CSCE 222-200 Discrete Structures for Computing

Fall 2024

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: `chen@cse.tamu.edu`
Office Hours: T+R 2:00pm–3:30pm

Teaching Assistant: Evan Kostov
Office: EABC Cubicle 6
Phone: (469) 996-5494
Email: `evankostov@tamu.edu`
Office Hours: MW 4:00pm-5:00pm

Assignment #3 Solutions

1. Give a big-O estimate for the number of number additions (i.e., the additions in the fourth statement $t = t + i + j$) in the following algorithm.

> $t = 0$;
> **for** $(i = 1; i \leq n; i{+}{+})$
>   **for** $(j = 1; j \leq n; j{+}{+})$
> $t = t + i + j$.

Solution. The for-loop "**for** $(i = 1; i \leq n; i{+}{+})$" iterates over n values of i, for $i = 1, 2, \ldots, n$. For each iterated value of i, the algorithm iterates in the for-loop "**for** $(j = 1; j \leq n; j{+}{+})$" over n values of j, for $j = 1, 2, \ldots, n$. As a result, for each value of i, the for-loop "**for** $(j = 1; j \leq n; j{+}{+})$" executes the statement "$t = t{+}i{+}j$" exactly n times. Since the algorithm executes the for-loop "**for** $(i = 1; i \leq n; i{+}{+})$" for n values of i, it executes the statement "$t = t + i + j$" in total $n \cdot n = n^2$ times. Since each execution of the statement "$t = t + i + j$" has two additions, we conclude that the algorithm executes $2n^2 = O(n^2)$ additions.

2. Give a big-O estimate for the number of arithmetic operations (i.e., additions and multiplications) in the following algorithm. What is the value of t at the end of the algorithm?

> $i = 1; \quad t = 0;$
> **while** $(i \leq n)$
> $\{\, t = t + i; \;\; i = 2i \,\}$

Solution. The only two arithemtic operations (one addition plus one multiplication) occur in the statement "$\{\, t = t{+}i;\ i = 2i \,\}$" in the while-loop in the algorithm. Thus, we only need to count how many times the statement is executed. The value i in the while-loop starts with $i = 1$, doubled in each execution of the statement "$\{\, t = t + i;$ $i = 2i \,\}$", and ends when the condition $i \leq n$ no longer holds. Therefore, for the following values of i: $1 = 2^0$, $2 \cdot 2^0 = 2^1$, $2 \cdot 2^1 = 2^2$, …, $2^r \leq n$, the statement

"$\{ t = t + i;\ i = 2i \}$" is executed, where r is the largest integer such that $2^r \le n$. Taking the logarithms on both sides of $2^r \le n$, we get $r \le \log_2 n$. Therefore, the largest integer satisfying $2^r \le n$ is $r = \lfloor \log_2 n \rfloor$.

Thus, the statement "$\{ t = t + i;\ i = 2i \}$" is executed by the algorithm exactly $\lfloor \log_2 n \rfloor$ times. Since each execution has two arithmetic operations, the total number of arithmetic operations executed by the algorithm is $2 \cdot \lfloor \log_2 n \rfloor = O(\log_2 n)$.

Now consider the value of t. As we discussed above, the while-loop is executed for the values $i = 1, 2, 2^2, \ldots, 2^{\lfloor \log_2 n \rfloor}$, while the statement "$\{ t = t+i;\ i = 2i \}$" adds each of these values to t, where t starts with an initial value 0 (by the statement "$t = 0$" in the first line). Therefore, at the end of the algorithm, the value of t is (using the summation formula of geometric sequences)

$$0 + 2^0 + 2^1 + \cdots + 2^{\lfloor \log_2 n \rfloor} = (2^{\lfloor \log_2 n \rfloor + 1} - 1)/(2 - 1) = 2^{\lfloor \log_2 n \rfloor + 1} - 1.$$

3. How much time does an algorithm take to solve a problem of size n if this algorithm uses $2n^2 + 2^n$ operations, each requiring 10^{-9} seconds, with each of the following values of n?

a) 20 **b**) 50 **c**) 100 **d**) 200

Solution. This question is to show that if the running time of an algorithm is of order such as 2^n, then even for reasonably small input sizes, the algorithm would become impractical.

(a) For value $n = 20$, we have $2n^2 + 2^n = 2 \cdot 20^2 + 2^{20} = 1,049,376$. Thus, the required time is $(2n^2 + 2^n) \cdot 10^{-9} = 0.01049376 \approx 0.01$ seconds (very fast).

(b) For value $n = 50$, we have $2n^2 + 2^n = 2 \cdot 50^2 + 2^{50} = 1,125,899,906,847,624$. Thus, the required time is $(2n^2 + 2^n) \cdot 10^{-9} \approx 1,125,900$ seconds ≈ 13 days (quite long, but probably still doable).

(c) For value $n = 100$, we have $2n^2 + 2^n = 2 \cdot 100^2 + 2^{100} \approx 1,267,650,600 \cdot 10^{21}$. Thus, the required time is $(2n^2 + 2^n) \cdot 10^{-9} \approx 1.268 \cdot 10^{21}$ seconds $\approx 40 \cdot 10^{12}$ years $= 40,000,000,000,000$ years (can you wait for the algorithm to finish?).

(d) For value $n = 200$, we have $2n^2 + 2^n = 2 \cdot 200^2 + 2^{200} \approx 1,606,938,044 \cdot 10^{51}$. Thus, the required time is $(2n^2 + 2^n) \cdot 10^{-9} \approx 1.6 \cdot 10^{51}$ seconds $\approx 5.1 \cdot 10^{43}$ years $= 51,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000$ years (will the Earth still exist?).

4. Devise an algorithm that on an array $A[1..n]$ of n integers prints out (using a statement `print(`i`)`) all indices i such that $A[i] > A[1] + A[2] + \cdots + A[i-1]$. What is the time complexity of your algorithm in terms of big-O notation?

Solution. The algorithm is given as follows.

$t = 0$;
for $(i = 1; i \le n; i{+}{+})$
  if $(A[i] > t)$ `print`(i);
  $t = t + A[i]$.

The variable t is used to hold the value of $A[1] + A[2] + \cdots + A[i-1]$ when the for-loop "**for** $(i = 1; i \le n; i{+}{+})$" reaches the current value of i. Thus,, t is initialized to 0 (in the first line of the algorithm), and increased by $A[i]$ when the element $A[i]$ is processed (in the fourth line of the algorithm, so to get ready for processing the next element $A[i+1]$). Moreover, when the algorithm processes the element $A[i]$, it compares the values of $A[i]$ and t, which is equal to $A[1] + A[2] + \cdots + A[i-1]$, and prints the index i if $A[i]$ is larger than $t = A[1] + A[2] + \cdots + A[i-1]$ (in the third line of the algorithm), as required by the question. As a result, the algorithm does exactly what the question asks for.

Now we consider the time complexity of the algorithm. Line 1 of the algorithm takes constant time, i.e., runs in time $O(1)$. For each value of i, the algorithm runs in time $O(1)$ in lines 3-4 (doing a comparison, a possible printing, and an addition). Since the for-loop "**for** $(i = 1; i \le n; i{+}{+})$" iterates exactly n times, we conclude that the time complexity of the algorithm is

$$O(1) + n \cdot O(1) = O(n).$$

5. Devise an algorithm for finding the first and second largest elements in an array $A[1,,n]$ of n integers. What is the time complexity of your algorithm in terms of big-O notation?

Solution. The algorithm is given as follows (assuming $n \ge 2$).

if $(A[1] > A[2])$
then $\{\ \max 1 = A[1];\ \max 2 = A[2]\ \}$
else $\{\ \max 1 = A[2];\ \max 2 = A[1]\ \}$;
for $(i = 3; i \le n; i{+}{+})$
  if $(A[i] > \max 1)$
  then $\{\ \max 2 = \max 1;\ \max 1 = A[i];\ \}$
  else if $(A[i] > \max 2)$ **then** $\max 2 = A[i]$.

The algorithm uses two variables $\max 1$ and $\max 2$, which hold, respectively, the largest and the second largest elements that have been processed so far. Initially for $i = 2$, $\max 1$ and $\max 2$ hold, correctly and respectively, the larger and the smaller of $A[1]$ and $A[2]$ (see lines 1-3 of the algorithm). The for-loop "**for** $(i = 3; i \le n; i{+}{+})$" in line 4 starts with $i = 3$ and iterates $n - 2$ times.

In an iteration of the for-loop for a value i, if $A[i] > \max 1$, then since $\max 1$ is the largest in $\{A[1], A[2], \ldots, A[i-1]\}$, $A[i]$ is the largest and $\max 1$ becomes the second largest in $\{A[1], A[2], \ldots, A[i-1], A[i]\}$. The algorithm records this change accordingly in line 6. On the other hand, if in line 7 the condition $A[i] > \max 2$ is satisfied, then (because of line 5) we have both $A[i] \leq \max 1$ and $A[i] > \max 2$, so $\max 1$ remains the largest but the element $A[i]$ becomes the second largest in $\{A[1], A[2], \ldots, A[i-1], A[i]\}$. This change is recorded by the algorithm in line 7. Note that the condition that both ($A[i] \leq \max 1$ and $A[i] \leq \max 2$ hold needs no change because in this case, $\max 1$ and $\max 2$ remain, respectively, as the largest and the second largest in $\{A[1], A[2], \ldots, A[i-1], A[i]\}$.

This shows that the algorithm solves the given problem correctly.

The time complexity of the algorithm is obvious. Lines 1-3 of the algorithm take time $O(1)$ (doing one comparison and two value assignments). The for-loop in line 4 iterates $n-2$ times, each again takes time $O(1)$ for doing one or two comparisons plus one or two value assignments. As a result, we conclude that the time complexity of the algorithm is

$$O(1) + (n-2) \cdot O(1) = O(n).$$