Comp 251: Assignment 4

Ian Benlolo 260744397
McGill University
November 19, 2017

1. **First Case:** $d < \alpha$:
   The base case, $T(1) = \Theta(1^\alpha) = \Theta(1)$ is true because $T(1) = c$.
   IH: Assume $T(n) = \Theta(n^\alpha)$. So the induction step follows:
   $T(bn) = aT(n) + c(bn)^d = \Theta(a(n)^\alpha) + \Theta((bn)^d) = \Theta(a(n)^\alpha)$ since $\alpha > d$.
   Hence by induction, if $d < \alpha, T(n) = \Theta(n^\alpha)$.

   Second Case $d = \alpha$:
   Base case: $T(i) = \Theta(i^\alpha \log(1)) = \Theta(1 \times 0)$ which is true as above.
   IH: assume $T(n) = \Theta(n^\alpha \log(n))$, the induction step follows:
   $T(bn) = aT(n) + c(bn)^d = \Theta(a(n)^\alpha \log(bn)) + \Theta((bn)^d) = \Theta(an^\alpha \log(bn)) + \Theta(a(n)^\alpha)$ since $\alpha = d \rightarrow T(bn) = \Theta(an^\alpha \log(bn))$

   Third case: $d > \alpha$:
   Base case: $T(i) = \Theta(1^d) = \Theta(1)$, same as above.
   IH: Assume $T(n) = \Theta(n^d)$ then $T(bn) = \Theta((bn)^d) = \Theta(an^d)$.
   It goes as follows:
   $T(bn) = aT(n) + c(bn)^d = \Theta(an^d) + \Theta((bn)^d) = \Theta(an^d)$ since $\alpha < d \rightarrow T(bn) = \Theta(an^d)$.
   Then by induction, if $d > \alpha, T(n) = \Theta(n^d)$.

2. (a) To solve this in $O(n^2)$ we would use a nested loop. The first one from $i = 0$ to the length of the array and the nested on from $j = i + 1$ also to the length of the array. We would keep a variable max within the nested loop we would check if $a_j - a_i > max$, then you would update max and store the values of i and j at these points.

   (b) We start by determining the base case which is if we have one day, we buy and sell on that day which gives no profit. If it isn't so, we split the array into half and the correct indices to buy/sell are in one of the following three cases:

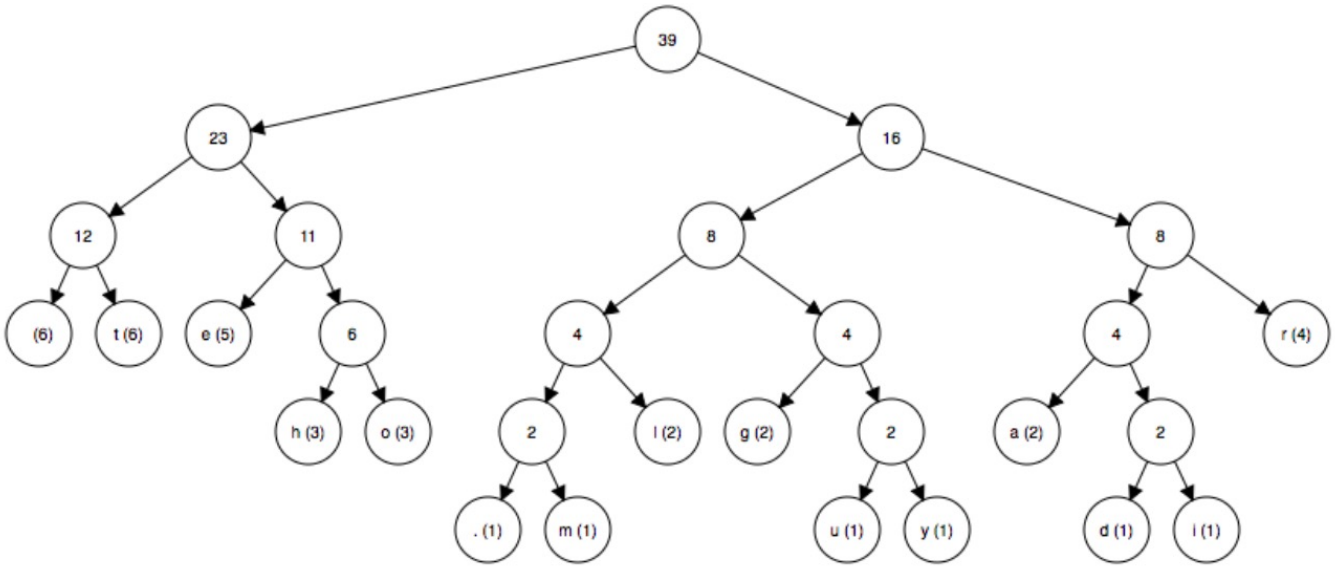
      I Both in the first half.
      II Both in the second half.
      III Each in one of the halves.

   I and II are trivial; they can be found by simply recursively calling our algorithm. III can be solved by a simple scan over the array and finding these two values. Our recurrence relation is therefore:$T(1) \leq O(1)$ and $T(n) \leq 2T(n/2) + O(n)$ which is $O(n * \log n)$ time (using the Master Theorem).

3. There are a total of 39 characters in this sentence. The frequency of each character goes as follows:

$$f_t = \frac{6}{39}, f_h = \frac{3}{39}, f_a = \frac{2}{39}, f_{space} = \frac{6}{39}, f_r = \frac{4}{39}, f_u = \frac{1}{39}, f_g = \frac{2}{39}, f_e = \frac{5}{39}, f_l = \frac{2}{39},$$
$$f_y = \frac{1}{39}, f_i = \frac{1}{39}, f_d = \frac{1}{39}, f_o = \frac{3}{39}, f_m = \frac{1}{39}, f_. = \frac{1}{39}$$

   Building a binary tree from this using Huffman code, we take the lest frequent nodes, make them siblings, and add their frequencies. Doing this we build the following tree:



Setting every left-going edge to a 0 and right-going edge to a 1 and encoding every letter we get the following:

$$t = 001,\ h = 0110,\ a = 1100,\ space = 000,\ r = 111,\ u = 10110,\ g = 1010,\ e = 010,$$

$$l = 1001,\ y = 10111,\ i = 11011,\ d = 11010,\ o = 0111,\ m = 10001,\ . = 10000$$

The sentence therefore translates to:
0010110110000100011110110101000011101011001001100110111000001110110101101 00
00001011001110001110111011110001000001011110100100010110010111 10000

4. If we had an input file with, say, n-chars, we'd have $(2^8)^n$ possible inputs.
Now suppose you could compress this for all inputs to i chars, with $i < n$, we would be able to form 2^{8i} compressions whereas 2^{8i} is at most $2^{8n} - 1$ possibilities.
By the PigeonHole Principle, with 2^{8n} items, and 2^{8i} containers, there would be at least one container with two items; two chars will have the same compressed string since $2^{8n} > 2^{8i}$.

5. Algorithm boardCoverDivideAndConquer input: integer n, cell p //n=size of the board, p=the cell thats missing We start by defining the base case, because we are using a recursive algorithm. If we have a 2×2 square with one cell missing we fill it with a tile.
We split the board to four $n/2 \times n/2$ squares. We set a tile in the center of the board such that it covers one tile in each of the quarters which do not contain the missing cell. We now have four boards, each containing one cell "missing"; either covered or the initial missing cell on the board.
We now recursively call this algorithm with input $n/2$ and the cell thats missing on each of the $n/2 \times n/2$ boards.

The time complexity of this $T(n) = 4T(n/2) + C$ which is $\Theta(n^2)$, using the Master Theorem.

6. Doing this is actually much simpler than it appears. It is taught in almost any basic linear algebra course that the (i, j) element in the n^{th} power of an adjacency matrix counts the number of paths of length n starting at i and ending at j. Since we are looking for paths of length 3 which start and end at the same node, we would take the 3^{rd} power of the adjacency matrix A. Now, the i^{th} diagonal element in the A^3 matrix counts the number of triangles which contains i as one of the nodes.
Therefore we would take cube the adjacency matrix, take the trace and divide it by 3 because a triangle can have 3 permutations. The time complexity of this would be the time complexity of matrix

multiplication, therefore $\Theta(n^{2.81\ldots})$ as seen in class, $O(n)$ for the addition and $O(n^{1.585\ldots})$ for the division as seen in class. Now,

$$\lim_{n \to \inf} \frac{n^{281\ldots} + n^{1.585\ldots} + n}{n^3} = 0$$

therefore this would in fact be $o(n^3)$.