CENG 567 Design and Analysis of Algorithms Homework 3

Ardan Yılmaz 2172195

November 20, 2022

1 Three-way sort

Algorithm 2 THREEWAYSORT(int [] L, int i, int j)

```
if L[i] > L[j] then
    swap(i, j)
end if
if (j − i + 1) > 2 then
    t = (j − i + 1)/3
    THREEWAYSORT(L, i, j − t)
    THREEWAYSORT(L, i + t, j)
    THREEWAYSORT(L, i, j − t)
end if
```

1.a Prove that this algorithm is correct.

Pf. By Induction
Base Case: $j - i + 1 = 2$
That is the list L has only two entries and the execution of the first if statement sorts L (if unsorted). And the function returns a sorted list.

Consider cases where $L.length \geq 3$**.**
Define three sublists of L:
$L_1 = L[i : i + t]$,
$L_2 = L[i + t : j - t]$,
$L_3 = L[j - t + 1 : j]$
The three recursive calls correspond to sorting $L_1 \cup L_2$ first, then $L_2 \cup L_1$, and finally $L_1 \cup L_2$ again.

Inductive Hypothesis:
THREEWAYSORT sorts sublists, $L_1 \cup L_2$, $L_2 \cup L_3$, and $L_1 \cup L_2$.
Inductive Step: Assuming THREEWAYSORT sorts sublists, show it also

sorts the whole list.
The first recursive call, THREEWAYSORT($L_1 \cup L_2$) places the greater values to L_2 and smaller ones to L_1.
Then, the second call THREEWAYSORT($L_2 \cup L_3$) places the greater values to L_3 and smaller ones to L_2.
That is, the greatest values in L are guaranteed to be placed in L_3. However, as a result of the second recursive call, the "small" numbers initially in L_3 are now placed in L_2. That is, another ordering between L_1 and L_2 is needed.
THREEWAYSORT($L_1 \cup L_2$) puts the smallest to L_1.
As a result, L_1 is guaranteed to have the smallest values, and L_3 has the greatest ones. Additionally, the sublists are sorted as per the assumption. Hence, L is sorted, which completes the proof.

1.b Let f(n) be the running time of THREEWAYSORT. Write a recurrence for f(n).

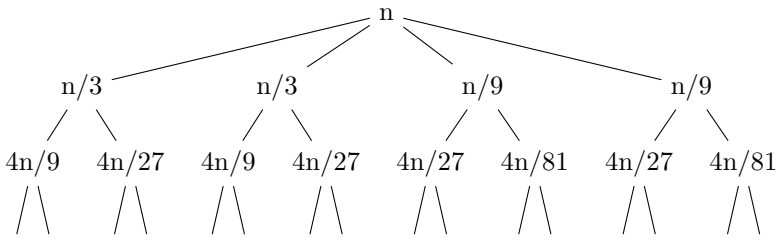
Each recursive call gets 2/3 of the whole list and there are 3 calls making an $O(1)$ swap operation in the end. Hence,
$f(n) = 3f(2n/3) + c$

1.c Solve your recurrence to determine the worst-case running time of threewaysort on lists of size n. How does it compare to other sorting algorithms you know?

Applying Master's Theorem, where $a = 3$, $b = 3/2$, and $d = 0$. $f(n) \in O(n^{log_{3/2}3})$, which is approximately $O(n^{2.7})$. The number of operations is regardless of the entries in the list, hence best/worst/average cases are all $O(n^{2.7})$. This is worse than all the other sorting algorithms I know (bubblesort, quicksort, mergesort, etc) except for a terribly naive solution, where all possible orderings are determined and the sorted one is chosen (an algorithm with exponential complexity).
This solution is based on [1].

2 Recurrences

2.a $T(n) = 2T(n/3) + 2T(n/9) + n$



Adding up the operations on each level:
0th level has: n
1st level has: 8/9n
2nd level has: $64/81 = (8/9)^2 n$
kth level has: $(8/9)^k n$
This, in total, makes: $\sum_{i=0}^{d}((8/9)^i n)$, where d is the depth of the tree. $d = log_3 n$

$\sum_{i=0}^{log_3 n}((8/9)^i n) = n * \frac{1-(8/9)^{log_3 n+1}}{1-8/9} = 9n * (1 - (8/9)^{log_3 n} * (8/9))$

$= 9n - 8n * n^{log_3 8-2} = 9n - 8 * n^{log_3 8-1}$

As $1 > log_3 8 - 1$, we can say $T(n) \in \Omega(n)$ and $T(n) \in O(n)$

Claim 1: $T(n) \in O(n)$

Pf. 1 Show $T(n) = 2T(n/3) + 2T(n/9) + n \leq 2cn/3 + 2cn/9 + n$
$= 8cn/9 + n = n(8c/9 + 1) \leq cn$ for $\forall c > 9$

This equation is upper-bounded by cn.
That is, $T(n) \leq cn$ also. And hence, $T(n) \in O(n)$

Claim 2: $T(n) \in \Omega(n)$

Pf. 2: $T(n) = 2T(n/3) + 2T(n/9) + n \geq n$. This is clearly true, hence, $T(n) \in \Omega(n)$.

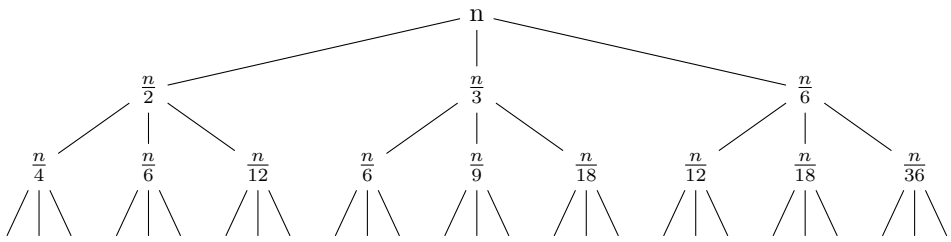
2.b $T(n) = 1/3(T(n-1) + T(n-2) + T(n-3)) + cn$

$T(n) = 1/3(T(n-1) + T(n-2) + T(n-3)) + cn$ is equivalent to $T(n) = 1/3(3T(n-1)) + cn$ and $T(n) = T(n-1) + cn$.

Show by unrolling:
$T(n) = T(n-1) + cn$

$= T(n-2) + +cn + cn = T(n-2) + 2c$
At k^{th} step: $T(n-k) + kc$ and when k = n, we arrive T(0), which is a constant.
$T(0) + kcn = T(0) + cn^2 \in O(n^2)$

2.c $T(n) = T(n/2) + T(n/3) + T(n/6) + n$



Adding up the operations on each level:
0th level has: n
1st level has: n
2nd level has: n
kth level has: n
This, in total, makes: $\sum_{i=0}^{d} n$, where d is the depth of the tree. $d = log_2 n$

$\sum_{i=0}^{lg_2 n}(n) = nlg_2 n$

Claim 1: $T(n) \in O(nlgn)$

Pf. 1: Show that the following holds:

$T(n) \leq cn/2 * lg(n/2) + cn/3 * lg(n/3) + cn/6 * lg(n/6) + n \leq dnlgn$

$cnlgn - cn(1/2 * lg2 + 1/3 * lg3 + 1/6lg6) + n = cnlgn - cn * (C') + n = cnlgn + n(-cC' + 1) \leq cnlgn$. This holds $\forall c > 0$.

Hence, $T(n) \in O(nlgn)$

Claim 2: $T(n) \in \Omega(nlogn)$

Pf. 2 By the equation of T(n), the following holds:
$T(n) \geq 3T(n/6)$, plug in $T(n) = nlogn$ as per the assumption.

$nlogn \geq 3n/6 * log(n/6) = n/2 * (logn - log6)$. This holds for $\forall n \geq 6$.
Thus, $T(n) \in \Omega(nlogn)$.

2.d $\sqrt{2n}T(\sqrt{2n}) + \sqrt{n}$

Changing variable $n = 2^m$, and hence achieving $\sqrt{n} = 2^{n/2}$ did not yield anything useful.

3 Decimal to Binary

3.a Design an algorithm to convert decimal number 10^n to binary, where n is a power of 2. What is the running time of the algorithm?

Given: fastmultiply(z, z) function executing binary multiplication in $O(n^{log_2 3})$

Algorithm 3 PWR2BIN(n)

```
if n == 1 then
    return 1010
end if
z = PWR2BIN(n/2)
return fastmultiply(z, z)
```

1. **Runtime Analysis**
   The algorithm divides the original problem into equal halves, then performs fastmultiply, an $O(n^{log_2 3})$ operation. Say T(n) is the total run-time of the algorithm, then the recurrence relation for T(n) is as follows:
   $T(n) = T(n/2) + O(n^{log_2 3})$.
   Applying Master's Theorem, $T(n) \in O(n^{log_2 3})$

2. **A worked example**
   Say n = 2, ie, convert 100 to binary.
   - call PWR2BIN(2)
   - This call does not enter the if. Proceeds to call z = PWR2BIN(1)
   - return z = 1010 to the last recursive call
   - return fastmultiply(z, z), which is the binary equivalent of multiplying 10*10.

3. **Proof of Correctness (By Induction)**
   Base Case: n=1: PWR2BIN(1) returns 1010, which is the true binary representation for 10^1.
   Inductive Hypothesis: Assume it successfully calculates the binary representation of 10^m, where m is a power of 2.
   Inductive Step: Show it also works for 10^{2m}
   PWR2BIN(10^{2m}) = fastmultiply(PWR2BIN(10^m), PWR2BIN(10^m)).
   This returns the true binary representation of $10^2 m$, as per the correctness assumption of PWR2BIN(10^m). (And the given function, fastmultiply, also runs correctly). This completes the proof.

3.b Convert a decimal integer x with n digits into binary, where n is a power of 2. What is the running time of the algorithm?

Given: binary(x) function performing a look up into a table containing the binary value of all decimal numbers from 0 to 9.

Algorithm 4 DEC2BIN(x)

```
if length(x) == 1 then
    return binary(x)
end if
Split x into xL the leading n/2 digits and xR, the trailing n/2 digits.
//Halves the operation: T(n/2)
return fastmultiply(PWR2BIN(n/2), DEC2BIN(xL)) + DEC2BIN(xR)
```

1. **Run Time Analysis:**
   As the algorithm is performed on two equal halves of the original input, followed by a $O(n^{log_2 3})$ operation, fastmultiply, the recurrence relation for its run-time is as follows:
   $T(n) = 2T(n/2) + O(n^{log_2 3})$.
   By Master's Theorem, $T(n) \in O(n^{log_2 3})$

2. **A worked example:** Say we call DEC2BIN(1234)
   (a) call DEC2BIN(1234)
   (b) xL = 12 and xR = 34
   (c) fastmultiply(PWR2BIN(2), DEC2BIN(12)) + DEC2BIN(34)
   (d) PWR2BIN(2) returns 100 in binary
   (e) call DEC2BIN(12)
   (f) xL = 1 and xR = 2
   (g) fastmultiply(PWR2BIN(1), DEC2BIN(1)) + DEC2BIN(2)
   (h) PWR2BIN(1) = 10, DEC2BIN(1) = 1, DEC2BIN(2) (as 1 and 2 have length 1)
   (i) fastmultiply(PWR2BIN(1), DEC2BIN(1)) + DEC2BIN(2) = 12 in binary. Returns to call DEC2BIN(12) on line (e)
   (j) backtrack to (c), it now calls DEC2BIN(34)
   (k) DEC2BIN(34) = fastmultiply(PWR2BIN(1), DEC2BIN(3)) + DEC2BIN(4)
   (l) PWR2BIN(1) = 10, DEC2BIN(3) = 3, and DEC2BIN(4) = 4, go back to line (k)
   (m) DEC2BIN(34) = 34 (in binary)
   (n) return to c, fastmultiply(2 (in binary), 12(in binary)) + 34 (in binary)

(o) return to (a), this calculates 1234 in binary

3. **Proof of correctness by induction on the length of input**
   Base case: length(x) = 1
   returns binary(x), which is correct.
   Inductive Hypothesis: DEC2BIN(x) works for integers x of length n where n is an integer greater than 1 and divisible by 2.
   Inductive Step: Show DEC2BIN(x') also works for integers, x', of length 2n.
   length(x') = 2n
   xL is the leading n digits of x', hence length(xL) = n.
   xR is the trailing n digits of x', hence length(xR) = n.
   DEC2BIN(x') = fastmultiply(PWR2BIN(n), DEC2BIN(xL)) + DEC2BIN(xR)
   This should return a correct answer as per the assumption made that the algorithm works on integers of length n. And, PWR2BIN(n) has proven to work correctly in part **3.a**.

4 Weighted path. Given a rooted binary tree, weight of a path is defined as the sum of the length of path from all nodes to the node that is closest to the root. Design a divide and conquer algorithm that finds the maximum weight among all simple paths in the tree

The following algorithm (an ensemble of 2 functions) developed mostly by Deniz Germen solves this problem. [2]

The following algorithm utilizes three different types of variables, ie, D, C, and M. D is the depth of the path with the current node, used to calculate how much of an update current root accounts for if added to the path. B holds the value of the max path, ie, how much it accumulates to the total weight if added to the path. And, M holds the maximum valued path. The subscripts used, L, R, and C, mean right, left, and current, respectively. The algorithm calls for left and right sub-trees recursively, until leaf nodes are reached. Each node is considered to be the "main node" (the node closest to the root) and its potential addition to a path is calculated. For this, we need depth information, as an addition of a node to a path accumulates a weight of worth as its distance from the "main node". Then, the max path value, B, is calculated to indicate how much potential value the best path up to that main node carries. Then, the real path value is calculated as M, which is the result.

Algorithm 6 MAX_PATH(T)

M, _, _ = helper(T)
return M

Algorithm 8 helper(T)

if (L → Left): M_L, B_L, D_L = helper(L → Left)
else: M_L, B_L, D_L = 0, 0, 0
if (L → Right): M_R, B_R, D_R = helper(L → Right)
else: M_R, B_R, D_R = 0, 0, 0
$D_c = 1 + max(D_L, D_R)$ //Accumulate the depth info
$B_c = max(B_L, B_R) + D_c$ //Calculate the potential value of the branch
$M_c = max(B_L + B_R, M_L, M_R)$ // Calculate the real weight of the path
return M_c, B_c, D_c

1. **Run-Time Analysis**

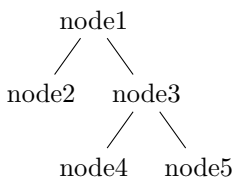
   (a) **Best Case** occurs when the tree is balanced, ie, it roughly all nodes, except the leaves, have both left and right sub-trees. This is when the problem is divided into sub-problems.
   $T(n) = 2T(n/2) + O(1)$
   $T(n) \in O(n)$ by the Master's Theorem.

   (b) **Worst Case** occurs when the tree grows only to one side. Hence, the size of the sub-problem is merely decreased by one node.
   $T(n) = T(n-1) + O(1)$. Applying unrolling method:
   $T(n) = T(n-1) + c = T(n-2) + c + c = ...$
   $T(n) = T(n-k) + kc$. Knowing T(0) = c, k = n is constant. Hence,
   $T(n) = T(0) + nc$, and $T(n) \in O(n)$.

   Since the number of operations is not dependent on n, ie, constant, the order of subdivision seems not important. Hence, all best, worst, and average run time complexities are O(n)

2. **A worked example**

   Say the following tree is given.



   The algorithm runs as follows:

(a) It first recursively searches through the left tree and finds node2. The algorithm returns $M_c = 0$, $D_c = 1$, and $D_c = 1$ back to node1.

(b) It proceeds with the right sub-tree, discovering node3 first and then node 4. The algorithm returns $M_c = 0$, $D_c = 1$, and $D_c = 1$ for node 4 back to node3.

(c) Proceeding with node 5, the algorithm returns $M_c = 0$, $D_c = 1$, and $D_c = 1$ for node 5 back to node3.

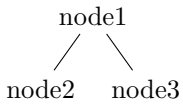
(d) Node 3 calculates $D_c = 1 + max(1, 1) = 2$, $B_c = max(1, 1) + 2 = 3$, and $M_c = max(2, 0, 0) = 2$. Returning back to node1.

(e) Node1 calculates $D_c = 1 + max(2) = 3$, $B_c = max(1, 3) + 3 = 6$, and $M_c = max(4, 0, 2) = 4$. Returning back to the initial function.

(f) Returned M value is 4.

3. **Proof of Correctness (By Induction on the number of nodes in the tree)**

   (a) **Base Case:** n = 3 (n=1 is trivial to show).

   Algorithm finds the path weight 2, which is correct.

   (b) **Inductive Hypothesis:** Assume the algorithm works correctly for trees with $m < n$ nodes.

   (c) **Inductive Step**
   Say we add another node as a leaf of the tree. This addition is likely to update result of the weight of the max path. Keeping the depth information of the tree provides the information of how much to update the path weight on an addition of a node to the path. Say we add a node with a path of length n to the "main node" (the node closest to the root), it should account for an update of worth n to the overall result. This is correctly accomplished with the addition of the depth information. In the event that the max-weight path is updated, though I am not sure if this is possible, the algorithm also detects and updates the true path, as it recursively finds the max valued paths of the left and right sub-trees as per the assumption. Hence, the algorithm works for a tree with n+1 nodes, which completes the proof.

5 REFERENCES

1. https://stackoverflow.com/questions/58479678/how-to-prove-correctness-of-recursive-algorithm

2. We have exchanged ideas with Yalgınay Yaltırık and Deniz Germen.