CS141, Fall 2021, Assignment #2 **Due: 11:59pm, Wed 10/20, 2021**
Name: Rishab Dudhia **Student ID: 862141444**

1 (18pts) Kayaking Again

1. With the given six students' weights as: 1, 3, 5, 7, 8, and 9, and the total weight per kayak being $w = 10$, the following solution with the given algorithm would be:

   ```
   (9,1)
   (8)
   (7,3)
   (5)
   ```

   We can see that starting at the end of the sorted weights, the first chosen student would be of weight 9. Then, the only student who can fit in the kayak to make a total weight of 10 is the lightest student of weight 1. Now, there are 4 students left with the next student to be seated being of weight 8. No other student can fit in the kayak with this student since the lightest student remaining is of weight 3, so the student of weight 8 will have a kayak of their own. The next heaviest student is of weight 7 who can fit in the kayak with the lightest remaining student of weight 3. This leaves the final student of weight 5 to have a kayak of his own. Finally, we have sat all the students down in 4 kayaks.

2. Claim: The optimal solution after making the choice of students (A, B) is (A, B) + the optimal solution of choosing from the rest of the $n - 2$ students.
   To prove this claim, we will follow by contradiction. Therefore, let's assume that the optimal solution after making the choice (A, B) is (A, B) + a different solution (a solution where you do not choose the heaviest remaining student that can be paired with the current student). For example, from the above problem, let $A = 9$ and $B = 1$. Then, choosing 8 to be alone will be the only option since all remaining students would cause the pair to be overweight. Finally, we are left with three students of weights: 3, 5, 7. If we are following our assumption, we do not make the optimal decision from above (which would be pairing students of weights 7 and 3) and instead must put the student of weight 7 in a kayak alone since the only other student is of weight 5 which would make the kayak overweight. This leaves the last two students whose weights combine to be less than the total maximum weight per kayak, but our assumption says that we do not make this "greedy decision" and instead use a different solution, so we would have to put each of these students in different kayaks. In total, using this different solution after pairing students of weights (9, 1), we sit the students down in 5 total kayaks which is not an optimal solution. In general, after selecting students (A,B), placing the rest of the students in kayaks not using the Kayaking Algorithm may yield in extra kayaks being needed. Therefore, the assumption made holds false and the optimal solution after making the choice of students (A, B) is (A, B) + the optimal solution (making the *greedy choice*) of choosing from the rest of the $n - 2$ students.

3. Now we will prove that the "greedy choice" mentioned in part (2) will always be the optimized choice. To prove this choice, we have to look at student i with weight a_i. Then, from the Kayaking Algorithm, i will be paired with student j where j is another student or $j = -1$ which means i will be alone in a kayak. From the algorithm, we know that j will be the heaviest available student that can sit in a kayak with student i while $a_i + a_j \leq w$.
   Now, let's assume that instead of picking student j (the heaviest available student to be paired with i without exceeding the weight limit), we pick student k (who must weigh less than student j since j was the heaviest possible that could be paired with i) to be paired with i. We know that picking k will work because $a_k < a_j$ and the pair (i, j) was valid, so lowering the second value will still have a total weight below the valid weight total. Since student k is partnered with student i now, there are

two possible scenarios for student j. Student j can either be in a kayak alone, or he can be in another valid pair with another student, let's call this student r.
Case 1:
Since student j is paired with no one, the current situation looks as such:

```
(i, k)      (j)
```

We have shown each kayak as a pair of parentheses such that there can be one or two elements between the parentheses signifying whether there are one or two students in the kayak. Above, the first kayak has students i and k and the second kayak has just student j. Earlier, we stated that student j was the heaviest possible student to be paired with student i, and any one student is allowed to be in a kayak alone. We can, therefore, swap students j and k, so that the kayaks now contain:

```
(i, j)      (k)
```

This is a valid swap since the weight limit w is not exceeded by either kayak and both kayaks have at least one student in them.
Case 2:
In case 2, we must consider that instead of student j being placed in a kayak alone, he is paired with another student, student r. The situation would then look as:

```
(i, k)      (j, r)
```

We know that neither kayak above is exceeding the weight limit from our assumptions stated earlier. Therefore, $a_i + a_k \leq w$ and $a_j + a_r \leq w$. We also stated earlier that student k weighs less than student j since student j was the heaviest possible student that could be paired with student i, and for k to be paired with i he must weigh less than j so that the weight limit is not exceeded. In other words,

$$a_k < a_j$$

Since we know that student j was the heaviest allowed student to be paired with student i, we can swap students k and j in their respective kayaks. We know we can swap these two students since we know that the first kayak will contain students i and j and we know that j is the heaviest student that can be paired with i without exceeding the weight limit. For the second kayak, we know that the total weight must be decreasing since $a_k < a_j$ and if the combination of (j, r) is valid, then the combination of (k, r) must also be valid. Finally, the situation would look as:

```
(i, j)      (k, r)
```

We now know that whether the situation is of Case 1 or Case 2, student i can always be paired with student j who is the heaviest student that can be paired with student i.

4. Making a certain decision (the "greedy choice") per sub-problem will yield an optimal solution after all sub-problems have been have been solved by making that greedy choice. To show this, we first must prove that making the greedy choice is always valid. For the Kayaking Algorithm, in part (3) we proved that pairing student i with the heaviest student that can sit in a kayak with i is always allowed. Then, we must show that for every sub-problem, the greedy choice will yield the optimal solution. In part (2), we proved that the Kayaking Algorithm does have optimal substructure, so the greedy choice for each problem will result in the optimal solution. From parts (3) and (2), we can confidently say that the Kayaking Algorithm will yield an optimal solution.

5. (Bonus)

2 (20 pts) Missed All the Deadlines!

1. The optimal solution of the problem would be 23 days late. We find this answer by sorting the assignments from lowest time taken for assignment to highest time taken. Then we choose to do the assignment that requires the least amount of time. Therefore, we would do the assignments in the order: 3, 4, 6. When we do this, the amount of late days is calculated as:

$$(3) + (3 + 4) + ((3 + 4) + 6) = 3 + 7 + 13 = 23 \tag{1}$$

2. A greedy algorithm that will help us lose the least amount of points is to sort the times required per assignment, t_i, from least to greatest. Then choosing to do the assignment that requires the least amount of time while completing all n assignments will result in losing the fewest amount of points. We will have three variables called currLate, pastTotal, and totalLateDays that will all be initialized to 0. The first variable will calculate the value of the current "term" to be added to the totalLateDays variable. The currLate value is calculated with the values that were added before. These values will be remembered via the pastTotal variable which will remember the sum of the previous added assignments. For example, when element three is read in the for loop from the problem above, pastTotal will have the value of 7 since the previous two necessary times were three and four. Then, currLate can be calculated as (pastTotal = 7) + (current element value = 6). 13 is the result and is saved in currLate which is then added to the totalLateDays variable. Once all elements in the array have been read in the for loop, the value in totalLateDays is returned as the minimum points that can be lost.

```
int minLate(int* timeNeeded, int size) {
    sort the array of times needed for assignments from least to greatest;
    currLate = 0;
    totalLateDays = 0;
    pastTotal = 0;

    for all elements of timeNeeded starting at beginning{
        currLate = pastTotal + current element's time;
        pastTotal = pastTotal + current element's time;
        totalLateDays = totalLateDays + currLate;
    }

    return totalLateDays;
}
```

3. To show that this algorithm will yield the least amount of points lost, we can go about the proof in two ways. Like the last problem, we can prove that choosing to do the assignment that will take the least amount of time is an appropriate greedy choice and then prove that the algorithm has an optimal substructure. Instead, we will take an approach that may be easier to understand, visually. To prove that choosing the assignment with the smallest t_i will provide an optimal solution, we will continue by contradiction. Let's assume, instead of choosing the assignment with the smallest t_i, we choose a different assignment. Going with the example above, instead of initially choosing the assignment with $t_i = 3$, let us choose either of the remaining assignments. For the second choice let's choose the assignment with $t_i = 3$. Let's assume that the order we pick the three assignments is: 4, 3, 6. The total lost points would then be calculated as:

$$(4) + (4 + 3) + ((4 + 3) + 6) = 4 + 7 + 13 = 24 \tag{2}$$

We know that this is not the optimal solution as we earlier stated that the optimal solution was 23 missed points.

Now, let's take a more holistic view at what we have just demonstrated. We know that we have a total of n assignments, and let's set another variable, j, which will represent which pick we are making. Initially, $j = 1$ since we will be making the first pick, and after a pick is made, j is incremented by 1. Then, after the first choice is made, j would be equal to two since we will be making the second pick. In lines (1) and (2), we can see that when an assignment is picked earlier, the time taken to complete that assignment is added to the total late points more times than the times required for assignments picked later. In line (2) above, we can see that since we picked assignment with $t_i = 4$ first, it is added to the total missed points three times. Then, the assignment with $t_i = 3$ will be added to the total two times since it is the second choice. However, in line (1) above, we picked the assignment with $t_i = 3$ first, so that assignment's time was added to the total late points three times. Then the following is obvious,

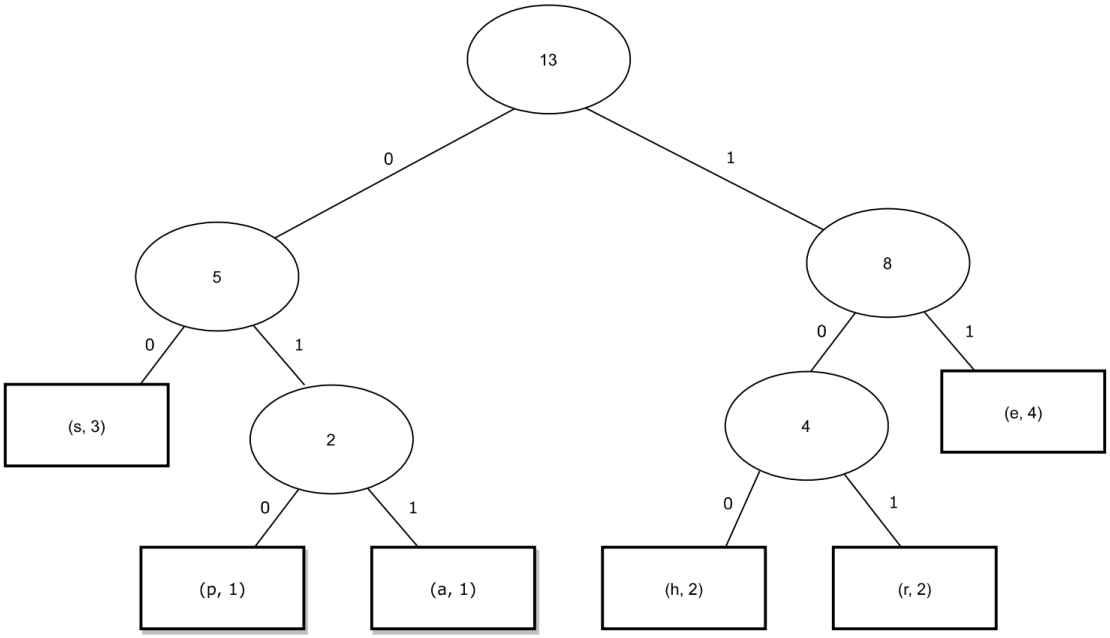
$$3 \cdot 3 < 3 \cdot 4$$

Therefore, to minimize the total late days, one would rather count the assignment that took three days three times (adding 9 days) towards their late total instead of adding three times the assignment that took four days (adding 12 days). In general, when an assignment is picked with pick j, it will be counted towards the total late days $n - (j - 1)$ times. For example, when the assignment with $t_i = 4$ is picked first, it will be added $3 - (1 - 1)$ times to the total late points, or, simply, three times. To minimize the total amount of lost points, one would want to choose to count multiples of an assignment that takes a lower amount of time as opposed to choosing to count multiples of an assignment that take more time. In other words, the assignment that takes the least amount of time should be counted n times, the second least time consuming assignment will be counted $n-1$ times, ... , until the most time consuming assignment is added only once. If the order is changed at all, that means an assignment that is more time consuming is done before an assignment that is less time consuming which means that the larger number will be added more times and the smaller number will be added less times resulting in a larger number of points lost. Therefore, always choosing the assignment with the lowest t_i will result in an optimal solution because we ensure that our calculation eliminates accounting for extraneous late days from doing longer assignments when shorter assignments are still needing to be done.

3 (12pts) Fun with Huffman Code

1. Prefix code is a way of encoding a message so that there is no ambiguity when decoding the message. Prefix code states that no encoded value can be a prefix of another encoded value. This completely avoids any ambiguity when decoding because once a sequence of values combine to be an encoded value, we can look at its respective decoded value and be confident that the result is the decoded value for the sequence. For example, let's say that this was not ensured by the encoding. Then, the character o could be encoded as 00 and the character p could be encoded as 001. When trying to decode a random string, let's use 00100, the first two values read 00, so can we say that these first two values represent o? We cannot say this because the code for o is a prefix for the code for p. Surprisingly enough, if we read one more character of the encoded message, we have an encoded value of 001. This is the value we set for p, so there is ambiguity because we do not know if we should read the next encoded bit to try to decode the message. If we wanted to convert our code to a prefix code, we would have to make sure that no encoded value is a prefix in a different encoded value. If this is ensured, there is complete confidence in decoding a message.

2. The word **sheepshearers** contains 6 different characters with respective frequencies: (s,3), (h, 2), (e, 4), (p, 1), (a, 1), (r, 2). To encode the word using Huffman Code, we must construct a trie based on frequencies of the characters. We must pick characters p and a as the leaves with the most depth because they have the lowest frequencies and build up from there. Then, the trie, would further be constructed as:



Now we can say that each character can be encoded as:

```
e = 11
s = 00
r = 101
h = 100
p = 010
a = 011
```

Then, the word **sheepshearers** would be encoded as:

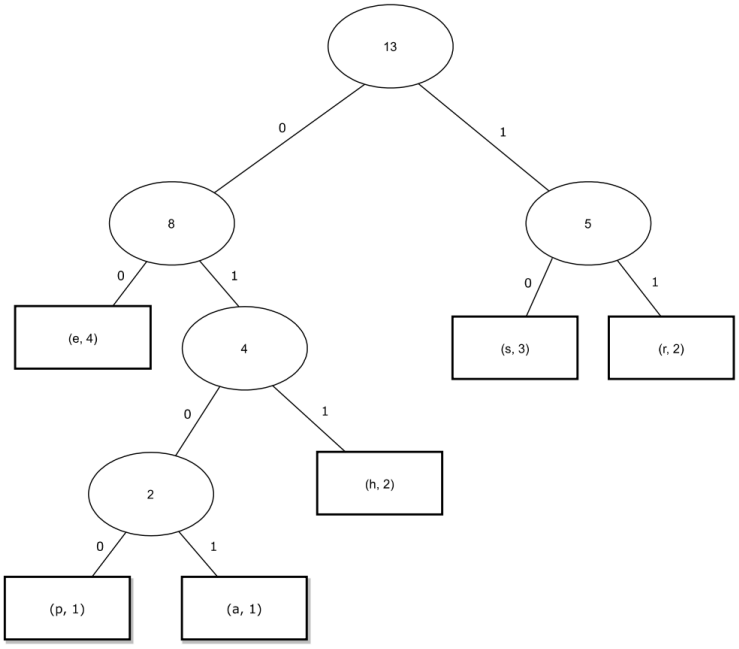
```
00  100  11  11  010  00  100  11  011  101  11  101  00
s    h   e   e   p   s   h   e   a   r   e   r   s
```

We can then see that the total length of the encoded message

0010011110100010011011101111010 0

is 32 bits.

3. My encoded codeword is not the only valid Huffman Code of the word. Another encoding of the word can be created with the following trie with the root node value being equal to our previous root node value in part (2):



From this new trie, we can state the following encoded values:

```
e = 00
s = 10
r = 11
h = 011
p = 0100
a = 0101
```

Then, the word **sheepshearers** would be encoded as:

```
10  011  00  00  0100  10  011  00  0101  11  00  11  10
s    h   e   e    p    s   h    e   a     r   e   r   s
```

Now with different values for the characters, we have a new final codeword. The length of this new codeword

10111100111001011110111011100110 10

is still 32 bits as it is still a Huffman Code of the word so it uses the minimum amount of bits required to encode the word.

4. (Bonus)