CS 170 HW 6

Due **2020-10-12, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Maximum Coverage

In the maximum coverage problem, we have m subsets of the set $\{1, 2, \dots n\}$, denoted $S_1, S_2, \dots S_m$. We are given an integer k, and we want to choose k sets whose union is as large as possible.

Give an efficient algorithm that finds k sets whose union has size at least $(1 - 1/e) \cdot OPT$, where OPT is the maximum number of elements in the union of any k sets. In other words, $OPT = \max_{i_1, i_2, \dots i_k} |\cup_{j=1}^{k} S_{i_j}|$. Just the algorithm description and justification for the lower bound on the number of elements your solution contains is needed.

(Recall the set cover algorithm from lecture, and use that $(1 - 1/n)^n \leq 1/e$ for all integers n)

Solution: Similar to set cover, we choose sets greedily, each time adding the set that includes the most elements that are not covered by any sets included in our solution so far.

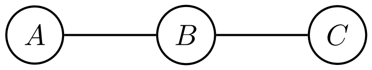
Let n_i be OPT minus the number of elements covered by the first i sets we choose. Initially, we have $n_0 = OPT$. The k sets forming OPT cover at least n_i elements that our solution does not cover, which means one of these sets covers at least n_i/k elements our solution doesn't cover. The set we add in iteration $i + 1$ can only cover more new elements than this set, i.e. we add a set covering at least n_i/k elements. So we have $n_{i+1} \leq n_i - n_i/k = n_i(1 - 1/k)$. In turn, we have $n_k \leq (1 - 1/k)^k OPT \leq \frac{1}{e} \cdot OPT$, or equivalently our solution covers at least $(1 - 1/e) * OPT$ elements.

3 Graph Game

This is a solo question.

Given an undirected, unweighted graph G, with each node v having a value $\ell(v) \geq 0$, consider the following game.

1. All nodes are initially *unmarked* and your score is 0.
2. Choose an unmarked node u. Let $M(u)$ be the *marked* neighbours of u. Add $\sum_{v \in M(u)} \ell(v)$ to your score. Then mark u.
3. Repeat the last step for as many turns as you like, or until all the nodes are marked.



For instance, suppose we had the graph:
with $\ell(A) = 3$, $\ell(B) = 2$, $\ell(C) = 3$. Then, an optimal strategy is to mark A then C then B giving you a score of $0 + 0 + 6$. We can check that no other order will give us a better score.

(a) Is the following statement true or false? **For any graph, an optimal strategy which marks all the vertices always exists.** Briefly justify your answer.

(b) Give a greedy algorithm to find the order which gives the best score. **Please give a 3-part solution for this part.**

(c) Now suppose that $\ell(v)$ can be negative. Give an example where your algorithm fails.

(d) Your friend suggests the following modified algorithm: delete all v with $\ell(v) < 0$, then run your greedy algorithm on the resulting graph. Give an example where this algorithm fails.

Solution:

(a) Marking a node can only ever increase your score, since all values are positive.

(b) *Main idea:* Sort the nodes by value largest-first (breaking ties arbitrarily), and mark them in that order.

Proof of correctness: We show this solution is optimal by an exchange argument. Suppose that we have an ordering $v_1, \ldots, v_n$ which is not sorted by decreasing value, and let i be such that $\ell(v_i) < \ell(v_{i+1})$. Note first that if we swap the order of these nodes, only the scores that we obtain in steps i and $i + 1$ could change, since in all other steps the set of marked nodes is unaffected. There are two cases.

(a) Case 1: v_i and v_{i+1} do not have an edge between them. Then the score when marking v_i is not affected by whether v_{i+1} is marked, and vice versa.

(b) Case 2: v_i and v_{i+1} do have an edge between them. Then if we swap them, v_{i+1}'s score decreases by $\ell(v_i)$, and v_i's score increases by $\ell(v_{i+1})$. Since $\ell(v_{i+1}) > \ell(v_i)$, the total score increases.

Thus, by the exchange argument, sorting the nodes will give us an optimal solution. The running time of this algorithm is $O(|V| \log |V|)$, since we just sort the vertices.

Runtime: Sorting takes $O(|V| \log |V|)$ time.

(c) We can take the example graph and set $\ell(A) = 1$, $\ell(B) = -1$, $\ell(C) = -2$. Then the greedy algorithm gives A, B, C with value 0. The optimum is A, B with value 1.

(d) Again we take the example graph and set $\ell(A) = \ell(C) = 3$, $\ell(B) = -1$. The modified algorithm gives A, C which has value 0. The optimum is A, C, B with value 6.

DP Solution Guidelines

Try to follow the following 3-part template when writing your solutions.

- Define a function $f(\cdot)$ in words, including how many parameters are and what they mean, and tell us what inputs you feed into f to get the answer to your problem.
- Write the "base cases" along with a recurrence relation for f.
- Prove that the recurrence correctly solves the problem.
- Analyze the runtime of your final DP algorithm.

4 Maximum weight independent set

This is a solo question.

Let $G = (V, E)$ be an undirected graph, with nonnegative weights $w(v)$ for each vertex $v \in V$. A subset of nodes $S \subset V$ is an *independent set* of G if no edge has both its endpoints in S.

Assuming that G is a tree, find an efficient dynamic programming algorithm for finding the maximum weight independent set in G, i.e. an independent set S of G such that $\sum_{v \in S} w(v)$ is maximized.

Please provide a 3-part solution.

Solution:

Main Idea:

Each vertex in the tree defines a subtree, and we define a subproblem for each subtree as follows.

$$I(v) = \text{total weight of maximum weight independent set of subtree rooted at } v.$$

Our goal is to compute $I(r)$, where r is the root of the tree. Our reccurence relation is as follows.

$$I(v) = \max\left\{ w(v) + \sum_{\text{grandchildren } u \text{ of } v} I(u), \sum_{\text{children } u \text{ of } v} I(u) \right\}.$$

The children of V can only be in the maximum weight independent set of the subtree rooted at v, if the set does not include v. If not, then the maximum weight independent set consists of v and the union of the maximum weight independent sets of the subtrees of the grandchildren of v.

We can also compute the maximum weight independent set in linear time and space as follows. For each subproblem $I(v)$, we store whether v is in the maximum weight independent set of the subtree rooted at v. The solution can then be constructed by traversing the tree from its root, and at each vertex v, we traverse its children if v is not included in the set, otherwise we traverse its grandchildren.

Runtime Analysis:

The running time is $O(n)$, where n is the number of vertices. Each vertex v is only processed 3 times: when the algorithm is processing v, when it is processing v's parent and when it is processing v's grandparent.

Proof of Correctness:

Base Case: The base cases are the leaves of the tree. For each leaf v, the maximum weight independent set consists of just v, so $I(v) = w(v)$.

Inductive Step: To compute $I(v)$, if v is in the maximum weight independent set, then the children of v cannot be in the maximum weight independent set. Hence we take the union of v with the maximum weight independent sets of the subtrees rooted at the grandchildren of v, so $I(v)$ is $w(v)$ plus $I(u)$ for the grandchildren u of v.

If v is not in the maximum weight independent set, then the children of v can be in the maximum weight independent set. The maximum weight independent set is the union of the maximum weight independent sets of the subtrees rooted at the children of v.

5 Egg Drop

This is a solo question.

You are given k identical eggs and an n story building. You need to figure out the highest floor $\ell \in \{0, 1, 2, \ldots n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor ℓ or lower, and always breaks if dropped from floor $\ell + 1$ or higher. ($\ell = 0$ means the egg always breaks). Once an egg breaks, you cannot use it any more.

Let $f(n, k)$ be the minimum number of egg drops that are needed to find ℓ (regardless of the value of ℓ).

(a) Find $f(1, k)$, $f(0, k)$, $f(n, 1)$, and $f(n, 0)$.

(b) Find a recurrence relation for $f(n, k)$. *Hint: Whenever you drop an egg, call whichever of the egg breaking/not breaking leads to more drops the "worst-case event". Since we need to find ℓ regardless of its value, you should assume the worst-case event always happens.*

Solution:

(a) We have that:

- $f(1, k) = 1$, since we can drop the egg from the single floor to determine if it breaks on that floor or not.
- $f(0, k) = 0$, since there is only one possible value for ℓ.
- $f(n, 1) = n$,since we only have one egg, so the only strategy is to drop it from every floor, starting from floor 1 and going up, until it breaks.
- $f(n, 0) = \infty$ for $n > 0$, since the problem is unsolvable if we have no eggs to drop.

(b) The recurrence relation is

$$f(n, k) = 1 + \min_{x \in \{1 \ldots n\}} \max\{f(x - 1, k - 1), f(n - x, k)\}.$$

Consider dropping an egg floor x when there are n floors and k eggs left. If the egg breaks, we only need to consider floors 1 to $x-1$, and we have $k-1$ eggs left since an egg broke, in which case we need $f(x-1, k-1)$ more drops. If the egg doesn't break, we only need to consider floors $x+1$ to n, and there are k eggs left, so we need $f(n-x, k)$ more drops. So in the worst case, we need $\max\{f(x-1,k-1), f(n-x,k)\}$ drops if we drop from floor x. Then, the optimal strategy will choose the best of the n floors, so we need $\min_{x\in\{1\dots n\}} \max\{f(x-1,k-1), f(n-x,k)\}$ more drops.

6 Motel Choosing

You are traveling along a long road, and you start at location $r_0 = 0$. Along this road, there are n motels at location $\{r_i\}_{i=1}^n$ with $0 < r_1 < r_2 < \cdots < r_n$. The only places you may stop are these motels, but you can choose which to stop at. You must stop at the final motel (at distance r_n), which is your destination.

Ideally, you want to travel exactly T miles a day and stop at a motel at the end of the day, but this may not be possible (depending on the spacing of the motels). Instead, you receive a *penalty* of $(T-x)^8$ each day, if you travel x miles during the day. The goal is to plan your stops to minimize the total penalty (over all travel days).

Describe and analyze an algorithm that outputs the minimum penalty, given the locations $\{r_i\}$ of the motels and the value of T.

Solution:

Main idea: Let $P[i]$ be the optimal (*i.e.* minimum) penalty for getting to hotel i; it holds that

$$P[i] = \min_{j<i} \left\{ P[j] + (T - (r_i - r_j))^8 \right\}.$$

The base case is given by $P[0] = 0$.

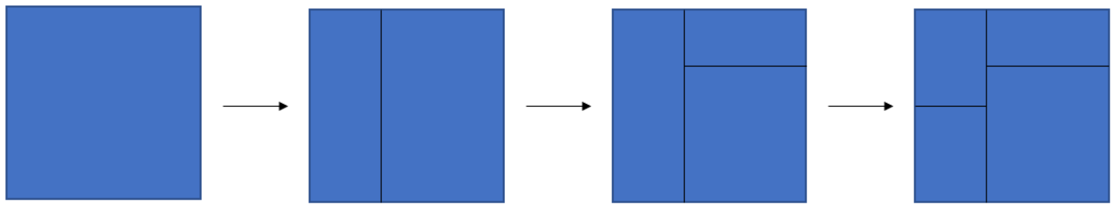
Proof of correctness: To prove correctness, note that the base case is trivial. Assume as an inductive hypothesis that $P[j]$ are computed correctly for all $j < n$. Then to finally get to the destination at location r_n, there has to be a penultimate stop is i along our route. Then each choice of the stop i lead to a penalty of $P[i] + \left(T - (r_i - r_n)^8\right)$. We assumed that the first term is the minimum penalty to get to i, and the second term is the necessary penalty from i to n. Therefore, minimizing this quantity over all i leads to the minimum penalty of the entire trip.

Runtime analysis: There are $O(n)$ subproblems, each takes $O(n)$ time to compute, so the runtime is $O(n^2)$.

7 Geometric Knapsack

Suppose we have a piece of cloth with side lengths X, Y, where X, Y are positive integers, and a set of n *products* we can make out of the cloth. Each product is a rectangle of dimensions $a_i \times b_i$ and of value c_i, where all these numbers are positive integers.

We want to cut our large piece of cloth into multiple smaller rectangles to sell as products. Any rectangle not matching the dimensions of one of these products gets us no value. To cut the cloth, we are using a machine that takes one piece of cloth and cuts it into two, where the dividing line between the two pieces must be a vertical or horizontal line going all the way through the cloth. For example, the following is a valid series of cuts:



Give an efficient algorithm that determines the maximum value of the products that can be made out of the single $X \times Y$ piece of cloth. You may produce a product multiple times, or none if you wish. Give a three-part solution.

Solution:

Main idea:

For $1 \leq i \leq X$ and $1 \leq j \leq Y$, let $C(i,j)$ be the best return that can be obtained from a cloth of shape $i \times j$. Define also a function `rect` as follows:

$$\texttt{rect}(i,j) = \begin{cases} \max_k c_k & \text{over all products } k \text{ with } a_k = i \text{ and } b_k = j \\ 0 & \text{if no such product exists} \end{cases}$$

Then the recurrence relation is

$$C(i,j) = \max\left\{\max_{1\leq k<i}\{C(k,j) + C(i-k,j)\}, \max_{1\leq h<j}\{C(i,h) + C(i,j-h)\}, \texttt{rect}(i,j)\right\}$$

Effectively, we look at every place we can cut the cloth and compute the resulting value, and also look at the value if we sell the cloth as one piece. We then take the max over these options. We use the convention here that if $i = 1$, $\max_{1\leq k<1}\{C(k,j) + C(i-k,j)\} = 0$ (and similarly for j).

The base case is $C(1,1) = \texttt{rect}(1,1)$. The final solution is then the value of $C(X,Y)$.

To implement the algorithm, we simply initialize a $X \times Y$ 2D array. Then we fill the DP table row by row from the top, starting from $C(1,1)$ and applying the recurrence relation for each entry.

Correctness:

For proving correctness, notice that $C(1,1)$ of course is the max value we can retrieve from a 1-by-1 piece of cloth, since all products have positive integer side lengths. Inductively, $C(i,j)$ is solved correctly, as a rectangle $i \times j$ can only be cut in the $(i-1) + (j-1)$ ways considered by the recursion or be occupied completely by a product, which is accounted for by the $\texttt{rect}(i,j)$ term.

Runtime analysis:

The running time is $O(XY(X+Y+n))$ as there are XY subproblems and each takes $O(X+Y+n)$ to evaluate. This can be improved to $O(XY(X+Y)+n)$ by precomputing `rect`, but this is not necessary for full credit.

8 (Extra Credit) Job Scheduling Redux

We have k supercomputers, and n jobs to run on these supercomputers. The ith job has processing time t_i. We want to come up with an assignment of jobs to supercomputers, so that the *maximum* total processing time of jobs assigned to any supercomputer is minimized. That is, if J_ℓ is the set of jobs assigned to the ℓ-th super computer, then we want to minimize $T_{max} := \max_\ell \sum_{i \in J_\ell} t_i$.

Let T^*_{max} be the minimum value of T_{max} across all assignments. Give an efficient algorithm that finds an assignment of jobs to machines such that T_{max} for this assignment is at most c times T^*_{max} for some constant $c \geq 1$. Give a three-part solution, and specify in the proof of correctness what c is for your solution.

Solutions where c is larger than the value of c obtained by the staff's solution will not get credit.

Solution:

Main idea: We iterate over the jobs in arbitrary order. For each job, we assign it to the supercomputer for which $\sum_{i \in J_\ell} t_i$ is currently the smallest.

Correctness: Let ℓ be the supercomputer with the most total processing time in our solution, and let i^* be the last job added to this supercomputer. For $A := \sum_{i \in J_\ell \setminus i^*} t_i$, we can express the total processing time on this machine as $A + t_{i^*}$.

When we added job i^* to this supercomputer, this was the supercomputer with the smallest total processing time and its total processing time was A, so the total processing time of all jobs is at least $k * A$. This means that in any assignment of jobs to machines, at least one machine has total processing time at least A, i.e. $T^*_{max} \geq A$.

On the other hand, any assignment of jobs to supercomputers must assign i^* to some supercomputer, so clearly $T^*_{max} \geq t_{i^*}$.

So for this assignment, $T_{max} = A + t_{i^*} \leq 2 * T^*_{max}$, i.e. $c = 2$.

Runtime analysis: This can be done in $O(n \log k)$ time: We can use e.g. a priority queue which holds the supercomputers sorted by $\sum_{i \in J_\ell} t_i$. This lets us determine which machine to add each job to in $O(1)$ time, and after assigning each job to a machine we can update the priority queue in $O(\log k)$ time.

Note: If we first sort the jobs in descending order of t_i, and then apply the above algorithm, c can be improved to $4/3$ (at the cost of the runtime increasing to $O(n \log n)$).

To see why: Without loss of generality, we can assume the last job on the machine with the largest processing time is also the last job added to a machine (and thus the shortest job), otherwise we could delete all jobs shorter than this one and our solution's cost would stay the same but T^*_{max} might decrease. If the last job added satisfies $t_{i^*} \leq T^*_{max}/3$, then the same argument as above gets $c = 4/3$ instead of $c = 2$. Otherwise every job has $t_i > T^*_{max}/3$, which implies there are at most $2k$ jobs (otherwise some supercomputer gets at least 3 jobs in the optimal assignment, and thus needs time strictly greater than T^*_{max}, a contradiction). The algorithm will put the $2k - n$ largest jobs on their own machines, and then of the remaining jobs, the longest and shortest will be on one machine, the second longest and shortest will be on another machine, etc. Using an exchange argument, one can show this is actually optimal.