CSE530: Algorithms & Complexity
Solutions to Exercise Set 1

Antoine Vigneron

March 5, 2018

Question 1. For each function $f(n)$ below, give a tight bound using the $\Theta(\cdot)$ notation. Your answer should be chosen from the following list: $\Theta(1)$, $\Theta(n)$, $\Theta(\log n)$, $\Theta(n \log n)$, $\Theta(n!)$, $\Theta(n^2)$, $\Theta(2^n)$, $\Theta(n^3)$, $\Theta(\sqrt{n})$, $\Theta(n \log^2 n)$, $\Theta(3^n)$. No justification is needed.

	$f(n)$	Answer
(a)	$10(n+1)(n-5)$	$\Theta(n^2)$
(b)	$n^2 + 2^n$	$\Theta(2^n)$
(c)	$n \log(n^2)$	$\Theta(n \log n)$
(d)	$\sqrt{n} + \log n$	$\Theta(\sqrt{n})$
(e)	$10^{23} + \log n$	$\Theta(\log n)$
(f)	$n \log_9 n + n^2$	$\Theta(n^2)$
(g)	$(\sqrt{n} \log n)^2$	$\Theta(n \log^2 n)$
(h)	$(n+1)^3 - (n-1)^3$	$\Theta(n^2)$
(i)	$\dfrac{n+\sqrt{n}}{n+1}$	$\Theta(1)$
(j)	$2n + n \sin(n)$	$\Theta(n)$

2. Let $f(n)$ and $g(n)$ be two functions that are nonnegative for every $n \geqslant n_0$, where n_0 is a constant. Using the basic definition of the Θ-notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Answer. First observe that $\max(x, y) \leqslant x + y \leqslant 2\max(x, y)$ for any two nonnegative numbers x, y. Indeed, if $x = \max(x, y)$, then $x \leqslant x + y \leqslant 2x$, and if $y = \max(x, y)$, then $y \leqslant x + y \leqslant 2y$.

So for every $n \geqslant n_0$, we have

$$0 \leqslant \max(f(n), g(n)) \leqslant f(n) + g(n) \leqslant 2 \max(f(n), g(n)).$$

It means that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

3. Suppose that $f(n) = O(g(n))$, and that there exists a constant n_1 such that $n \geqslant n_1$ implies that $f(n) \geqslant 2$ and $g(n) \geqslant 2$. Prove that $\log f(n) = O(\log g(n))$. (Remember that in this course, log means $\log_2$.)

Answer. By definition of the O notation, there exist positive constants c and n such that $n \geqslant n_0$ implies that $f(n) \leqslant cg(n)$.

Let $n_2 = \max(n_0, n_1)$, and let $c_2 = \max(c, 1)$. Then $n \geqslant n_2$ implies that $f(n) \geqslant 2$, $g(n) \geqslant 2$ and $f(n) \leqslant cg(n)$. Therefore,

$$\begin{aligned}
\log(f(n)) &\leqslant \log(c_2 g(n)) \\
&= \log(g(n)) + \log c_2 \\
&= \left(1 + \frac{\log c_2}{\log g(n)}\right) \log g(n) \\
&\leqslant \left(1 + \frac{\log c_2}{\log 2}\right) \log g(n) && \text{because } g(n) \geqslant 2 \text{ and } \log c_2 \geqslant 0 \\
&= (1 + \log c_2) \log g(n)
\end{aligned}$$

In addition, $\log f(n) \geqslant 0$ because $f(n) \geqslant 2$. So we have found positive constants n_2 and $c_3 = 1 + \log c_2$ such that $n \geqslant n_2$ implies that

$$0 \leqslant \log f(n) \leqslant c_3 \log g(n).$$

It means that $\log f(n) = O(\log g(n))$.

4. Given an input array $A[1 \ldots n]$ and a key x, the *searching problem* is to decide whether x is stored in A and if so, return i such that $x = A[i]$. The brute force approach for searching, which scans through the whole input array, is called *linear search.*

Write the pseudocode for linear search, and prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties

Answer. The pseudocode is given as Algorithm 1.

Algorithm 1 Linear Search

```
1: procedure LinearSearch($A, 1, n, x$)        ▷ Searches for $x$ in $A[1 \ldots n]$
2:     for $i \leftarrow 1, n$ do
3:         if $A[i] = x$ then
4:             return $i$
5:     return NotFound
```

We use the following **loop invariant**: For every $i \geqslant 2$, at the start of the ith iteration of the loop, the subarray $A[1 \ldots i-1]$ does not contain x.

We now prove the three properties of this loop invariant.

Initialization. We only reach the second iteration of the loop if $A[1] \neq x$. It means that the subarray formed by the single cell $A[1]$ does not contain x.

Maintenance. Suppose that the invariant is true before the ith iteration of the loop for $i \geqslant 2$. So the subarray $A[1, \ldots, i-1]$ does not contain x. We then reach the start of the $i+1$th iteration of the loop if and only if $A[i] \neq x$. So $A[1, \ldots, i]$ does not contain x.

Termination. When the loop terminates, we have $i = n + 1$, and thus $A[1, \ldots, n]$ does not contain x. On the other hand, if the loop does not terminate, it means that $A[i] = x$ and we return i at line 4, which is the correct answer.

5. Consider the problem of adding two n-bit binary integers, stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $(n + 1)$-element array C . State the problem formally and write pseudocode for adding the two integers.

Answer. A formal statement of the problem is below. It could also be stated using a loop invariant.

- **INPUT:** Two arrays $A[1 \ldots n]$ and $B[1, \ldots n]$ of n bits.
- **OUTPUT:** The array $C[1, \ldots n + 1]$ of $n + 1$ bits such that

$$\sum_{i=0}^{n} 2^i C[i+1] = \sum_{i=0}^{n-1} 2^i (A[i+1] + B[i+1]).$$

Two possible pseudocodes are given as Algorithm 2. and 3

Algorithm 2 Binary Addition

```
procedure BinaryAddition($A, B, n$)
    Create a new array $C[1, \ldots, n+1]$
    carry $\leftarrow 0$
    for $i \leftarrow 1, n$ do
        if $A[i] + B[i] +$ carry $\leqslant 1$ then
            $C[i] \leftarrow A[i] + B[i]$
            carry $\leftarrow 0$
        if $A[i] + B[i] +$ carry $= 2$ then
            $C[i] \leftarrow 0$
            carry $\leftarrow 1$
        if $A[i] + B[i] +$ carry $= 3$ then
            $C[i] \leftarrow 1$
            carry $\leftarrow 1$
    $C[i+1] \leftarrow$ carry
```

6. How can we modify almost any algorithm to have a good best-case running time?

Algorithm 3 Binary Addition

```
procedure BinaryAddition($A, B, n$)
    Create a new array $C[1, \dots, n+1]$
    carry $\leftarrow 0$
    for $i \leftarrow 1, n$ do
        $C[i] \leftarrow (A[i] + B[i] + \text{carry}) \bmod 2$          ▷ remainder
        carry $\leftarrow (A[i] + B[i] + \text{carry})$ div 2          ▷ integer division
    $C[n+1] \leftarrow$ carry
```

Answer. We can hardcode the solution to a special case. Then the algorithm first checks if the input is this special case, then we return the solution immediately. If we are not in this special case, we run the original algorithm.

Below is an example for merge sort. We check whether the array is already sorted, in which case we don't need to do anything. Otherwise we run merge sort. This new version of merge sort runs in linear time in the best case, and in time $\Theta(n \log n)$ in the worst case.

Algorithm 4 Modified merge sort

```
procedure ModifiedMergeSort($A[1 \dots n]$)
    for $i \leftarrow 1, n-1$ do
        if $A[i] > A[i+1]$ then
            return MergeSort($A, 1, n$)
```