Homework 2

Winston (Hanting) Zhang

September 16, by 23:00

General Instructions. The following assignment is meant to be challenging, and we anticipate that it will take most of you at least 10–15 hours to complete, so please allow yourself plenty of time to work on it. (This assignment is probably a bit harder than Homework 1.) We highly recommend reading the entire assignment right away — you never know when inspiration will strike. Please provide a formal mathematical proof for all your claims, and present runtime guarantees for your algorithms using asymptotic (big-$O/\Omega/\Theta$) notation, unless stated otherwise. You may assume that all basic arithmetic operations (multiplication, subtraction, division, comparison, etc.) take constant time.

Collaboration. Please carefully check the collaboration policy on the course website. When in doubt, ask an instructor.

Consulting outside sources. Please carefully check the policy regarding outside sources on the course website. Again, when in doubt, ask an instructor.

Submission. Homework submission will be through the Gradescope system. Instructions and links have been provided through the course website and Piazza. The only accepted format is PDF. For this homework, we will still allow photographs/scans of handwritten solutions, but if they are illegible, you may lose points. Starting with Homework 3, only typed solutions will be accepted, and we highly recommend producing your solutions using LaTeX (the text markup language we are also using for this assignment).

Recommended practice problems (do not hand in): KT Problems 4.3, 4.4, 4.5, 4.7, 4.13.

Problem 1. [15]

Lincoln Riley and Alex Grinch[1] have approached you for some help for the next USC football game. Specifically, they are hoping to utilize your algorithms prowess to help choose which defensive back guards which of the opponent's wide receivers.

They have abstracted the problem for you[2] as follows: on any given play, there will be n opposing wide receivers and n USC defensive backs on the field.[3] (This might change from play to play, but your algorithm will be called separately for each play.) They are playing man coverage (as opposed to zone coverage), which means that they want to match one defensive back each to one wide receiver. They will feed the algorithm the heights $r_1, r_2, \ldots, r_n$ of the opposing wide receivers in inches, as well as the heights $b_1, b_2, \ldots, b_n$ of the USC defensive backs in inches. They have learned from years of coaching experience how to predict the yardage gain of a play when a wide receiver of height r is defended by a back of height b. When $r > b$, the average gain is $1 + (r - b)^2$; when $r \leq b$, the average gain is $\frac{1}{1+(b-r)^2}$. Thus, the bigger the size advantage of the wide receiver, the larger the gain, and the bigger the size advantage of the defensive back, the smaller the gain. USC obviously prefers a smaller gain.

We assume that the opposing quarterback throws to a uniformly random receiver, regardless of how good the matchup is.[4]

Knowing all of this, your algorithm is now supposed to compute an assignment of defensive backs to receivers that minimizes the expected yardage gain of the opponent. Your assignment should be a matching, so you cannot double-team a receiver.

Give a polynomial-time algorithm for this problem, and analyze it (i.e., prove that it finds the best assignment, and briefly analyze the running time).

Proof. Let B be the set of defensive backs and R be the set of wide receivers. Sort B and R by increasing height and label them as $b_1, b_2, \ldots, b_n$ and $r_1, r_2, \ldots, r_n$. We claim that greedily assigning b_i to r_i, $1 \leq i \leq n$, minimizes the expected yardage gain. Call this algorithm G.

Suppose we have an optimal algorithm O, which matches each defensive back to wide receiver from shortest to tallest. For the sake of contradiction, suppose O choice of matchings differ from G. Let the first difference in matching occur at iteration i. Then we know that G assigned b_i to r_i. Furthermore, O assigned b_i to some r_j, with $j > i$, since the previous iterations already matched all the receivers before r_i. Now we claim that O is actually not optimal. Suppose O assigned b_k to r_i. Again we know $k > i$, since every defensive back before b_i was already matched. The two matches between (b_i, r_j) and (b_k, r_i) incurs expected gain in yardage of $2^{r_j - b_i} + 2^{r_i - b_k}$. However, if we instead matched (b_i, r_i) and (b_k, r_j), the expected gain is $2^{r_i - b_i} + 2^{r_j - b_k}$. Since our setup orders B and R in height order, $k > i$ implies that $b_k > b_i$. In particular, set $b_k = b_i + d$ for $d > 0$. Now we can compute, setting $2^{-d} = X$,

$$2^{r_j - b_i} + 2^{r_i - b_k} = 2^{r_j - b_i} + 2^{r_i - b_i - d} = 2^{r_j - b_i} + 2^{r_i - b_i} X = 2^{r_j - b_i} + 2^{r_i - b_i} + 2^{r_i - b_i}(1 - X)$$

and

$$2^{r_i - b_i} + 2^{r_j - b_k} = 2^{r_i - b_i} + 2^{r_j - b_i} X = 2^{r_i - b_i} + 2^{r_j - b_i} + 2^{r_j - b_i}(1 - X).$$

Note that the original matching and the modified matching differ only by where X is placed. Now $d > 0$ implies that $0 < X = 2^{-d} < 1$, so $0 < 1 - X < 1$.

Since $j > i$, we have $r_j > r_i$, so $2^{r_j - b_i} > 2^{r_i - b_i}$, so $2^{r_j - b_i} X < 2^{r_i - b_i} X$. Hence, we conclude that

$$2^{r_i - b_i} + 2^{r_j - b_k} < 2^{r_j - b_i} + 2^{r_i - b_k},$$

meaning that we have better defense when we modifies the matching given by O. This contradicts the optimality of O, and hence we conclude that O does not differ from G. Hence O is G, and greedy is optimal.

[1] Just in case: those are the head coach and the defensive coordinator for the USC football team.

[2] You didn't know they were that good at math, did you?

[3] No, you cannot assume that $n \leq 10$. Do you really think that Pac12 referees will notice if each team has 38 players on the field? Yeah, right!

[4] There's only one Caleb Williams in the Pac12.

The runtime of G is $O(n \log n)$ for the sorting, using a good sorting algorithm like quicksort or mergesort, plus $O(n)$ from iterating to create each matching. Hence the total runtime is $O(n \log n) + O(n) = O(n \log n)$. □

Problem 2. [10+5=15]

Imagine that you are an old-time vendor at a market place. When your customers buy — say — potatoes, you want to charge them by the pound. To do so, you have a scale, and to use it, you need to balance the scale against the potatoes. You do this by placing metal weights of known weight on the other side, and when the scale is balanced, you know how much the product weighs.[5] You want to bring only n different weights, but be able to measure as large of a consecutive (i.e., without any gaps) range $\{1, 2, \ldots, W\}$ of weights as possible, by using combinations (i.e., subsets) of the weights you brought. Here is a simple algorithm for doing this:

Algorithm 1 Weight Selection

Start with the empty set $S = \emptyset$ of weights.
for $i = 1, \ldots, n$ **do**
  Add to S the smallest integer weight w which you cannot yet get as a sum of a subset of S.

1. This algorithm looks reasonable, but it's very simple and greedy. Maybe it would be better to add a smaller weight w next which you can already get in some way from the weights you have, but which could possibly help you a lot later? Prove that this is not the case, by proving the following:

   Theorem 1. *The algorithm produces a set S such that among all possible sets T of size n, S produces the largest consecutive range $\{1, 2, \ldots, W\}$ of weights as sums of subsets.*

   Also briefly analyze the running time of the algorithm.

   Proof. **Note: We use the results of 2b here.**

   By (2b), the the algorithm produces the set $S = \{1, 2, \ldots, 2^{n-1}\}$ with range $2^n - 1$. Let T be any other set of size n. There are $2^n - 1$ nonempty subsets of T. Hence there is a maximum of $2^n - 1$ possible sums of subsets of T, implying that the range of T can be at most $2^n - 1$. Thus, the set S produced by the algorithm is optimal, as no other set T can improve upon its range.

   The runtime for this algorithm is $O(n)$. Within each iteration i, by (2b), we know that we must pick up weight 2^{i-1} without any extra computation in $O(1)$. Hence over the loop we accumulate $O(n)$ runtime. □

2. If you run the algorithm by hand, you will probably discover an interesting pattern in the weights w that it includes. State this pattern, and prove formally that this is the output of the algorithm.

   Theorem 2. *Algorithm 1 produces weights $1, 2, \ldots, 2^{n-1}$.*

   Proof. **Note: We do not use the results of (2a) here, so the proof is not circular.**

   We proceed by strong induction to prove that the i^{th} iteration of the loop adds weight 2^{i-1}. The base case $n = 1$ is clearly true as we must pick weight 1 on our first iteration.

   Now assume that on the k^{th} iteration, we have weights $1, 2, \ldots, 2^{k-1}$. We want to show that the $k + 1^{\text{th}}$ weight has weight 2^k.

   For any number $1 \leq m \leq 2^k - 1$, we can decompose m into its binary representation $\sum_{i=0}^{k-1} b_i 2^i$ for $b_i \in \{0, 1\}$, which is exactly a sum of the a subset of the weights $\{1, 2, \ldots, 2^{k-1}\}$. Hence the weights $\{1, 2, \ldots, 2^{k-1}\}$ have range $2^k - 1$. Thus on the $k + 1^{\text{th}}$ iteration, the algorithm adds the smallest number we cannot sum to, 2^k. This completes our induction. □

[5] A little less than you are charging the customer for, because of course, you have manipulated the scale a bit.

Problem 3. [0]

Chocolate Problem: 2 chocolate bars

Reminder: If you solve a chocolate problem (which you can do in groups of size up to 3), please e-mail David with the solution — do not submit it on Gradescope. Also, feel free to list preferences or dietary restrictions for/against particular types of chocolate.

Imagine that you are in a chocolate store which makes n different kinds of flavored chocolate truffles. They sell these truffles in m different kinds of "mix bags", where bag i contains a set $B_i \subseteq \{1, \ldots, n\}$ of distinct truffle types. Each bag costs 1 dollar, and you have k dollars with you. You would like to try as many different *types* of truffles as possible for your money. So you want to buy k bags which together have the largest variety of different kinds of truffles.[6]

There is an obvious greedy algorithm for choosing bags of truffles. In each iteration, add the bag which gives you the largest number of new types of truffles compared to what you already had in your earlier bag. (In the first iteration, this means just taking the bag with the largest number of different types.)

1. Warm-up (not worth any chocolate by itself): As we said in class, greedy algorithms are usually not optimal. This one is no exception. Construct an example input on which this algorithm does not get the largest number of distinct truffle types.

2. While the output of the greedy algorithm here is not optimal, it is actually not too far off. Prove that it gets at least a $(1 - (1 - 1/k)^k)$ fraction of the maximum possible number of truffle types you could have gotten with k bags.

   (Note that $(1 - 1/k)^k \to 1/e$ as $k \to \infty$, and for all k, you have that $(1 - 1/k)^k \leq 1/e$. (Here, $e \approx 2.71828$ is the Euler constant.) So the algorithm's output is always within a factor no worse than $1 - 1/e \approx 63\%$ of optimal.)

[6] Having duplicates of the same kind is not a problem, but it doesn't make you any happier, either.