Buffalo Hird
CS 124
Assignment 4

Problem 1: To solve this, we need to find a way of set covering which allows us to solve in k steps, but our greedy algorithm will use $logn$ steps, where n is the number of items in the domain. For $k = 3$, we can equally assign each item iteratively in the domain with n items, such that $x_i = 1, x_{i+1} = 2, ..., x_{i+k} = k$. We proceed incrementing i until we have supplied a value to each item in the domain so that there are $\lfloor \frac{n}{k} \rfloor$ items for each value. By doing this, we can construct 3 (or k) sets that cover the domain, by making a set of all 1's, all 2's, etc. for all k assignments.

We can then design sets such that we have a set for 2^{k-i} elements for $0 \leq i \leq k$ such that we have $log(n)$ sets as we divide n into decreasing powers of 2 which is the definition of log_2.

We then run through our greedy algorithm to prove that it will find a $log(n)$ set cover while a k set cover is available. We note that in each step for a given R remaining uncovered items, there will be at most $\lceil \frac{R}{k} \rceil$ covered by each k set, but $\frac{R}{2}$ covered by the next item of size logarithmic in terms of n. Therefore, our greedy algorithm will choose the next largest log set, subdividing the remaining uncovered domain by 2. As we have equally spaced our items in each set in k, we will always have $\sim \frac{R}{k}$ items which could be covered by each of the k ideal sets, but this value is smaller and greedily less useful than $\frac{R}{2}$.

As we will choose the largest available log set in each iteration, we will continually half the domain until we have covered the entire domain in $log(n)$ sets. However, our greedy algorithm misses that we could simply choose each of our k sets that divide the problem iteratively into subsets labeling each item from 1 to k as many times as it takes to cover all items with a value in this range. As a result, we will choose a larger value of sets with our greedy algorithm given that $k > log(n)$.

In the case of $k = 3$, we can create 16 items, such that we can cover them in a set of 6 1's, 5 2's, and 5 3's and have a 3 set covering. However, our greedly algorithm would first pick 8 items, then 4, then 2, then 1, then 1, to produce a 5 set covering.

Problem 2: To solve this problem we can consider the 2 machines as we greedily add items $j_1$through j_n to the machine with the smallest stack. We want to show that this has a worst case performance of $\frac{3}{2}$ of the ideal performance given a smarter non-greedy algorithm. We first give an example for this worst performance

Given list [1,1,1,1,1,1,1,1,1,1,10] our greedy algorithm will grow stacks 1 and 2 until we have value 5 and the last 10 is still to be assigned. This means that 5 units will occur after which the last 10 will be executed to give a time of 15. If we more smartly arranged this, however, we would run all 10 1's in one machine and the 10 in the other such that our runtime is 10. As 15 is $\frac{3}{2}$ 10 we have found this example.

This example is actually quite enlightening. We note that the completion time is:

1. At least as big as the biggest job
2. At least as big as half the sum of the jobs

We note that in our worst case example is when the biggest job is equal in size to the sum of all other jobs. We can formalize this by noting that the different in height between our two stacks as we greedily build them will never be greater than our max item. This is true, because we assign items to the smaller stuck, such that if we are adding to a stack it must be smaller or equal in size to the other stack. Therefore we only produce a new larger stack if $|smallerStack| + |newItem| > |largerStack|$. The largest item we can add is clearly the largest item, such that our gap is maximized when we have even stacks and arbitrarily add our largest item to one of them.

Now that we have shown the bounds of this gap we can show why this produces the worst-case results. This is the case because it inherently produces the longest time period in which one of the stacks is inoperational and we have divided the performance of our total machine by half as only one stack is operational. We can bound this value by consider the case:

We have the worst case where our max $= \frac{1}{2}$ of the sum without the max. We therefore let $y = maxItem$, $x = sumHalf$. We therefore have the cases where our stacks are either $[x, x][y]$ or $[x, y][x]$, meaning that either we calculate the rest of the sum in stack1 and max in the other, or calculate max after calculating each half of the sum concurrently. We can bound $x \leq y \leq 2x$ as if $y < x$ then our worst case scenario would execute in time $< 2x$ which is actually the faster case and therefore not worst case bounds.. If $y > 2x$, then it would dominate x, in that regardless of if we had $[x, x][y]$ or $[x, y][x]$ the runtime would be dominated by y. More formally, we can compare the calculation time of these sum halfs and the largest item to get the ratio:

$$\frac{y + x}{2x} = \frac{> 3x}{2x}$$

such that we spend a majority time executing y so executing it after x only adds runtime $< \frac{1}{2}$ of the total runtime. We find maximizing for our potential worst cases $x \leq y \leq 2x$ we get the largest runtime when $y = 2x$ such that our runtime is

$$\frac{y + x}{2x} = \frac{3x}{2x} = \frac{3}{2}$$

Explicitly, in this case executing concurrenctly $[x, x][y]$ we could finish in time $y = 2x = 2x$ but instead we finish in time $3x$. We have therefore proven our worst case bounds for our greedy algorithm is $\frac{3}{2}$ of our best-case running time.

For the general case with m machines we can show that this same solution strategy holds. Our worst case scenario still includes when all m machines are equally filled but we have our max operation still to compute. This means we must wait $\alpha x + y$ time for completion where x is $\frac{1}{m}$ of the stack excluding the largest item and y is the size of the largest item.We know the gap between stack heights is bounded by y because we places items onto the shortest stack such that a stack can never be shorter by more than y or otherwise we would have added to this stack instead of a now taller one. A new stack height is bounded by $|shortestStack| + |item| > |largestStack|$. To produce a gap larger than y we would have had to have added an item larger than the largest item which is impossible. We therefore maximize our gap when all our stacks are equal such that we get a new largest height from $|shortestStack| = |largestStack| \to |largestStack| + |maxItem|$. This produces the largest period where only 1 machine is operational and we reduce our performance to $\frac{1}{m}$.

The same previous logic holds for m instead of 2 such that we bound $x \leq y \leq (m+(m-1))x$. This is the case because given that we reallocate and assign the largest item to a machine, we can fit in the case of all equal stacks m as this is our tallest item and fill the additional $m-1$ stacks with m x items each to get $m-1$ additional x items. In the worst case where we have all equal stacks except one has the additional y element. To do this, we essentially rotate our x elements such that we have m stacks of $m-1$ x elements. We then have worst case speed of:

$$\frac{y + (m-1)x}{mx} = \frac{mx + (m-1)x}{mx} = \frac{2m-1}{m}$$

We have therefore bounded by the same logic as in the $m = 2$ case that our greedy algorithm is at worst $\frac{2m-1}{m}$ the speed of our ideal implementation.

Problem 3:

We can use a form of dynamic programming to solve this problem. We define variables to keep track of our current interval start and end, as well as a third variable to keep track of our current interval value. We likewise create an equivalent 3 variables to keep track of our current maximum intervals' start, end, and value.

NOTE: as per piazza I am assigning the max interval to be 0 for an all-negative array. However, we could support these by running through the array initially in n time to recording the maximum element and if this is negative return this as the max interval. Equivalently, any negative subinterval will receive value of 0, though this algorithm would work if we judged values ≤ 0 the way we judge 0 here.

We then iterate through our array, at each i setting our current interval to $max(interval_{i-1} + i, 0)$ such that if we ever dip to a negative interval, we know our interval up until this point is at least as good or better before this negative dip or at least as good or better after this dip. In both cases we exclude this negative region which only reduces our interval value. If our current interval' value is larger than our current maximum interval, we update our maximum interval to be the same as the value, start, and end of our current interval.

Time Analysis: We step through each of n array elements, doing at each constant time operations of setting our current interval variables and checking with and potentially modifying our maximum interval variables. As we do n sets of constant-time operations, we have runtime $O(n)$.

Proof of Correctness: We know that our algorithm will find the maximum interval given our finding method as we keep track of our maximum. We must then show that it will indeed find a correct maximum.

- In the all-positive case, our algorithm will simply create a sum of all items and will therefore be correct
- In the all-negative case, our algorithm will assign $max(0, interval_{i-1} + i)$ as 0 in each step as our i's will always be zero. This case is therefore correct.
- In the mixed case, we will continue to analyze an interval through a series of negative and positive integers which does not cause the interval value to dip below zero. It is possible that there will be a max interval that contains a negative value and our algorithm will catch this as the sum will be the largest found in iterating. When the interval value does dip below 0, we want to reset to 0 if this occurs. We would want to skip through each i while $i < 0$ as through this interval we will have a negative interval, after which we reset our interval to $i+1$. This is equivalent to considering this whole interval to have value 0. We can show this skipping is correct, because assuming $interval_{i-1} + i < 0$ where $interval_{x,i-1} > 0$ and $i < 0$, we could take the interval $[x, i-1]$ instead of $[x, i]$ and our interval would be larger. Equivalently given $interval_{i+1,y} > 0$, we could take the interval $[i+1, y]$ instead of $[i, y]$ and we would have an interval bigger by i. In the case where we have an interval $[x, y]$ which includes our i in which we dip below 0, as we have determined $[x, i]$ and $[i, y]$ are negative intervals, we will have either $[x, i] + [i+1, y]$ or $[x, i-1] + [i, y]$, both of which contain a positive interval added with a negative interval. Clearly, this is larger without the negative interval, and we have therefore proved that we will find the maximum interval skipping over these dips below 0 by resetting our sum to 0.

Problem 4:

(a) We design an algorithm adapted from Karatsuba's such that we can multiply two integers using only six multiplications on the smaller parts (instead of nine) given that we are given an integer divided equally into 3 pieces. We represent our two numbers n as $a_0 + b_0 + c_0$ and m as $a_1 + b_1 + c_1$. We then multiply out to get:

$$a_0a_1x^4 + (a_0b_1 + a_1b_0)x^3 + (a_0c_1 + a_1c_0 + b_0b_1)x^2 + (b_0c_1 + b_1c_0)x + c_0c_1$$

We then calculate a_0a_1, b_0b_1, c_0c_1. We note that the coefficients of x^2 are actually unfactored:

$$(a_0 + c_0)(a_1 + c_1) + b_0b_1 = a_0c_1 + a_1c_0 + b_0b_1$$

We have therefore calculated the coefficients of x^2 and can then similarly get the coefficients for x:

$$(b_0 + c_0)(b_1 + c_1) = b_0c_0 + b_0c_1 + b_1c_0 + b_1c_1$$

Since we have b_0b_1 and c_0c_1 we can get these values. We note that we can also calculate x^3 as we have already found a_0b_1 and a_1b_0.

This problem therefore requires only six multiplications:

$$a_0a_1, b_0b_1, c_0c_1, (a_0 + c_0)(a_1 + c_1), (b_0 + c_0)(b_1 + c_1), (b_0 + a_0)(b_1 + a_1)$$

(b) We give an asymptotoic running time of the algorithm. As we do 6 calculations recursively on each subdivision of 3 along with a constant amount of trivial work, we will have a recurrence equation of

$$T(n) = 6T(\frac{n}{3}) + c_n$$

Master's Theorem gives a runtime of $n^{log_3 6}$or $n^{1.63}$ for this problem. Because Master's Theorem would give $n^{log_2 3}$ or $n^{1.58}$, it is faster to only subdivide the numbers into 2 parts.

(c) We now consider only having five multiplications instead of six. If we could do this our recurrence equation would be:

$$T(n) = 5T(\frac{n}{3}) + c_n$$

By Master's Theorem this is $n^{log_3 5}$ or $n^{1.47}$. In this case, we would not want to only subdivide into 2 parts because we would then have $n^{1.585}$.