Buffalo Hird
CS 124
Assignment 4

Problem 1:

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 $|shortestStack| = |largestStack| \rightarrow |largestStack| + |maxItem|$. 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:

Problem 4: