Mock Exam - Answers
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).

   Answer: A comparison-based sorting algorithm determines the order of the array elements by repeatedly comparing two selected elements of the input array, i.e., for two elements $A[i]$ and $A[j]$ with $i \neq j$, the algorithm is only allowed to use the answer to the query $A[i] < A[j]$ (or $A[i] \leq A[j]$ or similar). No other operation on the array elements (except moving them) is allowed. For example, Countingsort is not a comparison-based sorting algorithm.

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

   Answer: Insertionsort is stable. It iterates through the array elements from left to right and inserts the current element $A[i]$ at the correct position in the already sorted prefix array $A[0, i-1]$. The insertion operation does not break stability: The current element x can always be inserted to the right of any element with the same value as x.

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 $O(n \log n)$.

   Answer: We need to ensure that the heap property is fulfilled at every node. To this end, we iterate through the array from right to left, which corresponds to iterating through the tree nodes from right to left and bottom to top. For every internal node x, we check whether the heap condition is fulfilled. If it is not, we switch the positions of x and its child with the larger value in the tree, thus moving x one level down in the tree. If the heap condition is still not satisfied for x we exchange positions of x and its child with the larger value again. Repeating this process, the heap condition for x will eventually be fulfilled: Either x remains an internal node and the heap condition is fulfilled, or it becomes a leaf (the heap condition is always fulfilled at a leaf).
   Since each node can move at most $O(\log n)$ steps in the tree (a complete binary tree has height $O(\log n)$), and the tree has $O(n)$ nodes, the runtime of this procedure is $O(n \log n)$.

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.

Answer: Since the best-case runtime of Quicksort is $\Theta(n \log n)$ (i.e., we do not have to worry that our algorithm runs faster than $\Theta(n \log n)$), we can pick an arbitrary input sequence (let's pick the already sorted input sequence $1, 2, 3, 4, 5, \ldots, n$) and we select a uniform random element as the pivot. As argued in the lecture, Quicksort runs in expected time $O(n \log n)$ when choosing a uniform random element as the pivot. Alternatively, we can run a linear time median selection algorithm as the method for picking a pivot. There are other examples: For example, we can select the already sorted input sequence $1, 2, 3, 4, 5, \ldots, n$ and select the element at position $A[\lceil n/2 \rceil]$ as the pivot.

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

Answer: We use the already sorted input sequence $1, 2, 3, 4, \ldots, n$ and we select the right-most element as the pivot. Observe that in the divide step of the algorithm, the input of length n is split into a subproblem of length $n-1$ and one of length 1. This happens for every subproblem of length larger than 1. If only bad splits happen, the runtime is $\Theta(n^2)$.

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.

Answer: We go through A from left to right starting at position 1 (and ignoring position 0) and we check for every element $A[i]$ whether $A[i] \geq A[i-1]$ holds. If this is not the case, we return false. If none of the checks evaluated to false, we return true. The check takes time $O(1)$ and since we perform this check $O(n)$ times, the runtime is $O(n)$.

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

Answer: The algorithm cycles through all permutations of the input array and checks whether the current permutation is sorted. In the worst case, the sorted permutation is the last permutation considered. Since there are $n!$ permutations of the input array, and IsSorted(A) takes time $O(n)$ (in fact $\Theta(n)$), the worst-case runtime is $\Theta(n \cdot n!)$.

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

Answer: In the best case, the input is already sorted. In this case, the algorithm runs in time $O(n)$. The best-case runtime therefore is $\Theta(n)$.

2 O-notation

1. Give a formal proof of the following statement:

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

Answer: We need to show that there are constants c, n_0 such that $4n + \frac{1}{2}n^2 \leq c \cdot 6n^2$ holds for every $n \geq n_0$. We compute:

$$\begin{aligned} 4n + \frac{1}{2}n^2 &\leq c \cdot 6n^2 \\ 4 &\leq (6c - \frac{1}{2})n \\ \frac{4}{6c - \frac{1}{2}} &\leq n \ . \end{aligned}$$

Hence, we can for example choose $c = 1$ and $n = 1$, since $\frac{4}{6 \cdot 1 - \frac{1}{2}} \leq 1$.

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)

Answer: Yes.

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

Answer: We need to show that there are constants c, n_0 such that $\log(f(n)) \leq c \cdot \log n$, for every $n \geq n_0$. Since $f(n) \in O(n)$, we know that there are constants c', n_0' such that $f(n) \leq c'n$, for every $n \geq n_0'$. Since $f(n) \leq c'n$, we obtain $\log(f(n)) \leq \log(c'n) = \log(c') + \log(n)$. We require that $\log(c') + \log(n) \leq c \log n$ or $c \geq \frac{\log(c')}{\log n} + 1$. The function $\frac{1}{\log n}$ is decreasing in n and thus takes its largest value at $n = n_0'$. We can hence set $c = \frac{\log(c')}{\log(n_0)'} + 1$ and $n_0 = n_0'$.

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

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

Answer:

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

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.

Answer: The algorithm recurses on the side where the larger of the two neighboring elements to the central element lies. That is, if $A[\lfloor n/2 \rfloor - 1] > A[\lfloor n/2 \rfloor + 1]$ then the algorithm recurses on the left side, otherwise on the right side.
Consider for example the array $1, 2, 3, 4, \ldots, n$. This array contains a single peak, which is the right-most element. If we recurse on the left side, then the algorithm will never consider the right-most element and thus not find it.

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).

Answer: Our guess is $f(n) \leq c \cdot C^n$, for constants c, C. We plug our guess into the recurrence:

$$f(n) = f(n-1) + f(n-2) + f(n-3) \leq cC^{n-1} + cC^{n-2} + cC^{n-3} \ .$$

It is required that the right side of the prevous inequality is bounded by cC^n. This yields the following inequality:

$$\begin{aligned} cC^{n-1} + cC^{n-2} + cC^{n-3} &\leq cC^n \\ C^2 + C + 1 &\leq C^3 \ . \end{aligned} \tag{1}$$

The smallest value for C is the solution of the cubic equation $C^2 + C + 1 = C^3$. One potential value for C that fulfills the inequality given in Inequality 1 is 2. We will thus use the value $C = 2$. It remains to verify the base cases. We have: $C^1 = 2 \geq f(1)$, $C^2 = 4 \geq f(2)$, and $C^3 = 8 \geq f(3)$. We can hence set $c = 1$ and all base cases are still fulfilled. We thus proved that $f(n) \leq 2^n$, for every $n \geq 1$. This implies that $f(n) \in O(2^n)$.

3. Consider the following algorithm:

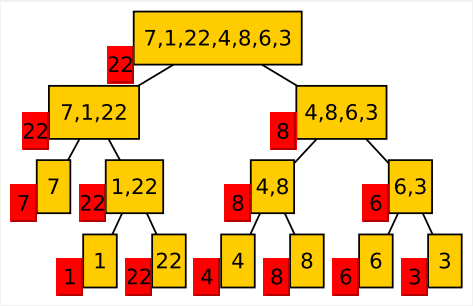
Algorithm 2 Algorithm ALG

```
Require: Integer array A of length n
  if n = 1 then
    return A[0]
  else
    return max{ALG(A[0, ⌊n/2⌋ − 1]), ALG(A[⌊n/2⌋, 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.

Answer:



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

Answer: The runtime is $O(n)$ (even $\Theta(n)$). Each invocation of the algorithm takes time $O(1)$ without the recursive calls. Overall, there are $O(n)$ recursive calls: Observe that the recursion tree has height at most $\log(n)+1$, since the size of the array roughly halves every step. The recursion tree is a binary tree. A binary tree of height k has at most $2^{k+1}-1$ nodes. Hence, the recursion tree has at most $2^{\log(n)+2}-1 = 4n-1$ nodes. The runtime is therefore $O(1) \cdot O(4n-1) = O(n)$.

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

Answer: The algorithm simply computes the maximum in the input array. This can be done non-recursively as follows: We set an auxiliary variable x equal to $A[0]$. We then go through the array from position 1 to $n-1$ and check whether the current element $A[i]$ is larger than x. If it is, then we update x and set it to $A[i]$. At the end of the loop, we return x.

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?

Answer: We use the following recursive definition: (for every $n \geq 8$)

$$T(n) := 1 + \min\{T(n-1), T(n-3), T(n-5), T(n-7)\} ,$$

and we use the values $T(1) = T(3) = T(5) = T(7) = 1$ and $T(2) = T(4) = T(6) = 2$. This can be implemented in a dynamic programming algorithm as follows: We define an array A of size $n+1$. We initialize the values $A[i] = T(i)$, for every $i \in \{1, 2, 3, 4, 5, 6, 7\}$. Then, we iterate from $i = 8$ to n and compute the value $A[i]$ by the formula: $A[i] = 1 + \max\{A[i-1], A[i-3], A[i-5], A[i-7]\}$. The runtime of this algorithm is $O(n)$, since computing each value $A[i]$ takes time $O(1)$ and there are $O(n)$ values to compute.