EE360C: Algorithms
University of Texas at Austin
Dr. Evdokia Nikolova & Dr. David Soloveichik

Exam #3 Practice Problems
Due: Preparation for Exam 3

Exam #3 Practice Problems

Instructions. No calculators, laptops, or other devices are allowed. This exam is **closed book**. **No additional materials are allowed.** Write your answers on the test pages. If you need more scratch paper, use the back of the test pages, but indicate clearly where your answers are. Write down your process for solving questions and intermediate answers that **may** earn you partial credit. **You may invoke theorems used in class without proof, but you cannot invoke in-class exercises, homework problems or quizzes (in other words, you need to write such solutions from scratch).**

If you are unsure of the meaning of a specific test question, write down your assumptions and proceed to answer the question on that basis.
You have **180 minutes** to complete the exam. The maximum possible score is 100.

Master Theorem:
Let $T(n)$ be defined by the recurrence $T(n) = aT(n/b)+f(n)$ (where $a \geq 1$ and $b > 1$ are constants). Then $T(n)$ can be bounded asymptotically as follows:

1. If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Problem 1: Divide and Conquer

You just started working at a local coffee shop, and they've been doing a terrible job of managing their staff, specifically with respect to having the right number of staff on hand at the busiest times. After just a few days on the job, you notice that the crowds follow a very specific pattern: the number of customers strictly increases up to a point and then strictly decreases. Your registers give you a print out of the average number of customers in every ten minute interval from open to close.

Given this array of the customer counts over a given day, write an efficient algorithm to find the specific 10 minute interval that is the "peak" traffic for your coffee shop. Write and solve the recurrence for your algorithm. Give as efficient algorithm as possible.

Solution

So this is basically a fancy binary search. Grab the middle value in the array. Look at the middle value, the one before, and the one after. If both the one before and the one after are smaller, return the middle one since you've found your peak. If the one before is smaller and the one after is larger, recurse on the second half of the array. If the one before is larger and the one after is smaller, recurse on the first half of the array.

The recurrence is $T(n) = T(n/2) + O(1)$. To solve this, we can use the master method. $\log_b a = \log_2 1 = 0$, which is the same as l ($l = 0$). So the running time is $O(n^l \log_2 n) = O(\log_2 n)$.

Problem 2: Bouncer Schedules

Each SXSW venue hires bouncers to check IDs at the door. A particular venue that wants to ensure that it has *two bouncers* scheduled to work at all times, from open (t_o) to close (t_c).

- All of the bouncers in Austin make their schedules available in some central repository.
- Bouncers make themselves available in shifts, and each shift has a start time and end time.
- A venue can hire a bouncer for a complete shift or for any fraction of an available shift.
- Assume that there are always at least two bouncers available at any given time.

From a venue's perspective, an optimal bouncer schedule minimizes the number of distinct shifts that have to be scheduled (e.g., hiring bouncer A to cover door 1 from open to close is preferred to scheduling two people to cover door 1).

Given the open and close time of our specific venue, describe an efficient algorithm for generating that venue's optimal bouncer schedule, given the set of available bouncer shifts. Recall that our venue needs to have two bouncers scheduled whenever the venue is open. Your algorithm should return the schedule of shifts for the venue, where each shift in the schedule includes the bouncer's name, the start time of the shift, and the end time of the shift.

State and briefly justify the runtime of your algorithm.

Solution

This is a modification of the tiling path problem. First, we make two arrays X_L and X_R, which contain the start times and end times of every available shift, both sorted so that the shifts with the earliest start times are first. Throw away any shifts that end before t_o and any shifts that start after t_c. To choose the first shift to schedule, take all of the shifts whose start times are at or before t_o, and choose the one of these shifts with the latest end time. Let's call this s_1, and its end time is s_1.end. Output the selected shift, including the bouncer's name, the start time (which should be t_o), and the end time (s_1.end). Remove every other shift that is completely overlapped by the selected shift (i.e., any other shift whose end time is before s_1.end). Then repeat the entire process, using s_1.end as the needed start time. When you output for the i^{th} shift, you will output the bouncer's name, the start time as s_{i-1}.end, and the end time as s_i.end. The only exception is the last shift. We stop the entire process when $t_c \leq s_i$.end; the end time output for this last shift should be t_c.

The other catch is that we need **two** bouncers. So after we do this once, we take the entire set of shifts again, remove all of the shifts that we selected to use, and repeat the process.

Note: at any point in the algorithm, if we do not find a tiling path, then we must output that it is impossible to schedule two bouncers for the whole time period.

The running time is $O(n \log n)$, where n is the number of shifts available. This is because the algorithm first sorts the shifts by start/end time (this takes $O(n \log n)$ time. After that, the algorithm iterates over all of the shifts, doing constant time work for each shift. It does this twice, for a total of $O(2n)$ work. This results in an overall running time of $O(n \log n)$.

Problem 3: Festival Traffic Planning

During SXSW, road closures and delays are caused by festival traffic and events. This makes it very difficult to figure out the best route to drive from one point to another.

To aid drivers, the city has implemented a system that will give the (correct) travel time on a given road segment, given a specific starting time within the festival. That is, given a road segment (e.g., the 600 block of N. Congress Avenue) and a time (e.g., 7pm on the Tuesday of the festival), the system returns a travel time for that road segment (e.g., 5 minutes).

Given a graph that represents the city's street network, this system can be represented as a function $f_e(t)$ that, for a given edge e, returns the travel time for road segment e, assuming the driver enters the segment at time t. Travel times in the city obey two rules: (1) travel times are always positive and (2) for $t_1 < t_2$, $t_1 + f_e(t_1) \leq t_2 + f_e(t_2)$ (that is, you cannot arrive earlier by waiting to start later).

Give an efficient algorithm to determine the earliest possible arrival time at a given destination d, given a starting location s and a departure time t_d. State and briefly justify the running time of your algorithm.

Solution

To solve this problem, we use a relatively straightforward modification of Dijkstra's algorithm, where we change the process to call our traffic system at runtime. As in Dijkstra's algorithm in class, S is the set of nodes to which the shortest path has already been determined. Initially, $S = \{s\}$, $s.d = 0$, and $v.d = \infty$, $\forall v \in V$ other than s. Also, as in our original statement of Dijkstra's we keep π for each $v \in V$ other than s; this is the previous node along the shortest path from s to v. Then all we basically do is update the RELAX procedure in use by Dijkstra's algorithm. To make this easy to represent, we assume that we can rewrite all of the functions f_e as a function f that takes two inputs: an edge (u, v), and t. The new relax procedure is simply:

```
RELAX(u, v, f)
1  if v.d > f((u, v), u.d)
2      then v.d ← u.d + f((u, v), u.d)
3           v.π ← u
```

Note: The rule (2) mentioned above is necessary for correct working of RELAX procedure. Everything else about the algorithm stays the same.
The running time remains $O(E \log V)$, assuming that the runtime of a single call to f is constant.

Problem 4: Mr. Smith

Mr. Smith loves eating at your restaurant. Every Friday he orders N dishes labeled from 1 to N. Your job as a great chef is to group the dishes in different plates. Since you have been a chef for many years you know that if the ordering of the dishes changes then the taste combinations you have carefully prepared will be spoiled. Thus you have decided that every plate will have sequential dishes of the form $(i, i+1, ..., i+k)$.

Every dish has a value $D = x_i$ which denotes how delicious it is. Initially, you thought that the value D of a plate with $(i, i+1, ..., i+k)$ dishes is calculated summing the respective x_is i.e. $D = x = x_i + x_{i+1} + ... + x_{i+k}$. But your cooking experience has led you to reconsider the formula producing the value D for a plate. Thus, you finally reached the conclusion that $D = ax^2 + bx + c$, with a, b, c known parameters ($a < 0$) and x as the sum of the respective x_is.

In order to keep Mr. Smith happy your goal is to group the dishes into different plates in order to maximize the sum of the D values over all the plates.

For example suppose you have 4 dishes, $x_1 = 2, x_2 = 2, x_3 = 3, x_4 = 4$. Also, let the parameters a, b, c take the values $-1, 10, -20$ respectively. For this instance the best way to group the dishes is into three plates:

The first plate should have dishes 1 and 2, the second plate should have dish 3 and the third plate should have dish 4. Thus, the values of D for the three plates are $4, 1, 4$ respectively which is the best possible for this instance of the problem.

Give an algorithm for the problem above with time complexity $O(n^2)$.

Solution

Let $f(x) = ax^2 + bx + c$.
Let $B(i)$ denote the optimal solution for the subproblem including only the i first dishes. Then:

$$B(1) = f(x_1)$$

$$B(k) = \max_{i \leq k} \{f(\sum_{j=i}^{k} x_j) + B(i-1)\}$$

Here we are checking all the possible starting points for the final plate and we are keeping the best of these options. Notice that having such a starting point fixed we can easily compute the optimal solution for our problem making use of the optimal solutions we have already computed for smaller subproblems.

In a preprocessing step we can compute all the sums we are going to need in the beginning of our algorithm. This can be done in $O(n^2)$ time. Finally, we have n subproblems in total (since $1 \leq k \leq n$) and in order to compute the optimal solution for each of these subproblems we are going to check at most n possible starting points for the final plate. Thus, the total time complexity of the algorithm is $O(n^2)$.

Problem 5: Shift Matching

A security company has to come up with an algorithm to take k guards and evenly assign them to n shifts. Each guard j has a list of shifts he/she is available for, L_j. Each shift i has a requirement that exactly g_i guards must be scheduled for the shift. Finally, the company wants an even assignment, which means that no guard should be scheduled for more than $\lceil (\sum_{0 \le i \le k} g_i)/k \rceil$ shifts.

(a) Write a polynomial algorithm to create this schedule and/or report whether it is possible to create a schedule. Analyze the runtime.

Solution

Convert the problem into a network flow problem. Each guard is a node in the graph and each shift is a node in the graph. Connect the source to each guard with a capacity of $\lceil \sum_{0 \le i \le k} g_i/k \rceil$ (the maximum number of shifts a guard should be assigned). Use an edge with capacity one to connect each guard to each shift that the guard is willing to work. Finally connect each shift i to the sink with an edge of capacity g_i (the number of guards needed for that shift). Run (capacity-scaled) Ford-Fulkerson on this graph to determine the max flow. If the max flow is less than the sum of all of the g_is (i.e., if $v(f) < \sum_{0 \le i \le n} g_i$) then a satisfying schedule is not possible. If the value of the max flow is exactly equal to the sum of all the g_is, then, for each edge between a guard and a shift that has flow, assign the guard to the shift.

The runtime of the algorithm is either $O(mC)$ or $O(m \log C)$, depending on whether it is capacity scaled or not. In our graph, there $m = O(nk)$ and $C = nk$. Therefore the runtime is either $O(n^2k^2)$ or $O(nk \log nk)$ depending on whether we use capacity scaled FF or not.

(b) The company has decided that guards should not have full control over their schedules, to prevent gaming the system. They add the constraint that a guard j can now be scheduled to e extra shifts outside his/her list L_j. Modify the above algorithm to account for this. Analyze the new runtime.

Solution

We start with the same construction as in the previous question, but we add an additional "gadget" node for every guard. We draw an edge from the guard node to the gadget with capacity e. We draw an edge with capacity one from each guards gadget to any shift node for shifts *not* listed in the guards preferred list.

As above, we run (capacity-scaled) Ford Fulkerson on the resulting network flow graph. As above, if the max flow is less than the sum of all of the g_is (i.e., if $v(f) < \sum_{0 \le i \le n} g_i$) then a satisfying schedule is not possible. If the value of the max flow is exactly equal to the sum of all the g_is, then, for each edge between a guard and a shift that has flow, assign the guard to the shift.

Exactly as above, the runtime of the algorithm is either $O(mC)$ or $O(m \log C)$, depending on whether it is capacity scaled or not. In our graph, there $m = O(nk)$ and $C = nk$. Therefore the runtime is either $O(n^2k^2)$ or $O(nk \log nk)$ depending on whether we use capacity scaled FF or not.

Problem 6: Swapping Contraband

After a few months in prison, life is easier for Fruitcake Frank. He's used his algorithms prowess to endear himself to the guards, so they're willing to look the other way with regards to minor transgressions. Fruitcake is surprised one day when he's assigned a new cell mate. It's Cupcake Carl! Before long, the two old associates have concocted a plan to start a micro-economy in the prison. Fruitcake and Cupcake want to get (what else?) candies by trading other inmates for hot commodities (cigarettes, ramen noodles (seriously, that's apparently a thing, Google it later), postage stamps, etc.).

More specifically, Fruitcake and Cupcake have n items they can trade away, $I_1, I_2, \ldots, I_n$. Each item is indivisible and can only be traded to one other inmate. There are m other inmates, and each of those inmates can place bids for one or more items. That is, the i^{th} bid specifies some subset of items, S_i and the number of candies c_i the bidder is willing to trade for that subset of items. Each bid is therefore a pair (S_i, c_i).

Fruitcake and Cupcake need an algorithm that *accepts* and *rejects* bids. If a bid i is accepted, the bidder gets to take all of the items in the set S_i and must give Fruitcake and Cupcake c_i candies. It therefore naturally follows that the sets S_i and S_j of two accepted bids i and j cannot have any items in common. Fruitcake and Cupcake want an algorithm that maximizes the number of candies they get. That's the *optimization* version of the **Contraband Swap** problem.

(a) Fruitcake suggests that there is a somewhat simpler *decision* version of this problem. Succinctly state the decision version of the **Contraband Swap** problem.

Solution

Given n items for trade $(I_1, I_2, \ldots, I_n)$ and m bids (each a pair (S_i, c_i)), determine whether there is a set of bids that Fruitcake and Cupcake can accept to get at least C candies.

(b) Because of his work in algorithms Fruitcake quickly realizes that this looks like an NP-Complete algorithm. When he mentions that, Cupcake vaguely remembers hearing something about that concept in a class he dozed through. He eagerly says the following:

> *We can prove that* **Contraband Swap** *is an NP-Complete problem by showing that* **Contraband Swap** *can be reduced to an existing NP-Complete problem in polynomial time!*

Is Cupcake right? Why or why not? (Hint: what are the implications of this reduction?)

Solution

Alas, no, Cupcake should have stayed awake in class. If we do this reduction, all we are proving is that the known hard problem is at least as hard as **Contraband Swap**. But we already know that the known problem is hard, what we want to know is whether the **Contraband Swap** problem is hard. The reduction doesn't tell us whether **Contraband Swap** is harder or easier than the known hard problem.

(c) The following problem can be assumed to be known NP-Complete problem.

The Independent Set Problem. *Given a graph G and a number k, does G contain an independent set of at least k? In a graph $G = (V, E)$, a set of nodes $S \subseteq V$ is* independent *if no two nodes in S are joined by an edge.*

Prove that your decision version of **Contraband Swap** is an NP-Complete problem. (Don't forget the first step!)

Solution

First step: **Contraband Swap** $\in$ NP. That is, given a "true" certificate of **Contraband Swap**, we can verify that it is a true instance in polynomial time. A true instance of **Contraband Swap** is a set of bids that are compatible with one another (i.e., can all be simultaneously accepted) whose candies (c_i) add up to at least C. Given a "true" certificate, we can check all of the sets S_i to ensure they have no overlapping items purchased and we can add up all of the c_i to make sure the total is at least C. This just involves some arithmetic over the sets; the check can easily be done in polynomial time (polynomial in the size of the input).

Second step: we will prove that **Contraband Swap** is an NP-Hard problem by reducing every instance of some known problem to an instance of **Contraband Swap**. For this, we will use **Independent Set**, i.e., we will show **Independent Set** $\leq_p$ **Contraband Swap**.

An input to **Independent Set** is a graph $G = (V, E)$, and a number k. To perform the mapping, we take every vertex in V and make it a bidder in the **Contraband Swap** problem. Every edge in E in the input to **Independent Set** is an item contained in the bid for the incident bidders. [Note, in this mapping, we're only hitting instances of **Contraband Swap** that have exactly two bidders per item. That's ok.] We will make every bid worth exactly one candy (i.e., $\forall\, i, c_i = 1$. [Another restriction in the instances of **Contraband Swap** that we hit, but again, ok. The goal is to "cover" every instance of **Independent Set**, so we just have to make sure it works for *every* graph.] Finally, we use the k of the input to **Independent Set** as the C of the input to **Contraband Swap**.

This reduction can be done in polynomial time in the size of the input to the **Independent Set** problem. It is just a straightforward mapping of that input onto the **Contraband Swap** input.

Now we argue that **Contraband Swap** returns true if and only if the mapped instance is a true instance of **Independent Set**. That is, Fruitcake and Cupcake can get C candies if an only if the original graph had an independent set of size k. If there is a set of acceptable bids that total $k = C$, then those bids to not share items (edges) and exactly the nodes corresponding to the successful bidders are the independent set in the original graph G. This same argument works in the other direction as well. If there is an independent set of size k in the graph G, then exactly those bids can be safely selected without causing a conflict on an item because, by definition of an independent set, those vertices do not share an edge, meaning the bidders do not share an item.