Mock Exam
COMS10007 Algorithms 2018/2019

10.05.2019

1 Sorting

1. What is a comparison-based sorting algorithm? Give an example of a sorting algorithm that is not comparison-based (only mention its name).

2. Is Insertionsort stable? If yes, explain why this is. If no, illustrate this with an example.

3. Heapsort interprets the input array A of length n as a binary tree. The first step of the algorithm is to heapify the tree (turn the tree into a heap). Argue that this can be done in time $O(n \log n)$.

4. Give an example input array A of length n and a pivot selection method so that:

   (a) Quicksort runs in time $\Theta(n \log n)$ on A.

   (b) Quicksort runs in time $\Theta(n^2)$ on A.

5. Consider the following algorithm:

Algorithm 1 Sorting algorithm

```
Require: Array A of length n
  while 1 do
    if IsSorted(A) then
      return A
    end if
    A ←Next-Perm(A)
  end while
```

We assume that the instruction $A \leftarrow$ Next-Perm(A) takes time $O(1)$ and returns the *next* permutation of A so that the sequence:

$$A, \text{Next-Perm}(A), \text{Next-Perm}(\text{Next-Perm}(A)), \ldots$$

cycles through all permutations of A. The function IsSorted(A) checks whether the input array A is sorted.

(a) Explain how IsSorted(A) can be implemented to run in $O(n)$ time.

(b) What is the worst-case runtime of the algorithm?

(c) What is the best-case runtime of the algorithm?

2 O-notation

1. Give a formal proof of the following statement:

$$4n + \frac{1}{2}n^2 \in O(6n^2) \ .$$

2. Consider two functions f, g with $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$. Does this imply that $f(n) \in \Theta(g(n))$? (no justification needed)

3. Let f be a function with $f(n) \geq 2$ for all n and $f(n) \in O(n)$. Prove that $\log(f(n)) \in O(\log n)$.

4. Order the following sets so that each is a subset of the one that comes after it:

$$O(n^2 + \log n), O((\log n)^n), O(7), O(\sqrt{2^{\log\log n}}), O(3^n), O(n^2 \log n), O(\log(n) - \log\log(n))$$

3 Algorithmic Design Principles

1. In the lecture we discussed a $O(\log n)$ time algorithm for finding a peak in a one-dimensional array A of length n. The algorithm checks whether the central element at position $\lfloor n/2 \rfloor$ is a peak. If it is then we are done. If it isn't then the algorithm recursively looks for a peak either in the left or the right half of the input array.

   Explain how the algorithm decides whether to recurse on the left or on the right half. Furthermore, give an example array A so that if we always recursed on the left half then the algorithm would not find a peak in A.

2. Consider the following recurrence:

$$f(1) = 2, f(2) = 4, f(3) = 7, \text{ and } f(n) = f(n-1) + f(n-2) + f(n-3) \text{ for } n \geq 4 \ .$$

   Use the substitution method to show that $f(n) = O(C^n)$, for some constant C (state such a constant explicitly). Show that the smallest possible value for C can be determined by a cubic equation (no need to solve the equation).

3. Consider the following algorithm:

Algorithm 2 Algorithm ALG

Require: Integer array A of length n
  if $n = 1$ **then**
    return $A[0]$
  else
    return $\max\{ALG(A[0, \lfloor n/2 \rfloor - 1]), ALG(A[\lfloor n/2 \rfloor, n-1])\}$
  end if

   We denote by $A[i, j]$ the subarray $A[i], A[i+1], \ldots, A[j]$.

   (a) Draw the recursion tree that corresponds to the invocation of ALG on the array $A = 7, 1, 22, 4, 8, 6, 3$. Annotate each node of the recursion tree with the value returned by the function call that corresponds to this node.

   (b) What is the runtime of this algorithm? Justify your answer.

(c) Describe (in plain English, no code or pseudo-code) a non-recursive algorithm with runtime $O(n)$ that computes the exact same output.

4. Suppose that we have an infinite supply of coins of values $1, 3, 5$ and 7. Given a number n, the goal is to select the least number of coins possible whose values sum up to n. For example, if $n = 8$ then one coin of value 7 and one coin of value 1 suffices (or one coin of value 3 and one coin of value 5).

   Let $T(n)$ be the smallest number of coins needed to make up the value n. Then, $T(1) = T(3) = T(5) = T(7) = 1$, $T(2) = T(4) = T(6) = T(8) = 2$, and $T(15) = 3$.

   Describe a dynamic programming algorithm that computes $T(n)$ bottom-up. What is the runtime of your algorithm? What is the recursive definition of $T(n)$ used in your algorithm?