

Let r be such that is an edge of G is an edge of and bcost is an edge of

G and bcost[r] + c[r,j] is minimum;

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

d[j] := r;

}

/ / Find a minimum-cost path.

p[1] := 1; p[k] :=n;

for j := k – 1 to 2 do p[j] := d[p[j+1]];

}

Complexity

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

10)N-Queens

**Algorithm place (k,I)**

//return true if a queen can be placed in k th row and I th column. otherwise it returns false .

X[] is a global array whose first k-1 values have been set

{

for j=1 to k-1 do

if ((X [j]=I) Or (abs (X [j]-I)=Abs (j-k))) then

return false;

return true;

}

**Algorithm Nqueen (k,n)**

//using backtracking it prints all possible positions of n queens in ‘n\*n’ chessboard. So

//that they are non-tracking.

{

for I=1 to n do

{

if place (k,I) then

{

X [k]=I;

if (k=n) then write (X [1:n]);

else nquenns(k+1,n) ;

}

}

}

Complexity

The complexity for all cases is O(n!)

11) Graph Coloring

**Algorithm:**

**Algorithm mColoring(k)**

// the graph is represented by its Boolean adjacency matrix G[1:n,1:n] .All assignments //of 1,2,……….,m to the vertices of the graph such that adjacent vertices are assigned //distinct integers are printed. ’k’ is the index of the next vertex to color.

{

repeat

{

// generate all legal assignment for X[k].

Nextvalue(k); // Assign to X[k] a legal color.

If (X[k]=0) then return; // No new color possible.

If (k=n) then // Almost ‘m’ colors have been used to color the ‘n’ vertices

Write(x[1:n]);

Else mcoloring(k+1);

}until(false);

}

**Algorithm Nextvalue(k)**

//\*X[1],……X[k-1] have been assigned integer values in the range[1,m] such that adjacent values have distinct integers.

A value for X[k] is determined in the range[0,m].X[k] is assigned the next highest numbers color while maintaining distinctness form the adjacent vertices of vertex K. If no such color exists, then X[k] is 0.

\*/

{ repeat

{

X[k] = (X[k]+1)mod(m+1); // next highest color.

If(X[k]=0) then return; //All colors have been used.

For j=1 to n do

{

// Check if this color is distinct from adjacent color.

If((G[k,j] 0)and(X[k] = X[j])) 

// If (k,j) is an edge and if adjacent vertices have the same color.

Then break;

}

if(j=n+1) then return; //new color found.

} until(false); //otherwise try to find another color.

}

Complexity

The time spent by Nextvalue to determine the children is (mn)

Total time is = (mn n).

M colors and N vertices

12) Sum of Subsets

Algo

sumOfSubsets ( *i*, *weightSoFar*, *totalPossibleLeft* )

**if** (promising ( *i* )) //may lead to solution

**then if** ( *weightSoFar ==* S) **then** print *include*[ 1 ] to *include*[ *i* ] //found solution

**else** //expand the node when *weightSoFar* < S

include [ *i* + 1 ] = "yes” //try including  
 sumOfSubsets ( *i* + 1,*weightSoFar + w*[*i* + 1],t*otalPossibleLeft* - *w*[*i* + 1] )

include [ *i* + 1 ] = "no” //try excluding  
 sumOfSubsets ( *i* + 1, *weightSoFar* , *totalPossibleLeft* - *w*[*i* + 1] )

boolean promising (*i* )  
 return ( *weightSoFar* + *totalPossibleLeft* ≥S) && ( *weightSoFar ==* S *||*  *weightSoFar +*  *w*[*i* + 1] ≤ *S* )

Prints all solutions

Complexity :

The Complexity is O(2^n)

13) Hiring Problem

Algorithm

HIRE ASSISTANT(a,n)

Best->0

For 1 <- to n from a

Do interview candidate i

If candidate i is better than candidate best

Then best<-i

Hire candidate i

Randomize(a)

N<-length[a]

For i <- 1 to n

Do swap a[i] <-> a[random(i,n)]

a <-candidates\_inputs

Hire normally

HIRE ASSISTANT(a,n)

Randomize(a)

HIRE ASSISTANT(a,n)

Complexity :

E[total cost]= for all n E[(fee to be paid for i-th candidate)]

=f. ln n