Algorithms

# Sorting

**Insertion-Sort**

Insertion-Sort(A):

for(j = 2 to A.length)

key = A[j]

i = j-1

while(i > 0 and A[i] > key)

A[i+1] = A[i]

i--

A[i+1] = key

At the start of each iteration of the for loop the subarray A[1...j-1] consists of the elements originally in A[1...j-1] but in sorted order

Worst case: O(n^2)

Best case: O(n)

efficient for sorting a small number of elements

**Bubble-Sort**

Bubble-sort(A):

for i = 1 to A.length - 1

xchanges = 0

for j = 1 to A.length -1 - i

if A[j] > A[j+1]

swap A[j] with A[j+1]

xchanges++

if xchanges == 0

break

With each iteration of the for loop the ith largest element will be moved to (*n - i)th* index, so subarray A[n-i...n] is in sorted order.

Best-case: O(n)

Worst-case: O(n^2)

**Selection-Sort**

Selection-sort(A):

n ← length(A)

for j = 1 to n-1

smallest ← j

for i = j+1 to n

if A[i] < A[smallest]

smallest = i

if (smallest != j)

swap A[j] with A[smallest]

With each iteration of the for loop the *jth*smallest element will be moved to *jth* index, so subarray A[0...j] is in sorted order.

Best-case: O(n^2)

Worst-case: O(n^2)

**Merge-Sort**

Divide & Conquer approach

* break the problem into several sub-problems that are similar to the original problem but smaller in size.
* solve the sub-problems recursively.
* combine these solutions to create a solution to the original problem.

**Divide ->** the problem into a number of sub-problems(2 or more) that are smaller instances of the same problem

**Conquer ->** the sub-problems by solving them recursively. However if the sub-problem sizes are small enough just solve the sub-problems in a straight-forward manner(brute force). (Smallest possible size of a sub-problem is the base case).

**Combine ->** the solutions of the sub-problems into the solution for the original problem.

Divide-Conquer(p):

if small(p)

return solve(p)

else

divide p into smaller instances p1,p2,...

apply Divide-Conquer to each of these subproblems

return combine (Divide-Conquer(p1), Divide-Conquer(p2),...)

Merge-Sort Algorithm

**Divide ->** divide the n-element sequence to be sorted into two sub-sequences of n/2 elements each.

**Conquer ->** sort the two sub-sequences recursively using merge-sort.

**Combine ->** merge the two sorted sub-sequences to produce the sorted answer.

Merge-Sort(A, p, r):

if (p < r)

q = (p + r) / 2 // divide

Merge-Sort(A, p, q) // conquer

Merge-Sort(A, q+1, r) // conquer

Merge(A, p, q, r) // combine

Merge(A, p, q, r):

n1 = q - p + 1

n2 = r - q

let 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 and j = 1

for k = p to r

if L[i] <= R[j]

A[k] = L[i]

i++

else

A[k] = R[j]

j++

Time-Complexity for divide and conquer

Suppose that our division of the problem yields ***a*** sub-problems, each of which is ***1/b*** the size of the original.

It takes T(***n/b***) to solve one sub-problem of size ***n/b***, so it takes ***a***T(***n/b***) to solve ***a*** of them.

If we take D(n) time to divide the problem into subproblems and C(n) time to combine the solution of the sub-problems into the solution to the original problem.

T(n) = ***a***T(***n/b***) + D(n) + C(n)

★ T(n) = ***a***T(***n/b***) + O(n^k) a >= 1, b > 1, k >= 0

* if a < b^k T(n) = O(n^k)
* if a = b^k T(n) = O(n^k \* log(n))
* if a > b^k T(n) = O(n^logba)

for Merge-sort: T(n) = 2\*T(n/2) + O(n)

a = 2, b = 2, k = 1 a = b^k so time-complexity T(n) = O(n^1 \* log(n))

**Quick-Sort**

**Divide ->** partition the array A[p..r] into two subarrays A[p...q-1] and A[q+1...r] such that each element of A[p...q-1] is less than or equal to A[q] which in turn less than or equal to A[q+1...r].

(Compute the index q as part of this partitioning procedure).

**Conquer ->** sort the two subarrays A[p...q-1] and A[q+1...r] by recursive calls to quick-sort.

**Combine ->** no work is needed to combine the subarrays, the entire array A[p...r] is already sorted.

Quick-Sort(A, p, r):

if (p < r):

q = Partition(A, p, r) // divide

Quick-Sort(A, p, q-1) // conquer

Quick-Sort(A, q+1, r) // conquer

Partition(A, p, r):

pivot = A[r]

i = p

for j = p to r-1

if A[j] <= pivot

swap A[i] with A[j]

i++

swap A[i] with A[r]

return i

Partition function moves element at *rth* index in subarray A[p...r] to its correct position in sorted order. Instead of picking the last index(*rth)* we can pick any index between p and r and swap it with the *rth* element.

Randomized-Partition(A, p, r):

i = Random(p, r)

swap A[i] with A[r]

return Partition(A, p, r)

Average-case: O(n \* log2n)

Worst-case: O(n^2)

Partition function use case => select ith order statistic from a set of n distinct numbers.

Randomized-Select returns the ith smallest element of array A[p..r]

Randomized-Select(A, p, r, i):

if (p == r)

return A[p]

q = Randomized-Partition(A, p, r) // divide

k = q - p + 1

if (i == k)

return A[q] // conquer

else if (i < k)

return Randomized-Select(A, p, q-1, i) // conquer

else

return Randomized-Select(A, q+1, r, i-k) // conquer

**Counting-Sort**

Sort n elements in range 0-k

A[1...n] Input array

B[1...n] holds the sorted output

C[0...k] provides temporary working storage space.

Counting-Sort(A, B, k):

let C[0...k] be new array with every element initialized as 0

for i = 1 to A.length

C[A[i]] = C[A[i]] + 1 // C[i] contains number of elements equal to i

for i = 1 to k

C[i] = C[i] + C[i-1] // C[i] contains number of elements less than or

// equal to i

for i = A.length down-to 1

B[C[A[i]]] = A[i]

C[A[i]]--

Time-Complexity: O(n+k)

**Radix-Sort**

Assume each element in the n-element array has d digits where digit *1* is the lowest order digit & digit *d* is the highest order digit.

Radix-Sort(A, d):

for i = 1 to d

use a stable sort to sort array A on digit i

if we do d passes of counting sort Time-Complexity: O(d\*(n+k))

Radix-Sort(A, n):

max\_element = maximum element of array A

let C[0..9] be new array

d = 0

while(max\_element)

max\_element /= 10

d++

mod = 10, div = 1

for i = 0 to d-1

C[0..9] = {0}

for j = 0 to n-1

C[(A[j]%mod) / div]++

for j = 1 to 9

C[i] = C[i] + C[i-1]

for j = n-1 to 0

B[C[(A[j]%mod) / div]]] = A[j]

C[(A[j]%mod) / div]]--

div = div\*10

mod = mod\*10

**Heap-Sort**

heap -> complete binary tree

parent(i) = [i/2] // parent of ith node

left(i) = 2\*i // left child of ith node

right(i) = 2\*i+1 // right child of ith node

max-heap: A[parent(i)] >= A[i]

min-heap: A[parent(i)] <= A[i]

Max-heapify: key to maintaining max-heap property O(log2n)

Build-Max-heap: produces a max heap from an unordered input array O(n)

Heap-Sort: sorts an array inplace O(n\*log2n)

*Maintaining the heap property*

Max-heapify(A, i):

l = left(i)

r = right(i)

if (l <= A.heap-size and A[l] > A[i])

largest = l

else

largest = i

if (r <= A.heap-size and A[r] > A[largest])

largest = r

if (largest != i)

swap A[i] with A[largest]

Max-heapify(A, largest)

*Building a heap*

we use the procedure Max-heapify in a bottom-up manner to convert an array A[1...n] into a max-heap.

the elements in the subarray A[[n/2]+1...n] are all leaves of the tree.

Build-Max-heap(A):

A.heap-size = A.length

for i = [A.length/2] down-to 1

Max-heapify(A, i)

*Heap Sort*

Heap-Sort(A):

Build-Max-heap(A)

for i = A.length down-to 2

swap A[1] with A[i]

A.heap-size = A.heap-size - 1

Max-heapify(A, 1)

# Backtracking

Naive Solution => trying all configurations(permutations) & output a configuration that follows given problem constraints.

Why?

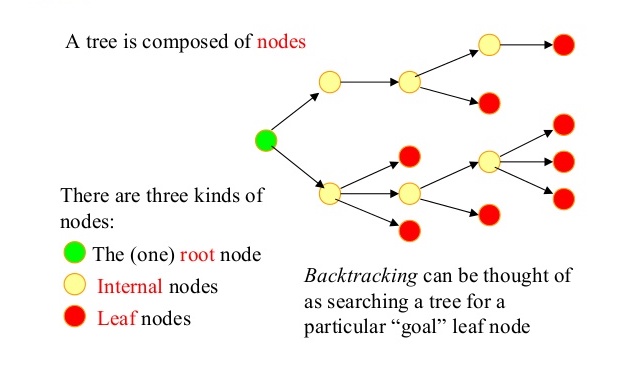
**Backtracking** works in an incremental way and is an optimization over naive solutions where all possible configurations are generated and tried. (way of trying out various sequences of decisions, until we find the one that “works”).

How?

* Typically we start from an empty solution vector & one by one add items.
* When we add an item we check if adding the current item violates the problem constraint, if it does then we remove the item & try other alternatives. If none of the alternatives work then we go to the previous stages & remove the item added in previous stages. If we reach the initial stage back then no solution exists.
* If adding an item doesn’t violate constraints then we recursively add items one by one. If the solution vector becomes complete then we print the solution.

When?

* we have to make a series of decisions among various choices.
* we don’t have much information to know what to choose.
* each decision leads to a new set of choices.
* some sequence of choices(possibly more than one) may be a solution to our problem



**To "explore" node N:**

1) If N is a goal node, return "success"

2) If N is a leaf node, return "failure"

3) For each child C of N,

3.1) Explore C

3.1.1) If C was successful, return "success"

4) Return "failure"

***Backtracking algorithm for Knight’s tour***

Knight’s tour: sequence of moves of a knight on a chessboard such that the knight visits every square exactly once

if all squares are visited

print the solution

else

a) Add one of the next moves to the solution vector and recursively check if this move leads to a solution. (A Knight can make a maximum of 8 moves. We choose one of the 8 moves in this step).

b) If the move chosen in the above step doesn't lead to a solution then remove this move from the solution vector and try other alternative moves.

c) If none of the alternatives work then return false (Returning false will remove the previously added item in recursion and if false is returned by the initial call of recursion then "no solution exists" )

bool isSafe(int x, int y, int sol[N][N]):

if (x >= 0 and x < N and y >= 0 and y < N and sol[x][y] == -1) return true

return false

int sol[N][N]

sol[i][j] = -1 ∀(i,j)

int x\_move[8] = {2, 1, -1, -2, -2, -1, 1, 2}

int y\_move[8] = {1, 2, 2, 1, -1, -2, -2, -1}

// x\_move[8], y\_move[8] defines next move of knight

// knight is initially at first block

sol[0][0] = 0

solveKnightTour(0, 0, 1)

bool solveKnightTour(int x, int y, int move):

if (move == N\*N) return true

for k = 0 to 7:

next\_x = x + x\_move[k]

next\_y = y + y\_move[k]

if isSafe(next\_x, next\_y, sol):

sol[next\_x][next\_y] = move

if solveKnightTour(next\_x, next\_y, move+1) return true

else sol[next\_x][next\_y] = -1 // backtrack

return false

***Backtracking algorithm for Rat in a maze***

Maze: N\*N binary matrix maze[i][j] is 0 for open cell and 1 for blocked cell

find if possible to move from source(0,0) to destination(n-1, n-1)

if destination is reached

print the solution

else

a) Mark current cell in solution matrix as 1. Add one of the next moves to the solution vector and recursively check if this move leads to a solution. (A rat can make 4 moves - horizontally and vertically. We choose one of the 4 moves in this step).

b) If the move chosen in the above step doesn't lead to a solution then remove this move from the solution vector and try other alternative moves.

c) If none of the alternatives work then return false (Returning false will remove the previously added item in recursion and if false is returned by the initial call of recursion then "no solution exists" )

bool isSafe(int x, int y, int maze[N][N]):

if (x >= 0 and x < N and y >= 0 and y < N and maze[x][y] == 0) return true

return false

int x\_move[4] = {1, 0, -1, 0}

int y\_move[4] = {0, 1, 0, -1}

// x\_move[4], y\_move[4] defines next move of rat

// rat is initially at source block

solveMaze(0, 0)

bool solveMaze(int x, int y):

if (x == N-1 and y == N-1) return true

for k = 0 to 3:

next\_x = x + x\_move[k]

next\_y = y + y\_move[k]

if isSafe(next\_x, next\_y, maze):

maze[next\_x][next\_y] = 1

if solveMaze(next\_x, next\_y) return true

maze[next\_x][next\_y] = 0 // backtrack

return false

# Dynamic Programming

Dynamic Programming like divide & conquer solves the problem by combining the solutions to subproblems.

When?

We apply DP to *optimization problems.* Such problems can have many possible answers, each answer has a value & we wish to find an answer with optimal value(min or max).

*Optimal Substructure Property:* Once we divide the problem into subproblems each subproblem is considered as an independent instance of the main problem. (Divide & Conquer also).

*Overlapping Subproblems:* Recalculate the same problem twice or more.

How?

Two ways to implement DP

* Top-down with memoization: In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in array or hash table). procedure first checks whether it has previously solved this subproblem, if so return the saved value if not procedure computes the value in usual manner.
* Bottom-up: We sort the subproblems by size & solve them in size-order, smallest first. When solving a particular subproblem, we have already solved all the smaller subproblems its solution depends upon & we have saved their solutions.

***Rod-cutting problem***

rod of length n inches

price pi for i = 1,2...n-1 // pi cutting rod of length i

determine maximum revenue rn obtained by cutting the road

rn = max(pn(making no cuts at all), p1 + rn-1, p2 + rn-2 … pn-1 + r1)

maximum revenue obtained by making an initial cut of the rod into two pieces of size i and n-i for i = 1, 2 … n-1 and then optimally cutting up those pieces further. Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem.

rn = max(pi + rn-i) ∀ 1 ≤ i ≤ n

*Recursive top-down implementation*

Cut-rod(p, n):

if (n == 0)

return 0

q = -∞

for i = 1 to n-1

q = max(q, p[i] + Cut-rod(p, n-i)

return q

*inefficient*: solves the same subproblem repeatedly.

*Top-down with memoization*

Memoized-Cut-Rod(p, n):

let r[0...n] be a new array with r[i] = -∞ ∀ i

return Memoized-Cut-Rod-Ans(p, n, r)

Memoized-Cut-Rod-Ans(p, n, r):

if (n == 0) return 0

if (r[n] >= 0) return r[n]

q = -∞

for i = 1 to n-1

q = max(q, p[i] + Memoized-Cut-Rod-Ans(p, n-i, r)

r[n] = q

return q

*Bottom-Up*

Bottom-Up-Cut-Rod(p, n):

let r[0...n] be a new array

r[0] = 0

for j = 1 to n

q = -∞

for i = 1 to j

q = max(q, p[i] + r[j-i])

r[j] = q

return r[n]

a problem of size i is smaller than a subproblem of size j. Thus the procedure solves subproblems of size j = 0, 1 ... n in that order.

Reconstructing a solution

We can extend a dynamic programming algorithm to record not only the optimal value computed for each subproblem, but also the optimal choice that led to the optimal value.

Extended-Bottom-Up-Cut-Rod(p, n):

let r[0...n] & s[0...n] be new arrays

r[0] = 0

for i = 1 to n

q = -∞

for j = 1 to i

if (q < p[j] + r[i-j])

q = p[j] + r[i-j]

s[i] = j

r[i] = q

return s

// print complete list of piece sizes an optimal decomposition of a rod of length n

Print-Cut-Rod-Solution(p, n):

s = Extended-Bottom-Up-Cut-Rod(p, n)

while n > 0

print s[n]

n = n - s[n]

***Fibonacci DP***

Non-DP(n):

if (n == 1 or n == 2)

return 1

else

return Non-DP(n-1) + Non-DP(n-2)

DP-Topdown(n):

if (n == 1 or n == 2)

return 1

if (memo[n] != 0)

return memo[n]

memo[n] = DP-Topdown(n-1) + DP-Topdown(n-2)

return memo[n]

DP-Bottom-Up(n):

memo[1] = memo[2] = 1

for i = 3 to n

memo[i] = memo[i-1] + memo[i-2]

return memo[n]

***Matrix Chain Multiplication***

Matrix-multiply(A, B):

if column[A] != column[B]

error

else

for i = 1 to rows[A]

for j = 1 to column[B]

C[i][j] = 0

for k = 1 to column[A]

C[i][j] = C[i][j] + A[i][k]\*B[k][j]

return C

Time-Complexity Matrix (Al\*m \* Bm\*n): O(l\*m\*n)

Input: Matrix A1A2A3 ... An, Array p0p1p2...pn Matrix Ai of size pi-1\*pi

Output: Fully parenthesized product A1A2A3 ... An that minimizes the number of scalar multiplication.

Checking all possible parenthesization take exponentiation time

Solving optimization problem by DP

1) Characterize the structure of an optimal solution. If a problem exhibits optimal substructure, involves optimally dividing the problem into >= 2 subproblems

Aij = AiAi+1Ai+2...Aj

Aij can be obtained by multiplying Ai...k & multiplying Ak+1...j & then multiplying them together there are j-i possible splits(i, i + 1, i + 2 ... j - 1)

2) Recursively define the value of an optimal solution(minimum cost of parenthesizing).

let m[i][j] = minimum number of scalar multiplications needed to compute AiAi+1Ai+2...Aj

i = j m[i][i] = 0

Aij breaking into Ai...k Ak+1...j

m[i][j] = m[i][k] + m[k+1][j] + pi-1\*pk\*pj i < j and i ≤ k < j

m[i][j] gives the optimal value

for providing all information need to construction optimal solution we define s[i][j] which store value of k at which we split the product Ai....j in an optimal parenthesization

3) Computing the value of an optimal solution.

matrix Ai has dimension pi-1\*pi

p = <p0, p1, p2 ... pn> p.length = n+1

Matrix-Chain-Order(A, p):

n = p.length - 1

let m[n][n] and s[n][n] be new matrices

for i = 1 to n

m[i][i] = 0

for l = 2 to n // l is chain length

for i = 1 to n-l+1

j = i+l-1

m[i][j] = ∞

for k = i to j-1

q = m[i][k] + m[k+1][j] + pi-1\*pk\*pj

if q < m[i][j]

m[i][j] = q

s[i][j] = k

return m, s

4) Constructing the optimal solution.

Ai...j = Ai...As[i][j]As[i][j]+1...Aj

Print-MCM(s, i, j):

if (i == j)

print ‘A’

else

print ‘(‘

Print-MCM(s, i, s[i][j])

Print-MCM(s, s[i][j]+1, j)

print ‘)’

If you can't visualize bottom-up, just modify the original top-down recursive solution by including memoization.

Memoized-Matrix-Chain(p):

n = p.length-1

let m[n][n] be a new matrix with each element initialized as ∞

return Look-Up-Chain(m, p, 1, n)

Look-Up-Chain(m, p, i, j):

if m[i][j] < ∞

return m[i][j]

if (i == j)

m[i][j] = 0

else

for k = i to j-1

q = Look-Up-Chain(m, p, i, k) + Look-Up-Chain(m, p, k+1, j) + pi-1\*pk\*pj

if q < m[i][j]

m[i][j] = q

return m[i][j]

***LCS (Longest Common Subsequence)***

Input: X <x1 x2 ... xm> Y <y1 y2 ... yn>

Output: Z longest common subsequence of X and Y

1) Characterize the longest common subsequence.

if xm = yn Zmn = xm or yn Zm-1,n-1 is an LCS of Xm-1 & Yn-1

xm ≠ yn Zm-1,n-1 is LCS of Xm-1 & Yn orXm & Yn-1

2) Recursively define the value of an optimal solution(length of LCS).

C[i][j] length of LCS of sequence Xi & Yj

if either i = 0 or j = 0 (one of the sequence has length 0) LCS has length 0.

C[i][j] = 0 if i = 0 or j = 0

C[i-1][j-1] + 1 if i,j > 0 xi = yj

max(C[i][j-1], C[i-1][j]) if i,j > 0 xi ≠ yj

3) Computing the value of an optimal solution.

LCS-Length(X, Y):

m = X.length

n = Y.length

for i = 1 to m c[i][0] = 0

for j = 0 to n c[0][j] = 0

for i = 1 to m

for j = 1 to n

if (xi == yj)

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

b[i][j] = 1

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

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

b[i][j] = 2

else

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

b[i][j] = 3

return b, c

b[i][j] corresponds to the optimal solution chosen while computing c[i][j]

Time-Complexity: O(m\*n)

4) Constructing the optimal solution.

Print-LCS(b, X, i, j):

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

if (b[i][j] == 1)

Print-LCS(b, X, i-1, j-1)

print Xi

else if (b[i][j] == 2)

Print-LCS(b, X, i-1, j)

else

Print-LCS(b, X, i, j-1)

Time-Complexity: O(m+n)

Improving the code(space wise) if we need only optimal value not optimal solution.

we can eliminate b, c[i][j] entry depends on only 3 other c table entries c[i-1][j-1], c[i-1][j], c[i][j-1]

we need only two rows of table C at a time the row being computed & the previous row.

***Zero-One Knapsack***

Input: N various items Vi(values) & Wi(weights) & maximum knapsack size MW.

Output: Maximum value of items that one can carry in a knapsack of size MW.

C[i][w] be the maximum value if available items are X1, X2, X3 ... Xi and the knapsack size is w.

if i = 0 or w = 0 (no item or knapsack is full) C[i][w] = 0

if Wi > w (skip this item too heavy for our knapsack) C[i][w] = C[i-1][w]

if Wi ≤ w (take the maximum of “not-take” or take) C[i][w] = max(C[i-1][w], C[i-1][w-Wi] + Vi)

solution is C[n][MW]

let C[n+1][MW+1], W[n], V[n]

for i = 0 to n

C[i][0] = 0

for w = 0 to MW

C[0][w] = 0

for i = 1 to n

for w = 1 to MW

if W[i-1] > w

C[i][w] = C[i-1][w]

else

C[i][w] = max(C[i-1][w], C[i-1][w-W[i-1] + V[i-1])

print C[n][MW]

***Edit Distance***

Input: Given two string cost for insertion, deletion and replace = 1

Output: minimum action to transform string1 to string2

d(string1, string2) be the distance between string1 and string2

m[i][j] = d(s1[1...i], s2[1...j])

m[0][0] = 0

for i = 1 to length(s1) m[i][0] = i

for j = 1 to length(s2) m[0][j] = j

for i = 1 to length(s1)

for j = 1 to length(s2)

val = (s1[i-1] == s2[j-1]) ? 0 : 1 // replace cost

m[i][j] = min(m[i-1][j-1] + val, // replace

min(m[i-1][j] + 1, // deletion

m[i][j-1] + 1)) // insertion

***Counting Change***

Input: list of denominations and a value N to be changed with these denominations.

Output: Number of ways to change N.

nway[N] = {0}

coin[m] = {.....} list of denominations in decreasing order

nway[0] = 1 // 1 way to change 0 (no coins)

for i = 0 to m-1

c = coin[i]

for j = c to N

nway[j] += nway[j-c]

print nway[N]

***Longest Increasing Subsequence (LIS)***

Input: given height sequence height[1...n]

Output: longest subsequence of given sequence such that all value in longest subsequence are strictly increasing.

for i = 1 to n-1

for j = i+1 to n

if height[j] > height[i] and length[i] + 1 > length[j]

length[j] = length[i] + 1

predecessor[j] = i

Time-Complexity: O(n\*n)

Non-DP solution Time-Complexity: O(n\*log(n))

*Strictly increasing sequence*

After ith iteration (in first i elements), jth index in set is the smallest possible element at the end of the increasing sequence of length j.

set<int> s

set<int>::iterator it

for i = 0 to n-1

s.insert(a[i])

it = s.lower\_bound(a[i])

it++

if (it != s.end()) s.erase(it)

return s.size()

*Non-strictly increasing sequence*

multiset<int> s

multiset<int>::iterator it

for i = 0 to n-1

s.insert(a[i])

it = s.upper\_bound(a[i])

if (it != s.end()) s.erase(it)

return s.size()

# Graph

graph: list of vertices adjacent to each vertex.

vector< vector <int> > g where g[i] is list of vertices adjacent to i

***Breadth First Search***

BFS: discovers each vertex that is reachable from s(source).

1. Computes the distance(smallest number of edges) from s to each reachable vertex.
2. Produces a Breadth first tree with root s that contains all reachable vertices. Breadth first tree from s to v corresponds to the shortest path from s to v.

At the start of the algorithm all vertices will be in initial state.

When a vertex is inserted in the queue its state will change from initial to waiting.

When a vertex is deleted from the queue its state is changed from waiting to visited.

BFS(g, s):

for i = 0 to n-1

visited[i] = 0

parent[i] = -1

distance[i] = 0

queue<int> q

q.push(s)

visited[s] = 1

while(!q.empty())

s = q.front()

q.pop()

for each v **∈** g[s]

if !visited[v]

q.push(v)

visited[v] = 1

parent[v] = s

distance[v] = distance[s] + 1

Print-Path(s, v):

if !visited[v]

print “no path”

vector<int> path

while (s != v and parent[v] != -1)

path.push\_back(v)

v = parent[v]

if parent[v] == -1

print “no path from s to v”

else

reverse(path.begin(), path.end())

print path

***Depth First Search***

DFS: explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it.

Once all the v’s edges have been explored, the search backtracks to explore edges leaving the vertex from which v was discovered.

Search continues until we have discovered all the vertices that are reachable from the original vertex source.

If any undiscovered vertices remain then DFS selects one of them as a new source & it repeats the search from that source.

DFS(g, s):

stack<int> st

dfs\_timer = 0

visited[s] = 1

time\_in[s] = dfs\_timer++

st.push(s)

while(!st.empty())

s = st.top()

st.pop()

time\_out[s] = dfs\_timer++

for each v **∈** g[s]

if !visited[v]

s.push(v)

visited[v] = 1

time\_in[v] = dfs\_timer++

***Disjoint Sets***

Init-Set():

for i = 0 to n-1

parent[i] = i

size[i] = 1

Find-Set(i):

if (parent[i] == i) return i

return parent[i] = Find-Set(parent[i])

Union-Set(i, j):

parent[Find-Set(i)] = Find-Set(j)

Extended-Union-Set(i, j):

i = Find-Set(i)

j = Find-Set(j)

if (i == j) return 0

if (size[j] > size[i]) swap(i, j)

parent[j] = i

size[i] += size[j]

return 1

***Dijkstra***

#define ii pair<int, int>

Dijkstra(G, s):

vector<int> D(N, ∞) // distance from start vertex to each vertex

priority\_queue< ii, vector<ii>, greater<ii> > Q

// priority queue with reverse comparison operator so top will return least distance

D[s] = 0 // initialize the start vertex suppose it’s 0

Q.push(ii(0, s))

while (!Q.empty())

ii top = Q.top() // fetch the nearest element

Q.pop()

v = top.second // v is vertex index

d = top.first // d is distance

if d <= D[v] // we analyze each vertex only once, the other occurrences

for each it **∈** g[v] // of it on queue(added earlier) will have

v2 = it->first // greater distance

cost = it->second

if D[v2] > D[v] + cost

D[v2] = D[v] + cost

Q.push(ii(D[v2], v2))