**i$^{th}$ order statistics** algorithm runs in O(n). That is the selection of the i$^{th}$ item can be done in O(n) running time.

**Dynamic programming**: Prove both *Optimal substructure* – (if an optimal solution to the problem contains within it optimal solutions to subproblems) and *overlapping subproblems* – (subproblems must be solved several time throughout solving the original problem.)

Optimal Substructure:

1. Show that the problem consists of making a choice that leaves one or more subproblems to solve.
2. Suppose that for a given problem you are given the choice that leads to the optimal solution.
3. Determine which subproblems ensue and how to best characterize the resulting space of subproblems
4. Show that the solutions to the subproblems used within the optimal solution to the problem must themselves by optimal by cut-n-paste contradiction method.

$$m[i,j]=\begin{cases} 0 & if\ i=j \\ \min\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & if\ i<j \end{cases}$$

$$c[i,j]=\begin{cases} 0 & if \quad i=0\ or\ j=0 \\ c[i-1,j-1]+1 & if \quad i,j>0\ and\ x_i=y_i \\ \max(c[i,j-1],c[i-1,j]) & if \quad i,j>0\ and\ x_i\neq y_j \end{cases}$$

**Transforming a recursive solution into a bottom-up dynamic programming solution**: Example

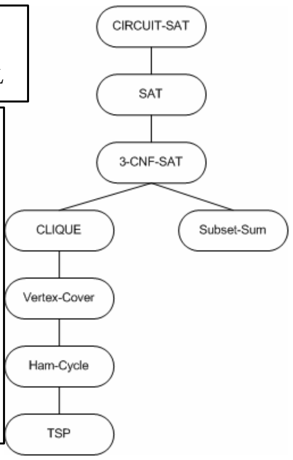
**MCM:** Matrix $A_i$ has dimensions $p_{i-1}$, $p_i$. m[i,j] is the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$. So

**LCS:** Let X = <x1, x2, …, xm>, we define the ith prefix of X, for i=0,…,m as Xi=<x1,…,xi>. Let Y=<y1,…yn> and Z=<z1,…,zk> be any LCS of X and Y. Then Optimal substructure argument

1. If xm=yn, then zk = xm=yn and Zk-1 is an LCS of Xm-1 and Yn-1
2. If xm≠yn, then zk≠xm → Z is an LCS of Xm-1 and Y.
3. If xm≠yn, then zk≠yn → Z is an LCS of X and Yn-1.

Recursive solution at right.

Ogden's lemma: $s=uvxyz$; *vy* contains at least one distinguished position; *wxy* contains no more than *n* distinguished positions; *x* contains at least 1 ( or 2) distinguished position(s) " m = 0, $uv^mxy^mz$ Î $L$



```
Matrix-Chain-Order(p)
n = length[p] –1
for i=n do {m[i,i]=0}
for L=2 to n
    for i=1 to n-L+1
        j=1+L-1
        m[i,j]=inf
        for k=i to j-1
            q=m[i,k] + m[k+1,j]+p[i-1]*p[k]*p[j]
            if q<m[i,j]
                m[i,j]=q
                s[i,j]=k
return m and s
```

```
LCS-Length
    m=length[X]; n=length[Y];
    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 x[i]=y[i] then
                c[i,j] = c[i-1, j-1]+1
                b[i,j] = "D" //diag
            else if c[i-1,j]≥c[i,j-1] then
                c[i,j]=c[i-1,j]
                b[i,j] = "U" //up
```

```
Optimal-BST(p,q,n)
for i=1 to n+1 {e[i,i-1]=q[i-1]; w[i,i-1]=q[i-1] }
for L=1 to n
    for i=1 to n-L+1
        j=i+L-1
        e[i,j]=inf
        w[i,j]=w[i,j-1]+p[j]+q[j]
        for k=1 to j
            t=e[i,k-1]+e[k+1,j]+w[i,j]
            if t<e[i,j] then
                e[i,j] = t
                root[i,j]=k
return e and root
```

**Optimal BST**: Let K=<k1,…,kn> be n distinct keys ∋ k1<…<kn. We also must have n+1 dummy keys representing all the values not in K. <d0,…,dn> ∋ d0 represents all values less than k1, d1 represents all values between k1 and k2, and dn represents all values greater than kn. Each ki has a probability pi and each di has a probability qi. Define: w(i,j) = $\sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$ and $e[i,j]=\begin{cases} q_{i-1} & if\ j=i-1 \\ \min_{i\le k\le j}\{e[i,k-1]+e[r+1,j]+w(i,j)\} & if\ i\le j \end{cases}$

**Greedy Algorithm:** Prove *Optimal substructure*, and *Greedy choice property* – (that any optimal solution may or must contain the greedy choice.)

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
2. Prove that there is always an optimal solution to the original problem that makes the greedy choice. → Greedy choice is safe.
3. Demonstrate that, …, what is left is a subproblem with the property that if we combine an optimal solution with the greedy choice we get an optimal solution.

Fractional Knapsack,
Super increasing coin problem…
Huffman Encoding

```
Huffman(C)
n = length of C //C linked list
Q=C //min priority queue
for i = 1 to n-1
…   allocate a new node z
    left[z]=x=extractMin[Q]
    right[z]=y=extractMin[Q]
    f[z]=f[x]+f[y]
…   Insert(Q,z)
return extractMin(Q) //return the
//root of the tree
```

**Flow Networks:** A flow network G = (V, E) is a directed graph in which each edge (u,v) ∈ E has a nonnegative capacity c(u,v)≥0. Each vertice v is on a path from (s)**Source** to (t)**Sink** → the graph is connected and |E| ≥ |V| – 1. **Flow** is defined by f(u,v) ≤ c(u,v) with the following properties: **Capacity constraint** above or $\Sigma_{u\in V}$f(u,v) = 0 for v∈ V-{s,t}, **Skew symmetry** ∀ u, v ∈ V, f(u,v) = -f(v,u), **Flow conservation** ∀ u ∈ V={s,t} $\Sigma_{v\in V}$f(u,v)=0. **Value of a flow** f denoted |f| = $\Sigma_{v\in V}$f(s,v). **Implicit Sum notation** f(X,Y)=$\Sigma_{x\in X}\Sigma_{y\in Y}$f(x,y). **Lemma 26.1**: ∀ X,Y,Z ⊆ V with X∩Y=0, f(X∪Y,Z)=f(X,Z)+f(Y,Z) and f(Z,X∪Y)=f(Z,X)+f(Z,Y). **Residual networks** are those nasty looking networks with back flow arrows instead of used/capacity numbers on the original edges. residual networks are defined as: $G_f$ = (V,$E_f$) where $E_f$ = {(u,v)∈ V×V : $c_f$(u,v)>0}. **Augmenting Paths:** p is a simple path from s to t in a residual network $G_f$. A flow is maximum if there does not exist any more augmenting paths. **Max-Flow, Min-Cut:** Given all cuts – see def. (S,T) where s∈ S, t∈T, the cut with the minimum flow f(S,T) is the maximum flow. Ford-Fulkerson = O(E|f*|) where f* is the max flow.
Edmonds-Karp algorithm uses a depth first search to find the shortest path when adding augmenting paths to the residual network and it runs in O(VE$^2$)

**Classification problems**: **P** (polynomial time solvable) **NP** (non-deterministic polynomial), **NPC**, **NP-Hard**

**Proving NP:** A language L ∈ NP means that given a certificate we can verify it in polynomial time. So: Precisely define the certificate and the verification algorithm, show that the algorithm verifies in polynomial time and cannot be fooled.

**Proving NP-Hard:** Given a language L and every language L'∈NP L'$\le_p$ L and possibly L∉NP. Thus we must only prove that a known NP-hard problem or NP-complete problem reduces to this one in polynomial time.

**Proving NP-Complete**: Given a language L, prove that L∈NP and prove that some known language L'∈NCP reduces to this language. NOTE: the reduction may seem totally arbitrary! what you have to do is show that (L'(x)→yes) ⇔ (L(x)→yes) , that is find a polynomial time algorithm to transform L' into L. Don't worry about anything except that ⇔ condition! Not all instances of your problem will cover the NPC problem! Don't worry about it!

For any graph G=(V,E) and subset V' Í V, the following statements are equivalent: (1) V' is a vertex cover for G. (2) V-V' is an independent set for G. (3) V-V' is a clique in the complement of G^{c} of G where G$^c$=(V,E$^c$) and E$^c$={{u,v}:u,v Î V and {u,v}? E}

**LIST OF NPC Problems and sketches of the reductions:**

Circuit-Sat : original problem we don't do this one!

Sat (Boolean formula sat) : label the wires and create formulas for each gate like

$xn \wedge (x3 \Leftrightarrow x1 \wedge x2) \wedge \ldots \wedge (x7 \Leftrightarrow x8 \vee x9)$

3-CNF-SAT :

Clique : A complete subgraph of G – thus a K-Clique is a complete subgraph of G with k=|V|. We reduce by creating a graph that has 3 vertice sets (same number as clauses) and put in an edge from each vertex to each vertex in other clauses that don't contradict it. If there is an n-clique where n is the number of clauses, it is satisfiable. Draw and convince yourself you should be able to reproduce it.

Vertex-Cover : If there is a k-clique in G, then there is a vertex cover of size n-k in the complement of G.

Subset-Sum : Evil and we don't have to do it!

Ham-Cycle: Evil and we don't have to do it!

TSP(Traveling Salesperson): TSP = {<G,c,k> : G = (V,E) is a complete graph, c is a function from V×V→Z, k∈Z, and G has a traveling-salesman tour with cost at most k}. The reduction is simple: Take an instance of ham-cycle G(V,E) and map it to a complete graph G'(V',E') where if (u,v)∈ E, c(u,v)=0, otherwise c(u,v)=1. Is there a TSP(G',c,0)?

GRAPH DEFINITIONS:

**Connected graph** :An *undirected graph* that has a *path* between every pair of *vertices*. **Strongly connected** graph: A *directed graph* that has a *path* from each *vertex* to every other vertex. **Degree** of a *vertex*: the number of *edges* connected to it. **Degree** of a *graph*: the maximum degree of any vertex **Residual network** : Instead of using flow/capacity flow is denoted by an arrow in the opposite direction and capacity is reduced.

```
BFS: Given a Graph G and a starting node s
1.  Color all the nodes white
2.  Distance for all nodes u to be  d[u]=infinity
3.  Parent of each node u to be  p[u]=nil
4.  Color s grey
5.  Enqueue  s→Q
6.  while Q is not empty
7.  u=dequeue(Q)
8.  find each white neighbor v of u do
9.   d[v]=d[u]+1
10.  p[v]=u
11.  engueue(v)
12.  color[u]=black
```

Augmenting paths: A path from s to t such that for each edge in the path c(u,v) > 0. **Cuts** (graphs/networks):A nonempty, *proper subset* of *vertices* of a *graph*. **Bipartite graph** : An *undirected graph* where *vertices* can be divided into two sets such that no *edge* connects vertices in the same set. **Trees**: Are zero based! **Bijection**: Given two non-empty sets X, Y, a function f: X→Y is a bijection if it is one-to-one (injective) and onto (surjective). **Vertex Cover**: of an undirected graph G=(V,E) is a subset V'⊆V such that if (u,v)∈ E, then u∈ V, or v∈ V' or both. That is a vertex cover is a set of vertices that cover the edges. Running time for BFS is **O(V+E)** BFS finds only those vertices that are reachable from ***S*** **.** The graph created from a BFS has no cycles. DFS may result in different trees based on the order in which adjacent verticies are visited, but this will not change the effectiveness of the results. **Running Time** Q(V+E). The Predecessor subgraph forms a **depth-first forest** composed of several depth-first trees. A **topological sort** of a DAG G=(V,E) is a linear ordering of all its vertices such that if G contains an edge (u,v), then u appears before v in the ordering. (If the graph is not acyclic, then no linear ordering is possible.) *A* ***strongly connected component*** *of a directed graph G=(V,E) is a* ***maximal*** *set of vertices C Í V such that "u,v Î C , we have both a path from u to v and v to u* .

```
Topological Sort(G)
1.  call DFS(G) to compute the finish times f[v] for each vertex v
2.  as each vertex is finished, insert it on the front of a linked list
3.  Return the linked list of vertices
(1) O(V+E) (2-3) O(1) = O(V+E)
```

```
Strongly-Connected-Components(G)
1)  call DFS(G) to compute finishing times f[u] for each vertex u
2)  compute G^T
3)  call DFS(G^T) , but in the main loop of DFS , consider the vertices in order of
    decreasing f[u] (as computed in line 1)
4)  output the vertices of each tree in the depth-first forest formed in line 3 as separate
    strongly connected components.
```

```
DFS(G)
1)  for each vertex u Î V[G]
2)    do color[u]=white
3)       p[u]=nil
4)    time=0
5)  for each vertex u Î V[G]
6)    do if color[u]=white
7)       then DFS_Visit(u)
 DFS_Visit(u)
1)   colorp[u]=gray
2)   time++
3)   discover[u]=time
4)  for each v Î Adj[u]  //explore edges
5)    do if color[v]=white
6)       then p[v]=u
7)          DFS_Visit(v)
8)  color[u]=black
9)  time++
10) finish[u]=time
```

```
MST-Kruskal(G,w)
1. A=?  // A is the set of edges that define the MST                     O(1)
2. for each vertex v∈V                                                    O(V)
3.    do Make-Set(v)
4. sort the edges of E into nondecreasing order by weight                 O(E log E)
5. for each edge (u,v)∈E, taken in nondecreasing order by weight          O(V+E)a(V)
6.    do if FindSet(u) ≠ FindSet(v)                                       ...
7.       Then A = A ∪ {(u,v)} //include (u,v) safe edge in MST            ...
8.          Union(u,v) //Union the sets u and v                           ...
9. return A                                                               O(1)
```

**Running Time :** where $a(V)=O(\log|V|)=O(\log|E|)$ (because $E<V^2$) is a slowly growing function defined on Cp511. So we get $O(V\log E+E\log E)=O(E\log E)$ or using the above logic $O(E\log V)$

```
MST-Prims(G(V,E), w(e∈E), r) // graph, weight, start       O(V)
1 for each u ∈ V
2.   key[u] = ∞                                              O(1)
3.   p[u] = Nil                                              O(1)
4 key[r] = 0 //decrease key                                  O(1)
5.Insert V into Q //a min priority queue                     O(1)
6.While Q != ?                                               O(V)
7.   u = Extract-Min(Q)                                      O(log V)
8.   for each v ∈ Adj[u]                                     2E times = O(E)
9.      if v ∈ Q and w(u,v) < key[v]                         O(1)
10.        p[v] = u                                          O(1)
11.        key[v] = w(u,v) //decrease key operation          O(log V)
RunningTime = O(E log V)
```

However MST running time can be decreased by using a Fibonacci heap where Extract-Min in O(log *v*) and Decrease-Key in O(1) , thus we can reduce line 11, and get a running time of O(E+V log V). **Shortest Paths:** Adjacency list representation is used for all these algorithms. B-F Allows: negative weight edges, but not negative cycles on the shortest path.

```
Initialize-Single-Source(G,s)
1.  " v Î V(G)
2.     d[v]=8
3.     p[v]=NIL
4.  d[s]=0          O(V)
```

```
Bellman-Ford(G,w,s)
1.   Initialize-Single-Source(G,s)       O(V)
2.   for i=1 to |V|-1                    O(V)
3.      for each edge (u,v)∈E            O(E)
4.         Relax(u,v,w)                  O(1)
5.   for each edge (u,v)∈E               O(E)
6.      if d[v] > d[u] + w(u,v)          O(1)
7.         return false                  O(1)
8.   return true                         O(1)
Running Time                             O(VE)
```

```
Relax(u,v,w)
if d[v]>d[u]+w(u,v)
   d[v]=d[u]+w(u,v)
   p[v]=u
O(1)
```

| DAG-Shortest Path(G,w,s) | |
|---|---|
| 1. Topologically sort the vertices of G | O(V+E) |
| 2. Initialize-Single-Source(G,s) | O(V) |
| 3. For each vertex u, taken in Topo. sorted order | O(V) |
| 4. for each vertex v ÎAdj[u] | O(E) agg analysis |
| 5. Relax(u,v,w) | O(1) |
| Running Time: O(V+E) | |

The running time for Dijkstra's algorithm is quite complex because it depends on how the Min-Queue Q is implemented. if we use an array and take advantage of the numbered list of vertices we have → Thus the **running time** is O(V²+E)=O(V²). This can be reduced to O(VlogV+E) by using a Fibonacci heap from chapter 20.

$$Insert(u)=O(1)$$
$$Decrease\text{-}Key(u)=O(1)$$
$$Extract\text{-}Min(u)=O(V)$$

```
Dijkstra(G,w,s)
1.   Initialze-Single-Source(G,s)     O(V)
2.   S=?                              O(1)
3.   Build Q from V                   O(V)
4.   While Q≠?                        O(V)
5.      u=Extract-Min(Q)              O(V)
6.      S=S∪{u}
7.      for each vertex v∈Adj[u]      O(E)
8.         Relax(u,v,w)               O(1)
```

**All-Pairs Shortest Paths:** Given a weight matrix *w* and a L$^{(i)}$ matrix where $L^{(0)}=\left[l_{ij}^{(0)}\right]=\begin{cases}0 & \text{if } i=j\\ \infty & \text{if } i\neq j\end{cases}$ Notice that the recursive formula $l_{ij}^{(m)}=\min\left(l_{ik}^{(m-1)}+w_{kj}\right)$ requires the pre-calculation of $L^{(m-1)}$ and we have three indices that go from 1 to *n* leading us to believe that the running time will be O(n⁴). Obviously L$^{(0)}$ contains the shortest path from every vertex to itself if there are no negative weight cycles. We extend this using the following algorithm:

```
Extend-Shortest-Paths (L,w)
n = # of rows in L                    O(1)
int L'[n][n] is an n x n matrix       O(1)
for i=1 to n                          O(n)
 for j=1 to n                          O(n)
  l'_ij = 8                             O(1)
  for k=1 to n                          O(n)
   l'_ij=min(l'_ij,l_ik+w_kj)            O(1)
return L'                              O(1)
Total Running Time                     O(n^3)=O(V^3)
```

After running Extend-Shortest-Paths *i* times, we find the shortest path between vertices of length *i* . To make sure that we find all shortest paths we must run this |v|-1 times. Thus the overall running time is going to be O(V²) . We can improve this by the following observation: **This algorithm is an operation (call it ∘ ) on matrices that is associative. We have**

$$L^{(1)} = w,$$
$$L^{(2)} = L^{(1)}\circ w = w^2$$
$$L^{(3)} = L^{(2)}\circ w = w^3$$
$$\ldots$$

Floyd-Warshall Algorithm for All Pairs Shortest Paths His recurssive characterization is ingenious:where $d_{ij}^{(k)}$ is the shortest path from i to j where the intermediate vertices come from the set {1,…,k) . Again this algorithm requires us to calculate $d_{ij}^{(k-1)}$ before we calculate $d_{ij}^{(k)}$, but here we are calculating only three indices in a manner that does not require repetitions we expect the running time to be O(n³) . Note: *w* is the adjacency matrix! Note: Dropping the superscripts allows us to diminish the space requirement from Q(n³)→Q(n²) without disrupting the algorithm. To construct the actual shortest paths, we can use the p's below.

$$d_{ij}^{(k)}=\begin{cases}w_{ij} & \text{if } k=0\\ \min\left(d_{ij}^{(k-1)},d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\right) & \text{if } k\geq 1\end{cases}$$

```
Floyd-Warshall(w)
n=# rows in w
D^(0) = w
for k = 1 to n
  for i = 1 to n
    for j= 1 to n
      d_ij^k = min(d_ij^(k-1), d_ik^(k-1)+ d_kj^(k-1))
return D^(n)
Running Time O(n^3)=O(V^3)
```

```
Ford-Fulkerson-Method
1 initialize flow to 0
2 while $p, an augmenting path
3 augment flow f along p
4 return f
```

$$p_{ij}^{(k)}=\begin{cases}p_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)}\leq d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\\ p_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)}> d_{ik}^{(k-1)}+d_{kj}^{(k-1)}\end{cases}$$

$$p_{ij}^{(0)}=\begin{cases}NIL & \text{if } i=j \text{ or } w_{ij}=\infty\\ i & \text{if } i\neq j \text{ and } w_{ij}<\infty\end{cases}$$