COMP523 Tutorial 1 - Solutions*

Coordinator: Aris Filos-Ratsikas Demonstrator: Michail Theofilatos

September 26, 2019

J. Kleinberg and E. Tardos, Chapter 2, Exercise 1

Solution

When the input size is doubled, the algorithms become slower by **(a)** 4 times, **(b)** 8 times, **(c)** 4 times, **(d)** 2 times and $2n$ operations and **(e)** a number of times equal to the previous running time.

When the input size is increased by one, the algorithms become slower by **(a)** $2n + 1$ operations, **(b)** $3n^2 + 3n + 1$ operations, **(c)** $200n + 100$ operations, **(d)** $\log(n + 1) + n\,[\log(n + 1) - \log n]$ operations and **(e)** 2 times.

J. Kleinberg and E. Tardos, Chapter 2, Exercise 2

Solution

In one hour, the computer can perform $3600 \cdot 10^{10}$ operations. Therefore, the running time is bounded by this number, which immediately yields the largest input size possible:

(a) $6 \cdot 10^6$.

(b) 33019.

(c) $600,000$.

(d) Approximately $9 \cdot 10^{11}$ (assuming the logarithm has base 2).

(e) 45.

(f) 5.

J. Kleinberg and E. Tardos, Chapter 2, Exercise 3

Solution

The functions in ascending order of growth are as follows:

$f_2(n)$, $f_3(n)$, $f_6(n)$, $f_1(n)$, $f_4(n)$, $f_5(n)$.

The idea here is that amongst polynomial functions, those with the smallest exponents have the smallest growth. Among exponential functions, those with the smallest bases have the smallest growth. The $\log n$ factor in $f_6(n)$ indicates that the function grows faster than n^2, but slower than n^α, for any $\alpha > 2$, since logarithms grow slower than polynomials.

*The solutions contain additional explanations that are not necessary, if you were to answer such a question in an exam.

J. Kleinberg and E. Tardos, Chapter 2, Exercise 6, Parts a and b

Solution

(a) We will choose the function $f(n) = n^3$ and prove that the running time of the algorithm is $O(n^3)$. First, notice that the outer loop runs for exactly n steps. The inner loops runs for steps $i, i+1, \ldots n$, which are at most n steps. Since these loops are nested, the number of steps taken to execute them is bounded by n^2. Inside each execution of the inner loop, there is an addition of the array entries from $A[i]$ to $A[j]$, which requires at most $c \cdot (j - i + 1)$ operations, for some constant c, which is obviously at most cn operations. The store operations for $B[i, j]$ take constant time. Overall, the running time of the algorithm is $O(n^3)$.

(b) Consider the case when $i \leq n/4$ and $j \geq 3n/4$. The addition of array entries $A[i]$ through $A[j]$ would require at least $j - i + 1$ operations, which, by those numbers are at least $n/4$ operations. There are $\left(\frac{n}{4}\right)^2$ choices of i and j that satisfy the constraints above, and therefore the inner loop will be accessed at least $\frac{n^2}{4}$ times. Overall, the algorithm will perform at least $\frac{n}{2} \cdot \frac{n^2}{4}$ operations, and its running time is $\Omega(n^3)$.

Problem 1

Sort the following functions according to the O ("big oh") and o ("small oh") order:

$\log n^{1/2}$, $\log(9n)$, $\log n^3$, $2^{\log n}$, $2^{3 \log n}$, $2^{\log(9n)}$, n^2, $n \log n$.

Solution

Let LOG $= \{\log n^{1/2}, \log(9n), \log n^3\}$. For any $f(n), g(n) \in$ LOG, it holds that $f(n) = O(g(n))$, i.e., the functions are *asymptotically equal*. It also holds that for any function $f(n) \in$ LOG and any function $g(n) \notin$ LOG from those given, $f(n) = o(g(n))$.

It also holds that $2^{\log n}$ and $2^{\log(9n)}$ are both $O(n)$ and therefore are asymptotically equal. Since n is $o(n \log n)$ and $o(n^2)$, it also holds that $2^{\log n}$ and $2^{\log(9n)}$ are both $o(n \log n)$ and $o(n^2)$. In turn, $n \log n = o(n^2)$. Finally, $n^2 = o(2^{3 \log n})$, since $2^{3 \log n} = n^3$ and $n^2 = o(n^3)$.

Problem 2

Recall that a *majority element* in an array of n numbers is one that appears more than $\lceil n/2 \rceil$ times. Design an algorithm that receives as input a *sorted* array A of integers and outputs YES if a majority element exists and NO otherwise. Present the algorithm in terms of pseudocode. The algorithm should run in (worst-case) time $O(\log n)$ and you should formally prove its asymptotic running time. For simplicity, you may ignore issues regarding whether numbers are divisible by 2 (the algorithms can be adjusted to account for that via the appropriate use of the $\lceil \cdot \rceil$ function).

Solution

We will use the BINARYSEARCH procedure that we saw in the lectures as a subroutine of our algorithm. A key observation is that if an element x is a majority element, then there must be an occurrence of x in the middle of the array, at position $n/2$. Therefore, we will first check whether the $(n/2)$'th element is in fact x. Then, if that is indeed the case, we can perform binary search to the subarrays to the left and to the right of the $n/2$'th element, to find the *first* and *last* occurrence of the value x. Then, we can simply subtract the two indices, and check whether the result is larger than $n/2$ or not.

To formalise this idea, we have pseudocode of Algorithm 1. Note that the algorithm uses two binary search procedures, one to find the *earliest* occurrence of x in the array and one to find the *latest* occurrence of x in the array.

Algorithm 1 Majority in Sorted Array

```
1: procedure BinarySearchLeft(x, i, j)
2:     if i = j then return i;
3:     else
4:         if x = A[(i+j)/2] then BinarySearchLeft(x, i, (i+j)/2)
5:         else BinarySearchLeft(x, (i+j)/2 + 1, j)
6: procedure BinarySearchRight(x, i, j)
7:     if i = j then return i;
8:     else
9:         if x = A[(i+j)/2] then BinarySearchRight(x, (i+j)/2, j)
10:        else BinarySearchRight(x, i, (i+j)/2 − 1)
11: procedure Majority(A)
12:    if BinarySearchRight(A[n/2], n/2, n)-BinarySearchLeft(A[n/2], 1, n/2 − 1) > n/2 then
13:        Return yes;
14:    else Return no;
```

In the lectures, we proved that BinarySearch runs in time $O(\log n)$ and the remaining operations of the algorithm run in constant time, so the overall running time is $O(\log n)$. We provide the formal analysis below.

BinarySearchLeft and BinarySearchRight have the same worst-case running time, so we will only perform the analysis for one of them. In each recursive call of the algorithm, there is a constant number of operations (e.g., checking if two elements are equal or returning an element), so the asymptotic complexity will be given by the number of times that the procedure is called. Also note that in each recursive call, the size of the input to the procedure is halved. More precisely, if $T(n)$ is the running time of the procedure on input size n, we can write

$$T(n) = T(n/2) + c,$$

where c is a constant number for the remaining operations. This gives us a recursive equation, which we can solve to obtain the value of $T(n)$. We will proceed by induction.

To be proven: $T(n) \leq 2c \log n$

Base Case: $n = 2$: Straightforward, $T(2) \leq 2c \leq 2c \log 2$.
Induction Hypothesis: Suppose $T(n/2) \leq 2c \log(n/2)$.
Inductive Step: We have that

$$\begin{aligned} T(n) &= T(n/2) + c \\ &\leq 2c \log(n/2) + c \quad (1) \\ &\leq 2c \log n + c - c \quad (2) \\ &= 2c \log n \end{aligned}$$

where Inequality 1 follows from the Induction Hypothesis and Inequality 2 follows from the fact that $\log(n/2) = \log(n) - \log 2$. Therefore, the two BinarySearch procedures together take time at most $4c \log n$ and the Majority algorithm also makes an additional subtraction and comparison, which only take constant time. Therefore, the running time of the algorithm is $O(\log n)$.