|  |  |  |
| --- | --- | --- |
| Algorithms | Time Complexity | Space Complexity |
| Selection sort | Average: Θ(n2) | O(1) |
| Best: Ω(n2) |
| Worst: O(n2) |
| Bubble sort | Average: Θ(n2) | O(1) |
| Best: Ω(n) |
| Worst: O(n2) |
| Insertion Sort | Average: Θ(n2) | O(1) |
| Best: Ω(n) |
| Worst: O(n2) |
| Merge sort | Average: Θ(nlogn) | O(nlogn) |
| Best: Ω(nlogn) |
| Worst: O(nlogn) |
| Quick sort | Average: Θ(nlogn) | O(1) |
| Best: Ω(nlogn) |
| Worst: O(n2) |
| Binary Search | O(logn) | O(1) |
| Ternary Search | O(log3n) | O(1) |
| DFS | O(V+E) [adjacency list] | same |
| O(V2) [adjacency matrix] |
| BFS | O(V+E) [adjacency list] | same |
| O(V2) [adjacency matrix] |
| Prims | O(E logV) | O(E) |
| Kruskals | O(E logV) | O(V+E) |
| Bellman Ford | O(VE) | O(V+E) |
| Dijkstra | O(E logV) | O(E) |
| Scheduling (Greedy) | O(nlogn) | O(n) |
| Knapsack(DP) | O(items\*capacity) | same |
| Coin Change(DP) | O(coins\*amount) | same |
| LCS(DP) | O(length of string1\*length of string 2) | same |
| Huffman Coding | O(length of string)  +  O(nlogn) [making the tree (n = distinct no. of chars)]  +  O(2n) [extracting codes from tree] | ------ |

N.B. logn means log2n

Binary Search

high = a.length-1

low = 0

binSearch (A[], x, low, high)

    if (low >= high)

        return false

    mid = (high + low)/2

    if (x == A[mid])

        return true

    else if (x < A[mid])

        binSearch (A[], x, low, mid-1)

    else

        binSearch (A[], x, mid+1, high)

Ternary Search

terSearch (A[], x, l, r)

    if (r >= 1)

        mid1 = l + (r-1)/3

        mid2 = r - (r-1)/3

        if (A[mid1] == x)

            return mid1

        if (A[mid2] == x)

            return mid2

        else if (x < A[mid1])

            return terSearch(A[], x, l, mid1-1)

        else if (x > A[mid2])

            return terSearch(A[], x, mid2+1, r)

        else

            return terSearch(A[], x, mid1+1, mid2-1)

Merge Sort

mergeSort (A[], l, r)

    if (l>r) return

    m = (l+r)/2

    mergeSort (A[], l, m)

    mergeSort (A[], m+1, r)

    merge(A[], l, m, r)

merge (A[], l, m, r)

    n1 = m-l+1

    n2 = r-m

    L[1...n1+1]

    R[1...n2+1]

    for(i = 1; i <= n1; i++)

        L[i] = A[l+i-1]

    for(i = 1; i <= n2; i++)

        R[i] = A[m+i]

    i=j=1

    for(k = l; k <= r; k++)

        if (L[i] <= R[i])

            A[k] = L[i]

            i++

        else A[k] = R[j]

            j++

Quick Sort

quickSort(A[], l, r)

    if (l >= r) return

    pivot = partition (A[], l, r)

    quickSort (A, l, pivot-1)

    quickSort (A, pivot+1, r)

partition (A[], l, r)

    i = l-1

    j = l

    while (j < r)

        if (A[j] > A[i])

            j++

        else if (A[j], A[i])

            swap (A[i+1], A[j])

            i++

            j++

    swap(A[i+1], A[r])

    return i+1

DFS

//declare and initialize color array with 0

//declare and initialize discovery,

//finishing and parent array with "NULL"

DFS\_visit (u)

    color[u] = 1

    time++

    discovery[u] = time

    for (v: v ∈ Adj[u])

        if (color[v] == 0)

            parent[v] = u

            DFS\_visit(v)

    time++

    finishing[u] = time

    color[u] = 2

BFS

//declare and initialize nodeList

BFS (s)

    for (x: x ∈ nodeList)

        color[x] = 0

        parent[x] = NULL

    distance[s] = 0

    //declare a Queue Q

    Q.add(s)

    color[s] = 1

    while (Q not empty)

        u = Q.pop()

        for (v: v ∈ Adj[u])

            if (color[v] == 0)

                Q.add(v)

                parent[v] = u

                distance[v] = distance[u] + 1

Prims

//declare and initialize nodeList

prim (G, r)

    for(x: x ∈ nodeList)

        key[x] = ∞

        parent[x] = NULL

    //declare a min heap mh

    //put all (key, node) in mh

    while (minheap not empty)

        (key, u) = extract\_min(mh)

        for (v: v ∈ Adj[u])

            if (v ∈ mh and w(u,v) < key[v])

                key[v] = w(u,v)

                parent[v] = u

Kruskal

Kruskal (G)

    for (x: x ∈ nodeList)

        parent[x] = x

    //sort all edges in ascending order of weight

    for ((u, v, w): (u, v, w) ∈ edgeList)

        par\_u = findPar(u)

        par\_v = findPar(v)

        par[par\_u] = par\_v

findPar(u)

    if (par[u] == u) return u

    return par[u] = findPar(par[u])

Strongly Connected Component finding

SCC (G

    //DFS on whole graph

    //Reverse the graph

    //DFS on reverse graph in descending order of finishing time

    //Store the nodes in graphs per DFS\_visit

Bellman Ford

BellmanFord (G,w,s)

    for (x: x ∈ V)

        d[x] = INF

    d[s] = 0

    for (i from 1 to V-1)

        for ((u,v,w): (u,v,w) ∈ EdgeList)

            Relax(u,v,w)

    for ((u,v,w): (u,v,w) ∈ EdgeList)

        if (d[u] + w < d[v])

            print (“Negative cycle is present”)

            return false

    return true

Relax (u,v,w)

    If (d[u] + w < d[v])

        d[v] = d[u] + w

Dijkstra

Dijkstra (s)

    for (x: x ∈ V)

        distance[x] = INF

        parent[x] = Null

    distance[s] = 0

    //declare minheap

    //put all (distance, node) pairs in minheap

    while(minheap not empty)

        pair(distance,u) = extract\_min(minheap)

        for(v: v ∈ adj[u])

            if(d[u]+w < d[v])

                d[v] = d[u] + w

                par[v] = u

------Greedy-------

Scheduling/Activity Selection

//Declare a class or struct

//containing three variables start, end, id

/\*

    jobs

        start

        end

        id

\*/

Schedule (jobs][])

    n = jobs.size

    //sort jobs according to ending times ascending

    //in case of time of ending randomly taken

    total\_jobs = 0

    active\_job\_end\_time = -1

    //initialize job\_list

    for (i = 1 to n)

        if (jobs[i].start >= active\_job\_end\_time)

            total\_jobs++

            job\_list.add(jobs[i])

            active\_job\_end\_time = jobs[i].end

Scheduling/Minimum Resource

//Declare a class or struct

//containing three variables start, end, id

/\*

    student

        start

        end

        id

\*/

min\_resource(Student[])

    n = student.size

    //eventList initialize

    //which contains

    for (i=1 to n)

        start = student[i].start

        end = student[i].end

        eventList.add(start,0)

        eventList.add(end,1)

    //sort eventList in ascending

    //order of time

    resource\_being\_used = 0

    min\_resource = 0

    for (i = 1 to 2n)

        if (eventList.type == 0)

            resource\_being\_used++

        else

            resource\_being\_used--

        min\_resource = max(min\_resource, resource\_being\_used)

    return

Fractional Knapsack

//Compute each items value/weight ratio and put in an array vw[]

//Sort vw in a descending order

//create w[] array and put total weight of the item at index i of vw[]

fracKnapsack (vw[], w[], capcity)

    weight = 0

    total\_price = 0

    for (i = 1 to n)

        if (weight + w[i] <= capacity)

            total\_price += vw[i]\*w[i]

            weight += w[i]

        else

            total\_price += (capacity - weight)\*vw[i]

            weight = capacity

            break

return total\_price

-------DP----------

Knapsack 0-1

n = total items

totalCapacity = 0

max\_profit (index,capacity)

    if (index >= n)

        if (dp[index][capacity] ≠ -1)

            return dp[index][capacity]

        if (capacity >= w[index])

            ret1 = max\_profit (index+1, capacity-w[index]) + p[index]

        else

            ret1 = 0                              // taken

        ret2 = max\_profit (index+1, capacity)     // not taken

        dp[index][capacity] = max(re1, ret2)

        return dp[index][capacity]

Coin Change

{Checking if it is possible}

//initialize dp[][] boolean array with NULL

if\_possible (index, amount)

    if (index >= n)

        if (amount == 0)

            return true

        return false

    if (amount >= c[index])

        ret1 = if\_possible (index+1, amount-c[index])

    else

        ret1 = false

    ret2 = if\_possible(index+1, amount)

    dp[index][amount] = ret1|ret2

    return dp[index][amount]

{Minimum coins needed (1 coin 1 times)}

//initialize dp[][] boolean array with NULL

min\_coin (index, amount)

    if (index >= n)

        if (amount == 0)

            return 0

        if (dp[index][amount] ≠ -1)

            return dp[index][amount]

        return INF

    if (amount >= c[index])

        ret1 = 1 + min\_coin (index+1, amount-c[index])

    else

        ret1 = INF

    ret2 = min\_coin (index+1, amount)

    dp[index][amount] = min (ret1,ret2)

    return dp[index][amount]

{Minimum coins needed (1 coin infinite times)}

//initialize dp[][] boolean array with NULL

min\_coin (index, amount)

    if (index >= n)

        if (amount == 0)

            return 0

        if (dp[index][amount] ≠ -1)

            return dp[index][amount]

        return INF

    if (amount >= c[index])

        ret1 = 1 + min\_coin (index, amount-c[index])  ;the only change

    else

        ret1 = INF

    ret2 = min\_coin (index+1, amount)

    dp[index][amount] = min (ret1,ret2)

    return dp[index][amount]

{Minimum coins needed (1 coin limited times)}

//initialize dp[][] boolean array with NULL

min\_coin (index, amount, freq)

    if (index >= n)

        if (amount == 0)

            return 0

        if (dp[index][amount][freq] ≠ -1)

            return dp[index][amount]

        return INF

    if (amount >= c[index] && freq > 0)

        ret1 = 1 + min\_coin (index, amount-c[index], freq-1)  ;the only change

    else

        ret1 = INF

    ret2 = min\_coin (index+1, amount, f[index + 1])

    dp[index][amount][freq] = min (ret1,ret2)

    return dp[index][amount][freq]