Warm-up

Problem 1. Construct the Huffman tree of the following phrase: "data structure"

Solution 1. We first determine the frequency of each character in the phrase (including the space):

character	frequency
a	2
c	1
d	1
e	1
r	2
s	1
t	3
u	2
space	1

After creating a single-node tree for each of these, the Huffman tree algorithm repeatedly merges the two trees with smallest total frequency. Assuming ties are broken lexicographically, it performs the following merges:

1. *c* and *d* form a new tree *cd* with frequency 2
2. *e* and *s* form a new tree *es* with frequency 2
3. space and *a* form a new tree *a*_ with frequency 3
4. *cd* and *es* form a new tree *cdes* with frequency 4
5. *r* and *u* form a new tree *ru* with frequency 4
6. *a*_ and *t* form a new tree *at*_ with frequency 6
7. *cdes* and *ru* form a new tree *cdersu* with frequency 8
8. and finally *cdersu* and *at*_ form a new tree with frequency 14

The resulting tree can be found below:

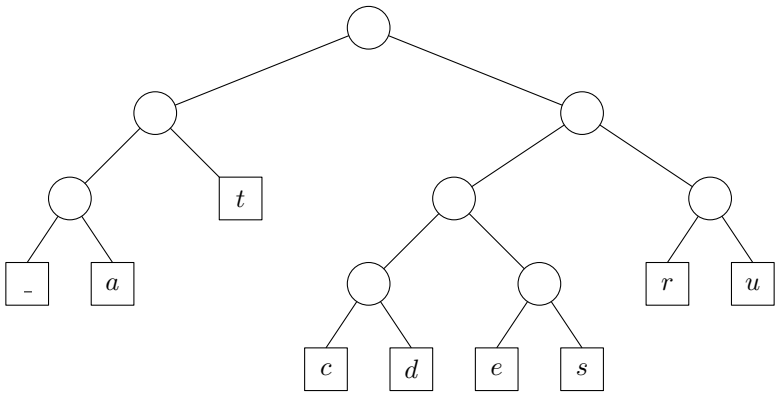

Figure 1: The resulting Huffman tree.

Problem 2. During the lecture we saw that the Fractional Knapsack algorithm's correctness depends on which greedy choice we make. We saw that there are instances where always picking the "highest benefit" or always picking the "smallest weight" doesn't give the optimal solution.

For each of these two strategies, give an infinite class of instances where the algorithm gives the optimal solution, thus showing that *you can't show correctness of an algorithm by using a finite number of example instances*.

Solution 2. The trivial solution that works for both strategies is any instance consisting of a single item with weight at most W (or any instance where the sum of the weights of all items is at most W).

A less trivial solution where not all items fit in the knapsack could for example be the following. For the "highest benefit" strategy, we construct the following class of instances: We have n items (numbered 0 to $n-1$) and item i has benefit $b_i = c \cdot 2^i$ (for and $c > 0$) and weight $w_i = 2^i$. For any integer W, picking the highest benefit items that fit in the knapsack corresponds to picking those i that are 1 in the binary representation of W, filling the knapsack entirely. The solution will have benefit $c \cdot W$.

For the "smallest weight" strategy, we can take the same items when we restrict the values of W. Specifically, by picking W such that $W = \sum_{j=0}^{k} 2^j$ for some positive integer k, we ensure that the k least significant bits of the binary representation of W are all ones and all others are zeroes. The "smallest weight" strategy picks the items starting from the smallest weight (i.e., the least significant bit) and solution will have benefit $c \cdot W$.

Problem solving

Problem 3. Given vectors $a, b \in R^n$, consider the problem of finding a permutation a' and b' of a and b respectively minimizing their total difference

$$\sum_{1 \le i \le n} |a'_i - b'_i|.$$

Prove or disprove the correctness (i.e., that it always returns the optimal solution) of the following algorithm that greedily tries to pair up similar coordinates.

```
1:  function GREEDY-MATCHING-V1(a, b)
2:      A ← multiset of values in a
3:      B ← multiset of values in b
4:      a' ← new empty list
5:      b' ← new empty list
6:      while A is not empty do
7:          x, y ← a pair in (A, B) minimizing |x − y|
8:          Append x to a'
9:          Append y to b'
10:         Remove x from A and y from B
11:     return (a', b')
```

Solution 3. The algorithm does not always return the optimal solution. Consider the vectors $a = (1, 4)$ and $b = (3, 6)$. Since 3 and 4 is the pair that minimizes their difference, the algorithm ends up with the following permutations: $a' = (4, 1)$ and $b' = (3, 6)$. These permutations have total difference of 6, where the optimal solution $a'' = (1, 4)$ and $b'' = (3, 6)$ has total difference 4.

Problem 4. Given vectors $a, b \in R^n$, consider the problem of finding a permutation a' and b' of a and b respectively that minimizes

$$\sum_{1 \leq i \leq n} |a'_i - b'_i|.$$

Prove or disprove the correctness (i.e., that it always returns the optimal solution) of the following algorithm that greedily tries to pair up similar coordinates.

```
1:  function GREEDY-MATCHING-V2(a, b)
2:      A ← multiset of values in a
3:      B ← multiset of values in b
4:      a' ← new empty list
5:      b' ← new empty list
6:      while A is not empty do
7:          x ← smallest in A
8:          y ← smallest in B
9:          Append x to a'
10:         Append y to b'
11:         Remove x from A
12:         Remove y from B
13:     return (a', b')
```

Solution 4. Note that the algorithm sorts the coordinates of a and b in non-decreasing order. In this case, the algorithm is optimal.

Let a^* and b^* be the optimal solution. We use an exchange argument. First note that we can assume that a^* is listed in non-decreasing order, because applying the same permutation to both a^* and b^* does not change the value of the objective. If b^* is not in sorted order, there must be an inversion, i.e., an index i such that $b^*_i > b^*_{i+1}$. The contribution of the indices i and $i + 1$ to the objective is

$$|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}|$$

since $a^*_i \leq a^*_{i+1}$, a case analysis argument (which is a bit involved, so we defer this to the end of the proof) can be used to show that

$$|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}| \geq |a^*_i - b^*_{i+1}| + |a^*_{i+1} - b^*_i|,$$

therefore, by swapping the entries b^*_i and b^*_{i+1} we have one fewer inversion without increasing the objective value.

We can keep on performing this exchange argument until b^* is sorted. Thus, our algorithm is optimal.

It remains to prove that $|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}| \geq |a^*_i - b^*_{i+1}| + |a^*_{i+1} - b^*_i|$. This is equivalent to proving that $|a^*_i - b^*_i| + |a^*_{i+1} - b^*_{i+1}| - (|a^*_i - b^*_{i+1}| + |a^*_{i+1} - b^*_i|) \geq 0$. Recall that we know that $a^*_i \leq a^*_{i+1}$ and $b^*_i > b^*_{i+1}$.

- Case 1: $a_i^* \geq b_i^*$ and $a_{i+1}^* \geq b_{i+1}^*$, which implies $b_{i+1}^* < b_i^* \leq a_i^* \leq a_{i+1}^*$. In this case $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| - (|a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|)$ can be rewritten to $a_i^* - b_i^* + a_{i+1}^* - b_{i+1}^* - (a_i^* - b_{i+1}^* + a_{i+1}^* - b_i^*)$. We can see that for every term occur both positively and negatively, thus this sums to 0.

- Case 2: $a_i^* \geq b_i^*$ and $b_{i+1}^* > a_{i+1}^*$. We have that $a_i^* \geq b_i^* > b_{i+1}^*$ and that $b_{i+1}^* > a_{i+1}^* \geq a_i^*$. These two contradict each other, so this case can't occur.

- Case 3: $b_i^* > a_i^*$ and $a_{i+1}^* \geq b_{i+1}^*$. In this case $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| - (|a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|)$ can be rewritten to $b_i^* - a_i^* + a_{i+1}^* - b_{i+1}^* - (|a_i^* - b_{i+1}^*| + |b_i^* - a_{i+1}^*|)$. Unfortunately, this isn't quite enough information yet, so we consider two cases again:

  - Case 3a: $b_{i+1}^* \geq a_i^*$, we now get $b_i^* - a_i^* + a_{i+1}^* - b_{i+1}^* - (b_{i+1}^* - a_i^* + |b_i^* - a_{i+1}^*|)$, which can be rewritten to $b_i^* + a_{i+1}^* - 2b_{i+1}^* - |b_i^* - a_{i+1}^*|$. Now, if $b_i^* \geq a_{i+1}^*$, this simplifies to $2a_{i+1}^* - 2b_{i+1}^*$, which is at least 0, since $a_{i+1}^* \geq b_{i+1}^*$. On the other hand, if $b_i^* < a_{i+1}^*$, this simplifies to $2b_i^* - 2b_{i+1}^*$, which is also at least 0, since $b_i^* > b_{i+1}^*$.
  - Case 3b: $b_{i+1}^* < a_i^*$, we now get $b_i^* - a_i^* + a_{i+1}^* - b_{i+1}^* - (a_i^* - b_{i+1}^* + |b_i^* - a_{i+1}^*|)$, which can be rewritten to $b_i^* - 2a_i + a_{i+1}^* - |b_i^* - a_{i+1}^*|$, which is at least 0. Now, if $b_i^* \geq a_{i+1}^*$, this simplifies to $2a_{i+1}^* - 2a_i^*$, which is at least 0, since $a_{i+1}^* \geq a_i^*$. On the other hand, if $b_i^* < a_{i+1}^*$, this simplifies to $2b_i^* - 2a_i^*$, which is also at least 0, since $b_i^* > a_i^*$.

- Case 4: $b_i^* > a_i^*$ and $b_{i+1}^* > a_{i+1}^*$, which implies $a_i^* \leq a_{i+1}^* < b_{i+1}^* < b_i^*$. In this case $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| - (|a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|)$ can be rewritten to $b_i^* - a_i^* + b_{i+1}^* - a_{i+1}^* - (b_{i+1}^* - a_i^* + b_i^* - a_{i+1}^*)$. We can see that for every term occur both positively and negatively, thus this sums to 0.

Hence, in all cases we have that $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| - (|a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|) \geq 0$ and thus $|a_i^* - b_i^*| + |a_{i+1}^* - b_{i+1}^*| \geq |a_i^* - b_{i+1}^*| + |a_{i+1}^* - b_i^*|$.

Problem 5. Suppose we are to construct a Huffman tree for a string over an alphabet $C = c_1, c_2, \ldots, c_k$ with frequencies $f_i = 2^i$. Prove that every internal node in T has an external-node child.

Solution 5. We prove by induction that at the start of the ith iteration of the algorithm, the set of trees the algorithm has is $\{c_1, \ldots c_i\}, \{c_{i+1}\}, \ldots \{c_k\}$. The base case $i = 1$ is trivial, since all trees are singletons. For the induction step, assume that after iteration $i-1$ we have $\{c_1, \ldots c_{i-1}\}, \{c_i\}, \ldots \{c_k\}$. In the ith iteration, the algorithm will merge $\{c_1, \ldots c_{i-1}\}$ and $\{c_i\}$. This is because $\sum_{1 \leq j < i} f_j = f_i - 2$.

Therefore, in the ith iteration the algorithm creates a new internal node that has the external node c_i as a child.

Problem 6. Design a greedy algorithm for the following problem (see Figure 2): Given a set of n points $\{x_1, ..., x_n\}$ on the real line, determine the smallest set of unit-length intervals that contains all points.

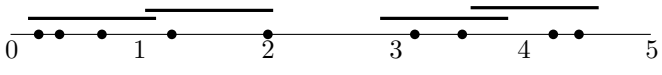

Figure 2: Covering points with unit intervals.

Solution 6. We sort the points from left to right. We start a new unit-length interval at the leftmost point and we find the leftmost point p that isn't covered by this interval. From p we start another interval. This process of finding the leftmost uncovered point and starting a new interval there is repeated until all points are covered.

In order to see that the algorithm is correct, we start from any optimal solution σ and convert it to our greedy solution. Let I be the leftmost interval (the interval with the leftmost left endpoint) of σ that is different from the greedy intervals. We shift interval I to the right until its left endpoint is at a point without leaving any points uncovered in the process (if there are points to its left, these are covered by intervals to the left of I, since those match the greedy solution). This way we can convert every segment in σ that differs from the greedy solution to segment in the greedy solution.

The running time of this algorithm is dominated by the sorting step, which takes $O(n \log n)$ time. Once the input is sorted, the rest of the algorithm just scans through the input, performing only $O(1)$ time operations in the process (checking if a point is contained in the current interval, and starting a new interval when needed).

Problem 7. Suppose we are to schedule print jobs on a printer. Each job j has an associated weight $w_j > 0$ (representing how important the job is) and a processing time t_j (representing how long the job takes). A schedule σ is an ordering of the jobs that tells the printer in which order to process the jobs. Let C_j^σ be the completion time of job j under the schedule σ.

Design a greedy algorithm that computes a schedule σ minimizing the sum of weighted completion times, that is, minimizing $\sum_j w_j C_j^\sigma$.

Solution 7. An optimal schedule can be obtained by sorting the jobs in increasing $\frac{t_j}{w_j}$ order; assume for simplicity that no two jobs have the same ratio.

To show that the schedule is optimal, we use an exchange argument. Suppose the optimal solution is σ and there are two consecutive jobs, say i and k such that $\frac{t_i}{w_i} > \frac{t_k}{w_k}$. We build another schedule τ where we swap the positions of i and k, and leave the other jobs unchanged. We need to argue that this change decreases the cost of the schedule. Notice that the completion time of jobs other than i and k is the same in σ and τ. Thus,

$$\sum_j w_j C_j^\sigma - \sum_j w_j C_j^\tau = w_i C_i^\sigma + w_k C_k^\sigma - w_i C_i^\tau - w_k C_k^\tau. \tag{1}$$

Let $X = C_i^\sigma - t_i$, the time when job i starts in σ, which equals the time when job k starts in τ. Then $C_i^\sigma = X + t_i$, $C_k^\sigma = X + t_i + t_k$, $C_k^\tau = X + t_k$, and $C_i^\tau = X + t_i + t_k$. If we plug these values into (1) and simplify, we get

$$\sum_j w_j C_j^\sigma - \sum_j w_j C_j^\tau = -w_i t_k + w_k t_i. \tag{2}$$

The proof is finished by noting that $\frac{t_i}{w_i} > \frac{t_k}{w_k}$ implies $-w_i t_k + w_k t_i > 0$ and therefore

$$\sum_j w_j C_j^\sigma > \sum_j w_j C_j^\tau.$$

Hence, the schedule σ is not optimal.

Finally, for the running time analysis, we note that sorting the jobs takes $O(n \log n)$ time. Once the jobs are sorted, we only need to output them in that order (which takes $O(n)$ time), hence the algorithm takes $O(n \log n)$ time.