

01:XXX:XXX - Homework n

Pranav Tikkawar

April 9, 2025

Contents

1	In person Notes on Midterm 2	2
2	Greedy Algorithms	3
2.1	Greedy Continued	5

1 In person Notes on Midterm 2

2 Greedy Algorithms

1. Coin Change

There are n possible denominations of coins: $a_1 < a_2 < \dots < a_n$, 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 $c_1 \geq 5$ 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 until $m - c_{100} * 100 < 100$ and do the same logic as before.

This proves that the greedy algorithm works for the US coin system. \square

2. Most Valuable Subset

Given a set of n distinct positive integers, $A = \{a_1, a_2, \dots, a_n\}$, 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 median 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. \square

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 $[\ell_i, r_i]$ be the first interval that is in O but not in G . We 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 are 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 by 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 solution 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.

$$f(k, p) = \min \begin{cases} f(k-1, p) \\ f(k-1, p-1) + t_k & \text{if } f(k-1, p-1) + t_k \leq d_k \end{cases}$$

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 met then 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, \dots, 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 at most k queries. We can quickly deal with 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 up to a point r_i where O evicts i and G evicts j . After this point O will behave optimally and doesn't incur any more cache misses than G . *** \square

11. Merging Stones

Given n piles of stones with sizes a_1, a_2, \dots, a_n . We want to merge all of them into one pile. In each step we can merge any two piles into one and pay a cost 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.

