Youssef Naguib <ymn4543@rit.edu>
CS261
Write-up for hw-4 questions 1-4

1. **Correctness argument:**
For this problem I focused on the interval scheduling algorithm we covered in class. It is a greedy o(nlogn) algorithm that first sorts intervals based on end time, and given a starting interval, proceeds to choose the next available compatible jobs. In this problem a two jobs are compatible if job A's end time is less than or equal to job B's start time. Since there are two employers and jobs must alternate, I first sorted the jobs in a single array, and kept track of the start time, end time, and employer for each. The algorithm is told if the first job should come from employer 1 or 2. An optimal output representing the most jobs possible to take given that they alternate employers is guaranteed due to the condition that each next job must come from a different employer and must have a compatible start time. Since the jobs are sorted by end time, this will produce the best result. I run the algorithm twice, once with the condition that the first job comes from employer 1, and vice versa. The max value from these two runs is the final answer. An optimal solution is guaranteed.

Running Time Estimate:
1. Merge Sort: O(nlogn)
2. Choose_Jobs: O(n)
3. Choose_Jobs: O(n)
Total complexity = O(nlogn) + O(n) + O(n)
= O(nlogn)

2. **Correctness argument:**
Part A) This is a greedy algorithm. To partition the array into subarrays with even sums, you must keep track of the number of odds. If there is an even number of odds, the answer is doubled and a 1 is added. If the total sum of the array is odd, the answer will always be 0. This is a valid algorithm because any even number of odd numbers will add up to an even number. It is a question of how many ways there are to select odd numbers so that they add up to an even sum, so in a way the even numbers can be ignored, since they are guaranteed to be valid.
Part B) This algorithm was a lot harder for me to grasp, and took multiple efforts, but in the end it was not so complicated. I start with a dynamic array, where each element i represents the number of possible odd sum partitions in the element array between index 0 and i. I then iterate through the array. If the element is even, it can only be attached to the final subsequence, so the value in the dynamic array will be s[i-1]. If the value is odd however, I set the value equal to s[i-1] and proceed to iterate back through the array, adding the elements backwards until I reach the start of the array. At each interval I check the sum, if it is an odd sum, it's index-1 in the dynamic array is added to the total, as this represents the number of ways to

partition the array if the final element was included in this particular subsequence. This algorithm is certainly more costly than the first and will likely take long for large outputs, but in theory is correct because it checks each possible subarray along with its sum.

Heart of the Algorithm:

S = dynamic array
S[i]: The total number of ways the array can be divided into consecutive odd sum partitions between index 0 and i.
Recurrence: S[j] = S[j-1] (if array[j] is even), S[j-1] + sum(all S[k-1] where k is an odd sum subsequence that ends at j and starks at k), if array[j] is odd.
Solution: S[n]

Running Time Estimate:

Even partitions:

1. Iterate through array: O(n)
2. For each element, do a constant amount of work: O(1)

Total = O(n)

Odd partitions:

1. Iterate through array to count number of odd and even elements: O(n)
2. Iterate through element array again to fill dynamic array. If the element is odd, An additional amount of linear work is done. O(n^2)

Total = O(n^2)

3.

Correctness argument:

This problem seemed trivial to me, and built upon the longest increasing subsequence problem from class. The algorithm is almost identical except for the fact that it takes an extra factor into account, the sum. The algorithm iterates through each element in the array and does the finds the maximum sum of an increasing subsequence that begins at that element, and adds it to a list. After this is computed, the answer is simply the maximum value in the list.

Running Time Estimate:

1. Initialize empty dynamic array: O(1)
2. For each element in array, check all other elements in array to find best possibility for that interval: O(n^2)
3. Add sums to an array and find the max: O(n)

Total = O(n^2) + O(n) + O(1)
= O(n^2)

4.

Correctness argument:

This problem was certainly one of the more challenging problems I've faced in the course, but upon completion, is not very difficult to understand. I used the sequence alignment algorithm from class to come up with my solution for the headache problem. I make 2 strings out of the lines, stringm for line 1 and stringn for line 2, and create an m x n dynamic array. I begin to fill in the base cases for the first row and the first column, for each choice there are only 2 options, include one and add 4, or include 3 and add 3 +appropriate cost of pairing. I then continue on to fill in the base cases for the second row and column, which are similar except this time there are 4 choices: take one from each line, take two from line, take one from other line, take one from line. Finally, all other cases can run based on 5 choices: one from each line, one from line 2, two from line 2, one from line1, or two from line 1. The minimum cost is always chose, and an optimal output is guaranteed.

Heart of the Algorithm:

S = dynamic array

S[i][j]: The minimum headache cost to allow employees from line 1[0,..i] and line 2[0,..j] on the ride.

Recurrence:
```
S[i][j] = min(s[i-1][j-1] + alpha(stringn[j-1],stringm[i-1]),  # take one from each line
          alpha(stringm[i-1],stringm[i-2])+s[i-2][j]+3,        # take two from line m
          alpha(stringn[j-1],stringn[j-2])+s[i][j-2]+3,        # take two from line n
          4 + s[i-1][j],                                       #  take one from line m
          4 + s[i][j-1])                                       # take one from line n
```

Alpha is a function that find the cost of the pairing. (E + N = 5, anything else = 0).

Solution: S[m][n] where m = length of line 1 and n = length of line 2

Running Time Estimate:

1. Initialize empty M x N dynamic array: O(1)
2. Fill in base cases of dynamic array: O(n)
3. Fill in rest of M x N dynamic array based on base cases: O(n) * O(m) = O(mn)

Total = O(mn) + O(n) + O(1)

= O(mn)