COMP 3804 — Solutions Assignment 1

Some useful formulas:

1. for any real number $x > 0$, $x = 2^{\log x}$.
2. for any real number $x \neq 1$ and any integer $k \geq 1$,

$$1 + x + x^2 + \ldots + x^{k-1} = \frac{x^k - 1}{x - 1}.$$

Question 1: Write your name and student number.

Solution: James Bond, 007

Question 2: Solve the following recurrences using the unfolding method we have seen in class. In each case, give the final answer using Big-O notation.

(2.1)

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 1 + 2 \cdot T(n/3) & \text{if } n \geq 3. \end{cases}$$

You may assume that n is a power of 3.

Solution: Write $n = 3^k$. Unfolding gives

$$\begin{aligned}
T(n) &= 1 + 2 \cdot T(n/3) \\
&= 1 + 2\left(1 + 2 \cdot T(n/3^2)\right) \\
&= (1 + 2) + 2^2 \cdot T(n/3^2) \\
&= (1 + 2) + 2^2\left(1 + 2 \cdot T(n/3^3)\right) \\
&= (1 + 2 + 2^2) + 2^3 \cdot T(n/3^3) \\
&= (1 + 2 + 2^2) + 2^3\left(1 + 2 \cdot T(n/3^4)\right) \\
&= (1 + 2 + 2^2 + 2^3) + 2^4 \cdot T(n/3^4) \\
&\vdots \\
&= (1 + 2 + 2^2 + 2^3 + \ldots + 2^{k-1}) + 2^k \cdot T(n/3^k) \\
&= (2^k - 1) + 2^k \cdot T(1) \\
&= (2^k - 1) + 2^k \\
&\leq 2 \cdot 2^k.
\end{aligned}$$

In the following, "log" denotes the logarithm in base 2. Since $n = 3^k$, we have $\log n = k \log 3$, thus $k = \frac{\log n}{\log 3}$, thus

$$2^k = 2^{\log n / \log 3} = \left(2^{\log n}\right)^{1/\log 3} = n^{1/\log 3}.$$

We conclude that

$$T(n) \leq 2 \cdot n^{1/\log 3} = O\left(n^{1/\log 3}\right).$$

By the way, since

$$\log_3 2 = \frac{\log 2}{\log 3} = \frac{1}{\log 3},$$

the following is also a correct answer:

$$T(n) = O\left(n^{\log_3 2}\right).$$

(2.2)

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 1 + 2 \cdot T(n-1) & \text{if } n \geq 2. \end{cases}$$

Solution: Unfolding gives

$$\begin{aligned} T(n) &= 1 + 2 \cdot T(n-1) \\ &= 1 + 2\left(1 + 2 \cdot T(n-2)\right) \\ &= (1+2) + 2^2 \cdot T(n-2) \\ &= (1+2) + 2^2\left(1 + 2 \cdot T(n-3)\right) \\ &= (1 + 2 + 2^2) + 2^3 \cdot T(n-3) \\ &= (1 + 2 + 2^2) + 2^3\left(1 + 2 \cdot T(n-4)\right) \\ &= (1 + 2 + 2^2 + 2^3) + 2^4 \cdot T(n-4) \\ &\vdots \\ &= (1 + 2 + 2^2 + 2^3 + \ldots + 2^{n-2}) + 2^{n-1} \cdot T(1) \\ &= (2^{n-1} - 1) + 2^{n-1} \\ &= 2^n - 1 \\ &\leq 2^n \\ &= O(2^n). \end{aligned}$$

(2.3)

$$T(n) = \begin{cases} 1 & \text{if } n = 2, \\ 1 + T(\sqrt{n}) & \text{if } n \geq 4. \end{cases}$$

You may assume that n is a power of a power of 2, so that $\log\log n$ is an integer.

Solution: We assume that n is a power of a power 2. Thus, we write $n = 2^{2^k}$. Observe that

$$\sqrt{n} = n^{1/2} = \left(2^{2^k}\right)^{1/2} = 2^{(1/2)\cdot 2^k} = 2^{2^{k-1}}.$$

Thus, we can rewrite the recurrence as

$$T\left(2^{2^k}\right) = 1 + T\left(2^{2^{k-1}}\right).$$

Unfolding gives

$$\begin{aligned}
T\left(2^{2^k}\right) &= 1 + T\left(2^{2^{k-1}}\right) \\
&= 1 + \left(1 + T\left(2^{2^{k-2}}\right)\right) \\
&= 2 + T\left(2^{2^{k-2}}\right) \\
&= 2 + \left(1 + T\left(2^{2^{k-3}}\right)\right) \\
&= 3 + T\left(2^{2^{k-3}}\right) \\
&\vdots \\
&= k + T\left(2^{2^0}\right) \\
&= k + T\left(2^1\right) \\
&= k + T(2) \\
&= k + 1.
\end{aligned}$$

Since $n = 2^{2^k}$, we have $k = \log\log n$. We conclude that

$$T(n) = k + 1 = \log\log n + 1 \leq 2\log\log n = O(\log\log n).$$

Question 3: In class, we have seen a fast algorithm that computes the product of two large integers. You may think that squaring an integer is "easier" than multiplying two integers. After all, for squaring, you get one input integer x, and have to multiply it by itself, whereas for multiplication, you have to compute the product of two input integers x and y. In this question, you will show that this is not the case: The time complexity of multiplication is the same (except for constant factors) as that of squaring.

1. You are given an algorithm $\mathcal{A}$ that takes as input an integer $x \geq 1$, written in binary, and returns the integer x^2. We denote the running time (i.e., the number of bit operations) of algorithm $\mathcal{A}$ by $S(n)$, where n is the number of bits in the binary representation of x.

2. You learned in elementary school that, given two integers $x \geq 1$ and $y \geq 1$, both written in binary and both having n bits, their sum $x + y$ and difference $x - y$ can be computed in $O(n)$ time.

3. Given an integer $x \geq 2$, written in binary, and given an integer $k \geq 1$, such that x is a multiple of 2^k, we can compute $x/2^k$ in $O(k)$ time, by just removing the rightmost k bits from the binary representation of x.

Describe (in plain English or pseudocode), an algorithm $\mathcal{B}$ that takes as input two n-bit integers x and y, and returns the product xy. Your algorithm $\mathcal{B}$ must use algorithm $\mathcal{A}$ as a subroutine, as well as the results in 2. and 3. above, and its running time must be $O(S(n+1))$.

It is easy to come up with an algorithm $\mathcal{B}$ that calls $\mathcal{A}$ three times. You will get more marks if your algorithm $\mathcal{B}$ calls $\mathcal{A}$ only twice. *Hint:* $(x+y)^2$.

Solution:

Call $\mathcal{A}$ three times: The magic equation is:

$$xy = \frac{(x+y)^2 - x^2 - y^2}{2}.$$

On input x and y, algorithm $\mathcal{B}$ does the following:

1. Compute $z_1 = x + y$.
2. Run algorithm $\mathcal{A}$ on input z_1. Let the output be z'_1.
3. Run algorithm $\mathcal{A}$ on input x. Let the output be z'_2.
4. Run algorithm $\mathcal{A}$ on input y. Let the output be z'_3.
5. Compute $z'_4 = z'_1 - z'_2$.
6. Compute $z'_5 = z'_4 - z'_3$.
7. Compute $z' = (z'_5)/2$. (Observe that z'_5 is an even integer; thus, the rightmost bit is 0. When dividing by 2, we just delete the rightmost bit.)
8. Return z'.

Step 1 takes $O(n)$ time, whereas Step 2 takes $S(n+1)$ time, because the binary representation of $z_1 = x + y$ has at most $n+1$ bits. Each of Steps 3 and 4 takes $S(n)$ time. Each of Steps 5 and 6 takes $O(n)$ time, because the integers involved have at most $n+2$ bits. Step 7 takes $O(1)$ time. Step 6 takes $O(n)$ time.

Thus, the total running time is $O(n) + S(n+1) + 2 \cdot S(n)$. Since $S(n) \leq S(n+1)$ and $S(n) = \Omega(n)$, it follows that the total running time is $O(S(n+1))$.

Call $\mathcal{A}$ twice: After some trying, you will come up with the magic equation:

$$xy = \frac{(x+y)^2 - (x-y)^2}{4}.$$

On input x and y, algorithm $\mathcal{B}$ does the following:

1. Compute $z_1 = x + y$.

2. Run algorithm $\mathcal{A}$ on input z_1. Let the output be z'_1.

3. Compute $z_2 = x - y$.

4. Run algorithm $\mathcal{A}$ on input z_2. Let the output be z'_2.

5. Compute $z'_3 = z'_1 - z'_2$.

6. Compute $z' = (z'_3)/4$. (Observe that z'_3 is a multiple of 4, thus, the two rightmost bits are 0. When dividing by 4, we just delete the two rightmost bits.)

7. Return z'.

Step 1 takes $O(n)$ time, whereas Step 2 takes $S(n+1)$ time, because the binary representation of $z_1 = x + y$ has at most $n + 1$ bits. Step 3 takes $O(n)$ time, whereas Step 4 takes $S(n)$ time, because the binary representation of $z_2 = x - y$ has at most n bits. Step 5 takes $O(n)$ time, because the numbers involved have at most $n + 2$ bits. Step 6 takes $O(1)$ time. Step 7 takes $O(n)$ time.

Thus, the total running time is $O(n) + S(n + 1) + S(n)$. Since $S(n) \leq S(n + 1)$ and $S(n) = \Omega(n)$, it follows that the total running time is $O(S(n + 1))$.

Question 4: Consider the following recursive algorithm Apple(n), which takes as input an integer $n \geq 1$:

Algorithm Apple(n):
if $n = 1$
then sing "an apple a day keeps the doctor away"
else eat one apple;
    choose an arbitrary integer m with $1 \leq m \leq n - 1$;
    Apple(m);
    Apple($n - m$)
endif

(4.1) Explain why, for any integer $n \geq 1$, algorithm Apple(n) terminates.

(4.2) Let $A(n)$ be the number of apples that you eat when running algorithm Apple(n). Determine the exact value of $A(n)$.

Solution: We start with the first part. A call to Apple(n) generates two recursive calls, one to Apple(m) and the other to Apple($n - m$). Since $1 \leq m \leq n - 1$, we have

$$1 \leq m < n$$

and

$$1 \leq n - m < n.$$

Thus, each recursive call is on an input which is a positive integer that is strictly less than n. Therefore, after a finite number of calls (deeper in the recursion), the algorithm will reach the base case (which is when the input is 1), in which case it terminates.

Here is an alternative proof by induction: If $n = 1$, then $\text{Apple}(n)$ terminates. Let $n \geq 2$ and assume that for each k with $1 \leq k \leq n-1$, $\text{Apple}(k)$ terminates. We show that $\text{Apple}(n)$ terminates. Running $\text{Apple}(n)$ generates two recursive calls:

- $\text{Apple}(m)$, where m is an integer with $1 \leq m \leq n-1$. By the induction hypothesis (with $k = m$), this recursive call terminates.

- $\text{Apple}(n-m)$, where m is an integer with $1 \leq m \leq n-1$. Since $1 \leq n-m \leq n-1$, it follows from the induction hypothesis (with $k = n-m$) that this recursive call terminates.

We conclude that $\text{Apple}(n)$ terminates.

Now the second part. We are asked to determine the exact value of $A(n)$. The actual computation (i.e., recursion tree) of $\text{Apple}(n)$ depends on the value of m that is chosen. As we will see, the number of apples that you eat when running $\text{Apple}(n)$ does not depend on the value of m that is chosen.

If you run algorithm Apple with several values of n and m, then you will probably guess that

$$A(n) = n - 1.$$

So we think that we know the answer and we can try to prove it by induction:

Base case: If $n = 1$, then you do not eat any apple, so $A(1) = 0$. Since $n - 1 = 0$, the base case is true.

Induction step: Let $n \geq 2$ and assume that $A(k) = k - 1$ for all k with $1 \leq k \leq n-1$. We have to show that $A(n) = n - 1$. Run algorithm $\text{Apple}(n)$ and consider the integer m that is chosen. Then it follows from the algorithm that

$$A(n) = 1 + A(m) + A(n-m).$$

- Since $1 \leq m \leq n-1$, we can apply the induction hypothesis with $k = m$. Thus, $A(m) = m - 1$.

- Since $1 \leq m \leq n-1$, we have $1 \leq n-m \leq n-1$. Thus, we can apply the induction hypothesis with $k = m$ and obtain $A(n-m) = n - m - 1$.

By combining these results, we obtain

$$A(n) = 1 + (m-1) + (n-m-1) = n - 1.$$

By the way, we have just proved the following result: Define a binary tree to be a rooted tree in which each node has zero or two children. A node with zero children is called a leaf, whereas a node with two children is called an internal node. Let T be an arbitrary binary

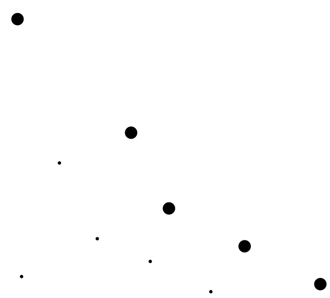

Figure 1: *The •-points are maximal; the ·-points are not maximal.*

---

tree with n leaves. Then the value of $A(n)$ is equal to the number of internal nodes of T. Thus, each binary tree with n leaves, regardless of how it is balanced, has exactly $n-1$ many internal nodes.

Question 5: Let S be a set of n points in the plane. Each point p of S is given by its x- and y-coordinates p_x and p_y, respectively.

A point p of S is called *maximal* in S, if there is no point in S that is to the north-east of p, i.e.,

$$\{q \in S \setminus \{p\} : q_x \geq p_x \text{ and } q_y \geq p_y\} = \emptyset.$$

See Figure 1 for an example. Observe that, in general, there is more than one maximal element in S.

Give a *divide-and-conquer*[1] algorithm (in plain English or pseudocode) that computes all maximal elements in S. The running time of your algorithm must be $O(n \log n)$. Explain why the running time of your algorithm is $O(n \log n)$ (you may use the Master Theorem).

Hint: At the start of the algorithm, sort the points of S from left to right. Use this ordering to divide the input set into two subsets. You don't have to give pseudocode for the sorting step. To avoid special cases, you may assume that no two points in S have the same x-coordinate, and no two points in S have the same y-coordinate.

Solution: There is a separate file with the solution for this question.

Question 6: *(Binary search in an infinite array)* Let $A[1\ldots]$ be an infinite array of real numbers such that

$$0 = A[1] < A[2] < A[3] < \ldots,$$

and let x be a positive real number.

In this question, you are asked to describe (in plain English or pseudocode) an algorithm that decides whether or not x occurs in A.

Let n be the index such that $A[n] \leq x < A[n+1]$. Hence, if x occurs in A then $x = A[n]$; if x does not occur in A, then $A[n] < x < A[n+1]$. The running time of your algorithm

[1]Your algorithm must use divide-and-conquer

must be $O(\log n)$. Keep in mind that the input consists *only* of the array A and the real number x; the index n is *not* part of the input. In other words, at the start of the algorithm, you *do not know* the value of n.

You are allowed to use the binary search algorithm as a subroutine, and you may use the fact that this algorithm has a logarithmic running time.

Explain why the running time of your algorithm is $O(\log n)$

Hint: In $O(\log n)$ time, compute an index h that is not "too large" and for which $A[h] > x$. (You may even show that such an index can be computed in $O(\log \log n)$ time, but the rest of the algorithm will still take $O(\log n)$ time.)

Solution: We want to apply binary search, which only works in a finite array. Our approach is as follows:

1. Find an index h such that $A[h] > x$. Since A is sorted, this implies that x does not occur in the infinite array $A[h \ldots]$. In other words, if x occurs in the infinite array $A[1 \ldots]$, then it occurs in the finite array $A[1 \ldots h-1]$.

2. Do a binary search for x in the finite array $A[1 \ldots h-1]$.

Let T be the time to find the index h. Since the binary search takes $O(\log h)$ time, we will obtain an overall time bound of

$$T + O(\log h).$$

So we must guarantee that $T = O(\log n)$ and $O(\log h) = O(\log n)$. In other words, we must show how to find, in $O(\log n)$ time, an index h such that

1. $A[h] > x$, and

2. $O(\log h) = O(\log n)$.

How to find h: We start with $h = 1$, and double it until $A[h] > x$:

$h = 1$;
while $A[h] \leq x$ **do** $h = 2h$ **endwhile**

Consider the final value of h. Then the previous value was $h/2$. Since the loop did not terminate at the previous value, we have $A[h/2] \leq x$. We know that n is the largest index such that the corresponding entry is less than or equal to x. Therefore $h/2 \leq n$, which implies that $h \leq 2n$.

Since $h \leq 2n$, we have

$$\log h \leq \log(2n) = 1 + \log n = O(\log n),$$

which implies that

$$O(\log h) = O(\log n).$$

How much time do we need to compute h? Since we start with h being equal to one, and double it repeatedly, we know that the final h is a power of two. Let k be the integer such that $h = 2^k$. We need $O(k)$ time to compute the final h. Since $k = \log h$, the time to find h is

$$O(k) = O(\log h) = O(\log n).$$

Here is a faster way to compute a value of h such that $A[h] > x$. Instead of doubling, we start with $h = 2$, and square it until $A[h] > x$:

$h = 2$;
while $A[h] \leq x$ **do** $h = h^2$ **endwhile**

Consider the final value of h. Then the previous value was $\sqrt{h}$. Since the loop did not terminate at the previous value, we have $A[\sqrt{h}] \leq x$. We know that n is the largest index such that the corresponding entry is less than or equal to x. Therefore $\sqrt{h} \leq n$, which implies that $h \leq n^2$.

Since $h \leq n^2$, we have

$$\log h \leq \log(n^2) = 2 \log n = O(\log n),$$

which implies that

$$O(\log h) = O(\log n).$$

How much time do we need to compute h? Since we start with h being equal to two, and square it repeatedly, we know that the final h is a power of a power of two. Let k be the integer such that $h = 2^{2^k}$. We need $O(k)$ time to compute the final h. Since $k = \log \log h$, the time to find h is

$$O(k) = O(\log \log h) = O(\log \log n).$$

Question 7: Let $A[1 \ldots n]$ be an array containing numbers in sorted order; you may assume that all these numbers are distinct. The following algorithm is a randomized version of the binary search algorithm. The input consists of the array A, its size n, and a number x. If x is in the array, then the algorithm returns the index p such that $A[p] = x$. Otherwise, the algorithm returns "not present".

Algorithm RandomizedBinarySearch(A, n, x):
$\ell = 1$; $r = n$;
while $\ell \leq r$
do $p =$ uniformly random element in $\{\ell, \ell + 1, \ldots, r\}$;
  if $A[p] < x$
  then $\ell = p + 1$
  else if $A[p] > x$
    then $r = p - 1$
    else return p
    endif

endif
endwhile;
return "not present"

Let T be the running time of this algorithm. Note that T is a random variable. Prove that the expected value of T is $O(\log n)$.

Solution: In one iteration of the while-loop, the algorithm searches for x in the subarray $A[\ell \dots r]$; this subarray has length $r - \ell + 1$. In each iteration, if the algorithm does not terminate, either ℓ increases or r decreases; thus, the next iteration searches a smaller subarray.

Let $i \geq 0$ be an integer. We say that the while-loop is in *phase* i if, at the beginning of this iteration,

$$(3/4)^{i+1} \cdot n < r - \ell + 1 \leq (3/4)^i \cdot n.$$

At the start of the first iteration, $r - \ell + 1 = n$ and, thus, the while-loop is in phase 0.

We first determine the largest possible phase number: If an iteration takes place in phase i, then $\ell \leq r$ (this is the condition in the while-loop) and, thus, $1 \leq r - \ell + 1$. It follows that

$$1 \leq (3/4)^i \cdot n,$$

which is equivalent to

$$(4/3)^i \leq n,$$

which is equivalent to

$$i \cdot \log(4/3) \leq \log n,$$

which is equivalent to

$$i \leq \frac{\log n}{\log(4/3)}.$$

Consider one phase i. Let $m = r - \ell + 1$. Divdide $\{\ell, \ell + 1, \dots, r\}$ into three pieces: The first $m/4$ elements, the middle $m/2$ elements, the last $m/4$ elements. If p belongs to the middle piece and if there is a next iteration, with values ℓ' and r', then

$$\ell' - r' + 1 \leq m - m/4 = (3/4) \cdot m \leq (3/4)^{i+1} \cdot n.$$

Thus, the next iteration is in a phase with number at least $i + 1$.

Let X_i be the random variable whose value is the number of iterations in phase i. Since p is in the middle piece with probability $1/2$, we have $\mathbb{E}(X_i) \leq 2$. (We have seen this in lecture 5.)

Let c be a constant such that one iteration takes at most c time. Let $L = \frac{\log n}{\log(4/3)}$. Then the running time T satisfies

$$T \leq \sum_{i=0}^{L} c \cdot X_i.$$

Thus,

$$
\begin{aligned}
\mathbb{E}(T) &\leq \mathbb{E}\left(\sum_{i=0}^{L} c \cdot X_i\right) \\
&= \sum_{i=0}^{L} c \cdot \mathbb{E}(X_i) \\
&\leq \sum_{i=0}^{L} 2c \\
&= 2c(L+1) \\
&= O(\log n).
\end{aligned}
$$