EECS 3101 - B Tutorial 4 Notes

Shahin Kamali

York University

September 2022

1. Given an array A of n integers and an integer x, we want to devise a divide & conquer algorithm that searches for x and returns the *first* index of x, and -1 if A does not contain x. For example, for $A = [3, 1, 3, 2, 5, 6, 7, 3, 7, 9]$, the output for $x = 3$ is 0 and for $x = 10$ is -1.
What is the best and worst-case time complexity of this algorithm?

   Answer: See the following pseudocode. The first call is Search$(A, 0, n-1)$. Note that this is a divide and conquer algorithm because the size of the sub-array at which we recurs is "divided" by a constant at each call.
   In the best case, the algorithm only recurses on the left subarray for every recursive call. For $n > 1$, the time complexity is $T(n) = T(n/2)+c$ for constant c, which solves as $T(n) = \Theta(\log n)$.
   In the worst case, we recurs on both left and right subarray, and for $n > 1$, the running time is $T(n) = 2T(n/2) + c$ for constant c; this solves as $T(n) = \Theta(n)$.

```
Search(A, lo, hi)
1.    if (lo = hi)
2.        if (A[lo] = A[hi])
3.            return lo
4.        else
5.            return −1
6.    mid ← (lo + hi)/2
7.    result ← Search(A, 0, mid)
8.    if (result = −1)
9.        result ← Search(A, mid + 1, hi)
10.   return result
```

2. A string is "palindrome" if it reads the same backward as forward, e.g., *madam*, *racecar*, *rotator*. Describe a recursive algorithm for detecting whether a given string of length n is palindrome. What is the time complexity of your algorithm?

   Answer: See the following pseudocode. The first call is IsPalindrome$(A, 0, n-1)$. Note that this is NOT a divide and conquer algorithm because the size of the sub-array at which we recurs is NOT "divided" by a constant at each call.
   In the best case, the first and the last character (at the first call) are not equal, and no recursive call is made, that is, the best-case running time is $\Theta(1)$.
   In the worst case, we recurs on a subarray of size $n-2$, and for $n > 1$, the running time is

$T(n) = T(n-2) + c$ for constant c; this solves as $T(n) = \Theta(n)$.

```
IsPalindrome(A, lo, hi)
1.    if (lo = hi) or (lo = hi − 1)
2.        if (A[lo] = A[hi])
3.            return true
4.        else
5.            return false
6.    if A[lo] = A[hi]
7.        result ← IsPalindrome(A, lo + 1, hi − 1)
8.    else
9.        result ←false
10.   return result
```

3. Given an array A of n integers, we want to report the length of the longest continuous subarray of A formed by increasing numbers. For example, for $A = [5, 15, -30, -10, -5, 40, 10]$, the longest increasing subarray is $[-30, -10, -5, 40]$, and the output is 4.
Devise a divide & conquer algorithm and analyze its time complexity.

Answer: See the following pseudocode. The first call is LCIS$(A, 0, n-1)$. The two recursive calls find the longest commons subsequences that are entirely on the left and right sub-arrays, respectively. The *helper* function finds the length of the longest commong subsequence that includes the middle point.
For the running time, note that we recurs twice on a subarrays of size $n/2$, and the running time of the helper function is $O(n)$. Therefore, for $n > 1$, the running time is $T(n) = 2T(n/2) + c$ for constant c; this solves as $T(n) = O(n)$. The running time is also $\Omega(n)$ because we are reading the entire array (each element is accessed at least once). Therefore, the running time of this method is $\Theta(n)$.

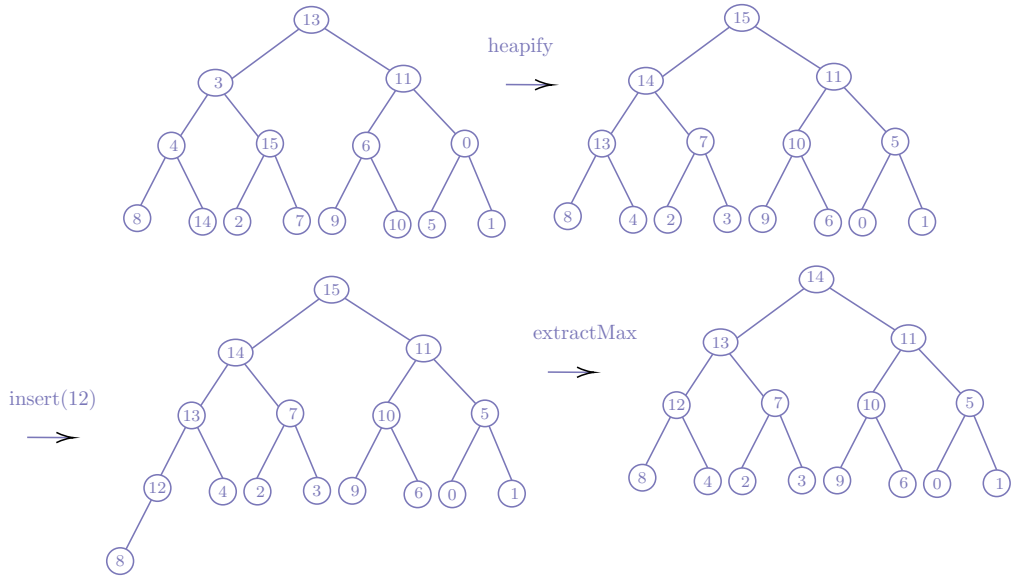
```
LCIS(A, lo, hi)
1.    if (lo = hi)
2.        return 1
3.    mid ← (lo + hi)/2
4.    option1 ← LCIS (A, lo, mid − 1)
5.    option2 ← LCIS (A, mid + 1, hi)
6.    option3 ← helper (A, mid, lo, hi)
7.    return max{option1, option2, option3}
```

```
helper(A, mid, lo, hi)
1.    result ← 1.
2.    j ← mid − 1
3.    while (j ≥ low) and (A[j] ≤ A[j + 1])
4.        result ← result + 1
5.        j ← j − 1
6.    j ← mid
7.    while (j ≤ hi − 1) and (A[j] ≤ A[j + 1])
8.        result ← result + 1
9.        j ← j + 1
10.   return result
```

4. Consider array $A = [13, 3, 11, 4, 15, 6, 0, 8, 14, 2, 7, 9, 10, 5, 1]$.

   - Apply the Heapify procedure on A.
   - On the resulting heap, apply operation *insert(12)*.
   - On the heap after insertion, apply *extract-Max*.

   Answer: Sea the figure below:



5. Justin presents the following algorithm for Heapify: Randomly shuffle the input, check if it is a correct heap (in linear time); if it is, stop and if it is not, try again (shuffle again and repeat). What is the best running time of the algorithm? What is the expected running time of the algorithm?[1]

   Answer: In the best case, after the shuffle, the resulting ordering of the numbers form a heap. Checking whether an array is in the heap form takes $\Theta(n)$; therefore, the best-case running time is $\Theta(n)$.

   For randomized algorithms, however, it is more appropriate to consider the expected running time (expectation is taken over all possible random outcomes). Let p denote the probability that a shuffled input is in the heap order. The expected running time of the algorithm is then $\Theta(p)$. To find the value of p, we note that there are $n!$ possible permutations. Among these, we have to figure how many are heaps. Let $H(n)$ denote the number of heaps. Note that the largest element must be the root of the heapy. Any other element may appear on the left or right subtree; therefore, there are $\binom{n-1}{(n-1)/2}$ ways to assign the remaining elements to left or right subtrees. For each subtree, there is $H((n-1)/2)$ possible ways to form a heap. Therefore, we can write $H(n) = \binom{n-1}{(n-1)/2} \times H((n-1)/2)^2$. One can verify that, although $H(n)$ grows exponentially, it is still exponentially slower than $n!$. Therefore, the value of $1/p$ grows exponentially, that is, the running time of Justin's algorithm is exponential to n (on average case).

[1] This question is discussed to provide a context for randomized algorithms. You do not need to study this for your exams.