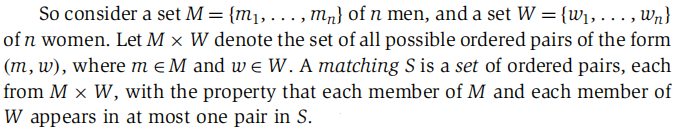
1. Stable Matching: Gale-Shapley Algorithm

**Problem definition:**

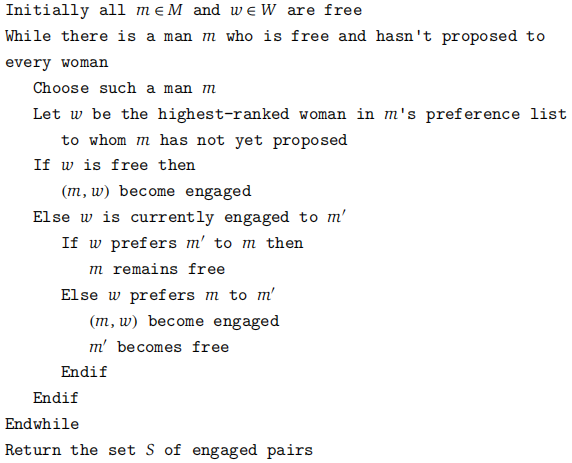




**Conditions of stable matching:**

1. Perfect matching**:** a matching with the property that each member of M and each member of W appears in exactly one pair in S`.
2. NoInstability**:** There are two pairs (m, w) and (m`, w`) in S with the property that m prefers w` to w, and w` prefers m to m`.

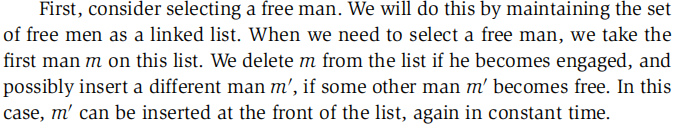
**Gale-Shapley algorithm:**

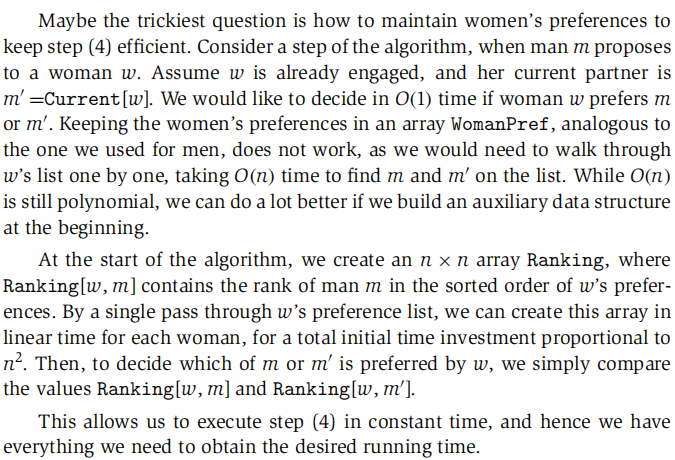
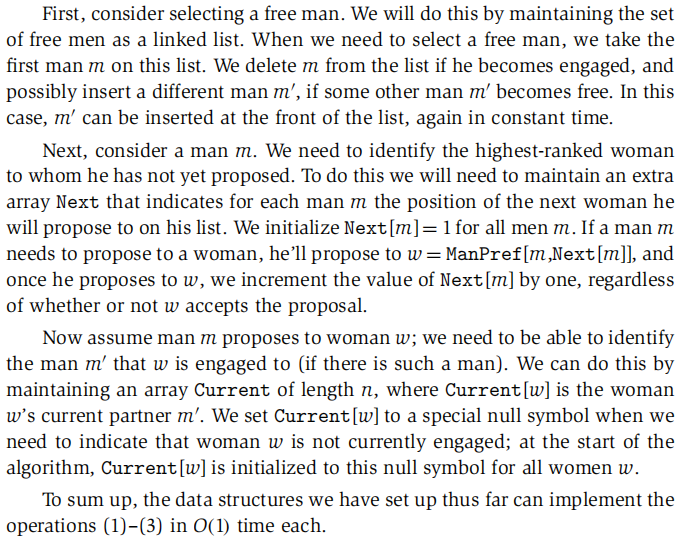


**Implementing: O(n2)**

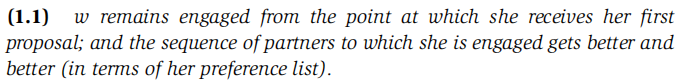
- ManPref[m,i]: denote the ith woman on man m’s preference list

- WomanPref[w,i]: denote the ith man on woman w’s preference list



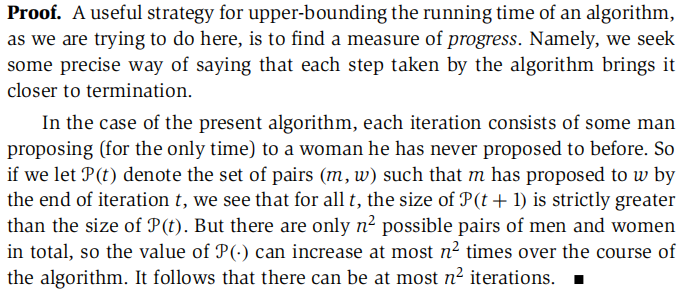


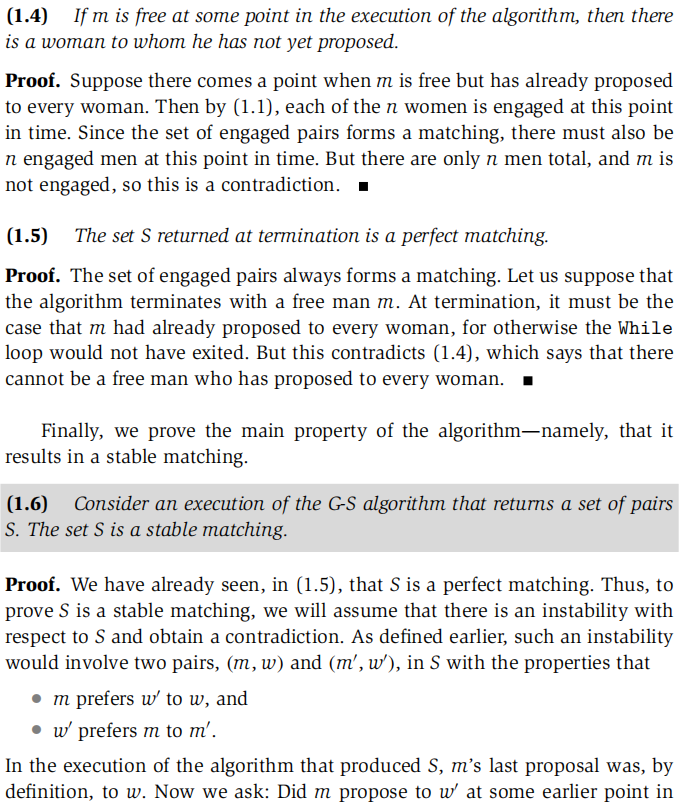
**Analyzing:**













**Extensions1: there could be multiple stable matchings**

- Unfairness:

favoring men. If the men’s preferences mesh perfectly (they all list different women as their first choice), then in all runs of the G-S algorithm all men end up matched with their first choice, independent of the preferences of the women. If the women’s preferences clash completely with the men’s preferences (as was the case in this example), then the resulting stable matching is as bad as possible for the women.

**-** Prove how general the “unfairness” is:

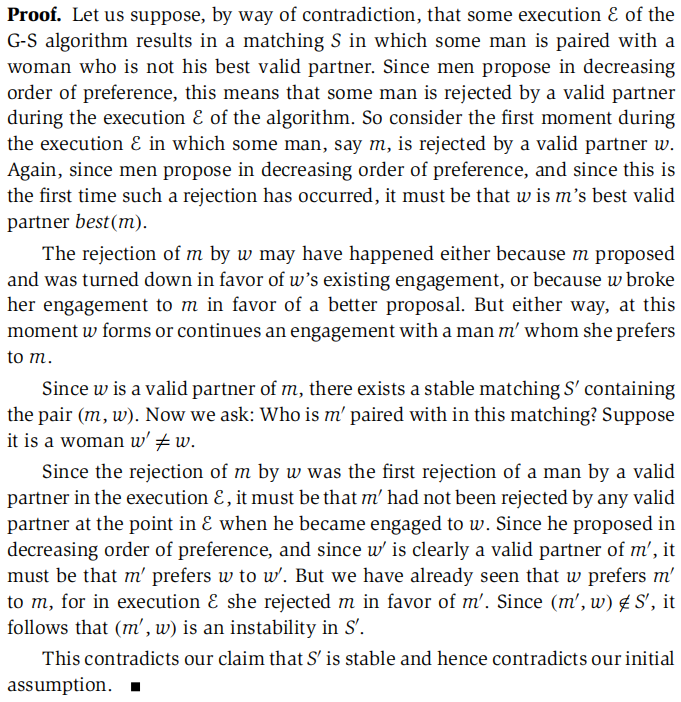
for any input, the side that does the proposing in the G-S algorithm ends up with the best possible stable matching (from their perspective), while the side that does not do the proposing correspondingly ends up with the worst possible stable matching.

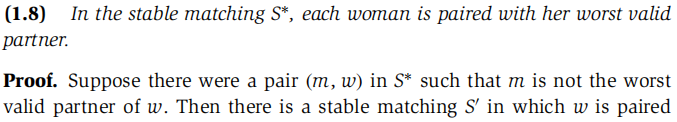
*valid partner*: a woman w is a valid partner of a man m if there is a stable matching that contains the pair (m, w).

*best valid partner*: w is the best valid partner of m if w is a valid partner of m, and no woman whom m ranks higher than w is a valid partner of his.

*set S\**: the set of pairs {(m,best(m)):mM}









***(1.9)*** *In a stable matching that G-S algorithm may end up with when men propose, none of the men end up with their highest-ranked woman and every women is matched with their most preferred man.*



**Extensions2: certain man-woman pairs are explicitly forbidden**

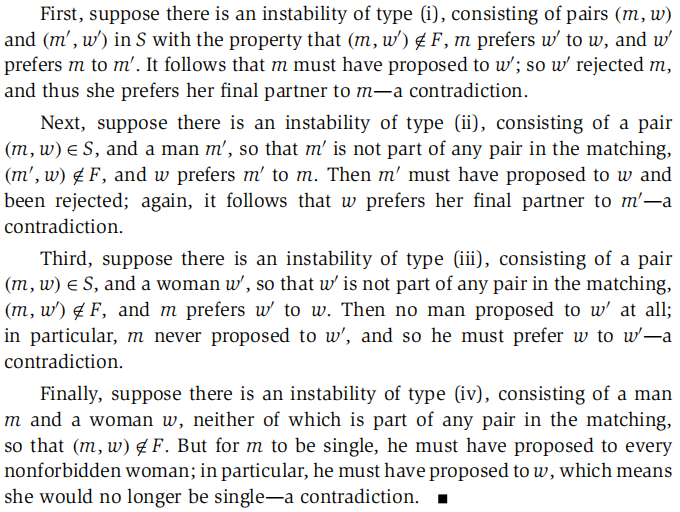
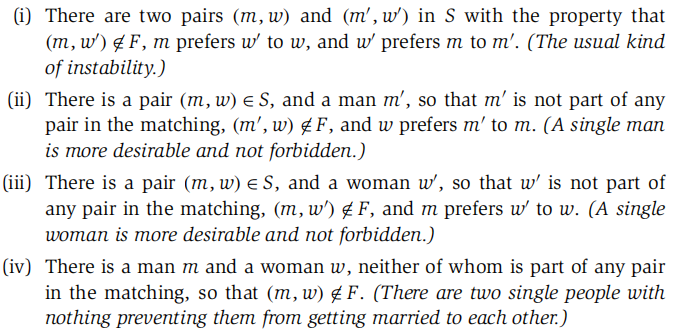
*Set* : pairs who are simply not allowed to get married.

- Solution:

1. algorithm: modify the While loop.

While there is a man m who is free and hasn’t proposed to every woman w for which (m, w) F.

1. Prove there is no instability with respect to the returned matching S.



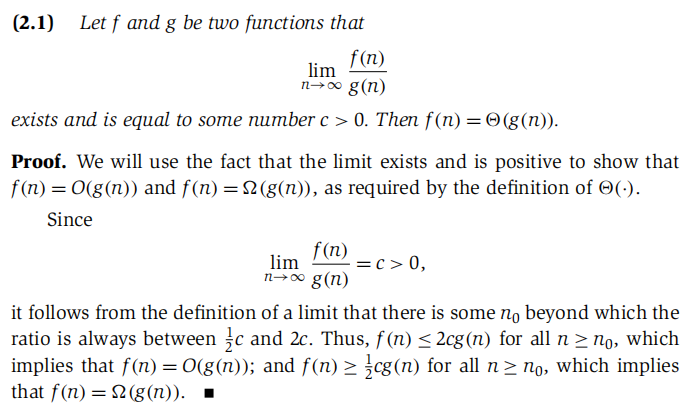
1. Time Complexity

T(n) : the worst-case running time of a certain algorithm on an input of size

**Asymptotic Upper Bounds:O(n)**

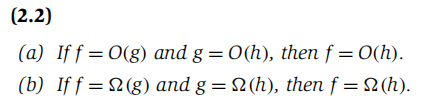
**Asymptotic Lower Bounds:(n)**

**Asymptotic Tight Bounds:(n)**

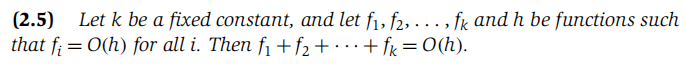


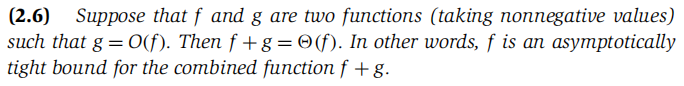
**Properties of Asymptotic Growth Rates**

- Transitivity



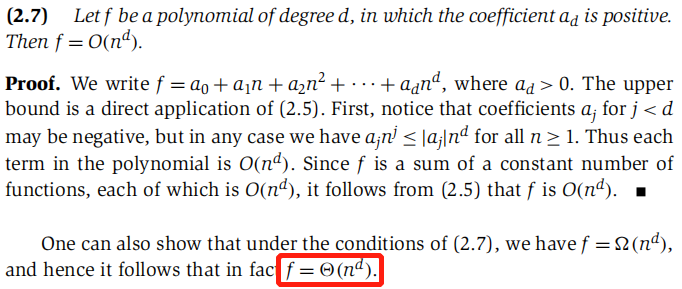
- Sum of Functions



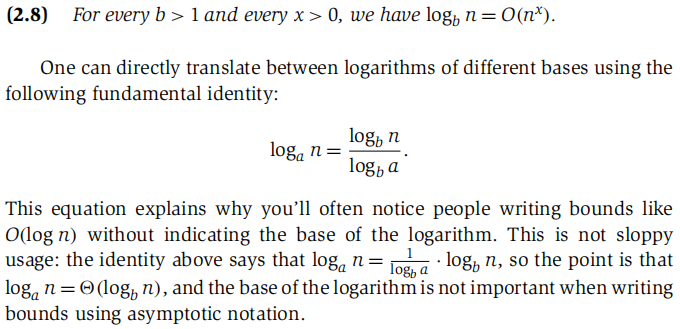


**Asymptotic Bounds for Some Common Functions**

- Polynomial-time algorithm: running time T(n) is O(nd) for some constant d, where d is independent of the input size



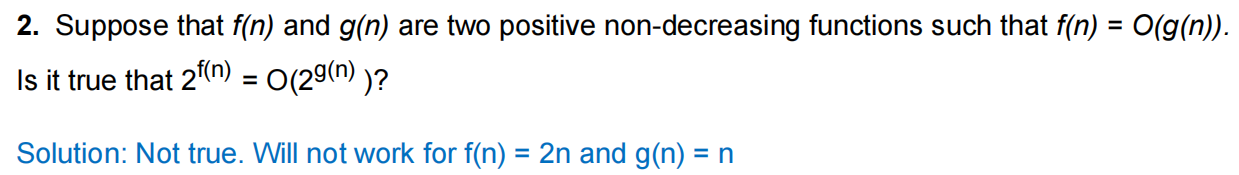
- Logarithms:



- Exponentials



**Sample Question:**



1. BFS&DFS

**Undirected graph**

**Basic Definitions:**

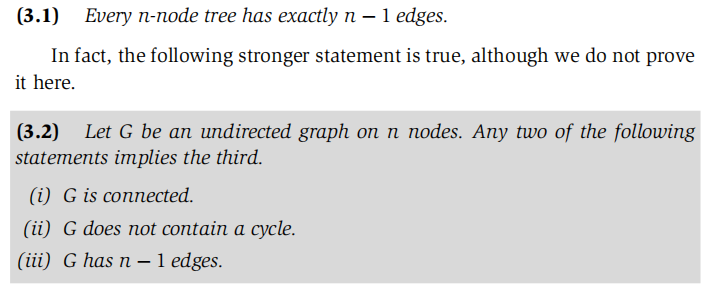
*Graph* *G=(V, E)*: it consists of a collection V of nodes and a collection E of edges.

*Path*: a sequence P of nodes v1,v2,...,vk with the property that each consecutive pair vi,vi+1 is joined by an edge in G.

*Cycle*: a path v1,v2,...,vk in which k>2, the first k-1 nodes are all distinct and v1 =vk.

*Connected Graph*: for every pair of nodes u and v, there is a path from u to v.

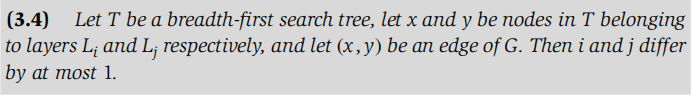
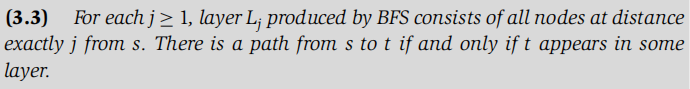
**Tree:** a graph is connected and does not contain a cycle.

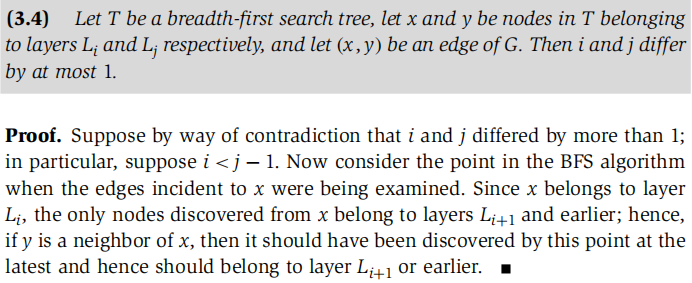


**BFS(Breadth-First Search):**

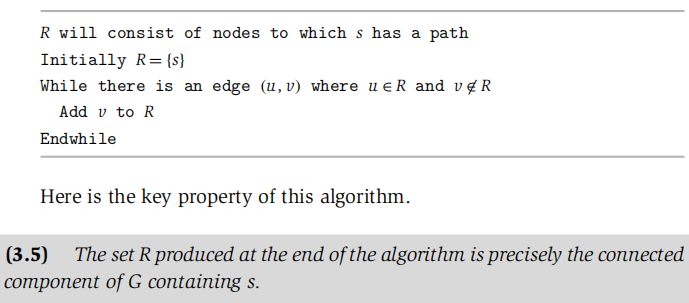
- explore outward from s in all possible directions, adding nodes one “layer” at a time.

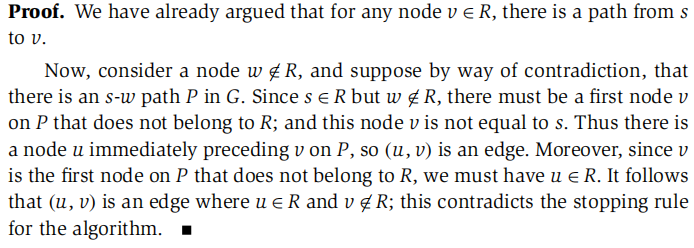
- Running time: O(m+n), in which m is no. of edges and n is no. of nodes.





*- connected component set R:*The set of nodes discovered by the BFS algorithm is precisely those reachable from the starting nodes s.

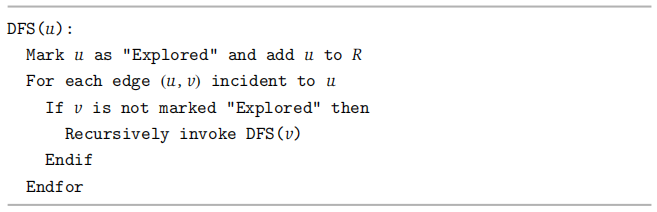


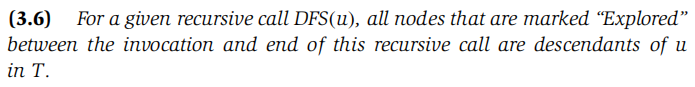


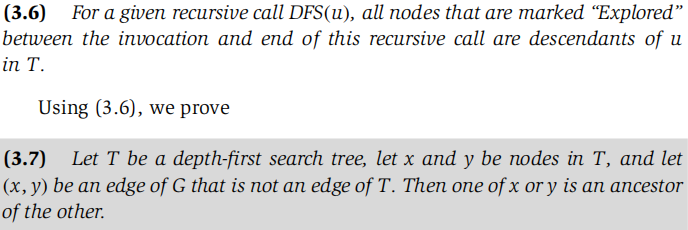
**DFS(Depth-First Search):**

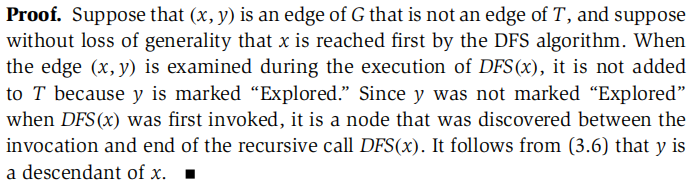
- it explores G by going as deeply as possible and only retreating when necessary.

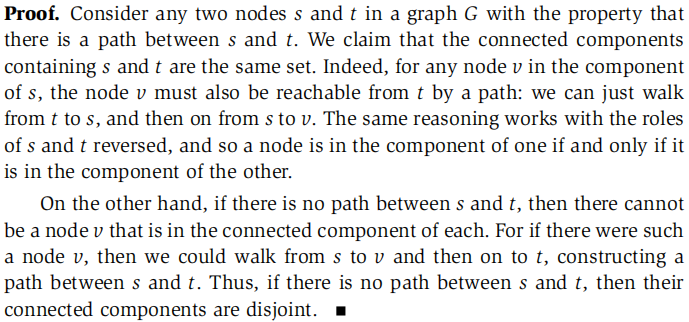
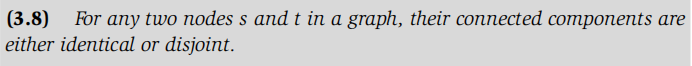
- Running time: O(m+n)



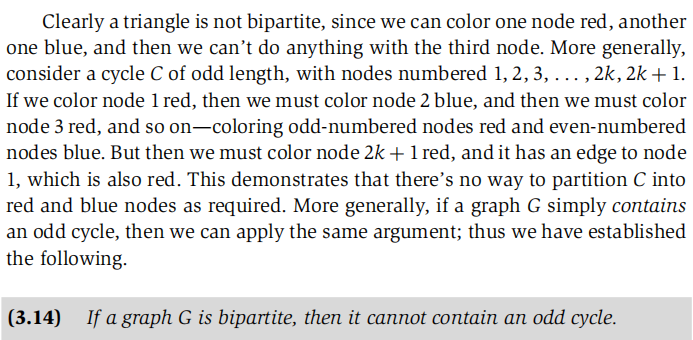
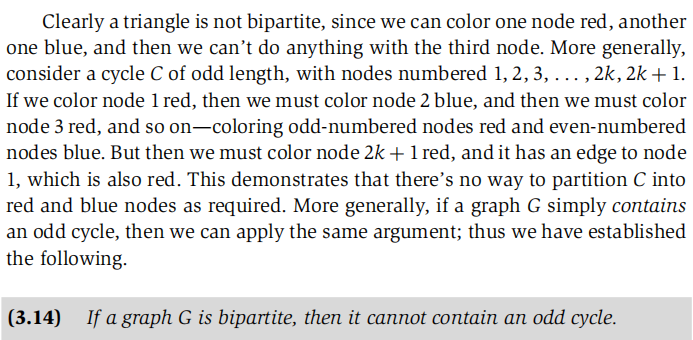








**Bipartite**



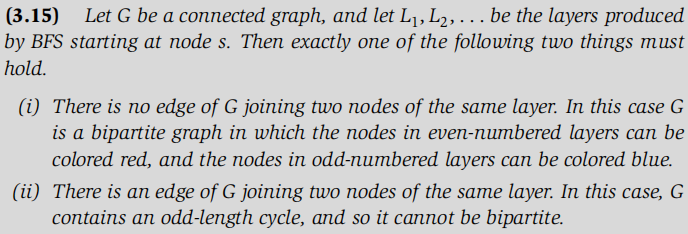
**Coloring algorithm**

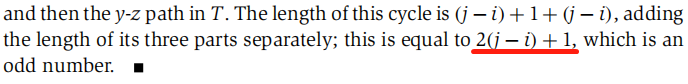
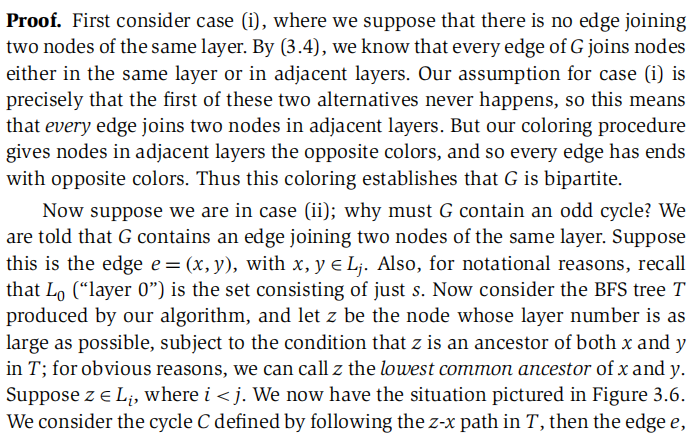
**-** Taking the implementation of BFS and adding an extra array Color over the nodes.

- Whenever we get to a step in BFS where we are adding a node v to a list L[i + 1], we assign Color[v] = red if i+1 is an even number, and Color[v] = blue if i+1 is an odd number.

- Finally, we scan all the edges and determine whether there is any edge for which both ends received the same color.

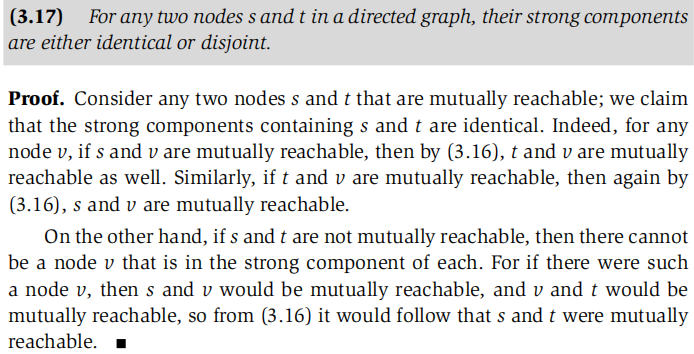
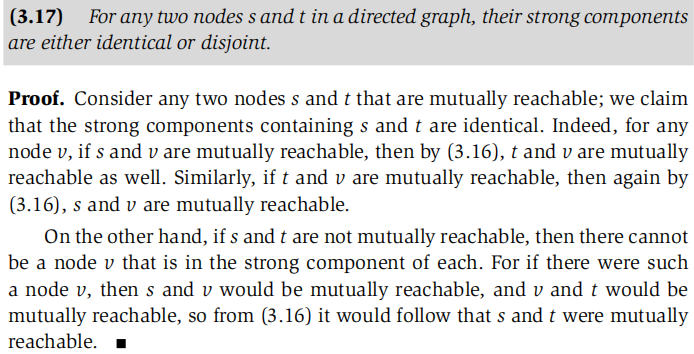
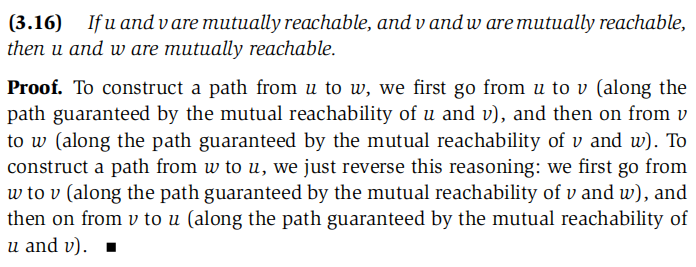
- Running time: O(m+n).





**Directed graph**

**Strong Connectivity**



**Algorithm to test if the graph is strongly connected**

- Brute Force:

Run BFS/DFS from each node.

Running time: O(n2+nm).

- Linear Algorithm:

Use BFS or DFS to find all nodes reachable from S (an arbitrary node) in G. If some nodes are not reachable from S, stop. The graph is not strongly connected.

Otherwise, Create GT (reversing the direction of every edge of G) and repeat the first step on GT .

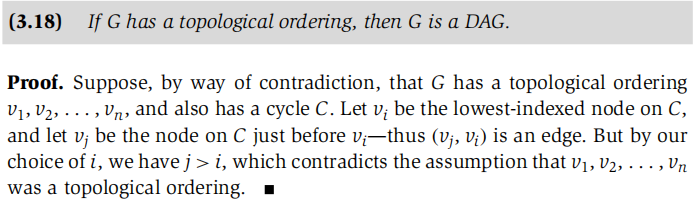
Running time: O(m+n).

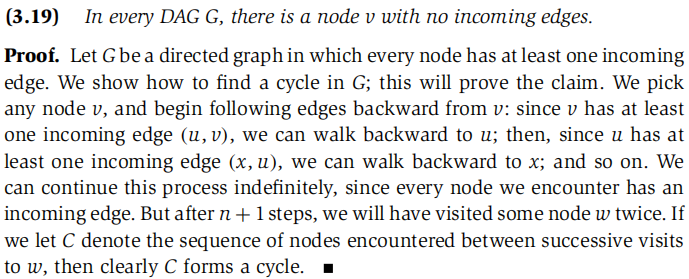
**DAG(Directed Acyclic Graphs)**

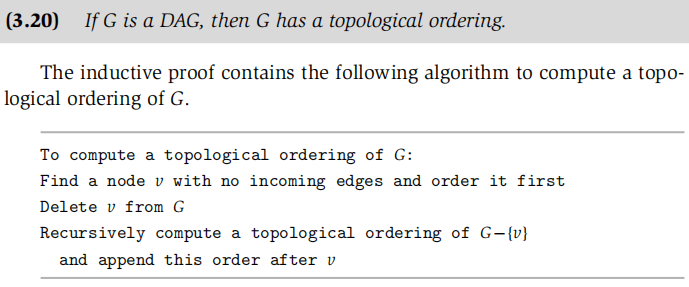
- a directed graph has no cycles

- *topological ordering:* an ordering of its nodes as v1,v2,...,vn so that for every edge

(vi, vj), we have i < j.



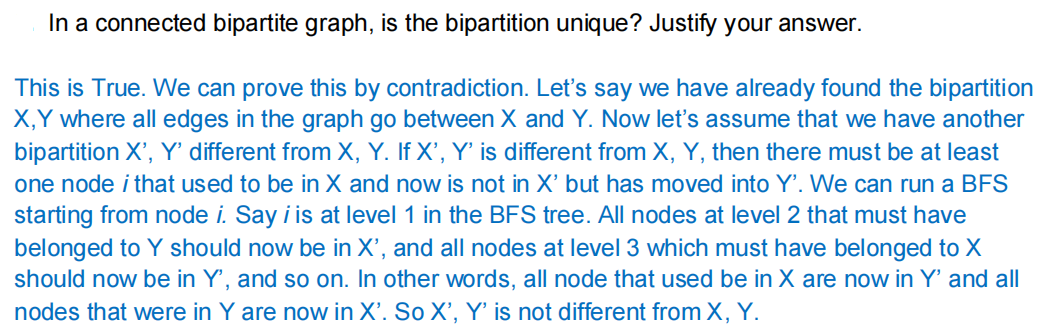




- finding the *longest simple path* in DAG

Find a topological order and see if there is a path that goes through the nodes of the graph in that topological order. If there is, then this will be the longest path we are looking for. In fact, if this path exists, it gives us a strict precedence order from the starting node to the end node on the path, and therefore the graph can only have one topological ordering. So, if our topological order does not provide such a path, then we can conclude that such a path does not exist in the graph.

**Sample Question:**



1. Greedy

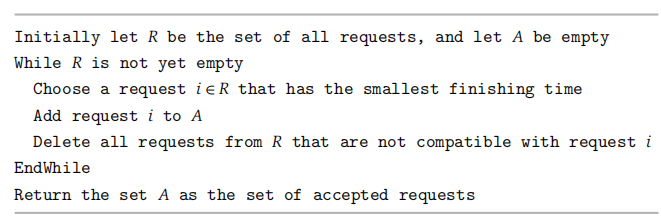
**Interval scheduling problem**

- Input: set of requests {1,...,n}

- ith request starts at s(i) and ends at f(i)

- Objective: To find the largest compatible subset of these requests. (optimal)

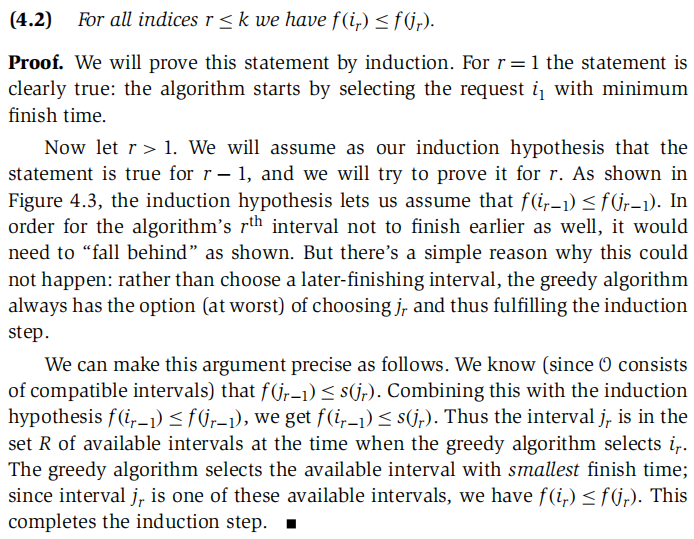
- Algorithm: Choose a request that has the earliest finishing time

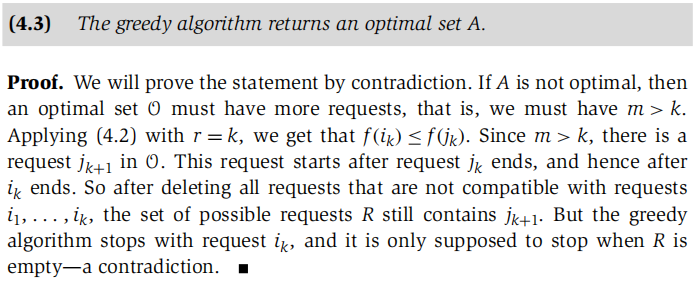


**Analyzing**

- let O be an optimal set of intervals, the set of requests in O be denoted by j1,j2...jm







- Running Time: sorting requests take O(nlogn), and select them take O(n)

**Fractional Knapsack**

- Knapsack has a weight capacity of W

- Input: a set of n objects with weight wi and value vi

- Objective: Fill up the knapsack to its weight capacity such that the value of items in knapsack is maximized.

- Solution: Calculate value per unit weight and choose the object that has higher value per unit weight.

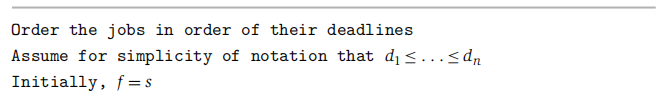
**Scheduling to Minimize Lateness**

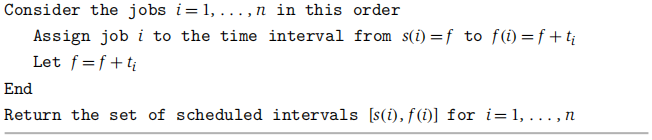
- Requests can be scheduled at any time and each of them has a deadline

- *lateness of a request:*

*-* Objective: Minimize the Maximum lateness L = maxili

- Solution: Schedule jobs in order of their deadline without any gaps between jobs.



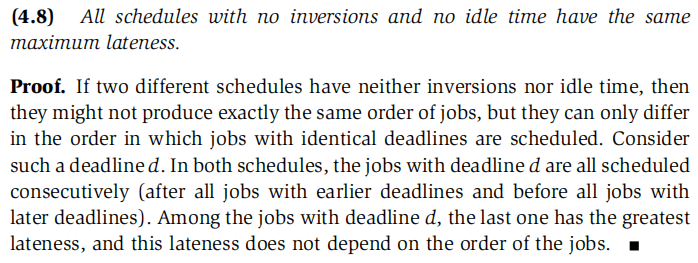


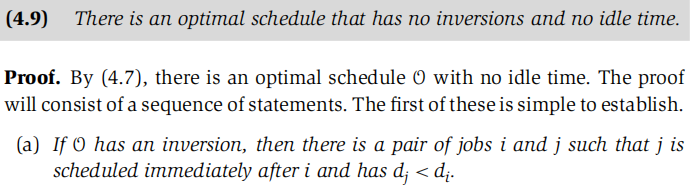
**Analyzing**

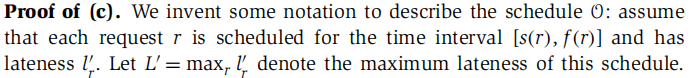
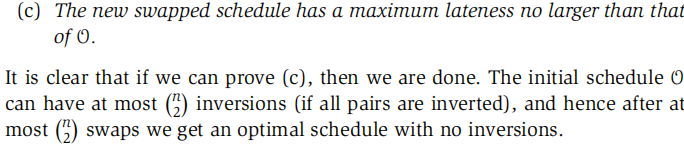
- *exchange argument:*let O be an optimal set of schedule and we gradually modify it, preserving its optimality at each step, but eventually transforming it into a schedule that is identical to the schedule A found by the greedy algorithm.

- *inversion*: a job i with deadline di is scheduled before another job j with earlier deadline dj < di.

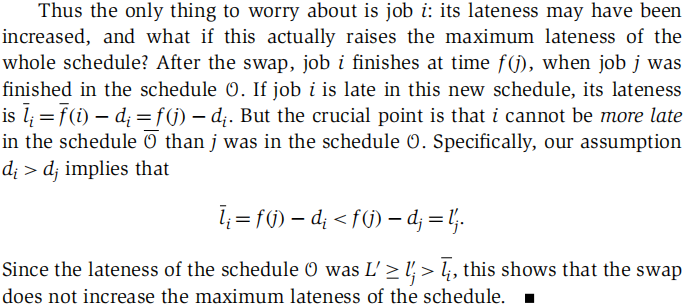


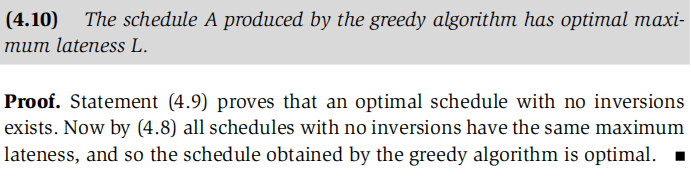






before the swap f(j) = after the swap f(i), Thus all jobs other than jobs i and j finish at the same time in the two schedules. Moreover, job j will get finished earlier in the new schedule, and hence the swap does not increase the lateness of job j.





1. Priority Queues

- *full binary tree*: a binary tree of depth k which has exactly 2k-1 nodes.

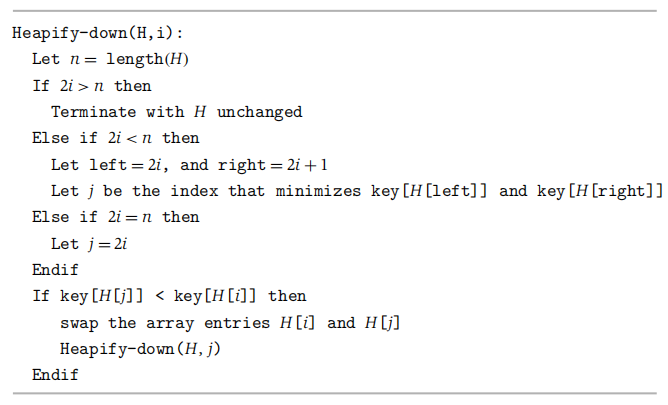
- *complete binary tree*: a binary tree with n nodes and of depth k, its nodes correspond to the nodes which are numbered 1 to n in the full binary tree of depth k,

**Binary Heap**

- a data structure for implementing a priority queue

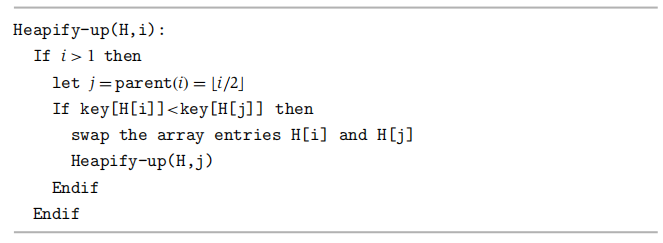
- *max heap*: a complete binary tree with the property that the value (of the key) of each node is at least as large as the values at its children.

**Delete heap elements (in min heap)**



Running time: O(logn)

**Insert heap elements (in min heap)**



Running time: Insert takes O(logn), Find-Min takes O(1)

**Construction:**

1. n inserts takes O(nlogn).
2. Bottom-up construction takes O(n) [eg. Merging two binary heaps of size n]

**Top k values in the array (k<n)**

- Input: An unsorted array of length n

- Constraints: no additional O memory and algorithm runs in time O(nlogk)

- Solution:

1. construction of heap takes O(k)
2. going through the rest of (n-k) elements & extract & insert takes O((n-k)logk).

**Binomial Heap**

- *binomial tree:* an ordered tree defined recursively, B0 consists one node and Bk consists of 2 binomial trees Bk-1 that are linked together.

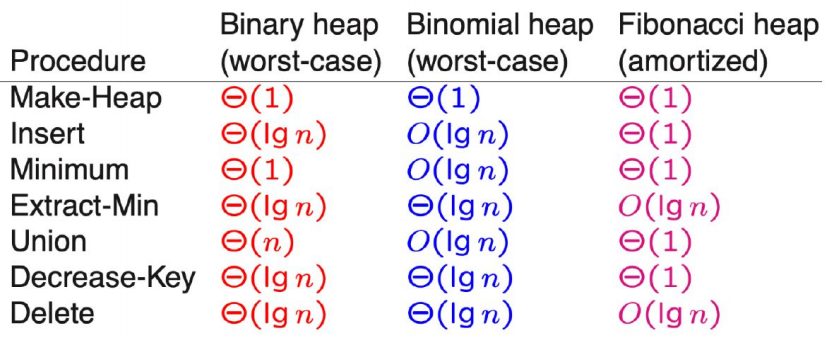
- *binomial heap* H is a set of binomial trees that satisfies the following properties:

1. Each binomial tree in H obeys the min-heap property.
2. For any non-negative integer k, there is at most one binomial tree in H whose root has degree k.

- Find-Min & Insert & Delete & Merge takes O(logn), and construction takes O(n)

**Fibonacci Heap: loosely based on binomial heaps**

- a collection of min-heap trees similar to binomial heaps, however, trees in a Fib heap are not constrained to be binomial trees. Also, trees in Fib heaps are not ordered.



- All construct takes O(n), binary heap takes O(n) to merge and Fib heap takes O(1).

**Amortized Cost Analysis**

- Guarantees the avg. performance of each operation in the worst case

1. *Aggregate Analysis*: a sequence of n operations (for all n) takes worst-case time T(n) total, so in the worst case, the amortized cost per operation will be T(n)/n.
2. *Accounting Method*: assign different charges to different operations. If the charge for an operation exceeds its actual cost, the excess is stored as credit which can later help pay for operations whose actual cost is higher than their amortized cost.

- Total credit at anytime = total amortized cost - total actual cost.

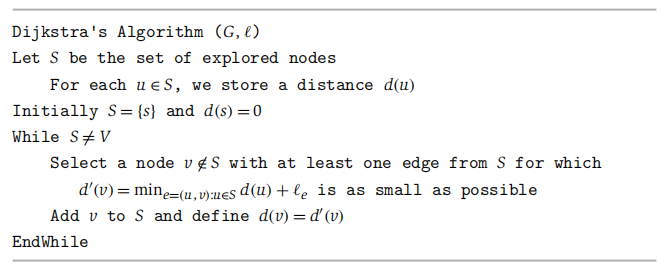
- Total credit can never be negative.

1. Shortest Path

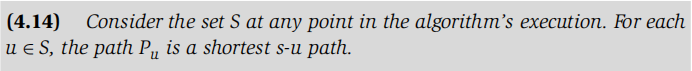
**Problem Statement**

- Given G = (V, E) with w(u,v)≥0 for each edge (u,v)E, find the shortest path from sV to V-S.

**Dijkstra’s Algorithm**

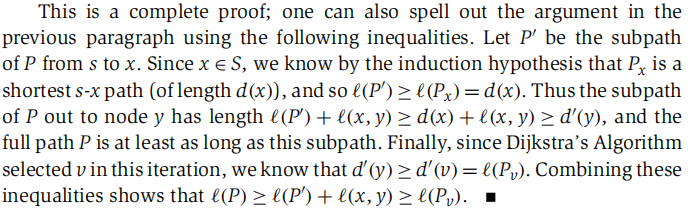


**Analyzing**



- Base case: |S|=1, S={s} and d(s) = 0

- Inductive step: Suppose the claim holds when |S|=k for some k≥1. We now grow S to size k+1 and prove that we have found the shortest path to the new node. let y be the first node on P that is not in S, and let xS be the node just before y.



**Implementation**

- Initialize priority queue Q with all nodes V where d(v) is the key value (all d(v)s are ∞, except for s where d(s)=0). [O(n)]

- While S≠V [O(n)]

v=Extract\_Min(Q)

S=S∪{v}

For each vertex u∈Adj(v) [O(n)]

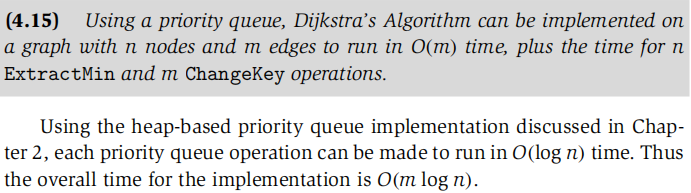
If d(v)＞d(v)+le

ChangeKey(Q,u,d(v)+le)

Endfor

Endwhile

- Running Time: O(n2) and using heap can take O(mlogn)

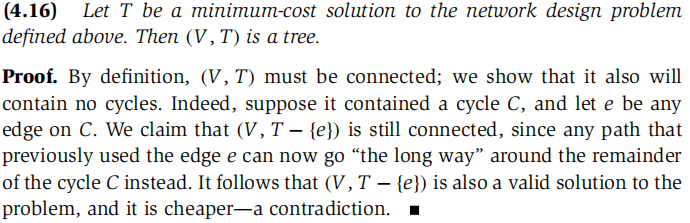


|  |  |  |  |
| --- | --- | --- | --- |
|  | Binary Heap | Binomial Heap | Fibonacci Heap |
| n.Extract\_Min’s | O(nlogn) | O(nlogn) | O(nlogn) |
| m.ChangKey’s | O(mlogn) | O(mlogn) | O(m) |
| Total | O(mlogn) | O(mlogn) | O(m+nlogn) |
| For sparse graph | O(nlogn) | O(nlogn) | O(nlogn) |
| For dense graph | O(n2logn) | O(n2logn) | O(n2) |

1. MST

- *spanning tree*: Any tree that covers all nodes of a graph.

- *minimum spanning tree (MST)*: A spanning tree with minimum total edge cost.



**Kruskal’s algorithm**

- Sort all edges in increasing order of cost. Add edges to T is this order as long as it doesn’t create a cycle. If it does, discard the edge.

**- Reverse-Delete algorithm:** Backward version of Kruskal’s. Start with deleting.

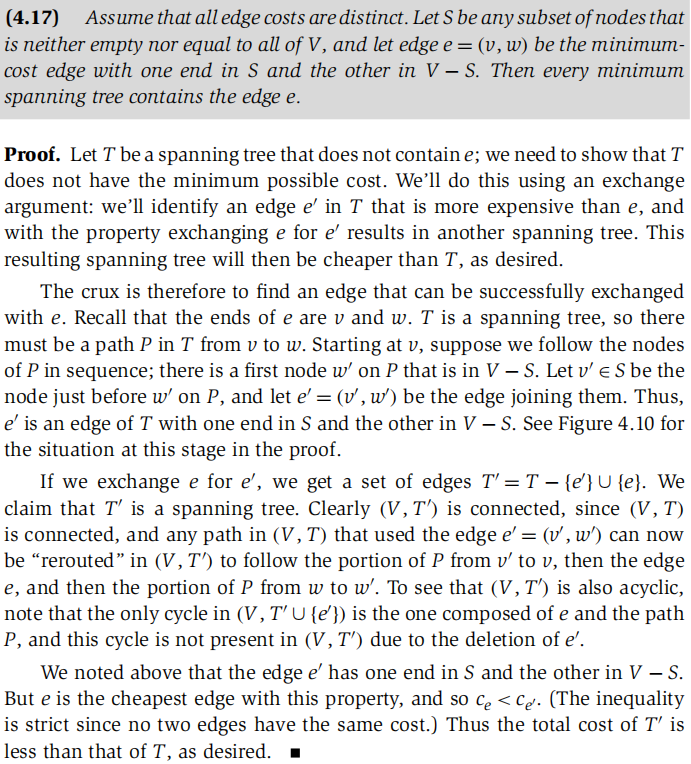
**Prim’s algorithm**

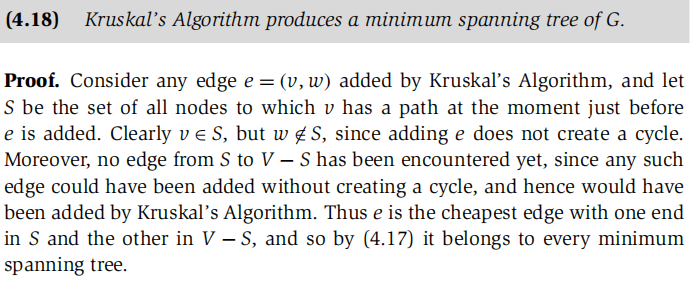
- Similar to Dijkstra’s algorithm, start with a node of a node set S (initially the root node) on which a MST has been constructed so far. At each step, grow S by one node, adding the node v that minimizes the attachment cost.

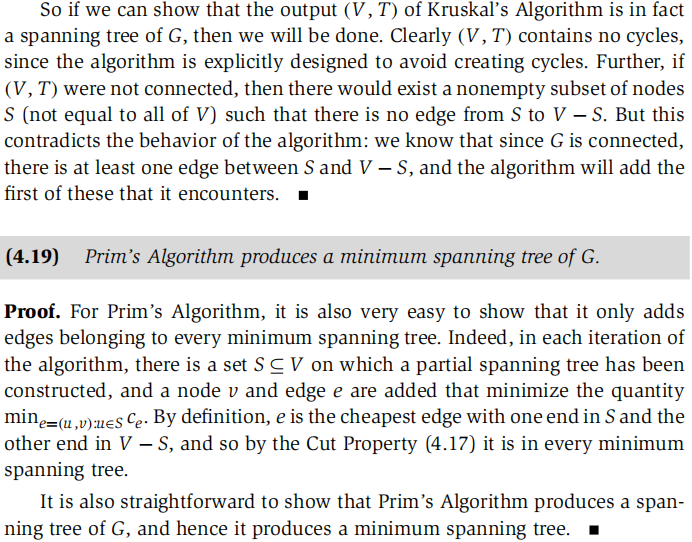
**Analyzing**

- The Optimality of Kruskal’s and Prim’s Algorithms:

The point is that both algorithms only include an edge when it is justified by the Cut Property (4.17).

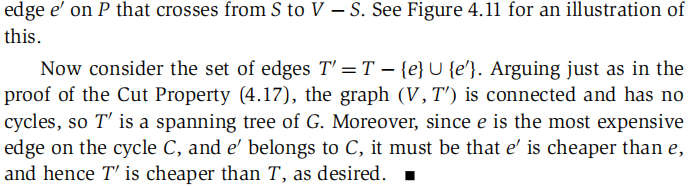
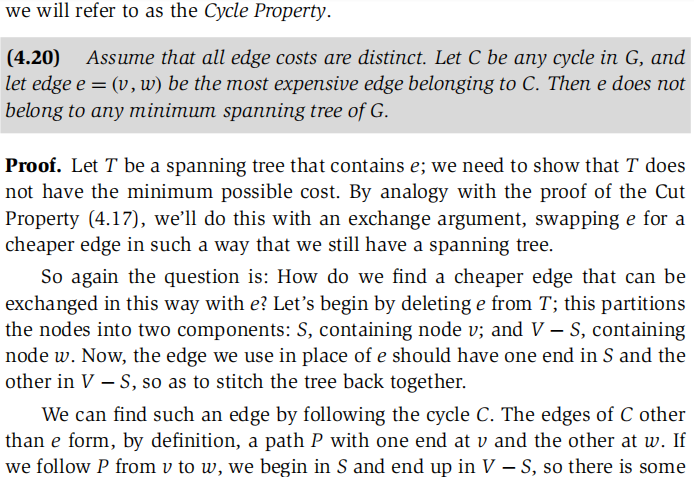


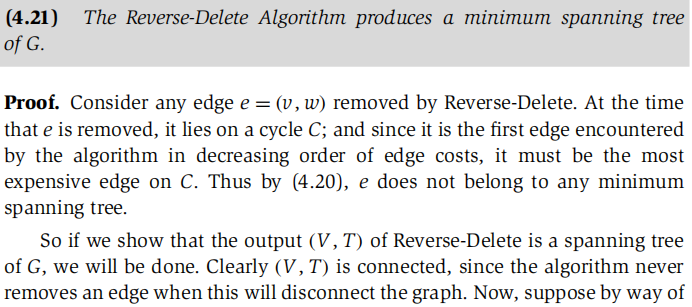


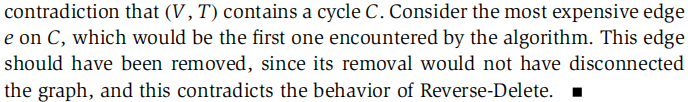


- The Optimality of Reverse-Delete Algorithms:

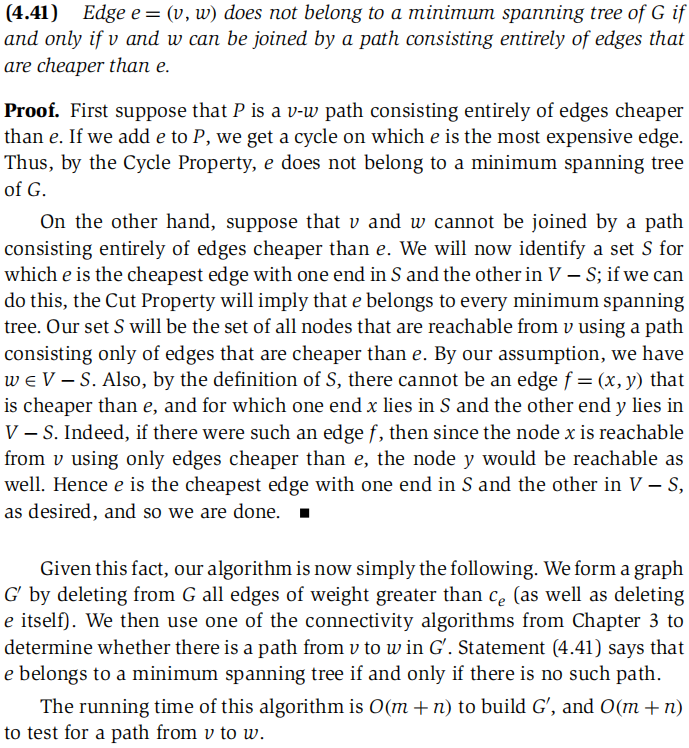
Reverse-Delete only adds an edge when it is justified by Cycle Property (4.20).







- Decide whether a particular edge e is contained in a minimum spanning tree of G.



**Implementation of Prim’s**

- Initialize priority queue Q with all nodes V where d(v) is the key value (all d(v)s are ∞, except for s where d(s)=0). [O(n)]

- While S≠V [O(n)]

v=Extract\_Min(Q)

S=S∪{v}

For each vertex u∈Adj(v) [O(m)]

If d(v)＞le

ChangeKey(Q,u,le)

Endfor

Endwhile

- Running Time: O(mlogn)

**Implementation of Kruskal’s: Union-Find**

- create an independent set for each node, and A=null. [O(n)]

For each vertex v∈V

Make-set(v)

End for

- sort edges in non-decreasing order of weight [O(mlogm)]

- for each edge (u,v)∈E, taken in this order [O(m)]

if Find-set(u) ≠ Find-set(v) then [O(logn)]

A = A∪{(u,v)}

Union(u,v)

Endif

Endfor

- Running time: O(n)+O(mlogm)+O(mlogn)=O(mlogm)



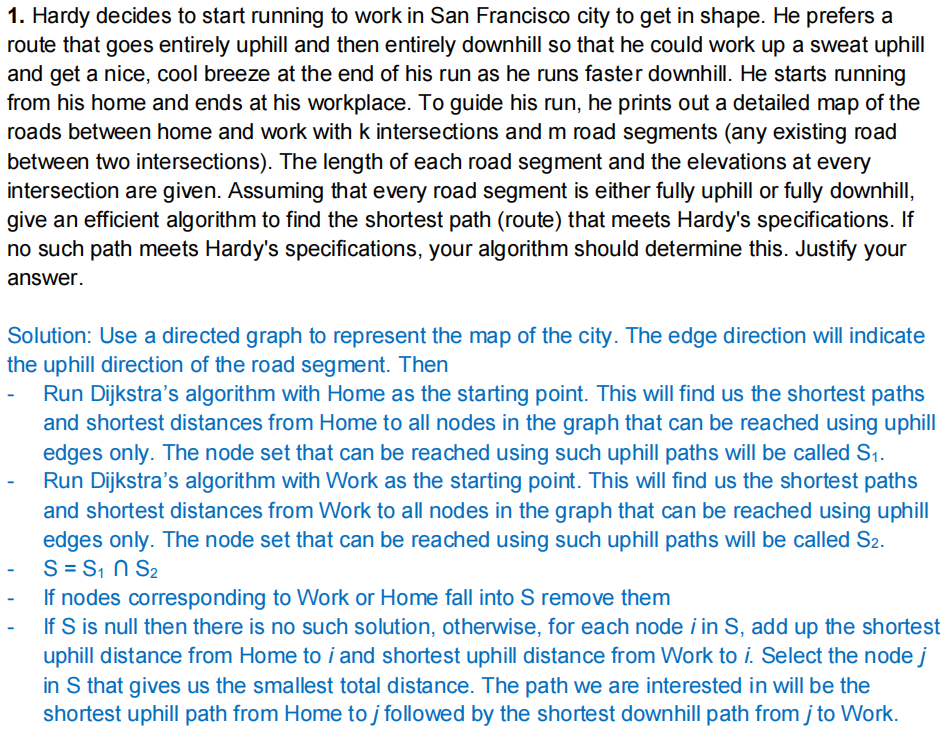
**Implementation of Reverse-Delete**

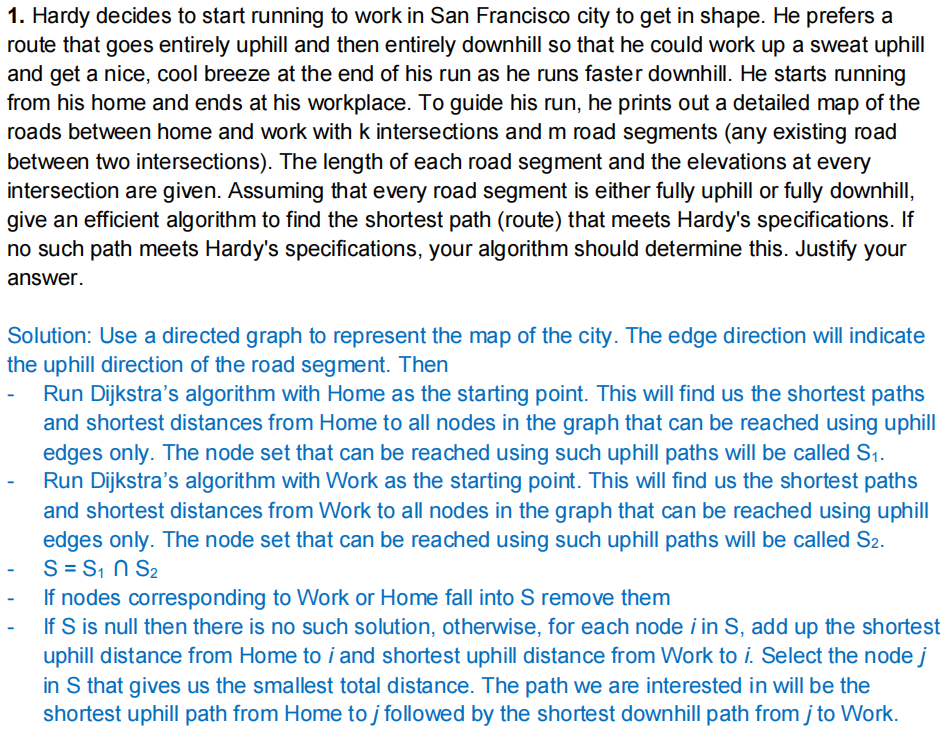
- Sort edges in decreasing order of cost and loop once edges in this order

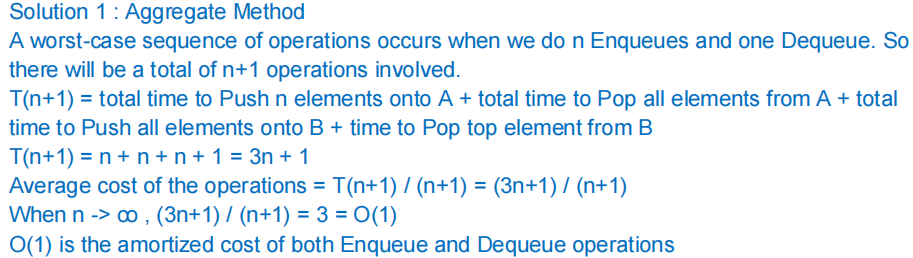
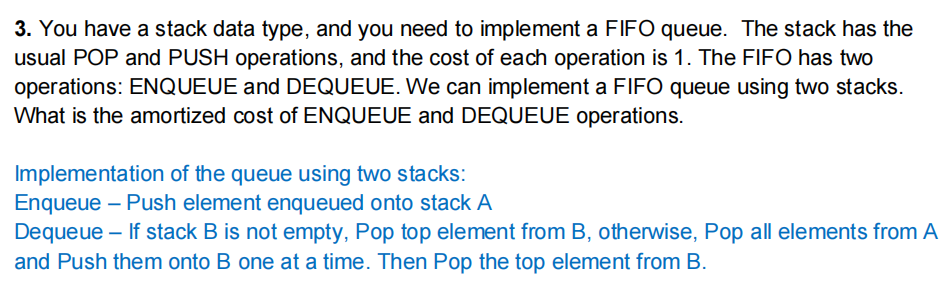
- Run BFS/DFS to find if it could be removed.

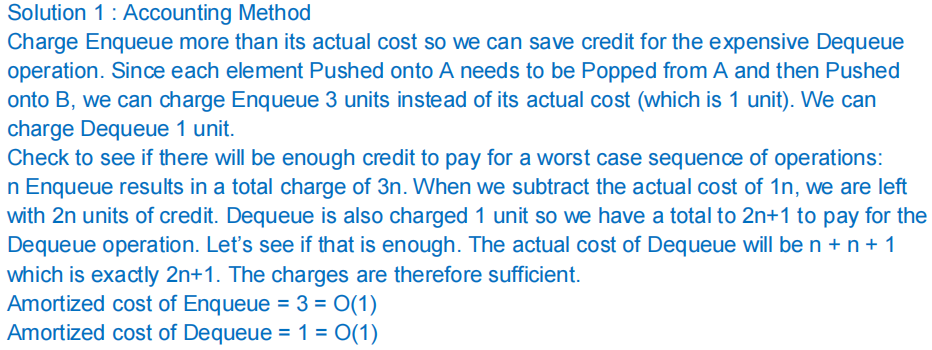
- Running Time: O(m2)

**Sample Question**









1. Divide and Conquer

- Divide problem into n sub-problems

- Conquer: i.e. solve the sub-problems recursively, or if trivial solve the problem itself

- Combine the solution to the sub-problems

**Merge-Sort**

MERGESORT(A,p,r)

If p<r then

Q=[(p+r)/2] [O(1)]

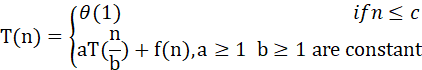
MERGESORT(A,p,q) [2T(n/2))]

MERGESORT(A,q+1,r)

MERGE(A,p,q,r) [O(n)]

Endif

**Master Theorem**



- , 

- a is number of sub-problems of each step and b is the size of each sub-problem





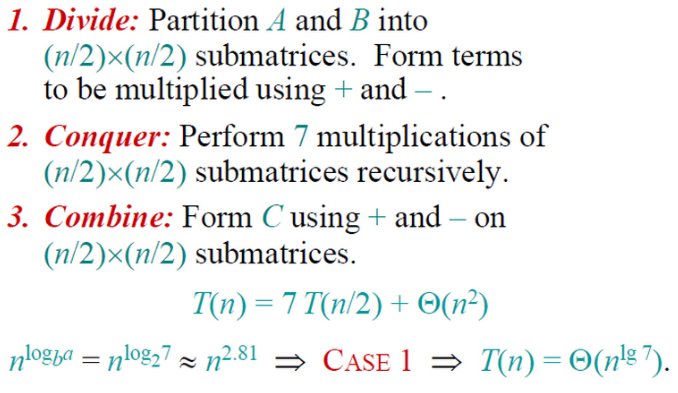
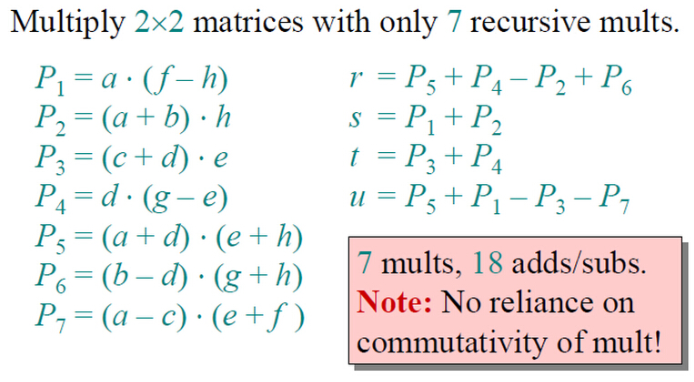
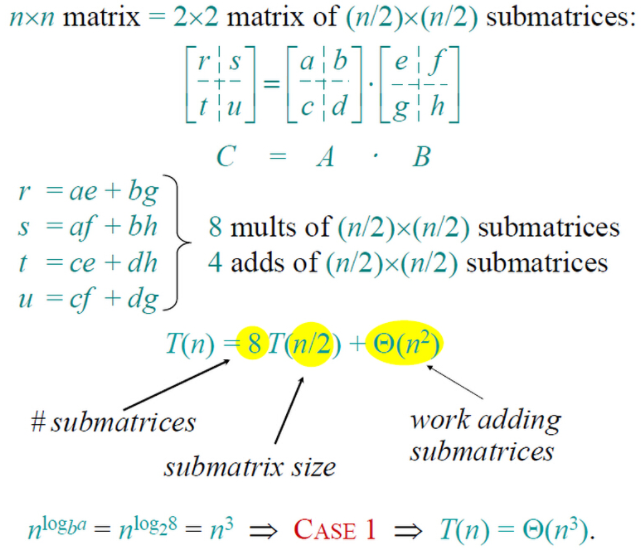


Closet pair of points problem (2D)

- Brute-force takes

- Divide and Conquer takes O(nlogn)

**Dense matrix multiplications：Strassen Algorithm**



**Sample Question**

