Buffalo Hird
CS124
Assignment 5

Problem 1:

We will solve 1a and use this to easily solve the other subproblems

Problem 1a:

We showed in class how to solve a text search problem for a substring m in n in time $O(nlog(n))$. Here, however, we want to reduce this runtime to $O(nlog(m))$, which should be faster given that we are searching for a smaller string in a bigger string (or it is not a particularly useful search). We want to divide up this problem such that our runtime grows logarithmically with m. We also note that our runtime will surely be multiplicative with n as we will need to look at all letters in our string we are searching through to detect matches.

We design a way such that we can do a divide and conquer set of operations for each divided item of $O(nlog(n))$ where n is now a smaller number the size of the subset of the problem we have divided to. We note that if we naively divide our string into segments of length m and search each of these, we might miss matches in which these subsets overlap. We must therefore design our subsets such that they uniquely contain any matches that might be found by overlap. We therefore create sets at each interval of m throughout the string each of length $2m-1$. By doing this, we will uniquely find each case of this searched-for string in our search string.

Runtime analysis: We divide our search string of length n into $\frac{n}{m}$ elements of length $2m-1$. By doing this, we execute $\frac{n}{m}$ searches in time $O(mlogm)$. We therefore have a final worst case runtime of $O(n * log(m))$.

Proof of correctness: Given we make a FFT convalution:

$$P^R = a_{k+1}, ..., a_0$$

$$T_i = C_0, ...C_{2k-1}$$

We will therefore compare the coefficients of P^R to all the coefficients of T_i for our first k elements which is the length of our searched-for string. After this point, P^R will run out of string to compare to and terminate such that only $k-1$ coefficients will be compared to in this second part. We then must show that this overlap will be sufficient and not cause duplicates. If we consider

$$P^R = a_{k+1}, ..., a_0$$

$$T_i = C_0, ...C_{2k-1}$$
$$T_j = C_k, ...C_{3k-1}$$

From this it can be seen that P^R will eventually terminate comparing with T_i but will continue to compare with T_j such that there are no gaps in C and it evaluates all possible substrings once. As we only index $k-1$ slots overlap, a substring contained in the overlap would need to have its first character in T_i and not T_j as this would be C_{k-1}. The converse can be said for a string ending at C_{2k}. Therefore, we will cover all strings without duplicates.

Problem 1b: We now add P potentially containing "don't care" symbols such that we consider this a character match regardless of what T contains. We consider the mapping from lecture

$$0 \to 01, 1 \to 10$$

From this we get mapped a 1 only if a character matches as $(0,1)*(0,1) = 1$ and $(1,0)*(1,0) = 1$ but $(1,0)*(0,1) = 0$. We can then map

$$* \to 11$$

Such that no matter what we dot product it with we get 1 as the output.

Problem 1c:

We now must deal with the case where our alphabet is larger than 2 items. We can't use this simple binary implementation from earlier and regardless binary produces large numbers if we represent them as 0 and 1s in base10. Instead, we represent the numbers as themselves, being instead clever in our comparison of these numbers. When we compare coefficients we now instead of multiplying them we take the square of their difference. If these are in fact equivalent coefficients then this result will be 0 as the difference between a number and itself is 0 and we square to account for negative and positive coefficients. This makes no asymptotic difference on the runtime as we were already multiplying a given x and y in the polynomial and now we do $(x-y)^2$ instead which is a constant number of additional computations.

Problem 1d:

We modify our solution in c and use our logic from b to let us support languages of more than 2 characters with "don't care" symbols. We want to modify the calculation in c such that we get 0 not just if $x = y$ but if either of them are a "don't care" . Since our desired outcome is 0, we map $* \to 0$. We can then modify our comparison function to return 0 if either x or y are *. We can simply multiply by both x and y, as if they are equal we will be multiplying them by 0 already, and if they are not equal, we cannot produce 0 by multiply non-zero numbers. We therefore get our comparison function of:

$$xy(x-y)^2$$

Problem 2:

We design an al gore rythm such that we can, given a dictionary that compresses strings, encode a string using the dictionary such that we compress it. We give an algorithm in $O(nmk)$time where n is the length of the string, m is the length of the dictionary, and k is the maximum length of a string in the dictionary.

We define a function miminumEncoding = $f(x)$ such that given a string x we compute $f(x) = min_{i \in 1,2...n} f(a - D_i) + 1$ such that we compute dividing and conquering the element in the dictionary that best reduces the string's length while this is possible.

Runtime analysis: We look at each of m elements in the dictionary and compare a string of at most length k for each of these comparisons. As we have n elements in our string, we will have n of these min calls which each have $m * k$ runtime. We therefore have asymptotic worst-case runtime of $O(n * m * k)$.

Proof of correctness:

We prove this inductively, noting our base cases:

We have our base case $f(0) = 0$ such that you cannot further reduce an empty string. We also have $f(n) = \infty$ when $n < 0$ as you cant reduce a string such that it is negative.

We then prove our inductive step $f(k) = min_{i \in 1...m} f(k - D_i) + 1$. Given that these values are valid, we are returning the min of these values, such that we are returning the smallest value possible following from step $k - 1$ plus an additional step for computing the current element.

We can now, assuming infinite precision complex arithmetic, we can improve our running time to $O(nmlog(k))$ using FFT. Using our result from problem 1, we know we can use FFT to compare a string of k of a long string of length n in time $nlog(k)$. We therefore begin by finding all occurences of each dictionary item of maximum length k in the string of length n. We accomplish this in time $m * nlog(k)$ as there are m strings we use FFT on to solve this. After we have accomplished this, we can compare each of these m dictionary elements to the string of length n such that we have a runtime $\tilde{} \; m * n * log(k) + m * n \rightarrow O(m * n * log(k))$.

Problem 3:

We can use a divide and conquer and dynamic programming approach to solve this problem. For a given desired (k dividers creating k+1) partitions we want to minimize our imbalance for an array of n elements which these dividers subdivide. We define the value of a partition as the sum of its elements such that we want to divide our array into k+1 slots such that we minimize the maximum distance any partition is from the entire array's average value.

We can use dynamic programming here to inductively build up our results for larger lists and k values. We begin for the k=1 case to more simply see this algorithm. We begin with our full length n array and attempt putting the

divider in each of $n-1$ slots and return the minimum imbalance produced by this. This process occurs n times as we try $\sim n$ positions. We note that trying each of these positions takes linear time as we will have already calculated the values for the subarrays. We can find this minimum imbalance then in linear time as we can sum as we move our divider such that we don't begin from the beginning where the divider is placed and sum $n-1*n-2*n-3*...*2$ suboptimally. We however have an additional multiplicative overhead of n for each of these dividers, as we must calculate the minimum imbalance for $k-1$ dividers leading up to this divider. We note that for $k=1$ this additional n overhead is useless information and $n=1$.

We then build up keeping t rack of our minimal imbalance as we build. We find the minimal imbalance for every divider position for length m and then build up to find the minimal balance for every length m from 1 to n. We note that there is symmetry in this approach and by slowly building up cases of larger m, we are building cases of smaller $n-m$ on the right side of our divisions being made. After this process then, we have stored in n^2 time the minimal balance for every subarray configuration and therefore can return the indices of the minimum of these maxes. When we continue to do this for our $k!=1$, we repeat this process for each k, using the lookup table to get values for $k-1$ such that we know where to add an additional divider such that we create a new minimal balance based on the problem so far.

In more mathematical terms we have the base cases:

$$\begin{array}{ll} 0 & n=1 \\ \Sigma A[0:n] & k=0 \end{array}$$

This means when we get to a single element array, it has no imbalance. When we are no longer adding dividers and we just want the minimal imbalance , we return its summed value such that we can compare its value to other subarrays.

We then define our equation as:

$$max(min_{i\in 1...n}[f(k-1,i-1)]-avg, \Sigma A[i:n]-avg)$$

Where avg is the average of an array. We therefore determine the average of our array and return the minimal value from our function from our lookup table (the mimimal imbalance so far) or the minimal imbalance from our array itself if it is larger than our current minimum imbalance and we can therefore achieve no better as we must include this.

Runtime analysis: As has been discussed, we do n operations to "recursively" determine the minimum imbalance for each of n divider positions for an array, and we repeat this k times until we have found minimum imbalance for k dividers. Our algorithm therefore runs in $O(n^2k)$.

If we redefine minimum imbalance, we have the same algorithm but now we solve for the minimum of the sum of imbalances which are the difference of each subarray's sum and the average. We essentially plug this new evaluation

function into the black box that is our algorithm to produce this desired output. The runtime should be the same runtime as we iterate through the subarrays equivalently but now compare different values.

Problem 4:

We note that the extra space on a line containing words $l_i$through $l_j$is $M - j + i - \Sigma_{k=i}^{j} l_k$. As the penalty is the sum over all lines except the last of the cube of the extra space at the end of the line, we can define our DP.

We have base cases for: $c(i,j) = \infty$ for $spaces(i,j) < 0$, where $spaces(i,j)$ is the funtion that determines the number of spaces left in a line from i to j. Obviously this is an impossible value that we assign infinity to such that we ignore it.

We have $c(i,j) = 0$ if $j = n$ for positive values from $spaces(i,j)$ such that we are already at the end of the last line such that no white space needs to be provided.

We then have our last case, which is cubed as defined in our problem, such that in all other cases $c(i,j) = spaces(i,j)^3$.

We can then consider our dynamic programing problem such that we can give C an index j and it will give an optimal cost of ordering the first j words. We then have $C[0] = 0$ intuitively as 0 words require 0 space. We can then determine the optimal cost for ordering the first j (for $j > 0$ as is defined possible by the previous cases) words as:

$$C[j] = min_{0 \leq i \leq j}(C[i] + c(i,j))$$

Runtime analysis: we run through all j in range n and for each of these j's we compare to the previous $j-1$ $i's$ such that we can find the minimal $C[i]+c(i,j)$ for each j. We therefore have a worst case runtime analysis of $O(n^2)$.