**Greedy Algorithm Summary**:

**Greedy General Idea**: Making the locally optimal choice at each stage, hope to find a global optimum. 在每一步寻求最好的解.

**Standard Correctness Proof:** Exchange Argument

**Examples of problem solved using a greedy approach**

Interval Scheduling / Partitioning

Scheduling to minimize lateness

Shortest path (Dijkstra’s Algorithm)

Minimum Spanning Trees (Cut Property, Cycle Property, Prim’s Algorithm, Kruskal’s Algorithm)

**Greedy Template**: Consider jobs in some order. We can use some counterexamples to eliminate some possibilities. Take each job provided it is compatible with the ones already taken. (要与前一个job兼容). Therefore, we need to sort the jobs in some order first. Sorting usually costs O(nlogn), prove by exchange argument or proof by contradiction.

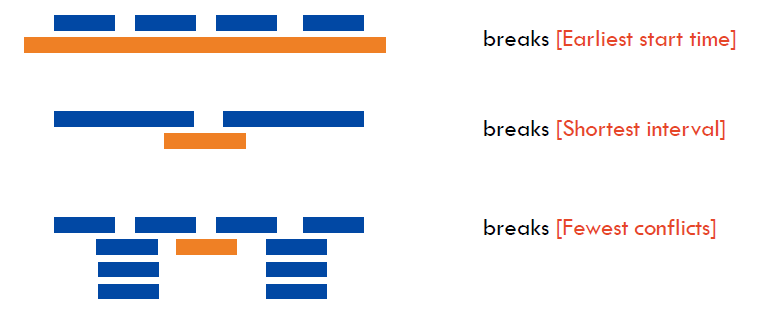
**Interval Scheduling Problem**:

Compatible: Two jobs are compatible if they don’t overlap in time.

Goal: find the maximum subset of mutually compatible jobs.

Greedy Algorithm: Increasing order of Earliest finish time is the optimal solution.

Time complexity O(nlogn)



**Exchange Argument:** Gradually transform any solution to the one found by the greedy algorithm **without hurting its quality**

**Proving the correctness of greedy algorithm**: Exchange Argument. Let suppose we have a greedy solution, and an optimal solution. And let’s suppose that k is the smallest index where the greedy solution is different from the optimal. Prove that the greedy is actually more optimal than that so-called optimal solution. Therefore, k is not the smallest index where the difference happens.

1. Define your solution: Greedy solution X and an optimal solution OPT
2. Compare solutions: if X!= OPT, then they must differ in a specific way.
3. Exchange pieces: Transform OPT by exchanging some piece of OPT for some piece of X. Prove the cost doesn’t increase, which means this exchange doesn’t make the solution worse.
4. Iterate: Argue optimality. By iteratively exchanging pieces one can turn Xopt into Xgreedy without impacting the quality of the solution. Or we can argue that there is a contradiction with the largest possible index of the same values of two solutions. (After exchanging, the solution is still feasible and optimal, but contradicts maximality of the index r, since r is already assumed to be the largest possible index, and this is done by proof by contradiction.)

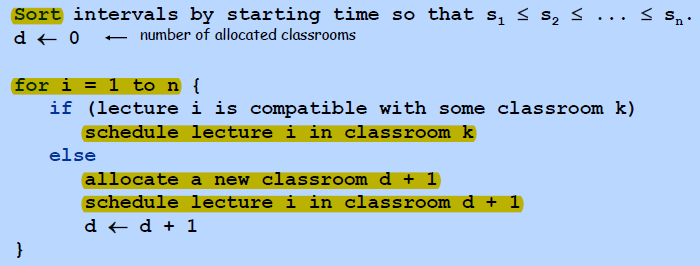
Theorem: There exist a greedy algorithm [Earliest Finish Time] that computes the optimal solution in O(n\*logn) time.

**Interval Partitioning**:

**Goal**: find the minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

**Observation**: Number of classrooms needed >= depth.

**Solution**: Sorted in earliest starting time, 分配到已有教室，或者新建教室



**Running time**: computes the optimal solution in O(nlogn) time. Done using Priority Queue.

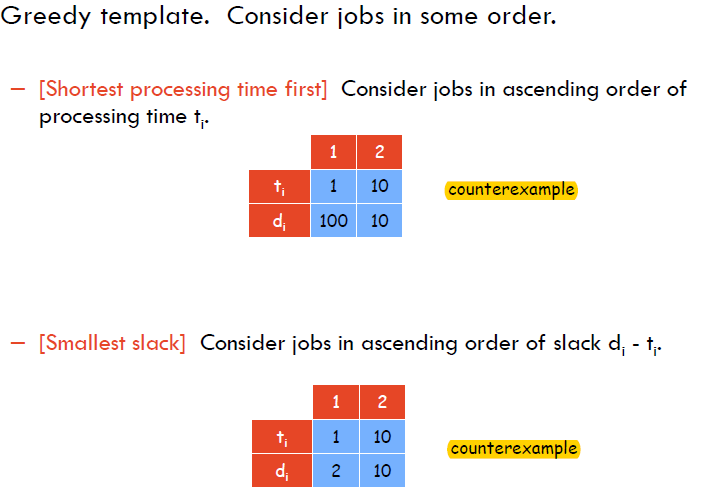
Theorem: There exists a greedy algorithm [Earliest Starting Time] that computes the optimal solution in O(n\*logn) time.

**Minimizing lateness**:

Job I requires ti units of processing time and is due at time di . It starts at time si and finishes at time fi = si + ti

Lateness: Li = max(0 , fi – di ).

Goal: To minimize maximum lateness L = max(Li)



**Observation (No idle time)**: The greedy schedule has no idle time.

**Theorem**: Done by sorted with earliest deadline first, computes in O(nlogn) time.

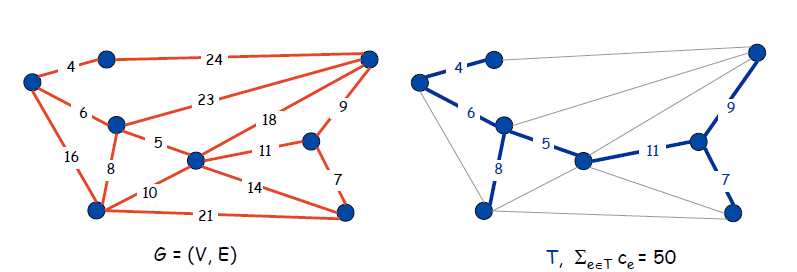
**Inversion**: a pair of job j and k such that j < k by deadline, but k is schedule before j.

**Observation (No Inversions)**: Greedy schedule has no inversions.

Observation: If a schedule (without idle time) has an inversion, it has on with a pair of inverted jobs scheduled consecutively.

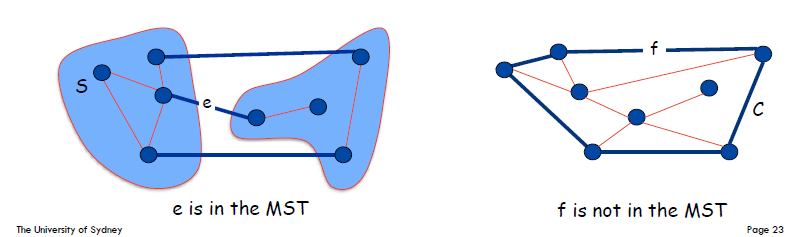
**Claim** (Swapping the inversions): Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

**Minimum Spanning Tree MST**: Given a connected graph G = (V, E) with real value weights ce , MST is a subset of edges such that this subset is a spanning tree whose sum of edge weights is minimized.



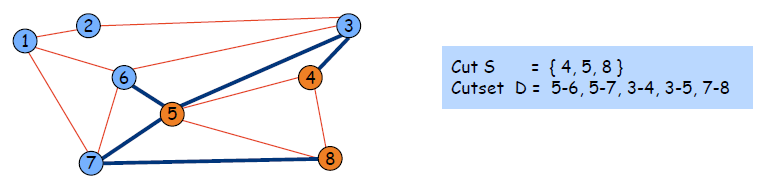
**Cut property**: Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoints in S. Then the MST contains e.

**Cycle Property**: Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f.



**Cut**: A cut is a subset of nodes S.

**Cut Set**: A cut set of S is the subset of edges with exactly one endpoint in S.



**Cut property**: Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T\* contains e. Prove this by exchange argument.

**Prim’s Algorithm**: Prim’s Algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. Using priority queue, every time choose the smallest weighted edge to enter the basis, then update, the repeat, remember to update the predecessor. O(mlogn) with binary heap. (Prim’s using Cut Property)

Firstly, choose a node randomly as the starting node, and add this node into the visited set. And then for all the nodes which is reachable from all the nodes in the visited set, choose one which has the smallest distance, and add this node into the visited set, and keep looping like this. Finally, this algorithm will produce the minimum spanning tree, which is a minimum weight connected graph without any cycles.

**Running time**: O(n2) using array, O(m\*logn) using a binary heap.

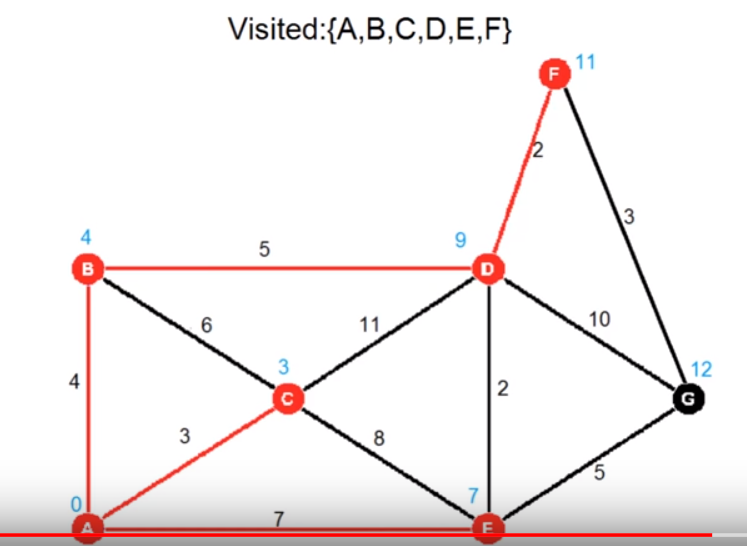
**Kruskal’s Algorithm:**

1. Consider edges in ascending order of weight. (Need to sort them in advance)
2. If adding e to MST creates a cycle, discard e according to Cycle Property
3. Otherwise, insert e into MST according to Cut Property

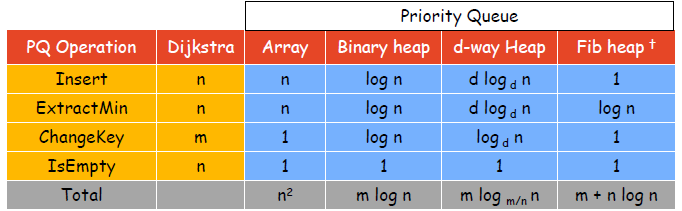
**Shortest path in the graph (Dijkstra’s Algorithm)**: Dijkstra’s Algorithm. Used to find the shortest path in a directed graph without negative cost edges. Runs in O(m+nlogn) time. M->edges, n->nodes. Implemented using Priority Queue.

**Procedures of Dijkstra’s Algorithm**:

Assign to every node a distance value: set it to 0 for initial node and to infinity for all other nodes. Keep a set of visited nodes, this set starts with just the initial node itself. For the current node, consider all of its unvisited neighbours and calculate the distance to the current node + distance from current node to the neighbour. If this is less than their current tentative distance value, replace it with the newly calculated value. When we finish calculating consider all the unvisited neighbours, choose the one with the smallest distance value, add it to visited set, and keep looping. If the destination node has been marked visited, the algorithm terminates.



**Analysis of Dijkstra’s Algorithm**: This algorithm runs in O(|E|+|V|log(|V|)), when using a binary heap. It has the space complexity of O(|V|), when using the binary heap.



**Proof**: Prove Dijkstra’s Algorithm by using Proof by Induction.