THE GREEDY METHOD

The greedy method of solving problems suggests working in stages considering one input at a time. At each stage, it is decided whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If inclusion of the input forms an infeasible solution it is discarded.

For some problems, it may be required to obtain a subset of the input set that satisfies some constraints. Any subset that does this is called a **feasible solution**. A feasible solution that maximizes or minimizes a given objective function is called an **optimal solution**.

Why is the greedy method called 'greedy'?

Initially, before starting to solve a problem using the greedy method, the function that one is going to optimize (called the **optimization measure**) is found out. And at each stage, we select the input that will optimize this optimization measure based on local information. Then, we insert this input into the solution. Thus we can see that this method behaves greedily. Surprisingly, many problems can be solved using this simple method.

Knapsack Problem (a.k.a. Fractional Knapsack problem)

Given n objects with profits, p_1, p_2, .., p_n and weights, w_1, w_2, .., w_n and a knapsack of capacity, M, the problem is to insert an object (a fraction of it is allowed) into the knapsack such that the total profit earned in maximized.

Maximize $\sum_{i=1}^{n}\left(p_i x_i\right)$

s.t. $\sum_{i=1}^{n}\left(w_i x_i\right) \le M,$

$0 <= x_i <= 1.$

p=[11, 16, 10]; w=[5, 6, 3]; M=11; n = 3
p/w = [11/5, 16/6, 10/3]

p/w = [2.2, 2.7, 3.3]

Initially, Cu(remaining capacity) =M

1. Greedy about profit only [Choose the objects in the decreasing order of profit]

Object(i)	Is w_i<= Cu?	x_i	Sum ($w_i x_i$)	Sum ($p_i x_i$)	Cu
2	Yes	1	6x1=6	16x1=16	11-6 = 5
1	Yes	1	6+5x1=11	16+11x1= 27	5-5 = 0
3	No	0	11	27	0

Total profit = 27. Solution vector = (x_1,x_2,x_3) = (1,1,0)

p=[11, 16, 10]; w=[5, 6, 3]; M=11

p/w = [2.2, 2.7, 3.3]

Cu = 11

2. Greedy about weight only [Choose the objects in the increasing order of weight]

Object	Is w_i<= Cu?	x_i	Sum ($w_i x_i$)	Sum ($p_i x_i$)	Cu
3	Yes	1	3x1=3	10x1=10	11-3 = 8
1	Yes	1	3+5x1=8	10+11x1=21	8-5 = 3
2	No	3/6	8+ 6 x (3/6)=11	21 + 16x(3/6)=29	3-3 = 0

Total profit = 29

p=[11, 16, 10]; w=[5, 6, 3]; M=11

p/w = [2.2, 2.7, 3.3]

3. Greedy about both profit and weight [Choose the objects in the decreasing order of profit/weight ratio]

Object	Is w_i<= Cu?	x_i	Sum ($w_i x_i$)	Sum ($p_i x_i$)	Cu
3	Yes	1	3x1=3	10x1=10	11-3 = 8
2	Yes	1	3+6x1=9	10+16x1=26	8-6 = 2
1	No	2/5	9+ 5 x (2/5)=11	26 + 11x(2/5)=30.4	2-2 = 0

Total profit = 30.4; Solution vector, (x_1,x_2,x_3) = (2/5,1,1)

Examples:

p=[20, 25, 18]; w=[4,7,6]; M=15

p=[11, 16, 10,…….]; w=[20, 100, 100,…….]; M=10
p/w = [0.55, 0.16, 0.10,……]
x=[1/2, 0, 0,0,…….]

---

p=[11, 16, 10]; w=[20, 100, 100]; M=1000
p/w = [0.55, 0.16, 0.10]
x=[1, 1, 1]

```
GreedyKnapsack(p,w,n,M)
begin
    //Initialize soln. vector, array x[1..n] to 0's.
    .Sort the objects in the decreasing order of their profit/weight ratios
    such that $(p_i/w_i) >= (p_{i+1}/w_{i+1})$ for all 1<= i <n.   Ω(nlogn)  O(nlogn)
    Cu = M;                                                 Ω(1)   O(1)
    profit = 0;                                             Ω(1)   O(1)
    for i = 1 to n do                                       Ω(1)   O(n)
        if w[i]<= Cu then                                   Ω(1)   O(n)
            x[i] = 1;// put in the whole of object i        Ω(0)   O(n)
            profit = profit + p[i].x[i];                    Ω(0)   O(n)
            Cu = Cu – w[i].x[i];                            Ω(0)   O(n)
        else
            x[i] = Cu/w[i];                                 Ω(0)   O(1)
            profit = profit + p[i].x[i];                    Ω(0)   O(1)
            Cu = 0;  //i.e., Cu = Cu - w[i].x[i];           Ω(0)   O(1)
            exit; //exit out of the for loop                Ω(0)   O(1)
        endif
    endfor
end
```

Assume that mergesort is used for sorting. So, the time taken by GreedyKnapsack is Ω(nlogn). It is also O(nlogn). Therefore, we can conclude that the time taken is Θ(nlogn).

Applications: E.g., in logistics, it can be used to determine the most efficient way to load a truck with a given set of items. The aim may be to maximize the total cost of items being loaded. Here, the truck has a limited capacity. In finance, it can be used to select which investments to make in order to maximize return while staying within a budget.

Job scheduling with deadline

Problem Statement

In job sequencing problem, the objective is to find a sequence of jobs, which is completed within their deadlines and gives maximum profit.

Solution

Let us consider a set of ***n*** given jobs which are associated with deadlines. A profit, $\boldsymbol{p_i}$ is earned iff a job *i* is completed by its deadline. To complete a job, one has to process the job on a machine for 1 unit of time. Only one machine is available for processing the jobs.

It may happen that all of the given jobs may not be completed within their deadlines.

Assume, deadline of **i**th job is $\boldsymbol{d_i}$ and the profit received from this job is $\boldsymbol{p_i}$. Hence, the optimal solution of this algorithm is a feasible solution with maximum profit. A feasible solution is a subset, J of jobs such that each job in J can be finished by its dateline.

The value of a feasible solution, J is the sum of the profits of the jobs in J, which is given by:

Maximize

$$\sum_{i \in J} (p_i)$$

Greedy algorithm:

Choose the objection function, $\sum_{i \in J} (p_i)$ as the optimization measure. Choose the job that increases the objection function value the most at each stage subject to the constraint that J is a feasible solution.

Problem instance:

p=[100,10,15,27]; d=[2,1,2,1]; n=4.

Brute-Force method: List out all the possible feasible solutions and choose the one that gives (exhaustive search) maximum profit:

For this, you consider all the permutations (2^n -1):

p=[100,10,15,27]; d=[2,1,2,1]; n=4.

Candidate	Feasible	Sequence	Profit
1) {1}	yes		100
2) {2}	yes		10
3) {3}	yes		15
4) {4}	yes		27
5) {1,2}	yes	<2,1>	110
6) {1,3}	yes	<1,3> or <3,1>	115
7) {1,4}	yes	<4,1>	127
8) {1,5}			
9)			
10)			
11)			
12)			
13)			
14)			
15)			

```
GreedyJobScheduling(n,p,d)
// Jobs are already sorted in the decreasing order of profit such that $p_i$ >= $p_{i+1}$ for
// all 1<= i <n.
.Sort in the decreasing order of profit to get P.                    O(nlogn)
.Sort in the increasing order of deadline to get D.                  O(nlogn)
begin
    J={1}; //Insert job 1 into J. J contains the final output.       O(1)
    for i = 2 to n do   //get a job fr om P                          O(n)
        if (all jobs in J U {i} can be completed by their deadlines) then
O($n^2$)
            J = J U {i};
        endif
    endfor
end
```

Time taken by the above algorithm is O (n^2).

Problem instance:

p=[10,25,15,40]; d=[3,1,2,1]; n=4.

Soln?

1) Apply brute-force method

2) Apply Greedy Method.

In the above algorithm, how do you check whether J is a feasible solution?

Try all possible permutations and check whether they can be processed in the order without violating their datelines?

In that case, for instance for J={1,2,3}, we have to check the possible permutations which are {1,2,3}, {1,3,2}, {2,1,3}, {2,3,1}, {3,1,2},{3,2,1}.

Thankfully, we need not try out all permutations, but only one of them. This is the permutation in which jobs are ordered in non-decreasing order of their datelines.

Theorem: Let J be a set of k jobs and T= i_i ,i_2,…,i_k a permutation such that d(i_i)<=d(i_2)<=… <=d(i_k). Then J is a feasible solution iff the jobs in J can be processed in the order T without violating any deadline.

Problem instance:

p=[100,10,15,27]; d=[2,1,2,1]; n=4.

Soln?

<u>Interval scheduling</u>

Problem Statement

Given a set of *n* job requests, a subset of requests is a feasible solution if no two requests overlap in time. The objective is to find the feasible solution of maximum size.

Solution

Let us consider a set of ***n*** job requests. Associated with each request, *i* are its start time, s_i and finish time, f_i.

Develop a greedy algorithm for solving this problem.

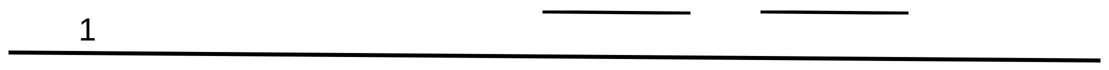
Select the inputs (requests) in some order.

i) Always select the available request that starts earliest. (sort in increasing order of start time). Order in which the inputs will be considered is =<1, 2, 3, 4, 5>

Resource : A classroom, R-1

Ex1., 2 3 4 5

- Ningrinla



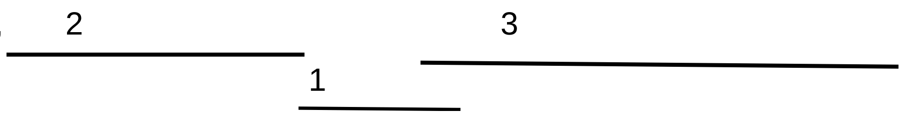
---------------→ t

Solution = {1}

Only request 1 will be in the solution since it has the earliest time, whereas we can get a bigger solution ({2,3,4,5}) which contains 4 requests. So this selection procedure fails.

ii) Always select the available request that has the smallest time interval. Order the requests in the increasing order of their duration. Order = <1,2,3>

Ex2.,

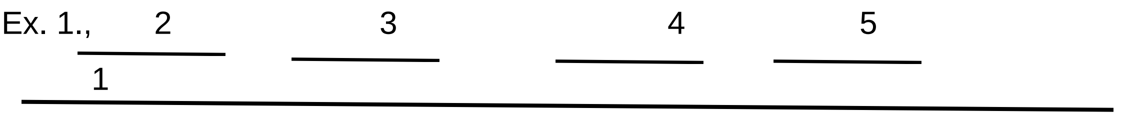


Solution= {1}

Only request, 1 will be in the solution as it has the smallest time interval, whereas we can get a bigger solution ({2,3}) which contains 2 requests. So this selection procedure fails.

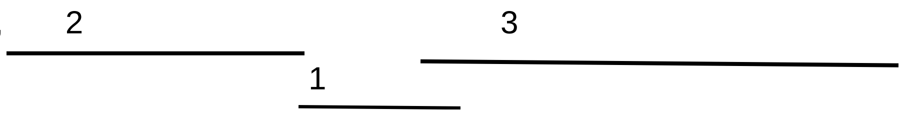
iii) Always select the available request that finishes earliest. The idea is to ensure that the resource becomes free as soon as possible while satisfying one request. This maximizes the time left to satisfy other requests. Sort in the increasing order of finish time. Order = <2,3,4,5,1>

Ex. 1.,



The solution obtained is ({2,3,4,5}) which is the optimal solution.

Ex.2,



Order = <2,1,3>

Solution= {2, 3}

The solution obtained is ({2,3}) which is the optimal solution.

```
GreedyIntervalScheduling(R,A)
begin
//R is the set of available requests; A is the set of accepted requests, i.e, A is the solution.
A = Φ          //A is an empty set initially
while R != Φ do //while there is a request in R
    Select a request i in R that has the smallest finishing time $f_i$.
    A = A U{i}
    Delete all requests from R that are not compatible with request i.
endwhile
return A
end
```

```
GreedyIntervalScheduling1(R,A,n)
begin
//R is an array of n requests; A is the set of accepted requests, i.e, A is the solution.
Sort the requests in R in the nondecreasing order of their finishing time such that  $f_i <= f_{i+1}$  where 1<=i<n. //R is in this sorted order now.
O(nlogn)
A = {1}                                                    O(1)
for i = 2 to n do // start from the 2nd request.           O(n)
    if (time interval of i does not overlap with that of any request in A) then
O($n^2$)
        A = A U{i}                                              O($n^2$)
    endif
endfor
end
Time taken =O($n^2$)
```

Minimum-cost Spanning Trees (or Minimum Spanning Trees)

Let G=(V, E) be a connected, undirected, weighted graph. V and E are the sets of vertices and edges respectively. A spanning tree is an undirected tree that connects all vertices in V. The cost of a spanning tree is the sum of the costs of its edges. Our goal is to find a spanning tree of minimal cost for G.

Cayley's Theorem : For a complete graph of n nodes, the number of spanning trees of the graph is $n^{(n-2)}$.
A subgraph T=(V, E') of G is a spanning tree of G iff T is a tree.

Kruskal's Algorithm

```
T = Φ
while T contains fewer than n-1 edges and E != Φ do
    choose an edge (v,w) from E of minimum cost
    delete (v,w) from E
    if (v,w) does not create a cycle in T then
        add (v,w) to T
    else
        discard (v,w)
    endif
end while
```

How can the checking of cycles in Kruskal's algorithm be implemented?

One possible way is to place all the vertices in the same connected component of T into a set. Two vertices are connected if they are in the same set. Therefore, initially we will have n sets, each of the sets containing a vertex, and representing a connected component. It is so because, initially no edge is inserted into the spanning tree. Thus when an edge (v,w) is taken as a candidate for insertion, we check whether v and w belong to the same set. For that, we use procedures *Find(v)* and *Find(w).* If *Find(v)* and *Find(u)* return the same value (say, set number) they are in the same set. If they do, they are already connected, so edge (v,w) is discarded because inclusion of it will form a cycle. On the other hand, if v and w happen to be in different sets, it is accepted and the two sets to which they belong are combined using procedure *Union(find(v),find(u))* to form one set. (A set represents a connected component.)

<u>Union-Find Data Structure</u>

a) Array Implementation:

Maintain an array Component that contains the name of the set currently containing each element. Thus,. Let S be a set, and assume that it has n elements denoted {1,…,n}. An array Component of size n is set up where Component[i] gives the name of the set (component) containing element (node). We initialize such that Component[i] = i for all i in S. To implement *Find(v),* it takes only O(1) time, since it is a simple lookup. However, to implement *Union(A,B),* it can take as long a O(n) as we have to update the values of Component[i] for all elements i in sets A and B.

b) Link/Pointer-based Implementation:

Union operation takes O(1) time and *Find* operation takes O(logn).

[Refer 'Algorithm Design' by Klienberg et. al., page no. 196]

We perform a total of at most 2e *Find* and n-1 *Union* operations over the course of Kruskal's algorithm, where e and n represent the number of edges and number of nodes respectively.

Using an array-based implementation, *Union* operation takes O(n) time and *Find* operation takes O(1). Thus, these operations over the course of the algorithm contribute $O(e) + O(n^2) = O(n^2)$. Here, e = |E|.

Using a pointer-based implementation, *Union* operation takes O(1) time and *Find* operation takes O(logn). Thus, these operations over the course of the algorithm contribute O(e logn) + O(n) =O(elogn).

The computing time of Kruskal's algorithm is, therefore, determined by the time for sorting the edges in the increasing order of their costs. (Please notice that we have to sort the edges first, because we have to consider the minimum cost edge first, then the next minimum-cost edge, and so on.) The time for sorting e edges (e = |E|) would be in the worst case O(e log e), assuming that we use some algorithm like the mergesort algorithm.

O(e log e) can be transformed to O(e log n) as follows.
Here $e <= n(n-1)/2$. So, $e < n^2$. So $e \log e = O(e \log n^2) = O(e (\log n + \log n)) = O(e \log n)$.

Hence, one can say the running time of Kruskal's algorithm is O(e log n).

Prim's Algorithm

The Prim's algorithm on the other hand takes $\theta(n^2)$, where n is the number of vertices in the graph.

Advantage: No cycle checking.

Prim's algorithm. Start with a start node, s (any node) and try to greedily grow a tree from s outward. At each step, simply add the node that can be attached as cheaply as possible to the partial tree we already have.

G=(V,E) //connected graph

Set S(selected vertices)   Set V(original)

Maintain a set S which is a subset of V on which a spanning tree has been constructed so far.

```
Prim(V, E)
// T is the minimum cost spanning tree, i.e., the set of edges in the minimum cost spanning tree.  --------------------------
begin
S = {s}
T = Φ
```

```
 //You can also start with the min-cost edge. In that case, mincost edge
(x,y).   //S = {x,y}, T= {(x,y)}
while S != V do
        Choose the edge (u,v) such that c(u,v) is the minimum-cost edge
        coming out of S (i.e., to V-S)
        Add (u,v) to T
        S = S U{v}
end while
end

Prim(V, E,n) //more detailed algorithm using array near for the vertices.
//From the book 'Fundamentals of algos' by Horowitz and Sahani
// T is the minimum cost spanning tree, i.e., the set of edges in the MST
//near[i] stores the vertex in T nearest to vertex i (which is not in T).
begin
Let (k,l) be the minimum cost edge in E
T = T U {(k,l)}
for  i = 1 to n do //Initialize near[]
        if cost(i,l) < cost(i,k) then
                near[i] = l
        else  near[i] = k
        endif
endfor
for  i = 2 to n-1 do //Find the additional edges for T
        Find vertex j such that near[j]≠0 and cost(j,near[j]) is minimum
        T = T U {(j,near[j])}
        near[j]= 0
        for k = 1 to n do  // update near for vertices not in T
                If near[k] ]≠0 and cost(k,near[k]) > cost(k,j) then
                        near[k]=j
                endif
        endfor
endfor
end
```

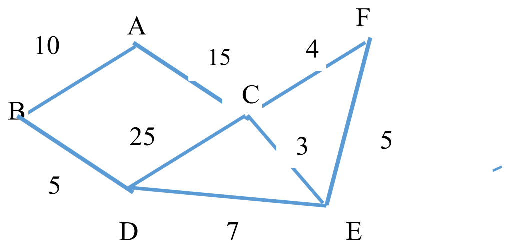
The time taken by Prim's algorithm is $O(n^2)$. It fact, it is $\Theta(n^2)$.

NOTE: Using a heap-based priority queue, we can implement Prim's algorithm and get an overall running time of O(e log n) [Refer 'Algorithm Design' by Klienberg and Tardo]

Ex. 5 vertices; c(A,B) =3; c(A,C)=4; c(B,D)=6, c(B,C)=5; c(C,D)=8, c(C,E)=10;c(D,E)=12. Cost of MST= 23.

Single Source Shortest Paths

Find the shortest path from a vertex v to all the other vertices.
Multi-stage solution: Build the shortest paths one by one.
Optimization measure: Sum of the lengths of the paths found so far. In order for this measure to be minimized, each individual path must be of minimum length.
So, generate these shortest paths in increasing order of path length.



ShortestPaths(v, cost, dist, n)
// dist[i], 1<=i<=n denotes the length of the shortest path from v to i in a //directed graph with n vertices.
//dist[v]= 0. G is represented by its cost adjacency matrix cost[n,n].
//s[i]=1, 1<=i<=n denotes that vertex i is selected (i.e., shortest path to vertex i //found).

```
for i = 1 to n do      //initialize set s to empty/                              O(n)
          s[i] =0; dist[i] = cost[v,i]                                           O(n)
endfor
s[v] = 1; dist[v] = 0 //put vertex v in the selected array, s                   O(1)
for i = 1 to n-2 do    //determine n-1 paths from v                              O(n)
          choose u such that dist[u] = min {dist[w]} where s[w]==0
                                              n-1 + n-2 + n-3+ ....+2  = n(n+1)/2 –n
-1=$O(n^2)$
                                                        n+n-1+n-2+....1 = n(n+1)/2
          s[u] = 1 // put u in the selected array.                               O(n)
          for all w with s[w]==0 do  //for unselected vertices, update distance
          $O(n^2)$
                    dist[w] = min (dist[w], dist[u]+cost[u,w]) //i.e., update is done
if
                                              //going through u makes w nearer.
          endfor                              n-2 + n-3+ ....+1 = n(n+1)/2 –n-(n-1)= $O(n^2)$
endfor
end
```

Time taken by the algorithm is $O(n^2)$.
How do you modify the above algorithm so that not only the costs of the shortest paths but also the shortest paths are generated?

```
ShortestPaths(v, cost, dist, n)
// dist[i], 1<=i<=n denotes the length of the shortest path from v to i in a
//directed graph with n vertices.
//dist[v]= 0. G is represented by its cost adjacency matrix cost[n,n].
//s[i]=1, 1<=i<=n denotes that vertex i is selected (i.e., shortest path to
vertex i //found).
for i = 1 to n do      //initialize set s to empty/                              O(n)
          s[i] =0; dist[i] = cost[v,i]                                           O(n)
endfor
s[v] = 1; dist[v] = 0 //put vertex v in the selected array, s                   O(1)
for i = 1 to n-2 do    //determine n-1 paths from v                              O(n)
          choose u such that dist[u] = min {dist[w]} where s[w]==0
                                              n-1 + n-2 + n-3+ ....+2  = n(n+1)/2 –n
-1=$O(n^2)$
```

```
                                                  n+n-1+n-2+....1 = n(n+1)/2
    s[u] = 1 // put u in the selected array.                          O(n)
    for all w with s[w]==0 do  //for unselected vertices, update distance
    O(n^2)
        if dist[w] >= dist[u]+cost[u,w] then //i.e., update is done if
            dist[w]= dist[u]+cost[u,w]
            predecessor[w] = u
        endif
    endfor                    n-2 + n-3+ ....+1  = n(n+1)/2 –n-(n-1)= O(n^2)
endfor
end
```

What is a shortest path spanning tree?

The edges on the shortest paths from a vertex v to all remaining vertices in a connected undirected graph G form a spanning tree of G. This spanning tree is called a shortest path spanning tree.