CSOR W4231 Analysis of Algorithms I - Spring 2021
Homework #1

Joseph High - jph2185

February 10, 2021

Problem 1

(a) Since N is an n-bit integer, $N \cdot (N-1)$ is *at most* a $2n$-bit integer, $N \cdot (N-1) \cdot (N-2)$ a $3n$-bit integer, ..., $N! = N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$ is *at most* an Nn-bit integer. At most, N is the largest number in the range of n-bit values; in this case there are $2^n - 1$ n-bit terms in $N \cdot (N-1) \cdot (N-2) \cdots 2 \cdot 1$, so the number of bits in $N!$ is $\Theta(Nn)$, or $\Theta(n2^n)$.

Algorithm 1 Compute $N!$

(b)
```
1:  FACTORIAL(N)
2:      N_Fac ← 1
3:      if N = 0 :
4:          N_Fac = N_Fac
5:      else:
6:          for i = 1 to N :
7:              N_Fac = *i
8:          end for
9:      end if
10:     return N_Fac
11: end
```

Algorithm 2 Alternative algorithm to compute $N!$

```
1:  FACT(N)
2:      if N ≠ 0 :
3:          return N*FACT(N − 1)
4:      else:
5:          return 1
6:      end if
7:  end
```

From part (a), it was determined that $N!$ is $N \cdot n$ bits long. In each iteration of the for-loop, an Nn-bit integer is multiplied by an n-bit integer (i), and there are a total of

N iterations in the for-loop. Therefore, the running time is $O(N^2n^2) = O((2^n)^2n^2) = O(4^nn^2)$.

Correctness: Base case ($N = 1$): $N = 1$, so the for-loop iterates exactly once and $N_{Fac} = 1$. Induction hypothesis: Now suppose that the algorithm returns $(N-1)!$ for all non-negative integers up to $N-1$. For N the algorithm computes $N*\textsc{Fact}(N-1)$. By the induction hypothesis, $\textsc{Fact}(N-1)$ returns $(N-1)!$, and $N*(N-1)! = N!$.

Problem 2

(a) i. Because the tree is a representation of a comparison-based sorting algorithm, every path must lead to a possible order/sequence of any given set of numbers input into the algorithm. Since every leaf corresponds to a possible arrangement of the numbers, every permutation must appear in the tree. The algorithm must be able to generate all possible permutations of n numbers to be correct. Suppose it was not the case that all permutations appeared in the tree. That is, suppose some permutation $p = (\alpha(x_1), \alpha(x_2), \ldots, \alpha(x_n))$ does not appear in the tree. Then for sets of numbers that follow the non-decreasing order in p, the algorithm either outputs an incorrect ordering or does not run to completion.

ii. In part (i), it was argued that every possible permutation of the n numbers must appear as a leaf in the tree. Thus, the tree has at least $n!$ leaves.

iii. The maximum number of comparisons performed corresponds to the length of the longest (simple) path starting at the root node (i.e., the depth of the binary tree). Indeed, at each node, exactly one comparison is performed, so the number of comparisons performed for a given path is equal to the number of edges traversed on that path. In other words, the maximum number of comparisons performed corresponds to the depth of the tree.

iv. Let L denote the set of all leaves in an arbitrarily chosen binary tree.

Claim: The maximum possible number of leaves occurs when *every* leaf has the same depth. Furthermore, $|L| \leq 2^d$.

Proof of Claim: Consider a binary tree where *every* leaf has the same depth. In such a tree, every node at levels $i \in \{0, 1, \ldots, d-1\}$ has exactly 2 child nodes (i.e., two possibilities for every comparison). Otherwise, all leaves would not have the same depth. This implies that each level $j \in \{1, \ldots, d\}$ has 2 times the number of nodes as level $j-1$. Level 1 has 2 nodes because level 0 has one node (the root node), which has exactly 2 child nodes. Since each node at level 1 has two child nodes, level 2 has $2 \cdot 2 = 2^2$ nodes. Similarly, level 3 has $2 \cdot 2 \cdot 2 = 2^3$ nodes. Continuing in this way, level j has 2^j nodes since every node at every level has two child nodes. Therefore, bottom level (level d) has 2^d nodes, but the last level are the leaves. Therefore, there are 2^d leaves in a binary tree where all leaves have the same depth.

Clearly, this is the maximum number of leaves a binary tree of depth d can possibly have. Indeed, consider a binary tree of depth d where some leaf occurs at level $d-1$, but all other leaves occur at level d. The number of leaves in such a tree is $2^d - 1$ since leaves do not have child nodes. The number of leaves decreases the further up the tree the leaf occurs. Therefore, the number of leaves in the tree is at most 2^d. That is, $|L| \leq 2^d$ □

v. In parts (i), (ii) and (iv), it was argued that the number of leaves in the tree is at least $n!$ leaves and at most 2^d. We also know that $n! \leq n^n$. Putting this all

together, we have the following:

$$\begin{aligned} 2^d \geq |L| \geq n! \geq n^n \quad &\iff \quad 2^d \geq n^n \\ &\iff \quad d \log_2 2 \geq \log_2(n^n) \\ &\iff \quad d \geq n \log_2(n) \\ &\implies \quad d = \Omega(n \log n) \end{aligned}$$

From part (iii), it was argued that the worst-case time complexity of the sorting algorithm corresponds to the depth d of the tree. Therefore, a lower-bound for the worst-time complexity of a comparison-based sorting algorithm is $\Omega(n \log n)$.

(Parts (b) and (c) are on the following page.)

(b) The proposed algorithm is inspired by the Counting Sort algorithm discussed in Chapter 8 of the course textbook.[1] Given an array of integers $A[1,\ldots,n] = [x_1,\ldots,x_n]$ of length n, define an array of zeros $K[1,\ldots,D+1] = [0,0,\ldots,0]$ of length $D+1$ where $D = \max_i x_i - \min_i x_i$. The array K will be used to store the counts of each integer in the input array A. Define an array of the same length as A. Traverse the elements in A: for each value in A, locate the index in K equal to the value in A and increment the corresponding entry in K by one. Next, sum up array values with the value of the predecessor value. Using the array R, map the values in K, according to the order in which they appear in A, with the corresponding indices in K, decrementing the values in K by one in each iteration. The output is the sorted array.

```
ArraySort(A)
  n = length(A)
  D = A.max − A.min
  K = array(D + 1)                    // empty array of length D + 1
  R = array(n)                        // empty array of length equal to length of A
  for i = 1 to D + 1 :
      K[i] = 0
  end for
  for i = 1 to n :
      K[A[i]] = K[A[i]] + 1
  end for
  for i = 1 to D + 1 :
      K[i] = K[i] + K[i − 1]
  end for
  for i = 0 to n :
      R[K[A[i]]] = A[i]
      K[A[i]]  = K[A[i]] − 1
  end for
end
```

Running time: Let $T(n)$ denote the running time of the above algorithm. Creating a zero matrix K of length $D+1$ requires $O(D)$ time. Traversing the elements in A and incrementing corresponding indices in K by one requires $O(n)$ time. Summing array values with the value of the predecessor value requires $O(D)$ time. Last, mapping the values in K with the indices, requires $O(n)$ time. Therefore,

$$T(n) = O(D) + O(n) + O(D) + O(n) = O(n+D)$$

(c) If D is on the order of $O(n)$, then the running time of the algorithm is $O(n)$. This contradicts the lower-bound $\Omega(n \log n)$ derived in part (a). Indeed, $n \neq \Omega(n \log n)$ and $n \log n \neq O(n)$.

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press.

Problem 3

f	g	O	o	Ω	ω	Θ
$\log^2 n$	$6 \log n$	no	no	yes	yes	no
$\sqrt{\log n}$	$(\log \log n)^3$	no	no	yes	yes	no
$4n \log n$	$n \log(4n)$	yes	no	yes	no	yes
$n^{3/5}$	$\sqrt{n} \log n$	no	no	yes	yes	no
$5\sqrt{n} + \log n$	$2\sqrt{n}$	yes	no	yes	no	yes
$n^5 4^n$	5^n	yes	yes	no	no	no
$\sqrt{n} 2^n$	$2^{n/2+\log n}$	no	no	yes	yes	no
$n \log(2n)$	$\frac{n^2}{\log n}$	yes	yes	no	no	no
$n!$	2^n	no	no	yes	yes	no
$\log n!$	$(\log n)^{\log n}$	yes	yes	no	no	no

Problem 4

We can use a divide and conquer approach to achieve a better running time than the "straightforward algorithm". In particular, let $T(n)$ denote the running time of the algorithm we seek, and partition $\nu = \begin{bmatrix}\nu_1 & \nu_2 & \cdots & \nu_n\end{bmatrix}^\top$ evenly into two vectors $\boldsymbol{\nu_1}$ and $\boldsymbol{\nu_2}$, both of length $n/2$, such that $\boldsymbol{\nu_1}$ consists of the first $n/2$ components in ν and $\boldsymbol{\nu_2}$ consists of the last $n/2$ components in ν. That is,

$$\nu = \begin{bmatrix}\boldsymbol{\nu_1} & \boldsymbol{\nu_2}\end{bmatrix}^\top = \begin{bmatrix}\underbrace{\nu_1 & \nu_2 & \cdots & \nu_{\frac{n}{2}}}_{=\boldsymbol{\nu_1}} & \underbrace{\nu_{\frac{n}{2}+1} & \cdots & \nu_n}_{=\boldsymbol{\nu_2}}\end{bmatrix}^\top$$

Consider the following multiplication of the Hadamard matrix H_k by ν:

$$H_k\nu = \begin{bmatrix}H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1}\end{bmatrix}\begin{bmatrix}\boldsymbol{\nu_1} \\ \boldsymbol{\nu_2}\end{bmatrix} = \begin{bmatrix}H_{k-1}\boldsymbol{\nu_1} + H_{k-1}\boldsymbol{\nu_2} \\ H_{k-1}\boldsymbol{\nu_1} - H_{k-1}\boldsymbol{\nu_2}\end{bmatrix} = \begin{bmatrix}H_{k-1}(\boldsymbol{\nu_1} + \boldsymbol{\nu_2}) \\ H_{k-1}(\boldsymbol{\nu_1} - \boldsymbol{\nu_2})\end{bmatrix}$$

The summation $\boldsymbol{\nu_1} + \boldsymbol{\nu_2}$ takes $O(n)$ time since there are $n/2$ entries to add. By the same argument, $\boldsymbol{\nu_1} - \boldsymbol{\nu_2}$ also takes $O(n)$ time. Both multiplication operations, $H_{k-1}(\boldsymbol{\nu_1} + \boldsymbol{\nu_2})$ and $H_{k-1}(\boldsymbol{\nu_1} - \boldsymbol{\nu_2})$, require $T(n/2)$ time to execute, as these are the same recursion operations of size $n/2$. To summarize:

1. Compute $\boldsymbol{\nu_1} + \boldsymbol{\nu_2}$ and $\boldsymbol{\nu_1} - \boldsymbol{\nu_2}$
2. Compute $H_{k-1}(\boldsymbol{\nu_1} + \boldsymbol{\nu_2})$
3. Compute $H_{k-1}(\boldsymbol{\nu_1} - \boldsymbol{\nu_2})$
4. Merge $H_{k-1}(\boldsymbol{\nu_1} + \boldsymbol{\nu_2})$ and $H_{k-1}(\boldsymbol{\nu_1} - \boldsymbol{\nu_2})$ into vector

Pseudocode:

Algorithm 3 Compute $H_k\nu$

```
1: MatrixVectorMultiply(H_k , ν)
2:     ν_1 = ν[1, n/2]
3:     ν_2 = ν[n/2 + 1, n]
4:     [[H_{k-1}, H_{k-1}], [H_{k-1}, H_{k-1}]] = Partition(H_k, blocks = 4, size = (n/2 × n/2))
5:     Top = MatrixVectorMultiply(H_{k-1} , ν_1 + ν_2)
6:     Bottom = MatrixVectorMultiply(H_{k-1} , ν_1 − ν_2)
7:     Product = [Top, Bottom]                    // Merge top and bottom into column vector
8:     return MatrixVectorMultiply(H_{k-1} , ν_1 + ν_2)
9: end
```

which has a running time of $T(n) = 2T(n/2) + O(n) = 2T(n/2) + cn$.

By the Master Theorem, $a = 2,\ b = 2,\ k = 1 \implies T(n) = O(n \log n)$

which is significantly better than $O(n^2)$.

Correctness: Induction on k
Base Case ($k = 0$): The dimension of H_0 is 1×1 and the length of ν is 1. Therefore, there is nothing to partition and the product $H_0\nu$ a scalar multiple, which returns the 1×1 vector $[H_0\nu]$.
Induction Hypothesis: Assume MatrixVectorMultiply(H_i, ν) returns the correct matrix-vector product for $i \in \{1, \ldots, k-1\}$.

Induction Step: For $n = 2^k$, consider the following multiplication of the Hadamard matrix H_k by ν:

$$H_k\nu = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{\nu_1} \\ \boldsymbol{\nu_2} \end{bmatrix} = \begin{bmatrix} H_{k-1}\boldsymbol{\nu_1} + H_{k-1}\boldsymbol{\nu_2} \\ H_{k-1}\boldsymbol{\nu_1} - H_{k-1}\boldsymbol{\nu_2} \end{bmatrix} = \begin{bmatrix} H_{k-1}(\boldsymbol{\nu_1} + \boldsymbol{\nu_2}) \\ H_{k-1}(\boldsymbol{\nu_1} - \boldsymbol{\nu_2}) \end{bmatrix}$$

By the induction hypothesis, MatrixVectorMultiply$(H_{k-1}, \boldsymbol{\nu_1})$ and MatrixVectorMultiply$(H_{k-1}, \boldsymbol{\nu_2})$ return the correct matrix-vector products $H_{k-1}\boldsymbol{\nu_1}$ and $H_{k-1}\boldsymbol{\nu_2}$, respectively. Then MatrixVectorMultiply$(H_{k-1}, \boldsymbol{\nu_1} + \boldsymbol{\nu_2})$ and MatrixVectorMultiply$(H_{k-1}, \boldsymbol{\nu_1} - \boldsymbol{\nu_2})$ both return the correct matrix-vector products $H_{k-1}(\boldsymbol{\nu_1} + \boldsymbol{\nu_2})$ and $H_{k-1}(\boldsymbol{\nu_1} - \boldsymbol{\nu_2})$, respectively. Therefore, the algorithm returns the correct matrix-vector product for all values up to k. Indeed, MatrixVectorMultiply(H_k, ν) merges the two products such that MatrixVectorMultiply$(H_{k-1}, \boldsymbol{\nu_1} + \boldsymbol{\nu_2})$ is stored in the first $n/2$ entries and the output from MatrixVectorMultiply$(H_{k-1}, \boldsymbol{\nu_1} - \boldsymbol{\nu_2})$ is stored in last $n/2$, and by the induction hypothesis, each of these were computed correctly.

Problem 5

Initialize at the root node, which we will denote by s. Probe both child nodes, c_1 and c_2, of s and compare x_s to both x_{c_1} and x_{c_2}. If $x_s < x_{c_1}$ and $x_s < x_{c_2}$, then s is a local minimum: Return s. Else, choose the child node with the smaller label; indeed, since all labels are distinct, one label must be strictly less than the other. Let u denote the node with the minimum label. Next, we repeat the steps noted above. In particular, probe both child nodes, v_1 and v_2, of u. If $x_u < x_{v_1}$ and $x_u < x_{v_2}$, then u is a local minimum. Indeed, x_u is smaller than the label of its parent node and smaller than the labels of both its children nodes. Return u. Otherwise, choose the child node with the smaller label. Repeat until a local minimum is identified.

Pseudocode:

Algorithm 4 Find local minimum $v* \in V$ for binary tree $T = (V, E)$

```
1: V_probed = {}                          // Storage for set of vertices which have already been probed
2: E_probed = {}                          // Storage for set of edges which have already been traversed
3: s = v.root                             // Define s to be the root node
4:
5:  LocalMinimum(T = (V, E) , s ∈ V)
6:     Probe s
7:     for children nodes u_1, u_2 of s :
8:         Probe u_1
9:         Probe u_2
10:        if (x_s < x_{u_1}) and (x_s < x_{u_2}) then:
11:            return s
12:        else:
13:            v = arg min{x_{u_1}, x_{u_2}}          // Define child node with smaller label
14:            V_probed = V_probed ∪ {s, arg max{x_{u_1}, x_{u_2}}}
15:            E_probed = E_probed ∪ {(s, arg max{x_{u_1}, x_{u_2}})}
16:            V' = V \ V_probed                      // Vertices in T that have not yet been probed
17:            E' = V \ E_probed                      // Edges in T that have not yet been traversed
18:            T' = (V', E')
19:            return LocalMinimum(T' = (V', E') , v)
20:        end if
21:    end for
22: end
```

Running time: Let $T(n)$ denote the running time of Algorithm 4. The algorithm iterates *at most* $n/2$ times since only one half of the tree is considered after the root node. This requires $T(n/2)$ time. Probing the root node before for-loop requires constant time $O(1)$. Therefore, the running time is $T(n) = T(n/2)+2O(1) = T(n/2)+c$. By the Master Theorem, with $a = 1$, $b = 2$, and $k = 0$:

$$T(n) = O(n^0 \log n) = O(\log n)$$

Correctness is argued in the discussion above the algorithm.