CSC 373 - Algorithm Design, Analysis, and Complexity **Summer 2016**

Assignment 1- Solutions: Due Sunday June 12, 10PM

Please follow the instructions provided on the course website to submit your assignment. You may submit the assignments in pairs. Also, if you use **any** sources (textbooks, online notes, friends) please cite them for your own safety.
You can use those data-structures and algorithms discussed in CSC263 (e.g. merge-sort, heaps, etc.) and in the lectures by stating their name. You do not need to provide any explanation or pseudo-code for their implementation. You can also use their running time without proving them: for example, if you are using the merge-sort in your algorithm you can simply state that merge-sort's running time is $\mathcal{O}(n \log n)$.

Every time you are asked to design an efficient algorithm, you should provide both a short high level explanation of how your algorithm works in plain English, and the pseudo-code of your algorithm similar to what we've seen in class. State the running time of your algorithm with a brief argument supporting your claim. You must prove that your algorithm finds an optimal solution!

1 Nap Time

Recall the music festival scheduling problem we saw in class. Before heading to FestTown where it's held, you wanted to plan your best nap yet during the festival[1]. You know that the quietest and least busy time, at the camp where you're staying, is when a maximum number of shows are happening simultaneously. They don't necessarily overlap from start to end, but there is a non-empty time interval in which they overlap. You're given the schedule ahead of time.

1/ Give an efficient algorithm that computes the maximum number of overlapping performances.

2/Prove your algorithm is correct.

Solution:

"Flatten" your intervals and sort them in non-decreasing order in a table ($2n$ entries) where you store the values and whether they are start or finish times:

$$s_1 < s_2 < f_1 < s_3 < \ldots < f_k$$

Keep two variables: a counter $c = 0$ and a max overlap $o = 0$. Scan the sorted list, if entry i is a start time, increase c by 1, if it is a finish time, decrease c by 1. Note that at the end $c = 0$. The max overlap variable o is updated every time c is incremented: If $o < c : o = c$. Return c.

The proof of correctness is straightforward. Suppose your algorithm is not optimal, that is o was not properly incremented. This means there is one or more overlaps we didn't take into account, i.e. a start time that was not considered. Since we process all the intervals (and thus all start times), this cannot happen.

Note 1: In interval scheduling, we study different types of overlap, and different intervals (whether they are opened or closed or both). The easiest, most well studied case is the one where we don't care about end points, i.e. we assume all end points are open. So if for some intervals I_i, I_j we have $s_i = f_j$, then the two

[1] who goes to festivals to nap...?

intervals are not considered overlapping. Your algorithms might return different values on the same input depending on how you process this scenario. In the algorithm above, one way to maintain this non-overlap is to order $f_j \leq s_i$ so we decrease the counter first before incrementing it again.

Note 2: Interval graphs are the graphs constructed from a set of intervals, where every interval is represented as a vertex, and two vertices are adjacent if and only if their corresponding intervals overlap. Interval graphs are known to be perfect. Perfect graphs are a family of graphs where for every induced subgraph, the clique number (i.e. the size of the largest clique) equals the chromatic number (i.e. the minimum number of colours needed to colour the vertices such that no two adjacent vertices receive the same colour). The above problem asks for the clique number of the graph, a different way to solve it is to compute the colouring number instead. Again, this would only work since the graph is perfect. it doesn't work for arbitrary graphs.

2 Hmm Cookies!

You got a baking gig on TV, and as you know, the most important thing about TV cooking is looks (taste comes second)! You need to make cookies live on-air. You lay the dough flat and cut out the pieces for each cookie, which you will then place on a baking tray with maximum length L (suppose all the cookies fit in the tray). But, as we know, we need to leave some space in between, since cookies grow when they bake. Suppose you leave a fixed length space between side-by-side cookies. One way to present the trays nicely is to fill the rows of the tray as much as possible, and move to the next row when the current cookie doesn't fit. One way to penalize "ugly" trays is to assign a penalty to each row that is not neat enough. For the sake of simplicity, suppose all the cookies have equal height.

Formally, you have trays of fixed length L, n cookies of width $d_1, d_2, , \ldots, d_n$ where $d_i \leq L$, for $i \in [n]$. A fixed distance $s = 1$ that represents the space you leave between cookies. If in row k, you place cookies i to j, then the unoccupied space in this row is computed as follows:

$$L - \sum_{h=i}^{j} d_h - (j - i)$$

Suppose your penalty function is computed as follows:

$$row\_penalty(i, j) = \begin{cases} \infty & \text{if cookies } i \text{ through } j \text{ do not fit in a row} \\ 0 & \text{if } j = n, \text{ last row} \\ (L - \sum_{h=i}^{j} d_h - (j - i))^3 & \text{otherwise} \end{cases}$$

Notice that you don't pay any penalty for the last row of cookies.

The total penalty for placing the cookies is the sum of the penalties of all rows. An optimal solution is a placement of the cookies into rows such that your total penalty is minimized. Notice that you do not shuffle the order of the cookies to achieve the best looking trays.

1/ Give a recurrence relation to compute the optimal penalty.

2/ Construct an efficient algorithm which solves the above problem. Give a high-level description of how your algorithm works, and prove that it is optimal.

Solution:

Note that $row\_penalty(i, j)$ is defined to be ∞ if the cookies i through j do not fit on a row to guarantee that no rows in the optimal solution overflow. (This relies on the assumption that the width of each cookie is not more than L.) Second, notice that $row\_penalty(i, j)$ is defined to be 0 when $j = n$, where n is the total number of cookies; only the actual last row has zero penalty, not the recursive last rows of subproblems, which, since they are not the last row overall, have the same penalty cost as any other row.

Now, consider an optimal solution of placing cookies 1 through n. Let i be the index of the first cookie placed on the last row of this solution. Then the placement of cookies $1, \ldots, i-1$ must be optimal. Otherwise, we could modify the ordering to obtain an optimal placement of these cookies and improve the total penalty cost of the solution, a contradiction. Also note that the same reordering argument can be applied if we take i to be the index of the first cookie placed on the k^{th} row, where $2 \leq k \leq n$. Therefore this problem displays optimal substructure.

Let $c(j)$ be the optimal penalty cost of placing cookies 1 through j. From the above argument, it is clear that given the optimal i (i.e. the index of the cookie placed on the last row of an optimal solution), we have:

$$c(j) = c(i-1) + row\_penalty(i, j)$$

However, since we do no know which i is optimal, we need to consider every possible i, so our recursive definition of the optimal penalty cost is:

$$c(j) = \min_{1 \leq i \leq j} \{c(i-1) + row\_penalty(i, j)\}.$$

To accommodate this recursive definition, we define $c(0) = 0$.

The algorithm would then compute the values of the array c from index 1 to n, bottom up, which can be done efficiently since each $c(k)$ for $1 \leq k < j$ will be available by the time $c(j)$ is computed. To keep track of the actual optimal arrangement of the cookies, we record an array p, where $p(k)$ is the i (in the recursive definition of c) which led to the optimal $c(k)$. Then, after the arrays for c and p are computed, the optimal cost is $c(n)$ and the optimal solution can be found by placing cookies $p(n)$ through n on the last row, cookies $p(p(n) - 1)$ through $p(n) - 1$ on the next to last row, and so on.

A good optimization can be obtained by noticing that computing $row\_penalty(i, j)$ takes in general $\mathcal{O}(j-i+1)$ time because of the summation in the formula. However, it is possible to do this computation in $\mathcal{O}(1)$ time with some additional pre-processing. We create an auxiliary array $A[0 \ldots n]$, where $A[i]$ is a cumulative sum of widths of cookies 1 through i:

$$\begin{aligned} A[0] &= 0 \\ A[i] &= A[i-1] + d_i \\ &\equiv \sum_{k=1}^{i} d_k \end{aligned}$$

Filling in this array takes $\mathcal{O}(n)$ time using recursion. In order to then compute $row\_penalty(i, j)$ in $\mathcal{O}(1)$ time, we modify the formula as follows:

$$row\_penalty(i, j) = \begin{cases} \infty & \text{if cookies } i \text{ through } j \text{ do not fit in a row} \\ 0 & \text{if } j = n, \text{ last row} \\ (L - (j-i) - (A[j] - A[i-1]))^3 & \text{otherwise} \end{cases}$$

3/ Consider the case where the penalty function is just the free space in each tray, that is:

$$row\_penalty(i,j) = \begin{cases} \infty & \text{if cookies } i \text{ through } j \text{ do not fit in a row} \\ 0 & \text{if } j = n, \text{ last row} \\ L - \sum_{h=i}^{j} d_h - (j-i) & \text{otherwise} \end{cases}$$

Devise an algorithm to compute the optimal solution, and show that it is correct.

Solution:

We use a straightforward greedy algorithm, which puts as many cookies as possible on each row before going to the next row. The algorithm clearly runs in linear time.

We need to show that any optimal solution has the same penalty cost as the solution obtained by this greedy algorithm. Consider some optimal solution. If this solution is the same as the greedy solution, then we are done. If it is different, then there is some row i which has enough space left over for the first cookie of the next row. In this case, we move the first cookie of row $i+1$ to the end of row i. This does not change the total penalty cost, since if the width of the cookie moved is d, then the reduction to the cost of row i will be $d+s = d+1$, for the cookie and the space before it, and the increase of the cost of row $i+1$ will also be $d+1$, for the cookie and the space after it. (If the moved cookie was the only cookie on line $i+1$, then by moving it to the previous row the total cost is reduced, a contradiction to the assumption that we have an optimal solution.) As long as there are rows with enough extra space, we can keep moving the first cookies of the next rows back without changing the total cost. When there are no longer any such rows, we will have changed our optimal solution into the greedy solution without affecting the total penalty cost. Therefore, the greedy solution is an optimal solution.

3 Work Work Work Work Work

You work at Amazon. Every day at work, a set of orders are assigned to you. Each order requires the same steps before it is shipped: pack, label and mail. Each order o_i has a processing time p_i. The orders are distributed every morning at work and you do not leave until your orders have been shipped for the day. **Employee Of The Month** is selected based on how long it takes an employee to ship an order[2]. The faster the better of course. Your goal this month is to win this title[3].

1/Describe the function you need to optimize in order to win. Formalize the input and output of the problem.

Solution:

Input: A set of orders with processing times $p_1, p_2, \ldots, p_n$.
Output: An ordering of the jobs that minimizes the average completion time.

Let c_i be the completion time of order o_i with processing time p_i for all $i \in [n]$. Our objective function is:

$$\min \frac{1}{n} \sum_{i=1}^{n} c_i$$

[2]On average of course
[3]Apologies for such a cr*ppy job...

2/ In this first scenario, you are not allowed to switch back and forth between orders. Once you start working on an order, you cannot move to the next one until the current one is done. Give an algorithm that will help you win your title. What is the complexity of your algorithm? Prove that your algorithm is optimal.

Solution:

Run orders in shortest processing time order. This can be done by sorting the elements using heap sort or merge sort and then process them in the order of increasing processing times. This algorithm takes $\mathcal{O}(n \log n)$.

This algorithm uses a greedy strategy. It is shown to be optimal as follows:

$$c_{avg} = \frac{c_1 + c_2 + \ldots + c_n}{n}$$

This cost can also be expressed as:

$$c_{avg} = \frac{1}{n}[p_1 + (p_1 + p_2) + (p_1 + p_2 + p_3) + \ldots + (p_1 + p_2 + \ldots + p_n)]$$

p_1 is added in the most times, then p_2, etc. As a result, p_1 should have the shortest processing time, then p_2, etc. otherwise, you could "cut and paste" in a shorter processing time and produce a faster algorithm. As a result, our algorithm is correct.

3/ Suppose now the orders are not handed to you all at once, but rather come in one at a time. In addition to the processing time p_i, each order o_i has a release time r_i. In this scenario, you can switch back and forth between orders.

Example: You get an order $o_i(r_i = 1, p_i = 8)$. You can for instance work on this order from time 1 to 3, get back to it from time 6 to 11 and then finish it from time 14 to 15, for a total of 8 units of time.

Give an algorithm that allows you to win in this case. Does switching between tasks improve the complexity? Prove that your algorithm is optimal.

Solution:

This problem also exhibits the greedy property which can be exploited by processing orders in shortest remaining processing time order. Use a priority queue which prioritizes based on the order with the shortest time remaining. Each time a new order comes up, insert it into the queue and if it would take less time to do that task then the one you are on, do the shorter task. Each time you finish an order, the next order you should do is the one with the least remaining time until completion. The priority queue can be maintained in $\mathcal{O}(n \log n)$ time.

This algorithm minimizes the average completion time and the proof is similar to part (2). If we do not process using the greedy algorithm based on remaining processing time, then we will be able to swap two time slots which would then improve the sum of the completion times and thus result in a contradiction. For example, assume you have two orders at time t, where order i has x processing time remaining and j has y processing time remaining where $x > y$. Assume for the purposes of contradiction that the optimal answer has order i running before order j. If i is done before j then $c_i = t + x$ and $c_j = t + y + x$. The average completion time is $\frac{2t+2x+y}{2}$. However if j were done before i,then $c_i = t + y + x$ and $c_j = t + y$. The average completion time is now $\frac{2t+2y+x}{2}$ which is less the average completion time for the "optimal" solution since $x > y$. As a result, the order with the lowest time remaining should be done first.

4 Some Graph Theory .. Because you haven't seen enough.[4]

Let $G(V, E, w)$ be a directed edge-weighted graph: $w : E \to \mathbb{R}$. For a given cycle $\mathcal{C} = \{e_1, e_2, \ldots, e_k\}$ in G, let $\mu(\mathcal{C})$ denote the value:

$$\mu(\mathcal{C}) = \frac{1}{k} \sum_{i=1}^{k} w(e_i)$$

For all cycles $\{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_h\}$ in G, let $\tilde{\mu}$ be the minimum over $\mu(\mathcal{C}_i), i \in [h]$.

Let $s \in V$ be a source vertex in G, and suppose that every vertex $v \in V$ is reachable from s. Let w_{sv} (resp. w^k_{sv}) denote the weight of a shortest sv path, (resp. sv k-path[5]) in G. If no such k-path exists between s and v, we set w^k_{sv} to ∞.

1/ Prove that if $\tilde{\mu} = 0$, the following holds:

1. $w(\mathcal{C}) = \sum_{e \in \mathcal{C}} w(e) \geq 0$ for all cycles $\mathcal{C}$ in G.
2. $w_{sv} = \min_{0 \leq k \leq n-1} w^k_{sv}$ for all $v \in V$.
3. $\max_{0 \leq k \leq n-1} \frac{w^n_{sv} - w^k_{sv}}{n-k} \geq 0$.

Solution:

If there were a negative-weight cycle, then $\tilde{\mu} < 0$ because the minimum would have to be negative, therefore there are no negative weight cycles. Given that there are no negative weight cycles, then the shortest path will not take any cycles and can only be at most $n - 1$ edges long.

For part 3, we know $n - k$ is strictly positive because $k \leq n - 1$. Then $w^n_{sv} - w^k_{sv} \geq 0$ because the shortest path when no negative weight cycles exist is going to cost more with n nodes than the shortest path with fewer nodes that is the actual shortest path.

2/ Let $\mathcal{C}^*$ be a cycle of total weight 0, and let x, y be two vertices on $\mathcal{C}^*$ where $w(P_{xy}) = d$, the weight of the xy path in $\mathcal{C}^*$. Show that $w_{sy} = w_{sx} + d$.

Solution:

We know $w_{sx} \leq w_{sy} + d$ because the shortest path from s to x might use y. Alternatively we also know that $w_{sy} \leq w_{sx} - d$ because the shortest path to y from s might go through x and around the zero weight cycle for a cost of $-d$. Therefore with these two inequalities we know $w_{sy} = w_{sx} + d$.

3/ Suppose $\tilde{\mu} = 0$ and let $\mathcal{C}^*$ be the cycle to achieve it (i.e. $\tilde{\mu} = \mu(\mathcal{C}^*)$), show that:

1. For some vertex v on $\mathcal{C}^*$, $\max_{0 \leq k \leq n-1} \frac{w^n_{sv} - w^k_{sv}}{n-k} = 0$.
2. $\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{w^n_{sv} - w^k_{sv}}{n-k} = 0$

[4] Seriously though..
[5] A path using exactly k edges

Solution:

3.1/ Get to the cycle along some shortest path and then extend the path along the cycle to make a shortest path of length n. If v is the vertex we end up at, then $w_{sv}^n = w_{sv}$. Then since we took the shortest possible path to the cycle, there cannot exist any shorter path to the node with fewer steps, only equal path lengths.

3.2/ We know that there exists some vertex with a maximum difference of 0, from part 3.1, and all differences are greater than 0, so the minimum must be 0.

4/ How does $\tilde{\mu}$ change if we increase the weight of all edges by a constant c? Show that $\tilde{\mu} = \min_{v \in V} \max_{0 \le k \le n-1} \frac{w_{sv}^n - w_{sv}^k}{n-k}$.

Solution:

Adding c to each edge increases $\tilde{\mu}$ by c. It also increases w_{sv}^n by nc and decreases $-w_{sv}^k$ by kc. Manipulate, and both sides increase by c and the equation is maintained. Thus by picking $c = -\tilde{\mu}$, we get the second part.

5/ Devise an algorithm to compute $\tilde{\mu}$. Show that your algorithm runs in $\mathcal{O}(mn)$ time.

Solution:

Compute w_{sv}^k for $k \in \{0, 1, \ldots, n\}$ in $\mathcal{O}(mn)$ time by evaluating the recurrence $w_{sv}^{k+1} = \min_u w_{su}^k + w(u, v)$. In $\mathcal{O}(n^2)$ time, determine the minimum of the maximum of the fraction in part 4.4. This algorithm is known as Karp's minimum mean-weight cycle algorithm.