Nevin Liang
Ty Koslowski
Nhat Ho

Homework 5

Problem 1

We first proceed by simplifying the problem to a more visually understandable one. Let there be an array of n integers, where each integer corresponds to the length of a word in the text. We would like to find the optimal partition of this array so that the sum of L minus the sum of the numbers in each partition squared is minimized. Namely,

$$S = \sum_{p_x \in p} (L - \sum_{i \in p_x} c_i)^2$$

where p_x is each segment in the parition p, and c_i is the value of the index i inside the segment p_x. For example, if L = 13, and our array was 7, 4, 5, 2, 6, 7, 1, 3, 5, 5, 4, 9, 2: our partition could be (7, 4), (5, 2, 6), (7, 1, 3), (5, 4), (9, 2). And the total error (which we are supposed to minimize) would be 2^2 + 0^2 + 2^2 + 4^2 + 2^2 = 28.

Note: for this solution we will let our array be 1 indexed, with the 0 index holding a value of 0.

Let us define our dp array to be dp[i], where i represents the index of the word in the array we are currently on, and dp[i] representing the sum S for all the partitions of the first i words.

The first thing we do is calculate all values of pref where pref[i] is the sum of the first i + 1 values in the array. We will use this later on to calculate the sum of ranges in O(1) time.

The dynamic programming relation we can derive from using our definition of dp[i] is that

$$dp[i] = \min(sum(j + 1, i)^2 + dp[j])$$

where j runs from 0 to i-1 inclusive, and sum(x, y) is the sum of all the array values from index x to index y inclusive. Here is a short example of the code:

```
int partition(vector<int> c, int L, int n) {
    vector<int> dp(n + 1, INT_MAX), pref(n + 1, 0);

    for (int i = 1; i <= n; i++)
        pref[i] = pref[i - 1] + c[i - 1];

    dp[0] = 0;
    for (int i = 1; i <= n; i++)
        for (int j = 0; j < i; j++) {
            int slack = L - (pref[i] - pref[j] + i - j - 1);
            if (slack < 0) continue;
            dp[i] = min(dp[i], slack * slack + dp[j]);
        }
    return dp[n];
}

int main() {
    vector<int> c = {4, 4, 5, 2, 6, 5, 1, 3, 5};
    cout << partition(c, 13, 9) << endl;
}
```

A few key points to notice about this code is that when calculating slack, we add an additional i - j - 1 because that is the number of spaces in the words from j + 1 to i inclusive. We also 1-index pref and dp, so pref[i] and dp[i] correspond to c[i - 1]. We do this to make the dp array easier to handle, as we can now put dp[0] = 0. Our prefix array is used to calculate sum(j + 1, i) in O(1) time. To fit the problem specs, all you have to do is calculate the integer array d by using a simple string.length() function. This part of the algorithm takes O(N^2) time.

Note that the quick code I wrote above only calculates the minimum value of the summation, and does not actually find the partition.

In order to find the partition, all we need to do is have another array, point, which is updated in the inner for loop. Instead of the min statement, we write an if statement that says, if slack * slack + dp[j] < dp[i], then #1 update dp[i] to the new min, and #2 update point[i] to equal j.

This way, when we reach the end of the dp, we can trace the partitions back and see what the actual minimum partition was.

So, if slack[9] = 7, slack[7] = 5, slack[5] = 2, and slack[2] = 0, then we know that the partition is (1, 2), (3, 4, 5), (6, 7), (8, 9). This part of the algorithm takes a maximum of O(N) time.

Problem 2

We start from smaller size rods and build up until we get the original size rod. For sure, the price of a rod of length 1 maxes out at 1. For a rod of length 2, however, the price is the max of a combination of 2 length 1 rods and the length 2 rod. It appears that the length 2 rod sells for more. In fact,

$$P_i = \max(a_i, \max(P_{i-k} + a_k)\,|_{k\in([1,i]\cap\mathbb{Z})})$$

where a_i is the price of the ith rod.
Thus, all we have to do is loop through every value of $i : \{0,1,\ldots,n\}$, and calculate the value of P_i.

Code should look something like this:

```
int maxPrice(vector<int> prices, n) {
    vector<int> dp(n + 1, 0);
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= i; j++)
            dp[i] = max(dp[i], dp[i – j] + prices[j])

    return dp[n];
}
```

where prices is the array of prices and index i corresponds to the price of a rod of length i. The complexity of this algorithm should be $O(n^2)$.

Problem 3

For this problem, there are a few possible complexities we can deal with, namely O(n), O(nlogn), O(n^2), O(n^2logn), and higher complexities. The following algorithm is O(nlogn), and I believe it cannot be optimized to O(n), although I am not entirely sure.

Algorithm: Create a set of all prefixes of the string (there are n of them). Reverse them. For example, for the example string in the problem statement “abababa,” the prefixes are a, ab, aba, abab, ababa, ababac, abababa. Reversing these gives the set {a, aba, ababa, acababa, ba, baba, cababa}.

If we sort this in alphabetical order, then for a given string like ababa, we can test if a, ab, aba, abab, and ababa are all in the set in one single loop of O(n) complexity. All we have to do is mark down whether we have found it in the set and continue with the next letter.

Thus, the algorithm goes like this:

```
int* longest_prefix_suffix(string s) {
    // get set of all prefixes
    set<string> prefixes;
    for (int i = 1; i <= s.length(); i++)
        // reverse because we are comparing backwards
        prefixes.insert(s.substr(0, i).reverse());

    // loop through every index and figure out the longest prefix/suffix
    int longest[s.length()] = {0};

    // we start from i = 1, because i = 0 has no possible prefix
    for (int i = 1; i < s.length(); i++) {
        string temp = "";
        for (int j = i; j >= 0; j--) {
            temp += s[j];
            // search for it in the prefixes array
            if (prefixes.find(temp) != prefixes.end()) {
                longest[i] = max(longest[i], i - j + 1);
            }
        }
    }
    return longest
}
```

The complexity of this is then O(n^2) because for every index we look at it takes O(n) time to figure out what the longest prefix/suffix is.

Problem 4

It is possible to do this scheduling in O(n^2) time. We know that n = 2^k. The following algorithm uses iterative recursion, which is dynamic programming. Running my algorithm in reverse will also work and the dp aspect of it will seem more apparent.

One interesting idea appears when we decide to split the players in half (fairly obvious as there are literally 2^k players lol). Split the players in half, and now we have two groups of 2^(k - 1). Let these to teams be Red and Blue for sake of terminology. There are now two types of games: matches between Red and Blue players and matches where the two players are the same color.

The number of rounds between Red and Blue players, so that each person from Red plays each person from Blue once, is just the number of players in each group. Pretend A, B, C, D were in Red and 1, 2, 3, 4 were in Blue. Round 1 is (A1, B2, C3, D4). Then, we just rotate: Round 2 is (A2, B3, C4, D1); Round 3 is (A3, B4, C1, D2); Round 4 is (A4, B1, C2, D3).

Thus, the number of rounds between players of opposite colors is 2^(k - 1). Now, what about between players of the same group. Well, we're back to the original problem: trying to schedule 2^k people, except now we have half the number of people.

With this combination of splitting rounds into two parts, we can schedule as shown in the example below:

	1	2	3	4	5	6	7
A	E	F	G	H	C	D	B
B	F	G	H	E	D	C	A
C	G	H	E	F	A	B	D
D	H	E	F	G	B	A	C
E	A	D	C	B	G	H	F
F	B	A	D	C	H	G	E
G	C	B	A	D	E	F	H
H	D	C	B	A	F	E	G

In the table above, there are three colors: green, orange and teal. Green represents the matches where members from {A, B, C, D} are going against members from {E, F, G, H}. Everything after that is intra-group matches. Columns 5 and 6, the orange columns, are inter-group matches between groups {A, B} and {C, D}. Similarly, Column 7, or the teal column is just the match between {A} and {B}. Scheduling this way leads to a perfect 2^k - 1 matches, as

$$2^{k-1} + 2^{k-2} + 2^{k-3} + \ldots = 2^k - 1$$

Running this backward, we can start out with 1v1 matches and then clump them together into groups. So, A vs B, C vs D, E vs F, G vs H would be day 1, and then we would clump the matches that have happened into groups. So, {A, B} would become a group and so on. Then, we would do inter-group matches and proceed from there repetitively until we reached a group of size 4 vs a group of size 4 which would take 4 matches as shown in the green columns.

Note that inter-group matches between two groups of size x requires x days because each person must play every person in the other group once. We accomplish this by rotating as described before.