CSCE 586 - Design and Analysis of Algorithms
Date: Tuesday 20th November, 2018
Name: Micah Hayden
Assignment: Homework #3
Documentation: I discussed a possible way of solving question #7 with 2Lts Mireles, Hanson, and Magness, specifically the method of dividing the matrix into smaller quadrants for the recursion.

Chapter 5, Problem #2

Problem Statement:

Give an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings, where a significant inversion occurs when $i < j$ and $a_i > 2 \cdot a_j$.

Description of Algorithm:

This algorithm will utilize the standard divide and conquer algorithm for counting inversions, but will utilize a different compare. It combines the two algorithms Merge-and-Count and Sort-and-Count [1].

Algorithm 1 Counting Significant Inversions - Merge-and-Count(A, B)

```
Merge-and-Count(A,B):
Maintain a $Current$ pointer into each list, initialized to point to the front elements
Maintain a $Restart$ pointer for each list, pointing to the front elements.
Maintain a variable $Count$ for the number of inversions, initialized to 0
while both $Current$ pointers point to items in lists $A$ or $B$, respectively do
    Let $a_i$ and $b_j$ be the elements pointed to by the $Current$ pointer.
    Let $a_i$ be the element pointed to by the $Inversion$ pointer.
    if $2 \cdot b_j < a_i$ then        ▷ If $2 \cdot b_j < a_i$ then $2 \cdot b_j <$ every other element in $A$.
        Increment $Count$ by the number of elements remaining in $A$
        Advance $Current$ pointer in $B$
    else
        Advance the $Current$ pointer in $A$
    end if
end while
$Current_A \leftarrow Restart_A$
$Current_B \leftarrow Restart_B$
while both lists are nonempty do
    Let $a_i$ and $b_j$ be the elements pointed to by the $Current$ pointers.
    Append the smaller of these two to the output list $L$.
    Advance the $Current$ pointer in the list from which the smaller element was selected
end while
return $Count$ and the merged list $L$
```

Algorithm 2 Counting Inversions - Sort-and-Count(L)

```
if the list has one element then
    There are no inversions
else
    Divide the list into two halves:
    - A contains the first $\lceil n/2 \rceil$ elements
    - B contains the remaining $\lfloor n/2 \rfloor$ elements.
    $(r_A, A) = Sort - and - Count(A)$
    $(r_B, B) = Sort - and - Count(B)$
    $(r, L) = Merge - and - Count(A, B)$
end if
return $r = r_A + r_B + r$ and the sorted list $L$.
```

Proof:

Algorithm #2 is a direct adaptation from the given algorithm to count inversions given in *Algorithm Design*. Algorithm #2 has no changes and thus needs no proof. The only change to Algorithm #1 is the introduction of a second while loop. The first while loop counts the number of significant inversions, which occurs when $2 \cdot b_j < a_i$. The second while loop merges the two lists A and B into the output list L in the same manner as the text, thus it is correct [1].

Because every $a_{i+1}...a_n > a_i$; if $2 \cdot b_j < a_i \rightarrow 2 \cdot b_j < (a_{i+1}...a_n)$. Thus, the algorithm will return the correct count for the number of significant inversions.

Asymptotic Analysis:

Worst Case: The comparison for an inversion still occurs in constant time. The Merge-and-Count procedure takes $O(n)$ time for a list of n elements, plus n operations from the introduction of the second while loop. Thus, Merge-and-Count takes $O(2 \cdot n) \rightarrow O(n)$. [1]. Sort-and-Count satisfies the recurrence $T(n) \leq 2 \cdot T(n/2) + c \cdot n$ when $n > 2$ and $T(2) \leq c$. Thus, the algorithm runs in $O(n \log n)$ time [1].
Best Case: Because the algorithm utilizes a divide-and-conquer approach, it will recursively need to operate until a list has only one element. The depth of recursion and running time is not affected by the number of inversions present in the list. Thus, even under a best-case scenario where there are no significant inversions, it will run in $\Omega(n \log n)$ time.
Average Case: Given that the best and worst case running times are equivalent, the function is tightly bounded by $\Theta(n \log n)$. Regardless of the input model, the average running time will be the same because it cannot be better or worse.
Difficulty: 4
Time: 30 Minutes

Chapter 5, Problem # 6

Problem Statement:

Show how to find a local minimum of T using only $O(\log n)$ probes to the nodes of T.

Description of Algorithm:

There is a complete binary tree T of n nodes, where $n = 2^d - 1$ for some d. Each node v of T has a distinct integer label x_v. Let $x = probe(v)$ be the function to determine the integer label x of v. Assume v knows whether or not it has children (pointers to children are null).

Algorithm 3 Find-Local-Minimum(v)

```
Let the node $v$ be the current node.
Let $a$ and $b$ be the left and right children of $v$, respectively.
if $v$ has no children then
    return $v$
end if
$x_v \leftarrow probe(v)$
$x_a \leftarrow probe(a)$
if $x_a < x_v$ then
    Find-Local-Minimum(a)
else
    $x_b \leftarrow probe(b)$
    if $x_b < x_v$ then
        Find-Local-Minimum(b)
    else
        return $v$                                  ▷ $x_v < x_a$ and $x_v < x_b$, thus, $v$ is a local min.
    end if
end if
```

Asymptotic Analysis:

Worst Case: The worst case execution of this algorithm would occur when the local minimum occurs in a leaf of the tree. However, at each step we only take one of the two possible branches. Each recursive call occurs in constant time: at most 3 probes and 3 comparisons. The subproblem size decreases by half ($n' = \frac{n}{2}$, and there is a single subproblem. By **(5.16)**, this algorithm is bounded $O(\log n)$ [1].
Best Case: The best case would occur when T has a single node. This is a full binary tree, satisfying $n = 2^d - 1$:

$$n = 1 = 2^d - 1|_{d=1} = 2^1 - 1 = 1$$

The algorithm would terminate in constant time because it needs to only check the existence of v's children a and b. Thus, the algorithm runs in $\Omega(1)$.
Average Case: The average case of this algorithm would occur when a local minimum occurs somewhere in the interior of the tree. Assume this is at depth $d' = \frac{d}{2}$. This would simply return when it finds the local minimum, and not continue searching through the tree. This situation would execute in $\frac{\log n}{2} \rightarrow O(\log n)$.

Proof:

Proof of Termination:
The algorithm will always run to termination because there must exist a local minimum in any tree T: either at the root, interior, or at a leaf because all node values x_v are distinct. Branches are only taken if $x_c < x_v$ where c is a child of node v. Thus, at node v, x_v must be less than x of its parent node. If v has no children, it is the local minimum, otherwise the process recursively continues. The algorithm returns whenever the first local minimum is found.

Proof of Correctness:

Anytime a recursive call is made, $x_a < x_v \cup x_b < x_v$. The algorithm recursively calls itself using a subtree T' of which a or b is the root. Because the algorithm follows the smallest probed value, $\exists v \in T'$, such that v is a local minimum of T'.

The number of probes k in any recursive call must satisfy $k \leq 3$: one for the root, and two for the children. The maximum depth of the tree is given by solving the equation $n = 2^d - 1$ for d, which gives the following:

$$d = \log_2(n+1) \tag{1}$$

Thus, the maximum number of probes $f(n)$ is:

$$f(n) = 3 \cdot d = 3 \cdot \log_2(n+1) \tag{2}$$

Let $g(n)$ be $\log_2 n$. $f(n)$ is $O(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ and $c \geq 0$.

$$\begin{aligned} \lim_{n\to\infty} \frac{f(n)}{g(n)} &= c \\ \lim_{n\to\infty} \frac{3 \cdot \log_2(n+1)}{\log_2 n} &= \infty \qquad \text{Apply l'Hopital's rule because limit} = \frac{\infty}{\infty} \end{aligned}$$

The following are the values of $f'(n)$ and $g'(n)$, and the remainder of the limit calculation:

$$\begin{aligned} f'(n) &= \frac{3 \cdot \log_2(e)}{n+1} \text{ and } g'(n) = \frac{\log_2(e)}{n} \\ \frac{f'(n)}{g'(n)} &= \frac{3 \cdot n}{n+1} \\ \lim_{n\to\infty} \frac{3 \cdot n}{n+1} &= \frac{\infty}{\infty} \qquad \text{Use l'Hopital's rule again} \\ \lim_{n\to\infty} \frac{3}{1} &= 3 = c \geq 0 \end{aligned}$$

Thus, $f(n)$ is $O(g(n)) = O(\log(n))$ because $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 3$. Where $f(n)$ is the number of probes to find the solution.

Difficulty: 6

Time: 60 Minutes, mostly showing the proof of the # of probes.

Chapter 5, Problem # 7

Problem Statement:

Suppose you're given an $n \times n$ grid graph G. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G.

Description of Algorithm:

This algorithm will probe each element on the center column, center row, and the boundary of the $n \times n$ matrix. When it finds that minimum value, it checks that node's un-probed neighbors. If the node is the local minimum, the algorithm returns it as the solution. Otherwise, the algorithm will recurse into the quadrant containing the minimum-value neighbor [2]. This will convert the subproblem size from a $n \times n$ matrix to an $\approx \frac{n}{2} \times \frac{n}{2}$ matrix.

Algorithm 4 Find-Local-Min(G)

```
Probe all nodes $v_n$ on the boundary, middle column, and middle row of $G$
$v_{min} \leftarrow$ minimum value of the previous step's probes.
**if** $n \leq 3$ **then**
    All nodes in $G$ have been probed
    **return** $v_{min}$
**else**
    **if** $v_{min} <$ neighbors **then**
        **return** $v_{min}$ as local minimum
    **else**
        Let $G'$ be the $n' \times n'$ matrix of the quadrant containing $v_{min}$'s minimum-value neighbor, where $n' \leq \lceil \frac{n-3}{2} \rceil$.
        Find-Local-Min($G'$)
    **end if**
**end if**
```

Note: G' has $n' \leq \frac{n-3}{2}$ because 3 rows and 3 columns have already been probed, and its dimension contains at most half of the remaining unprobed rows and columns.

Asymptotic Analysis:

Worst Case: Because this algorithm reduces the size of the subproblem by half at each recursion, and there is only one subproblem considered, the algorithm is bounded by $O(n)$ by **(5.5)** [1].
Best Case: The best case scenario occurs when $n = 1$, in which case there is a single operation $\Omega(1)$. However, because $n = 1$, this can be written as $\Omega(n)$. Thus, because the best and worst case bounds are the same, the function is tightly bound by n, making its run time $\Theta(n)$.
Average Case: Because the algorithm is tightly bound by $\Theta(n)$, its average case is trivial, and will still run proportional to the dimension of the matrix n. However, the average run time would depend on the dimension of the matrix, and how the values of each node are arranged (some might find it on the first pass, on the boundary, or buried somewhere within the matrix).

Proof:

Proof of Correctness: The algorithm begins by searching the entire boundary, the center row, and the center column. Let v_{min} be the minimum value found. If v_{min} is smaller than its neighbors, it is a local minimum. Otherwise, let v'_{min} be its smallest neighbor. Because $v'_{min} < v_{min}$, v'_{min} is smaller than every other value so-far probed in graph G. When the recursive call is made, v'_{min} will lie on the boundary of sub-graph G'.
If the algorithm determines that there exists a local minimum, k, on the boundary of G', then $k \leq v'_{min}$, and is thus smaller than its neighboring values in G because $v'_{min} < v_{min}$.
If G is a 3x3 matrix, all nodes will be examined and will return v_{min}, the local minimum.

At each step of the algorithm, we follow the smallest value that has been found up to that point. This will converge by the principle of steepest descent [3].
Proof of $O(n)$ Probes: In each function, there are $\approx 6 \cdot n$ probes (in reality, it is slightly smaller because that would indicate double counting some of the nodes), approximating to a larger number will not affect the asymptotic analysis. Each subsequent iteration has a $\frac{n-3}{2} \times \frac{n-3}{2}$ dimension matrix, which can be approximated using the slightly larger $\frac{n}{2} \times \frac{n}{2}$. This approximation allows us to calculate the number of probes by modeling it as a geometric sequence with $r = \frac{1}{2}$. The infinite sum of a geometric sequence is $\frac{a}{1-r}$.

$$\begin{aligned} \# Probes &= \frac{a}{1-r} \\ &= \frac{6 \cdot n}{1-0.5} \\ \# Probes &= \boxed{12 \cdot n \rightarrow O(n)} \end{aligned}$$

Difficulty: 8
Time: 90 Minutes

References

[1] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Pearson Education. ISBN: 978-93-325-1864-3.

[2] *Lecture 2*. http://courses.csail.mit.edu/6.006/spring11/lectures/lec02.pdf. Accessed: 2018-10-30.

[3] Donna C. Llewellyn and Craig A. Tovey. “Dividing and conquering the square”. In: *Discrete Applied Mathematics* 43.2 (1993), pp. 131–153. ISSN: 0166-218X. URL: http://www.sciencedirect.com/science/article/pii/0166218X93900048.