GREEDY ALGORITHMS

The reason we are starting with greedy algorithms is that they are straightforward, shortsighted, taking decisions on the basis of information immediately at hand (not worrying about the future). They are easy to "invent", easy to implement and efficient (when they work of course). Typically used to solve optimization problems. Let's begin with an everyday example.

6.1 Making change:

We live in a country where the following coins are available:
Dollars (100 cents) - Quarters (25 cents) - Dimes (10 cents) - Nickels (5 cents) and pennies (1 cent).

Our problem is to devise an algorithm for paying a given amount to a customer using the smallest possible number of coins.

For instance: If you must pay \$2.89 (289 cents), the best solution is to give the customer 10 coins: 2 Dollars - 3 Quarters - 1 Dime - 4 Pennies
The algorithm can be formalized as follows.

```
function {make-change} (n): set of coins
  {makes change for n units using the least possible number of
   coins. The constant C specifies the coinage}
   Constant C = {100, 25, 10, 5, 1}
   S ← Ø  {S set that will hold the solution}
   s ← 0  {s sum of element in S}
   while s ≠ n do
       x ← the largest item in C such that s + x ≤ n
       if no such item exists
            return "No Solution found"
       S ← S U {a coin of value x}
       s ← s + x
   return S
```

This algorithm always produces an optimal solution to our problem.
If the supply of some of the coins is limited, the greedy Algorithm may not work.
The algorithm is greedy because at every step it chooses the largest coin it can. It never changes its mind: once a coin is chosen, it is there for good.

6.2 <u>General characteristics of Greedy algorithms:</u>
Greedy algorithms are characterized by most or all of the following features.
- You have a problem to solve in an optimal way - set of candidates are available
- As the algorithm proceeds, 2 others sets are accumulated. One contains candidates that have already been considered, the second is of the rejected ones
- There is a function that checks whether a particular set of candidates provides a "solution" to our problem, ignoring questions of optimality for the time being (e.g. Do the coins chosen add up to the amount to be paid?) +
- A second function checks whether a set of candidates is "feasible", that is whether or not it is possible to complete the set by adding further candidates so as to obtain at least one solution to our problem. (Here too, not concerned much with optimality yet.)
- Yet another function, the "selection function", indicates at any time which of the remaining candidates, that have neither been chosen nor rejected, is the most promising.
- Finally an objective function gives the value of a solution we have found (e.g. number of coins used to make change ...). Unlike the three functions mentioned previously, the objective function does not appear explicitly in the greedy algorithm.

To solve our problem, we look for a set of candidates that constitute a solution, and that optimizes the value of the objective function.

A greedy algorithm proceeds step by step. Originally, the set of chosen candidates is empty. Then at each step we consider adding to this set the best remaining untried candidate, our choice being guided by the selection function. If the enlarged set of chosen candidates would no longer be feasible, we reject the candidate we are currently considering. In this case the candidate that has been tried and rejected is never considered again. However if the enlarged set is still feasible, then we add it to the set of chosen candidates. Each time we add a chosen candidate we check whether it now constitutes a solution to the problem.

```
function greedy (C: set): set
  {C is the set of candidates}
  S ← Ø   {construct solution in S}
  while C ≠ Ø and not Solution(S) do
     x ← select (C)
     C ← C - {x}
     if feasible (S U {x}) then S ← S U {x}
  if Solution (S) then return S
                  else return "there is no solution"
```

6.3 Graphs: Minimum Spanning trees (MST)

Let $G = (N, A)$ be a connected, undirected graph. Each edge has a nonnegative "length cost." The problem is to find a subset T of A such that all the nodes remain connected when only the edges in T are used and the sum of the lengths of the edges in T is as small as possible. G is connected, then at least one solution exists. If one edge is of length 0, then several solutions may exist. If 2 solutions occur, then we consider the one with the least number of edges.

Let $G' = (N, T)$, $G' \subseteq G$ (G' subgraph of G), $|N| = n$

G' must have at least $n-1$ edges to remain connected. This is minimum number of edges in T. More edges means, cycles. Thus T with n or more edges cannot be optimal. T must have n-1 edges.

G' is called a minimum Spanning tree for the graph G.

At first 2 lines of attack seem possible if we hope to find a greedy algorithm for this problem. Clearly, A is the set of candidates:

i) One possible tactic is to start with an empty set T and select at each stage the shortest edge that has not been chosen or rejected.

ii) 2nd line is to choose a node and build a tree from there, selecting the shortest (every stage) available edge that can extend the tree to an additional node.

Both approaches work just fine.

The general scheme of a greedy algorithm apply in this case.

1. Candidates: edges in G
2. A set of edges is a solution if it constitute a spanning tree in N.
3. A set of edges is feasible if it does not include a cycle.
4. Selection function used varies with the algorithm
5. Objective function: Minimize the total length of the edges in the solution.

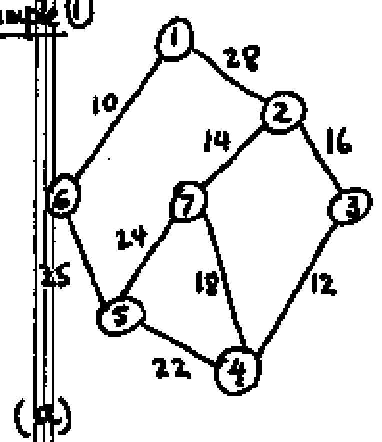
6.3.1 Kruskal's Algorithm:

Initially set T is empty. As the program progresses, edges are added to T. Kruskal's algorithm selects the "n-1" edges one at a time using the greedy criterion:

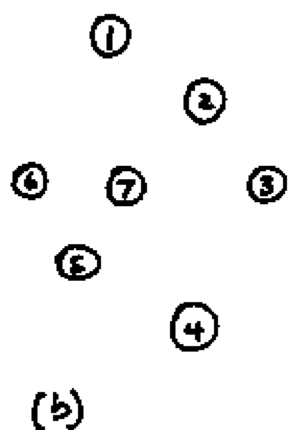
"From the remaining edges, select the shortest (least-cost) edge that does not result in a cycle when added to the set of already selected edges."

To illustrate how this algorithm works:

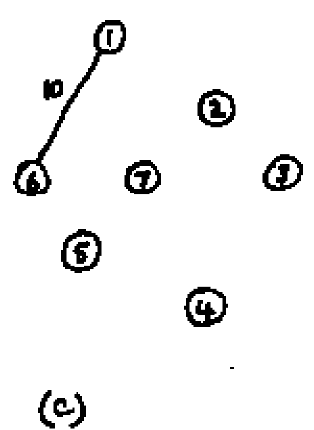
Example ①



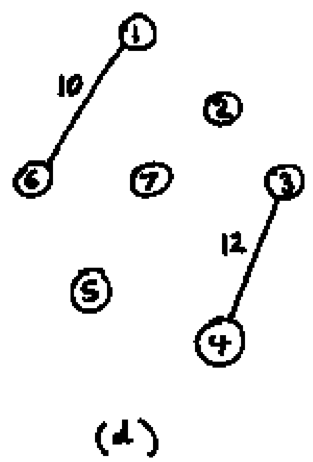
(a)



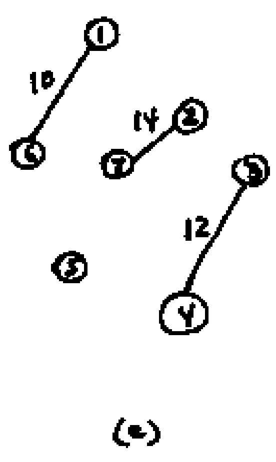
(b)



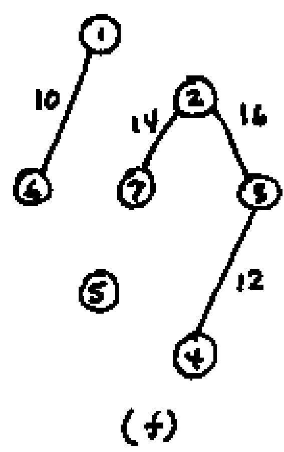
(c)



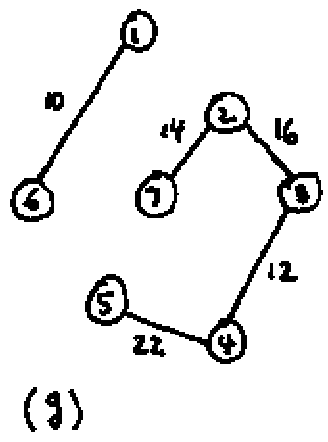
(d)



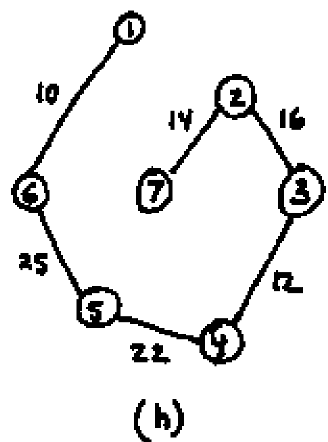
(e)



(f)

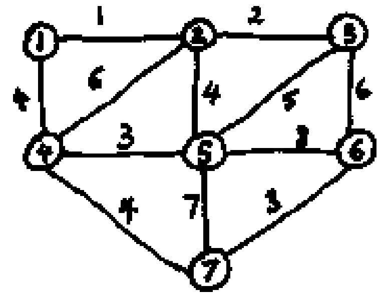


(g)



(h)

Example ②



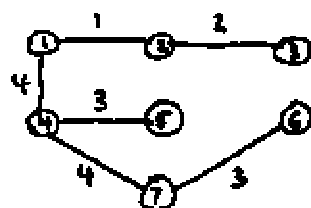
Edges are, in increasing order:

{1,2}, {2,3}, {4,5}, {6,7}, {1,4}, {2,5}
{4,7}, {3,5}, {2,4}, {3,6}, {5,7} and {5,6}.

The algorithm proceeds as follows:

Step	Edge considered	Connected components
Initialization	—	{1} {2} {3} {4} {5} {6} {7}
1	{1,2}	{1,2} {3} {4} {5} {6} {7}
2	{2,3}	{1,2,3} , {4} {5} {6} {7}
3	{4,5}	{1,2,3} {4,5} {6} {7}
4	{6,7}	{1,2,3} {4,5} {6,7}
5	{1,4}	{1,2,3,4,5} {6,7}
6	{2,5}	rejected
7	{4,7}	{1,2,3,4,5,6,7}

at this end the MST is shown across.



<u>Theorem</u>: *Kruskal's Algorithm finds a minimum spanning tree.*

To implement the algorithm, we have to handle a certain number of sets, namely the nodes in each connected component. Two operations have to be carried out rapidly: find(x), which tells us in which connected components the node x is to be found, and merge(A,B), to merge 2 connected components. For this algorithm it is preferable to represent the graph as a vector of edges rather than as a matrix of distances. Here is the algorithm:

```
function Kruskal (G = (N, A): graph; length: A → R^{+*}) set of edges
    {initialization}
    Sort A by increasing length
    n ← number of nodes in N
    T ← ∅ {will contain considered edges}
```

```
{Greedy loop}
repeat
   e ← {u,v}      shortest edge not yet considered
   ucomp ← find(u)
   vcomp ← find(v)
   if ucomp ≠ vcomp then
       merge (ucomp, vcomp)
       T ← T ∪ {e}
until T contains n-1 edges
return T
```

Execution time: $|N| = n$, $|A| = a$, the number of operations is in;

- $\theta(a \log a)$ to sort the edges which is equivalent to $\theta(a \log n)$ because $(n-1) \leq a \leq \frac{n(n-1)}{2}$;
- $\theta(n)$ to initialize the n disjoint set;
- $\theta(2a\,\alpha(2a,n))$ for all find and merge operations
- at worst $O(a)$ for the remaining operations

Since $\theta(\alpha(2a,n)) \subset \theta(\log n)$, we conclude the total time is in $\theta(a \log n)$.

It is preferable to keep the edges in an inverted heap, so the shortest edge is at the top (at the root of the heap). This allows the initialization to be carried out in a time in $\theta(a)$, although each search for a minimum in the "repeat" loop now takes a time in $\theta(\log a) = \theta(\log n)$. This is particularly advantageous if the MST is found at a moment when a considerable number of edges remain to be tried. In such cases, the original algorithm wastes time sorting those useless edges.

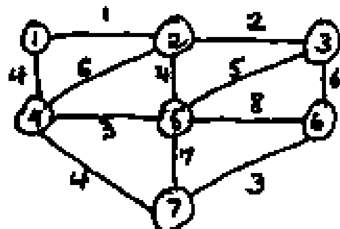
6.3.2 Prim's Algorithm:

Prim's algorithm, like Kruskal's, constructs the MS Tree by selecting edges one at a time. The greedy criterion used to determine the next edge to select is "from the remaining edges select the least cost/short length edge whose addition to the set of selected edges forms a tree". By contrast, the set of selected edges in Kruskal's algorithm forms a forest at each stage.

Let B be a set of nodes, and T a set of edges. Initially B contains a single arbitrary node, and T is empty. At each step Prim's Algorithm looks for the shortest possible edge $\{u, v\}$ such that $u \in B$ and $v \in N - B$. It then adds v to B and $\{u, v\}$ to T. In this way the edges in T form at any instant a MST for the nodes in B. We continue as long as $B \neq N$. Here is a simplified statement of the algorithm.

```
function Prim (G=(N,A): graph; length: A → R+): set of edges
    {initialization}
    T ← ∅
    B ← {an arbitrary member of N}
    while B ≠ N do
        find e = {u, v} of minimum length such that
            u ∈ B and v ∈ N - B
        T ← T ∪ {e}
        B ← B ∪ {v}
    return T
```

Let's illustrate the algorithm on the following example.



step	{u,v}	B
Initialization	–	{1}
1	{1,2}	{1,2}
2	{2,3}	{1,2,3}
3	{1,4}	{1,2,3,4}
4	{4,5}	{1,2,3,4,5}
5	{4,7}	{1,2,3,4,5,7}
6	{7,6}	{1,2,3,4,5,6,7}

The main loop of the algorithm is executed $n-1$ times, and each time it takes a time in $\theta(n)$. Thus Prim's algorithm takes a time in $\theta(n^2)$.

We saw that Kruskal's algorithm takes a time in $\theta(a \log n)$, where $a = |A|$. For a dense graph, a tends towards $\frac{n(n-1)}{2}$. In this case Kruskal's algorithm takes a time in $\theta(n^2 \log n)$ and Probably Prim's is better. For a sparse graph, a approaches n, and Kruskal's takes a time in $\theta(n \log n)$.

There exists more efficient algorithms than both.

6.4 Shortest Path - Graphs

Consider $G = (N, A)$ where G is a directed graph. Each edge has a nonnegative length. One of the nodes is designated as the "source" node. The problem is to determine the length of the shortest path from the source to each of the other nodes of the graph.

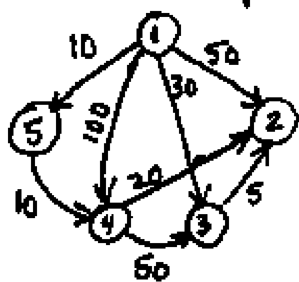
This problem can be solved by a greedy algorithm called "Dijkstra's Algorithm". It uses 2 sets of nodes, S and C. At every moment the set S contains those nodes that have been chosen (as we will see the minimal distance from the source is already known for every node in S). The set C all of the other nodes whose minimal distance from the source is not yet known, and which are candidates to be chosen at some later stage.

Hence the invariant property $N = S \cup C$. At the outset, S contains only the source itself; when the algorithm stops, S contains all the nodes of the graphs and our problem is solved. At each step we choose the node in C whose distance to the source is least and add

For simplicity we assume nodes of G are numbered 1 to n. So, $N = \{1, 2, \ldots, n\}$
We can suppose that node 1 is the source. Suppose that matrix L gives gives the length of each directed edge: $L[i,j] \geq 0$ if edge $(i,j) \in A$
$L[i,j] = \infty$ otherwise. Here is the algorithm:

```
function Dijkstra (L[1..n, 1..n]): array [2..n]
    array D[2..n]
    {initialization}
    C ← {2,3,...,n}          {S = N - C exists only implicitly
    for i ← 2 to n do D[i] ← L[1,i]
    {greedy loop}
    repeat n-2 times
        v ← some elements of C minimizing D[v]
        C ← C - {v}     {and implicitly S ← S U {v}}
        for each w ∈ C do
            D[w] ← min (D[w], D[v] + L[v,w])
    return D
```

The algorithm proceeds as follows on the following graph:



step	v	C	D
initialization	—	{2,3,4,5}	[50, 30, 100, 10]
1	5	{2,3,4}	[50, 30, 20, 10]
2	4	{2,3}	[40, 30, 20, 10]
3	3	{2}	[35, 30, 20, 10]

"Dijkstra's algorithm finds the shortest paths from a single source to the other nodes of a graph".

Suppose Dijkstra's algorithm is applied to a graph with n nodes and a edges. Using the representation used up to now, the instance is given in a form of a matrix $L[1..n, 1..n]$. Initialization takes a time in $O(n)$. Choosing v in the repeat loop requires all the elements of C to be examined, so we look at $n-1, n-2, \dots, 2$ values of D, on successive iterations, giving a total time in $\theta(n^2)$. The inner "for" loop does $n-2, n-3, \dots, 1$ iterations for a total in $\theta(n^2)$. The time required by this version is in $\theta(n^2)$..

6.5. The Knapsack Problem:

Given n objects, and a knapsack. For $i = 1, 2, \dots, n$ object i has a positive weight w_i and a positive v_i. The knapsack does not carry more than W. The problem is to maximize the value of included objects while respecting the weight. In this problem we assume that objects can be broken into smaller pieces. We may decide to carry only a fraction x_i of object i, $0 \leq x_i \leq 1$. The problem is stated the following way:

$$\text{maximize} \sum_{i=1}^{n} x_i v_i \qquad \text{subject to} \qquad \sum_{i=1}^{n} x_i w_i \leq W$$

where $v_i > 0$, $w_i > 0$, $0 \leq x_i \leq 1$ for $1 \leq i \leq n$. We are going to use Greedy algorithm to solve the problem. Candidates are the different objects, and the solution is a vector $(x_1, x_2, \dots, x_n)$. We assume that $\sum w_i > W$. In an optimal solution $\sum_{i=1}^{n} x_i w_i = W$. Our global strategy will be to select each object in turn in some suitable

order, to put as large a fraction as possible of the selected object into the knapsack, and to stop when full. Here is the algorithm:

```
function Knapsack (w[1..n], v[1..n], W): array[1
    {initialization}
    for i=1 to n do x[i] ← 0
    weight ← 0
    {greedy loop}
    while weight < W do
        i ← the best remaining object {see below
        if weight + w[i] ≤ W then x[i] ← 1
                                  weight ← weight + w[i]
            else x[i] ← (W - weight)/w[i]
                 weight ← W.
    return x.
```

There are 3 plausible selection function for this problem:

- -i- Choose the most valuable remaining object
- -ii- choose the lightest on the ground that it uses capacity slowly
- -iii- Choose objects whose value per unit weight is as high as poss

The following example illustrate how this works.

Exple: $n=5$ (objects), and $W=100$ (maximum weight)

w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2

if we choose in decreasing value order : $66 + 60 + \frac{40}{2} = 146$

if we choose in increasing weight : $20 + 30 + 66 + 40 = 156$

finally if we select the objects in order of decreasing v_i/w_i, we choose first object 3, then 1, then 2 and finally fill the knapsack with $4/5$ of object 5, the value is then

$$20 + 30 + 66 + 0.8 \times 60 = 164$$

This example shows that the solution obtained by a greedy algorithm that maximizes the value of each object it selects is not necessarily optimal, nor the solution obtained by minimizing the weight of each object. The solution can be summarized

select	x_i					Value
Max v_i	0	0	1	0.5	1	146
Min w_i	1	1	1	1	0	156
max v_i/w_i	1	1	1	0	0.8	164

The 3rd option is optimal.

- Implementation of the algorithm is straightforward. If the objects are already sorted, then the greedy loop clearly takes a time in $O(n)$
- Total time to sort is in $O(n \log n)$ (put objects in a heap).
- Creating a heap takes $O(n)$.
- Each trip round the greedy loop takes in $O(\log n)$ since the heap property must be restored after the root is removed

6.6 Scheduling:

6.6.1 Minimizing time in the system:

A single server, such as a processor, a gas pump, or a cashier in a bank, has n customers to serve. The service time required by each customer is known in advan[ce]: customer i will take time t_i, $1 \le i \le n$. We want to minimize the average time that a customer spends in t[he] system. Since n, the number of customers is fixed, this [is] the same as minimizing the total time spent in the system by all the customers. In other words, we want to minimize

$$T = \sum_{i=1}^{n} (\text{time in system for customer } i)$$

Suppose we have three customers with $t_1 = 5$, $t_2 = 10$, $t_3 = 3$. There are of course 6 permutations/orders.

order	Time T
123	$5 + (5+10) + (5+10+3) = 38$
132	$5 + (5+3) + (5+3+10) = 31$
213	$10 + (10+5) + (10+5+3) = 43$
231	$10 + (10+3) + (10+3+5) = 41$
312	$3 + (3+5) + (3+5+10) = 29$ ← OPTIMA[L]
321	$3 + (3+10) + (3+10+5) = 34$

The optimal schedule is obtained when the 3 customers are served in order of increasing service time.

This algorithm is optimal. All is needed is to sort the customer in nondecreasing order of service time which takes a time in $O(n \log n)$.

6.6.2 Scheduling with Deadlines:

We have a set of n jobs to execute, each of which takes a unit time. At any time $T = 1, 2, \ldots$ we can execute one job.

Job i earns us a profit $g_i > 0$ if and only if it is executed no later that time d_i.

Example: $n = 4$

i	1	2	3	4
g_i	50	10	15	30
d_i	2	1	2	1

The schedules to consider and the corresponding profits are

Sequence	Profit	Sequence	Profit	
1	50	2,1	60	
2	10	2,3	25	
3	15	3,1	65	
4	30	4,1	80	← OPTIMAL
1,3	65	4,3	45	

The sequence 3,2 has not been considered because job 2 would be executed at time $t = 2$, after its deadline $d_2 = 1$.

A set of job is feasible if there exists at least one sequence (also called feasible) that allows all the jobs in the set to be executed no later than their respective deadlines.

An obvious greedy algorithm consists of constructing the schedule step by step adding at each step the job with the highest value of g_i among those not considered yet (feasible of course). In our example choose job 1 first, then job 4; the set $\{1, 4\}$ is feasible because it can be executed in the order 4,1. Then try $\{1, 3, 4\}$ which is not feasible, then reject job 3. Try $\{1, 2, 4\}$, not feasible reject 2.

This algorithm always finds an optimal schedule, and its analysis is straightforward. Sorting the jobs in decreasing profit takes a time in $\theta(n \log n)$. The worst case for the algorithm is when this procedure turns out also to sort the jobs by order of decreasing deadline, and when they can all fit into the schedule. In this case, when job i is being consider the algorithm looks at each of the $k = i-1$ jobs in the schedule to find a place for the newcomer, and then moves them all along one position.