CSCC73 A3

Saad Makrod

September 2022

Q1 Huffman's Algorithm

Part A:

Let $\Gamma = \{a, b, c, d\}$ and the frequency function f be as follows: $f(a) = 2/5$, $f(b) = 1/5$, $f(c) = 1/5$, $f(d) = 1/5$.

Then Huffman's algorithm can construct the tree as follows:

First create an internal node with two letters in Γ with frequency of 1/5, the choice does not matter. Then create a node with a and the left over letter with frequency 1/5. Lastly merger these nodes. This is a valid optimal code by Huffman's algorithm.

So as an example let d and c be extracted to create an internal node with priority 2/5 (d on the left, c on the right). Now let b and a be extracted to create an internal node with priority 3/5 (b on the left, a on the right). Now finally, the nodes with weight 2/5 and 3/5 are extracted and merged to create a node of priority 1 (c,d on left, and b,a on right). Thus we have the following codewords: $a = 11$, $b = 10$, $c = 01$, and $d = 00$. Note that all codewords have length 2.

Thus this is an example where Huffman's algorithm could produce a tree in which no codeword has length 1.

Part B:

The proof will be done by contradiction.

Suppose that this is not the case and Huffman's algorithm does not produce any codeword of length 1. This means that all codewords are at least of length 2. This means that level at depth 2 there will be 4 nodes, let them be denoted by n_1, n_2, n_3, n_4. I know that the sum of the priorities of these nodes must be 1, i.e. $priority(n_1) + priority(n_2) + priority(n_3) + priority(n_4) = 1$. If this were not the case that would mean the total frequency would be less than or greater than 1 which is impossible.

Let the two nodes that are merged together first be n_1 and n_2. By Huffman's algorithm this means that $priority(n_1), priority(n_2) \leq priority(n_3), priority(n_4)$. There are two cases here, either the node with priority greater than 2/5 is in $\{n_1, n_2\}$ or it is in $\{n_3, n_4\}$.

The case where the node with priority greater than 2/5 is in $\{n_1, n_2\}$ is impossible. This is because I know that $priority(n_1), priority(n_2) \leq priority(n_3), priority(n_4)$ so if the node with priority greater than 2/5 is in $\{n_1, n_2\}$ then $2/5 \leq priority(n_3), priority(n_4)$. This implies that $1 < 6/5 < priority(n_1) + priority(n_2) + priority(n_3) + priority(n_4)$, which is impossible. Thus this is a contradiction since $priority(n_1) + priority(n_2) + priority(n_3) + priority(n_4) = 1$.

In the case where the node with priority greater than 2/5 is in $\{n_3, n_4\}$, there are three nodes that can be merged, one has $priority(n_1) + priority(n_2)$, another has $priority(n_3)$, and the last node has $priority(n_4)$. Note that one of n_3 or n_4 contains the symbol with priority greater than 2/5. Let n_3 be that node. n_3 must be merged next or it will have a codeword of length 1. If n_3 is merged with n_4 this means that $priority(n_1) + priority(n_2) \geq 2/5$. Due to this it must be the case that either $priority(n_1) > 1/5$ or $priority(n_2) > 1/5$. This implies that $priority(n_4) > 1/5$ since $priority(n_1), priority(n_2) \leq priority(n_3), priority(n_4)$. Thus we have $priority(n_1) + priority(n_2) + priority(n_3) + priority(n_4) > 1/5 + 2/5 + 2/5 = 1$ which is a contradiction. If we have n_3 merged with the node of $priority(n_1) + priority(n_2)$ this means that $priority(n_4) \geq priority(n_3)$. Note that n_4 is a tree and not a symbol since we cannot have a codeword of length 1. Furthermore, note that any symbol in n_4 can only be depth $\geq$ depth of n_3, this is because since n_4 is a tree its symbols will start at least at depth 2 and n_3 is on depth 2, we will call n_4's symbol at depth

2 x. There are three sub cases here, $priority(x) < priority(n_3)$, $priority(x) = priority(n_3)$, and $priority(x) > priority(n_3)$. If $priority(x) \geq priority(n_3)$ there is a contradiction since this implies the remaining symbols in n_4 are extremely small in frequency. This implies that they would have been merged with node of $priority(n_1) + priority(n_2)$ not n_4. Thus n_4 is actually a symbol and we have a codeword of length 1 which is a contradiction. In the case where $priority(x) < priority(n_3)$ note that $priority(n_1) + priority(n_2) < 1/5$ since both $priority(n_4), priority(n_3) > 2/5$. This implies that the frequency of x must be less than 1/5 since otherwise the node it was merged with would be merged with node of $priority(x) < priority(n_3)$ instead. However, similarly we cannot let the node that x was merged with be greater 1/5 in priority otherwise x would be merged with the node of $priority(n_1) + priority(n_2)$. This implies that $priority(n_4) < 2/5$ which cannot be the case because by Huffman's algorithm we must have $2/5 < priority(n_3) \leq priority(n_4)$. Thus we have a contradiction here as well. Thus the case where n_3 merged with the node of $priority(n_1) + priority(n_2)$ cannot occur.

Thus we have a contradiction in both cases, so we must have a codeword of length 1.

QED Proof

Q2 D&C Algorithm For Sorting Assignments

Part A:

I know that there are k piles of n papers. At each iteration i Vassos will merge a pile of size in with a pile of size n, the resulting work done is $(i+1)n$. This is done a total of $k-1$ times since k piles must be merged, thus the algorithm continues until $(i+1)=k$. For example if $k=4$ then I first merge two piles of size n which takes $2n$ work, then I merge a pile of size $2n$ with a pile of size n which is $3n$ work, and finally I merge a pile of size $3n$ with a pile of size n which is $4n$ work. So at each iteration $(i+1)n$ work is done.

This can be captured in the summation $\sum_{i=1}^{k-1}(i+1)n$, since at each stage $(i+1)n$ work is done. Note that the sum $\sum_{i=1}^{k-1}(i+1)n = \sum_{i=2}^{k} in$.

Thus since this sum captures the work done by the algorithm, I can evaluate it to find the complexity.

$\sum_{i=2}^{k} in = n\sum_{i=2}^{k} i < n\sum_{i=1}^{k} i = n\frac{k(k+1)}{2} \in O(nk^2)$.

Thus the time complexity of the algorithm is $O(nk^2)$.

Part B:

Consider the following algorithm:

Divide the k piles into pairs and merge all pairs together. Recurse, and repeat this operation until all piles are merged into a single pile.

So if $k=4$, then there are two pairs which are merged into two piles of size $2n$, then the two piles of size $2n$ form a pair and are merged into 1 pile of size $4n$.

The number of times this algorithm will recurse is $\lceil log(k) \rceil$. At each step the work done is always bounded by kn which is the number of papers. This can be captured in the summation $\sum_{i=1}^{\lceil log(k) \rceil} nk$.

Evaluating the sum,

$\sum_{i=1}^{\lceil log(k) \rceil} nk = nk\sum_{i=1}^{\lceil log(k) \rceil} 1 = \lceil log(k) \rceil nk \in O(nklog(k))$.

Thus this is a more efficient algorithm that Vassos can use.

Q3 D&C Algorithm For Largest Interval Overlap

Consider the following algorithm, where I is the list of intervals:

```
greatestOverlapDriver($I$)
    $L$ = intervals in $I$ sorted in increasing order by $\ell$
    $R$ = intervals in $I$ sorted in increasing order by $r$
    return greatestOverlap($L$,$R$)
greatestOverlap($L$,$R$)
    if $|L| \leq 3$
        calculate and return the two intervals with the greatest overlap through brute force
    $L_1$ = first half of $L$
    $L_2$ = second half of $L$
    $R_1$ = first half of $R$
    $R_2$ = second half of $R$
    $boundary$ = last interval in $R_1$
    $(i_1, i_2)$ = greatestOverlap($L_1$, $R_1$)
    $(j_1, j_2)$ = greatestOverlap($L_2$, $R_2$)
    $(s_1, s_2) = \textit{pair of intervals with greatest overlap between } (i_1, i_2), (j_1, j_2)$
    $\delta = \textit{overlap between } (s_1, s_2)$
    for every interval $i$ in $L_2$ that overlaps with $boundary$
        if overlap between $boundary$ and $i > \delta$
            $\delta$ = overlap between $boundary$ and $i$
            $(s_1, s_2) = (boundary, i)$
    return $(s_1, s_2)$
```

Algorithm Explanation

The algorithm will first sort the set of intervals I into two sets, one which contains all the intervals listed in increasing order of r called R, and the other has increasing order of ℓ called L. In other works, L has the intervals sorted by their "start" value and R has the intervals sorted by their "end" value.

Next the function which actually handles the D&C is called. First if L is has less than three intervals you can calculate the greatest overlap through brute force and return.

If this is not the case then split L and R into two halves. We define *boundary* to be the last interval in R_1 which will be used later in the algorithm.

Now recursively solve for the intervals with the greatest overlap in L_1 and L_2. After this is done we compare the two pairs of intervals and take the pair with the greater overlap and name the pair (s_1, s_2). After this there is one case left, we have found the greatest overlap between L_1 and L_2 but we have not considered that it can be in the intersection of L_1 and L_2.

This is where *boundary* is used. We only need to consider the intervals which overlap with *boundary* because as by the definition of R_1, *boundary* is the interval in L_1 that goes deepest into L_2. In other words, if any interval in L_1 overlaps with an interval in L_2, it will also overlap with *boundary*.

Thus once we complete this check all three cases are considered and the pair of intervals with greatest overlap can be returned.

Algorithm Justification

This was touched in the section above, there are essentially three cases based on the D&C algorithm: either the greatest overlap is in L_1, the greatest overlap is in L_2, or the greatest overlap is in the intersection of L_1 and L_2.

The first two cases, the greatest overlap is in L_1, or the greatest overlap is in L_2 is covered by the recursion of the D&C. At some point the brute force will compute the greatest overlap and return it. So we will receive the greatest overlap in L_1 and L_2. Next the algorithm computes the greater overlap between these pairs so we definitely get the pair of intervals with the greater overlap. I will refer to this pair as (s_1, s_2).

The case where greatest overlap is in the intersection of L_1 and L_2 is handled by the for loop. We use the earlier computed (s_1, s_2) and compare it to all possible overlaps in the intersection. This is done by using *boundary*. The reason that we do not need any other interval from L_1 is straight-forward from the definitions. Any interval in L_1 starts before or at the same time as any interval in L_2. Furthermore, R_1 sorts the intervals in L_1 by increasing r, meaning the one which ends the latest is the one which is last in R_1. Meaning that *boundary* ends the latest out of all intervals in L_1. Thus since this is the case any other interval in L_1 will only overlap with a subset of intervals that are overlapped by *boundary*. It is impossible for any other interval in L_1 to overlap a greater amount with an interval in L_2 because *boundary* will always end at least at the same time as the L_1 interval and will always start before or at the same time as the interval from L_2. Thus we only need to compare overlaps with *boundary* to check if there is a greater overlap, and if there ever is we set (s_1, s_2) to that pair ensuring that we always keep track of the greatest overlap.

At the end (s_1, s_2) is returned and since we justified that all cases are covered it is clear that the pair of intervals returned will have the greatest overlap.

Algorithm Complexity

For arbitrary n we can define the recurrence relation as follows:

$$T(n) = 1T(\lceil \tfrac{n}{2} \rceil) + 1T(\lfloor \tfrac{n}{2} \rfloor) + cn^1 = T(\lceil \tfrac{n}{2} \rceil) + T(\lfloor \tfrac{n}{2} \rfloor) + cn$$

This is because we always create two sub problems of size $n/2$, thus we get $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$. Furthermore, we will list the complexity of all other operations:

- Sorting I into L and R can be done in $O(n)$ time
- Calculating the overlap of two intervals is $O(1)$
- Calculating the greatest overlap between ≤ 3 intervals is $O(1)$
- Dividing L and R into halves can be done in $O(1)$
- Comparing the greatest overlap between two pairs of intervals is $O(1)$
- The for loop will run $O(n)$ times since it bounded by the number of intervals. Thus in the case of the for loop we run an $O(1)$ operation $O(n)$ times which will be $O(n)$.

Clearly the sum of all the other operations is $O(n)$, thus we have cn in the recurrence relation.

The recurrence relation can be evaluated using Master Theorem. By Master Theorem we have $a = 1 + 1 = 2$, $b = 2$, and $d = 1$, thus we have $a = b^d$ which is the second case. Thus $T(n) \in \theta(n^d log(n)) = \theta(nlog(n))$ as wanted.

Thus the complexity of the D&C algorithm is $\theta(nlog(n))$ by Master Theorem.