# CSCI270 Homework 2

Jacob Ma

September 14, 2022

**Problem 1: 15 points**

Lincoln Riley and Alex Grinch[a] 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[b] as follows: on any given play, there will be $n$ opposing wide receivers and $n$ USC defensive backs on the field.[c] (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 $b$. The average gain is $2^{r-b}$. 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.[d]

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).

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

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

[c] 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!

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

**Solution: Algorithm**

The algorithm is as follows 1. We will firstly prove this finds the best assignment, and then analyze the running time.

**Algorithm 1** Defensive Back and Receiver Assignment Algorithm

Sort USC's $n$ defensive back by non-decreasing heights, get a non-decreasing array $D[n]$

Sort opponent's $n$ wide receivers by non-decreasing heights, get a non-decreasing array $R[n]$

Start with an empty set $M = \{\}$, storing all the matched pairs of back and receivers

**for** $i = 0, \ldots, n-1$ **do**

  Add the pair of $(D[i], R[i])$ into set $M$

**end for**

*Proof.* We try prove this algorithm finds the best assignment.

Let's say our algorithm above is called $f$, and would find assignment for some defensive back $b$ a receiver $r$, $f(b) = r$. We assume there exist an optimal algorithm $f'$ which would find different assignments than our algorithm, we try to prove that our algorithm finds assignment no worse than the optimal algorithm.

Consider a receiver $r_1$, in our algorithm, $f(r_1) = b_1$, but in the optimal algorithm,$f'(r_1) = b_2$. Because according to our algorithm, $b_1$ must be the smallest height $r_1$ can find, thus $b_2 \geqslant b_1$. And in $f'$, $b_1$ is matched with another receiver $r_2$, in other words, $f'(r_2) = b_1$. Because in our algorithm, $r_1$ is the smallest height $b_2$ can find, thus $r_2 \geqslant r_1$. Thus, now we have $b_2 \geqslant b_1, r_2 \geqslant r_1$.

Consider we change the optimal algorithm to more like our algorithm, by exchanging this assignment. Originally, we have $\{(r_1, b_2)(r_2, b_1)\}$, now we change it into $\{(r_1, b_1), (r_2, b_2)\}$. The change of the average gain is calculated as follows:

Before the exchange, the average gain is

$$2^{r_1 - b_2} + 2^{r_2 - b_1}$$

After the exchange, the average gain is

$$2^{r_1 - b_1} + 2^{r_2 - b_2}$$

Thus, the difference between before and after the exchange is

$$\begin{aligned} 2^{r_1-b_1} + 2^{r_2-b_2} - 2^{r_1-b_2} - 2^{r_{r_2-b_1}} &= \frac{2^{r_1}}{2^{b_1}} + \frac{2^{r_2}}{2^{b_2}} - \frac{2^{r_1}}{2^{b_2}} - \frac{2^{r_2}}{2^{b_1}} \\ &= \frac{2^{r_1} - 2^{r_2}}{2^{b_1}} - \frac{2^{r_1} - 2^{r_2}}{2^{b_2}} \qquad \because b_2 \geqslant b_1, r_2 \geqslant r_1 \\ &= \frac{|2^{r_1} - 2^{r_2}|}{2^{b_2}} - \frac{|2^{r_1} - 2^{r_2}|}{2^{b_1}} \\ &\leqslant 0 \end{aligned}$$

Thus, the matching after the exchange is no worse than the optimal exchange before the exchange.

Let's define the distance between any matching and our matching be counting how many $r$ did not get matched with $f(r)$. In each swap, we had at least one pair of $r$ and $f(r)$ got matched. Because matching is bijective, since $r$ and $f(r)$ are matched in our algorithm, both original matching containing either $r$ or $f(r)$ is not a matching could be get by our algorithm. Thus, each swap/exchange would at least reduce the distance by 1.

Thus, we know that by repeating this exchange, the matching get from the optimal algorithm would become the matching get from our algorithm. Thus, the matching of our algorithm is no worse than the optimal matching.

The conclusion follows. □

**Solution: Run time Analysis for Problem 1**

The algorithm first take two sorting, which both takes $O(n \log n)$ (if using algorithms like quick sort with good pivot point choice).
And than, the declaration and initialization of the empty set would take $O(1)$.
As for the for loop, it takes $n$ iterations, and has a add method of set in each iteration, which takes $O(\log n)$ if implemented using C++ set, which takes $O(n \log n)$ for the whole for section.
Thus, the algorithm would take $O(n \log n)$ because $O(n \log n) > O(1)$.

**Problem 2: 10+5=15 points**

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.[a] You want to bring only $n$ different weights, but be able to measure as large of a consecutive (i.e., without any gaps) range $\{1, 2, ..., W\}$ of weights as possible, by using combinations (i.e., subsets) of the weights you brought. Here is a simple algorithm for doing this2.

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 0.1**

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

   Also briefly analyze the running time of the algorithm.

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.

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

**Algorithm 2** Weight Selection

Start with the empty set $S = \varnothing$ 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 $S$.
**end for**

*Proof.* As for 2-1, we prove the theorem by induction.
**Base Case:** Let's consider two base cases, where $n = 0$ and $n = 1$.

(1) When $n = 0$: There are no elements in set $S$, $S = \varnothing$, thus, the consecutive range of weights as sums of subset would be $\varnothing$ as well. Because this $S$ is the only possible set $T$ while $n = 0$, thus $S$ must produce the largest consecutive range of weights as sums of subsets. Thus, the theorem holds for $n = 0$.

(2) When $n = 1$: There are 1 element in set $S$. According to Algorithm **??**, we would choose 1 as our only element in $S$, and produce the consecutive range $\{1\}$. For any other possible $T$, the range would also be of size 1, because there is only one element in any $T$. Thus, the theorem holds for $n = 1$.

**Inductive Hypothesis:** Assume the algorithm produces a set $S$ such that among all possible sets $T$ of size $n = k$, where $k \in \mathbb{N}$, $S$ produces the largest consecutive range $\{1, 2, \cdots, W\}$.

**Induction Step:** We need to prove the statement holds for $n = k + 1$. We prove this by contradiction.

According to this algorithm, because we can produce every weights in $\{1, 2, \cdots, W\}$, next element we add into $S$ would be $W + 1$. Because we can produce every element from $W + 2$ to $2W + 1$ by adding this element to the previous sum, we can get consecutive range $\{1, 2, \cdots, 2W + 1\}$.

Let's assume there exists another possible set $S' \in T$ of size $k + 1$, could produce consecutive range $\{1, 2, \cdot, 2W + 1, \cdot, 2W + i\}, i > 2, i \in \mathbb{N}$. This set $S'$ would produce a larger consecutive range than $S$, meaning the element $2W + 2 \in \{1, 2, \cdot, 2W + i\}$. Consider the element $2W + 2$, which is a sum of subsets in $S'$:

According to the induction hypothesis, the maximum sum of subsets of a set of size $k$ is $W$. Thus, if we want to get a sum of $2W + 2$ when we consider a set of $k + 1$ element, we need to add a new element $W + 2$. In this case, we could get $2W + 2$ by adding $W + 2$ to the subsets whose sum is $W$. However, according to the Induction hypothesis, the original set with $n$ elements could only generate range from 1 to $W$, and there is no way of generating $W + 1$ if the $k + 1^{th}$ element is $W + 2$, thus, the new range is not consecutive. Thus, there exists a contradiction.

Thus, the statements holds true when $n = k + 1$.

The conclusion follows the induction. □

**Solution: Runtime Analysis for Problem 2-1**

The initialization of an empty set $S$ takes constant time $O(1)$.

Consider the for loop, there are $n$ iterations in this for loop. Inside each iteration, we will add $2^i$, which will be proved in next problem 2-2, which is a constant time operation. Thus, the whole for loop would take $O(n)$.

Because $O(n) > O(1)$, the algorithm would take $O(n)$ time.

**Solution: Problem 2-2**

The pattern gives a set of elements looks like this

$$1, 2, 4, 8, ..., 2^{n-1} \qquad n \in \mathbb{N}$$

When we are adding the $n^{th}$ element, we are always adding a element of value $2^{n-1}$. And it will produce a range of $2^n - 1$.

*Proof.* We prove this by strong induction.

**Base case:** n=1

The smallest sum we cannot get right now is 1, thus we add 1.

$$1 = 2^{1-1} = 2^0$$

And it produces consecutive range $\{1\}$.

Thus, the statement holds for $n = 1$.

**Induction hypothesis:** For all $i \leqslant k, i \in \mathbb{N}$, the $i^{th}$ element we add according to greedy algorithm is $2^{i-1}$, and after adding the $i^{th}$ element, the elements can generate a consecutive range $\left\{1, 2, \cdots, 2^i - 1\right\}$

**Induction step:** We need to prove that the $(k+1)^{th}$ element we add according to greedy algorithm is $2^{k+1-1} = 2^k$, and it would produce a consecutive range $\left(1, 2, \cdots, 2^{k+1} - 1\right)$

According to Induction Hypothesis, because we can generate a consecutive range of $\left\{1, 2, \cdots, 2^k - 1\right\}$ of the first $k^{th}$ element, the new $k^{th}$ element would be $2^k - 1 + 1 = 2^k$.

Thus, according to the greedy algorithm, because we can get every sum consecutively from 1 to $2^k - 1$, the next element, $(k+1)^{th}$ we will add is the next smallest element, $2^k - 1 + 1 = 2^k$. Thus, the statement holds true for $n = k + 1$.

We could construct any number between $2^k + 1$ to $2^{k+1} - 1$ by adding all the elements whose sum was from $\left\{1, 2, \cdots, 2^k - 1\right\}$ by induction hypothesis to $2^k$. The largest number it could produce is $2^k + 2^k - 1 = 2^{k+1} - 1$, and any consecutive number in the range stated above could be constructed. Visually, it looks like this

$$1, 2, \cdots, 2^k, 2^k + 1, 2^k + 2, \cdots, 2^k + 2^k - 1$$

Thus, after adding $2^k$, the new set with $k + 1$ element generates consecutive range between 1 to $2^{k+1} - 1$.

Thus, the conclusion follows the induction. □