Jeremy Cristobal

604982952

CS180 Final

Discussion 1B

Problem 1 Algorithm

Anything following '//' and surrounded by () are comments that try to help explain the given line

n = number of requests
m = number of months the company can work
a = pay for each programmer
R = {R_1, R_2, … , R_n} // (Set of all requests)
P = {p_1, p_2, … , p_n} // (Set of all profits)
M = {m_1, m_2, … , m_n} // (Set of months it takes for each request)
H = empty set // (Set of months it takes for each request if **H**ired)
for (x = 1 to n) {
    h = (m_x / 2)
    Add h to the end of set H
} // (H now equals {h_1, h_2, … , h_n}
A = 2D array that is (n + 1) by (m + 1) // (Used to keep track for non-hired)
B = 2D array that is (n + 1) by (m + 1) // (Used to keep track for hired)
for (w = 0 to m) {
    A[0, w] = 0
    B[0, w] = 0
}

for (i = 1 to n) {
    for (w = 1 to m) {
        if (m_i > w) {
            A[i, w] = A[i - 1, w]
        }
        else {
            A[i, w] = max{ A[i - 1, w] , (p_i + A[i - 1, w - m_i]) }
        }
        if (h_i > w) {
            B[i, w] = B[i - 1, w]
        }
        else {
            B[i, w] = max{ B[i - 1, w] , (p_i + B[i - 1, w - h_i]) }
        }
    }
}

```
c = A[n, m]                                  // (Maximum value for non-hired)
d = B[n, m] - (10 * a * m)                   // (Maximum value for hired)
Y = empty set                                // (Will copy max of hired or non-hired)
Z = empty 2D array that is (n + 1) by (m + 1)      // (Will copy max of hired or non-hired)
S = empty set        // (Will hold the solution: whether to hire or not and which jobs to do)

if (c > d) {
        Y = M                    // (Y now is the set of months it takes for non-hired)
        Z = A                    // (Z now is the 2D array for non-hired)
        add “Don’t Hire” to set S
}
else {
        Y = H                    // (Y is now the set of months it takes for hired)
        Z = B                    // (Z is now the 2D array for hired)
        add “Hire” to set S
}
                                 // (Y now equals {y_1, y_2, … , y_n})
while (n > 0 && m > 0) {
        if (Z[n, m] == Z[n - 1, m]) {        // (If highest value is equal to one above)
                n = n - 1
        }
        else {
                add R_n to set S
                m = m - y_n
                n = n - 1
        }
}

Return S
```

Problem 1 Explanation/ Complexity

This problem is basically the knapsack problem but instead of having a limit to the capacity of the knapsack, the limit is the amount of months available to work (m). So, I used the knapsack algorithm as the basic outline for the first page of my algorithm.

However, this problem also asks you to choose whether or not to hire a group of programmers, so I created an additional set (H) that contained how long it would take for each job if the programmers were hired, and divided each element of M by 2 in order to calculate that. Similarly, I created an additional 2D array (B) specifically for the situation in which the programmers would be hired. So, while the nested loops are running and the knapsack algorithm is working as expected, two 2D arrays (A and B) will be filled up instead of just one.

After this portion, we need to decide whether or not to hire the programmers. Because the programmers each get paid $a, there are 10 programmers, and they all work for m months, we subtract (10 * a * m) from the max profit we calculated for the 'hire programmers array' (B). We then compare this value to the max value for the non-hired array, and we choose whichever is higher.

If it is more profitable to hire them, we will set our new sets equal to B and H (our hired arrays), and if not we will set our new sets equal to A and M (our non-hired arrays). Because sets can hold any values, we will also make a new set S and set its first value to "Hire" or "Don't Hire" depending on our calculations.

Then, we use the 2D array that we chose and go to the bottom right (the max value). If that value is equal to the one above it, then the corresponding job in set R is not included in the max, so we move up one row. If it is not equal to the one above it, then the corresponding job in set R IS included in the max. So, we add it to set S, move up one row, and move left the amount of months that the selected job will take. This will work for both the hired and non-hired sets because we created different sets at the beginning and have already chosen which to use.

Once n or m becomes 0, we know that we have found all the jobs for what we calculated to be our max profit, so we leave the loop. We then return S, which contains "Hire" or "Don't Hire" at the start, followed by all the jobs the software company should take.

The most complex part of my algorithm is the nested for loop that I used to fill in both of my 2D arrays. Because the outer goes from 1 to n and the inner goes from 1 to m, the complexity of my algorithm is:

O(nm)

Problem 2 Algorithm

Anything following '//' and surrounded by () are comments that try to help explain the given line

```
n = number of processing jobs
J = {J1, J2, … , Jn}                    // (Set of all processing jobs)
q = {q1, q2, … , qn}                    // (Set of desktop time for each job)
t = {t1, t2, … , tn}                    // (Set of specialized time for each job)
A = empty set                  // (Will be used to store (desktop + spec. time, job) )
B = empty set                           // (Will be used to store MergeSort values)
C = empty set                           // (Will be used to temporarily store values from A)

for (i = 1 to n) {
        v = qi + ti
        Add (v, Ji) to end of set A
}                                       // (A now equals {(v1, J1), (v2, J2) , … , (vn, Jn)})
                        // (A can also be read as {a1, … , an}; a1 = (v1, J1) )

MergeSort( l = left barrier, r = right barrier ) {          // (Both l and r are integers)
        if (l == r) {                           // (When we are looking at only one element)
                return
        }

        m = lower bound of { (l + r) / 2 }          // (Find middle)
        MergeSort (l, m)                            // (Use MergeSort on left half)
        MergeSort (m + 1, r)                        // (Use MergeSort on right half)
        Merge (l, m ,r)

        for (i = l to r) {              // (Copy sorted values from set C into set A)
                ai = ci
        }
}
```

```
Merge( l = left barrier, m = middle, r = right barrier ) {          // (All are integers)
        x = l                                   // (Will be used to go through left side)
        y = m + 1                               // (Will be used to go through right side)
        z = l                                   // (Will be used to keep track of set C)

        while (x ≤ m && y ≤ r) {
                if (v_x > v_y) {
                        c_z = a_x
                        x = x + 1
                }
                else {
                        c_z = a_x
                        y = y + 1
                }
                z = z + 1
        }
        while (x ≤ m) {                         // (Fill in remaining left side if all right used)
                c_z = a_x
                x = x + 1
                z = z + 1
        }
        while (y ≤ r) {                         // (Fill in remaining right side if all left used)
                c_z = a_y
                y = y + 1
                z = z + 1
        }
}

MergeSort (1, n)                                // (Use MergeSort on all elements of set)
S = empty set                                   // (Will be used to deliver the solution)
for (k = 1 to n) {
        Add J value of a_k to end of set S
}                                               // (Now that list is sorted, extract ordered Jobs)
Return S
```

Problem 2 Explanation/ Complexity

My solution to this problem stems almost entirely from the fact that there is only one supercomputer that every job needs to go through. Because there exist more than enough desktops/ specialized computers, the biggest limitation is the supercomputer. The time it takes for every computer to finish its turn with the supercomputer is constant no matter the order they go through it, assuming that the supercomputer is used by the next job immediately when it is available. As a result, the time we need to minimize is the time after the supercomputer has completed all its jobs.

Because we want to minimize the time after and the time for the supercomputer to finish will always be the same, we need to find the jobs with the shortest total time not including its supercomputer aspect. The job with the shortest desktop + specialized computer time should be the last to go through the supercomputer, as it ensures that it will give the shortest time if it is the last to work on its final two phases and it allows the longer jobs to go earlier and maximize time.

Thus, the solution here is obvious: sort the Jobs in order of nonincreasing desktop + specialized computer time. The first step is to add the two times for each job, so I stored this value in a set of ordered pairs with its respective job.

I then used a version of a MergeSort algorithm in order to sort this new set. Instead of putting the smallest value first, I sorted the set so that the ordered pair with the highest added time would be first. I chose MergeSort because of its worst-case time complexity compared to other sorting algorithms.

After we have achieved this sorted set, we extract the Jobs themselves in order from the ordered pair of (desktop + specialized, Job) and put that into a new set called S. This set is then returned, and the result is the order in which the jobs should go through the supercomputer.

The most complex part of my algorithm is the MergeSort function itself, which relies on recursion. As a result, the complexity of my algorithm is:

O(n*log(n))

Problem 3 Algorithm

Anything following '//' and surrounded by () are comments that try to help explain the given line

n = number of companies
G = set of all companies // (I'm assuming this is a random assortment with no order)
A = empty set //(Will hold all companies; consecutive numbers are competing)
V = empty set //(Will hold value of each corresponding company)

Pick any element of G, call it g
while (G is not empty) {
    Add g to end of set A
    Add value of company g to set V
    h = competitor of g in set G
    Remove g from set G
    g = h
} // (A now equals $\{a_1, a_2, \dots, a_n\}$; V now equals $\{v_1, v_2, \dots, v_n\}$)

B = $\{v_1\}$ // (Will hold values of companies 1 through n-1)
C = empty set // (Will hold values of companies 2 through n)
for (i = 2 to (n-1)) {
    Add v_i to end of set B
    Add v_i to end of set C
}
Add v_n to end of set C
// (C = $\{c_1, \dots c_n\}$; B = $\{b_1, \dots, b_n\}$)
P = B // (Will use these sets to store values of B/C added together)
Q = C
$p_0 = 0$ // (Sets P and Q now start with 0 and have an extra element)
$q_0 = 0$ // (P = $\{p_0, p_1, \dots, p_n\}$; Q = $\{q_0, q_1, \dots, q_n\}$)

for (j = 3 to (n - 1)) { // (Start at 3 because one and two are incompatible)
    P[j] = (max{ P[j - 2] , P[j - 3] } + B[j])
    Q[j] = (max{ Q[j - 2] , Q[j - 3] } + C[j])
} // (Add value to either 2 or 3 before)

```
w = 0                   // (Will use to find max value)
x = 0                   // (Will use to find max value)
for (k = 3 to (n - 1) {
        w = max{ P[k], Q[k] }
        x = max{w, x}
}                       // (We now have max value out of both sets)

first = true            // (Will change whether max value is in first or second set)
Y = empty set           // (Will be set to either B or C depending on max value)
Z = empty set           // (Will be set to either P or Q depending on max value)
S = empty set           // (Will be used to hold the companies in our solution)
d = 0                   // (Will be used to store where max value is)

for (k = (n - 1) to 3) {
        d = k
        if (P[k] == x) {
                first = true
                Y = B
                Z = P
                exit loop
        }
        if (Q[k] == x) {
                first = false
                Y = C
                Z = Q
                exit loop
        }
}               // (We now know which set our max value is in)
```

```
while (d > 2) {          // (1 or 2 is starting value, anything higher has had addition)
        if (first == true) {
                Add A[d] to set S
        }
        else {
                Add A[d + 1] to set S
        }                       // (We need the +1 here because this set goes from a2 to an)

        if (Z[d] - Y[d] == Z[d - 2]) {
                d = d - 2
        }
        else {
                d = d - 3
        }               // (This works because we only ever increment by 2 or 3)
}

if (first == true) {
        Add A[d] to set S
}
else {
        Add A[d + 1] to set S
}       // (We do this once more b/c we did not add our final element in above loop)

Return S
```

Problem 3 Explanation/ Complexity

This was a very interesting problem to solve. It seems similar to a weighted interval scheduling problem but with some key twists. The first hurdle was thus figuring out how to deal with the problem now that it is cyclic, unlike the weighted interval scheduling problem.

After thinking for a while, my solution was to first order the companies such that each company is surrounded on either side by its competitor, for example the competitors of a_3 are a_2 and a_4. Also, a_1 and a_n are competitors of each other. As such, a_1 and a_n will never be allowed to be in the same solution set, so my idea was to create two different arrays to iterate through (B and C). The first contained v_1 through v_{n-1} and the second contained v_2 through v_n. This avoids allowing the edges to be in the same set and gives us a situation we are familiar solving through the weighted interval scheduling problem without having to worry about a cycle.

So, after creating these two sets, we create two more sets that are set equal to it (P and Q). We will use these two sets in order to store values and avoid having to calculate the same problem multiple times. Additionally, the added value will always be from 2 or 3 below because 1 below is a competitor and 4 below could only be increased by adding 2 below. We give these two new sets (P, Q) the value 0 at the 0 index in order to avoid looking for a nonexistent value.

After we fill in these two sets, we go back through the two sets to find our highest calculated value, to find which set our highest value is in, and where in that set it is located. We change some values depending on whether it is in our first or second set, then continue on.

From there, we add the company at the location we found to our solution set S. If we are working with the second set, we must add 1 to our variable value because they are all offset (for example, b_1 corresponds to a_1 but c_1 corresponds to a_2). Then, we subtract the value of said company from our total value and see if it is 2 or three below and change our variable value accordingly. Our loop leaves 1 iteration early in order to avoid checking for a nonexistent value, so we add our final element to the solution set S. We then return S, which is the set of companies that satisfy MPP.

Though the algorithm may seem lengthy and at times messy, it was done in order to optimize the time complexity. The most complex part of my algorithm is the first while loop, so the complexity of my algorithm is:

O(n)