01:XXX:XXX - Homework n

Pranav Tikkawar

April 9, 2025

Contents

1 In person Notes on Midterm 2

2 Greedy Algorithms

1. **Coin Change**
   There are n possible denominations of coins: $a_1 < a_2 < \cdots < a_n$, and all are positive integers. What is the minimum number of coins that sum up to m? Note we are using the US-coin system, which has denominations of 1, 5, 10, 25, 50, and 100 cents.

   Solution: Take as many of the largest denomination coins as possible until you reach the target amount. Then, take as many of the next largest denomination coins as possible until you reach the target amount. Repeat this process until you reach the target amount.

   Proof. Let m be the target amount. Let $1, 5, 10, 25, 50, 100$ be the denominations of coins. Let c_x be the number of coins of denomination x used.
   $c_1 < 4$ since if $c_1 >= 5$ then we can replace 5 coins of denomination 1 with 1 coin of denomination 5 resulting in a smaller number of coins.
   Similarly $c_5 < 1$
   $c_{10} < 2$
   $c_{25} < 1$
   $c_{50} < 1$
   Then if $m > 100$ then we can take as many 100 coins untill $m - c_{100} * 100 < 100$ and do the same logic as before.
   This proves that the greedy algorithm works for the US coin system. □

2. **Most Valuable Subset**
   Given a set of n distict positive integers, $A = \{a_1, a_2, \ldots, a_n\}$, and a positive integer k, find a subset of A with size at most k that has the largest sum.

   Solution: We use the method of meadian of medians to find the k-th largest element in A. Then we take all the elements greater than the k-th largest element. This will give us the largest sum.

   Proof. Let O be the optimal solution. If $|O| < k$ then we can add any element of A to make the sum of O larger. Thus $|O| = k$.
   Suppose for contradiction that O does not contain one of the largest k elements of A. Then we can replace one of the elements of O with the largest element of A to get a larger sum.
   Thus the optimal solution must contain the largest k elements of A and the greedy algorithm works. □

3. **Interval Scheduling** There are n jobs. The job i starts at time ℓ_i and ends at time r_i (distinct). Find the maximum number of jobs we can take without any overlap.

Solution: We can first sort the jobs in increasing order of deadline r_i and then remove all the intervals that overlap with it.

Proof. Let O be the optimal solution. And G be the greedy solution. Let us also sort the jobs in increasing order of r_i.
Let Let $[\ell_i, r_i]$ be the first interval that is in O but not in G.We know that know that in the corresponding greedy solution G we have chose an interval j with $r_j < r_i$ and it must be compatible with the rest of the intervals in O.
Thus we can replace j with i in G to get a same size feasible solution.

□

4. **Weighted Interval Scheduling** There ar n jobs. The job i starts at time ℓ_i and ends at time r_i and has value v_i. Find the maximal total value of jobs we can take without any overlap.

Solution: We can do this using DP.
sort the jobs in increasing order of r_i. Then we take the function $f(k) = \max_{j<k:r_j<\ell_k} f(j)+ v_k$
This runs in $O(n^2)$ time. We can also do this in $O(n \log n)$ time using binary search. To do this we would need to keep track of the last job that does not overlap with the current job.

5. **Scheduling to Meet Deadlines** There are n jobs. The job i takes time t_i and must be done be the deadline d_i. We can only do one job at a time. Starting at time 0, Is it possible to complete all jobs before their deadlines?

Solution: The last job will need to be finished at time $T = \sum_{i=1}^{n} t_i$. then we can see that if no job i has $d_i \geq T$ then it is impossible to schedule the job i before its deadline.
Thus we can sort the jobs in increasing order of d_i and then check if $d_i \geq T$ for all jobs.
Then we can schedule a job i at time $T - t_i$ and recursively schedule the rest of the jobs.

Proof. Let O be an optimal solution. If O is not a greedy soltution there there is a consecutive inversion i, j such that $d_i > d_j$ and $t_i < t_j$. Then we can swap the jobs to get a feasible solution with a smaller makespan.
Thus the greedy solution is optimal. □

6. **Hook Chain**
There are n hooks. The hook i has weight w_i and weight limit l_i Is it possible to chain all the hooks together without exceeding the weight limit of any hook?

Solution: This is similar to the Interval scheduling problem. We can order the hooks by $d_i = w_i + l_i$ and have our $t_i = w_i$ we can then use the same greedy algorithm as before which puts them in order and returns false if there is an overlapping interval.

7. **Interval Covering**
There are n intervals $[a_i, b_i]$. What is the minimum number of intervals that we have to pick in order to cover the entire interval $[0, m]$?

Solution: We can sort the intervals in order of b_i and pick the last interval that overlaps with the previous interval until you reach a $b_i > m$.

Proof. Sort the intervals in order of increasing b_i. Let O be the optimal solution and let G be the greedy solution.
Let $[a_i, b_i]$ be the first interval that is in O but not in G. We know the corresponding step, the greedy algorithm must have chose an interval with $b_j > b_i$ and it must overlap with the prior interval. Thus we can replace j with i in G to get a same size feasible solution.
Thus the greedy algorithm is optimal. □

2.1 Greedy Continued

8. **Minimizing Lateness**
There are n jobs. The job i takes time t_i and has a soft deadline d_i. We can only do one job at a time. Starting at time 0, what is the minimum lateness L so that we can complete all the jobs while no job is late for more than L time?

Solution: We can sort the jobs in increasing order of d_i and then schedule the jobs in that order.

Proof. Let O be an optimal solution. If O is not a greedy solution then there is a consecutive inversion i, j such that $d_i > d_j$ and $t_i < t_j$. Then we can swap the jobs to get a feasible solution with a smaller makespan.
Thus the greedy solution is optimal. □

9. **Meet Most Deadlines**
There are n jobs. The job i takes time t_i and has a deadline d_i. We can only do one job at a time. Starting at time 0, what is the maximum number of jobs we can complete before their deadlines?

Solution: Sort the jobs in increasing order of d_i. We can use DP to find the maximum number of jobs that can be completed before their deadlines.
Let $f(k, p)$ be the minimum time needed to complete p jobs on time among the first k jobs.
$f(k,p) = \min \begin{cases} f(k-1,p) \\ f(k-1,p-1)+t_k \quad \text{if} f(k-1,p-1)+t_k \leq d_k \end{cases}$
We can do this a greedy implementation:
We maintain a set P of picked jobs starting from $\emptyset$. We can add a job i into P in increasing order of d_i and if the deadline cannot be men the remove the job in P that takes longest to do.
This can be done by a SBBST or a binary heap.

10. **Optimal Offline Caching**
There is a stream of n queries: $q_1, q_2, \ldots, q_n$. We need to answer each query when it arrives.
We have a cache of size k. We can use the cache to store the answers to a most k queries. We can quickly deal wth any query in the cache. Otherwise, we have to cache it, possibly evict something in the cache, and then answer the query.
How to minimize the number of cache misses if we know the stream of queries beforehand (offline)?

Solution: Whenever we have to evict something in the cache, evict the item that is needed farthest into the future.

Proof. Let O be the optimal solution and G be the greedy solution. We can use the exchange argument to show the optimality of the greedy solution.
Suppose O and G are the same uop to a point r_i where O evicts i and G evicts j. After this point O will behave optimally deosnt incurre any more cache misses. than S *** □

11. **Merging Stones**
Given n piles of stones with sizes $a_1, a_2, \ldots, a_n$. We want to mergae all of them into one pile. In each step we can merge any two piles into one and pay a cose equal to the sum of sizes of the two piles. What is the minimum cost to merge all the piles into one?

Solution: We want to sort the piles in increasing order of size and then merge the two smallest piles together. We can do this using a min heap.

Proof. □

3 Minimum Spanning Tree

Definition (MST)**.** A minimum spanning tree of a graph is a spanning tree with the minimum weight. A spanning tree of a graph is a subgraph that is a tree and contains all the vertices of the graph.

Definition (Cut Set)**.** A cut of a graoh is the partition of the vertices in a graoh into two disjoint subsets. The cut set is the set of edges that cross the cut.

Theorem 1 (Cut Property)**.** *In any cut-set of the graph if an edge e is strictly lighter than every other edge it must belong to every minimized spanning tree.*

Definition (Cycle Property)**.** In any simple cycle of the graph if an edge e is strictly heavier than every other edge it must not belong to any minimum spanning tree.

Definition (Prim's Algorithm)**.** Prim's algorithm is as follows:
1. Maintain a set of reachble vertices S initialize with an arbitray vertex v.
2. In eah step add the edge with the smallest weight that connects S and $V \setminus S$ to S and update S 3. Repeat until $S = V$.
This works since It is repeated application of the cut property

Definition (Kruskal's Algorithm)**.** Kruskal's algorithm is as follows:
1. Sort the edges in increasing order of weight.
2. Initialize $T = \emptyset$
3. if there is an edge that connects two seperate connected components add it to the solution.
4. Repeat until T is a spanning tree.

Definition (Boruvka's Algorithm)**.** Initially ever vertex is a seperate group
In each round we dint the lightest edge from each group and merge the groups.
Repeat until there is only one group left.

4 Shortest Path

Definition (Dijkstra's Algorithm)**.** Dijkstra's algorithm is as follows:
Maintain a set S of decided vertices. Initially S is just the source s and we maintain a distance for each vertex d_v where d_v is the shortest path from s to v only given known vertices.
We update distances as $\min(d_u, d_v + e(u, v))$

Definition (Bellman-Ford Algorithm)**.** The Bellman-Ford algorithm is as follows:
It is a DP algorithm that works for negative weights.
$f(k, v)$ is the shortest distance from s to v with at most k edges.

$$f(k,v) = \min \begin{cases} f(k-1,v) \\ f(k-1,u) + e(u,v) \quad \text{if} (u,v) \in E \end{cases}$$

This runs in $O(nm)$ time.

Definition (Floyd-Warshall Algorithm)**.**

5 MISC

12. **Longest Increasing Subsequence**
Given a sequence of n distinct integers $a_1, a_2, \ldots, a_n$, find the length of the longest increasing subsequence.

Solution: We can this using DP. where we have the function $f(t)$ being the longest increasing subsequence ending at t.

$$f(t) = 1 + \max_{0<i<t:a_i<a_t} f(i)$$

This runs in $O(n^2)$ time.

6 Practice Midterm

Decide whether each of the following statements is true or false. If it is true, give a brief explanation. If it is false, give a counterexample.

1. Every simple undirected graph with 10 vertices cannot have exactly 11 different minimum spanning trees. (An undirected graph is simple if there is at most one edge between each pair of vertices, and no edge connects a vertex to itself.)

   Solution: False, we can make a graph where every edge has the same weight and then there are $C(10, 2) = 45$ edges. Then we can take any 11 edges and they will all be the same weight.
   Thus we can have $C(11, 2) = 55$ different minimum spanning trees.

2. In the single-source shortest path problem where every edge has a positive weight, the s–t shortest path cannot change if we replace the weight of every edge w_e by its square w_e^2.

   Solution: False since we can take agraph with 3 vertexes A, B, C with weights $1/3, 1/3, 1/2$ for AB, BC, AC respectively. Then the shortest path from A to C is AC with weight $1/2$. But if we square the weights then the shortest path is $AB + BC$ with weight $1/9 + 1/9 = 2/9 < 2/4$.

We have an array of n distinct integers $(a_1, a_2, \ldots, a_n)$ and a parameter $k \in \{1, 2, \ldots, n\}$. The goal is to find k increasing subsequences that are pairwise disjoint, so that the total number of elements in them is maximized. We only need to find the maximum number, not the subsequences themselves. For example, if the input is $n = 8$, $k = 2$, $a = (1, 3, 6, 2, 4, 9, 7, 0)$, then the output should be 7. (These two subsequences can be $(1, 3, 6, 7)$ and $(2, 4, 9)$.)

1. When $k = 1$, the problem is equivalent to the longest increasing subsequence problem. Give an algorithm for it that runs in at most $O(n^2)$ time, and briefly justify its correctness.

   Solution: We can use DP where we have the function $f(t)$ being the longest increasing subsequence ending at t.

   $$f(t) = 1 + \max_{0<i<t:a_i<a_t} f(i)$$

   This runs in $O(n^2)$ time.

2. Consider the following greedy algorithm for the case where $k = 2$: We find an arbitrary longest increasing subsequence s_1, delete it from the original array, and then find another longest increasing subsequence s_2 in the remaining array and output the sum of lengths of s_1 and s_2. Decide whether this algorithm is (always) correct. If so, give a brief explanation, and if not, give a counterexample.

Solution: No, it is not always correct. Is is possible that there is a sequence in the first round that takes up elements that would be better fit in the second round. For example, if we have $5, 1, 2, 3, 6, 7, 8, 4$ then the first round would take $1, 2, 3, 5, 6, 7, 8$ and the second round would take 4 but we could have taken $1, 2, 3, 4$ in the first round and $5, 6, 7, 8$ in the second round.
Thus the greedy algorithm is not optimal.

Continuing with the setting in Problem 2, we aim to design algorithms for it when k is larger. Clearly describe your algorithms and briefly justify their correctness.

1. Assume $k = n - 1$. Give an algorithm that runs in $O(n)$ time.

Solution: What we can do is we can go linearly through the list with a counter of a min and recaclulating the min up to the ith element. If there is a certain point where we dont recaclulate the min then there is an increasing pair and take that as a subsequence and the rest as singleton subsequences.
This results in n subsequences if sequence is not decreasing and $n - 1$ if it is.

2. Now we drop the assumption that $k = n-1$ and consider a general $k \in \{1, 2, \ldots, n\}$ instead. Give an algorithm that runs in polynomial time.

Solution: We can constrc