COMP 3804 — Solutions Assignment 1

Some useful facts:

1. for any real number $x > 0$, $x = 2^{\log x}$.
2. For any real number $x \neq 1$ and any integer $k \geq 1$,
$$1 + x + x^2 + \cdots + x^{k-1} = \frac{x^k - 1}{x - 1}.$$
3. For any real number $0 < \alpha < 1$,
$$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}.$$

Master Theorem:

1. Let $a \geq 1$, $b > 1$, $d \geq 0$, and
$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ a \cdot T(n/b) + O\left(n^d\right) & \text{if } n \geq 2. \end{cases}$$
2. If $d > \log_b a$, then $T(n) = O(n^d)$.
3. If $d = \log_b a$, then $T(n) = O(n^d \log n)$.
4. If $d < \log_b a$, then $T(n) = O(n^{\log_b a})$.

Question 1: Write your name and student number.

Solution: Al Gorithm, 007

Question 2: Consider the following recurrence, where n is a power of 8:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ n + 64 \cdot T(n/8) & \text{if } n \geq 8. \end{cases}$$

- Solve this recurrence using the *unfolding method*. Give the final answer using Big-O notation.
- Solve this recurrence using the *Master Theorem*.

Solution: We write $n = 8^k$. Unfolding gives

$$\begin{aligned}
T(n) &= n + 64 \cdot T(n/8) \\
&= n + 64\left((n/8) + 64 \cdot T(n/8^2)\right) \\
&= (1+8)\,n + 64^2 \cdot T(n/8^2) \\
&= (1+8)\,n + 64^2\left((n/8^2) + 64 \cdot T(n/8^3)\right) \\
&= \left(1+8+8^2\right) n + 64^3 \cdot T(n/8^3) \\
&= \left(1+8+8^2\right) n + 64^3\left((n/8^3) + 64 \cdot T(n/8^4)\right) \\
&= \left(1+8+8^2+8^3\right) n + 64^4 \cdot T(n/8^4) \\
&\vdots \\
&= \left(1+8+8^2+\cdots+8^{k-1}\right) n + 64^k \cdot T(n/8^k) \\
&= \frac{8^k-1}{8-1} \cdot n + 64^k \cdot T(1) \\
&= \frac{8^k-1}{7} \cdot n + 64^k \\
&\leq \frac{8^k}{7} \cdot n + \left(8^k\right)^2 \\
&= \frac{n}{7} \cdot n + n^2 \\
&= \frac{8}{7} \cdot n^2 \\
&= O\left(n^2\right).
\end{aligned}$$

Using the Master Theorem: We have $a = 64$, $b = 8$, and $d = 1$. Since

$$\log_b a = \log_8 64 = 2 > d,$$

the Master Theorem tells us that $T(n) = O(n^{\log_b a}) = O(n^2)$.

Question 3: Professor Justin Bieber has observed that only five multiplications are needed to compute the square of a 2×2 matrix:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a^2 + bc & b(a+d) \\ c(a+d) & bc + d^2 \end{pmatrix}$$

Professor Bieber remembers Strassen's algorithm from COMP 3804. Based on this, he claims that, for any given $n \times n$ matrix A, the matrix $A^2 = AA$ can be computed by a divide-and-conquer algorithm that makes five recursive calls, resulting in a running time of $O(n^{\log 5})$.

Is Professor Bieber's claim correct? As always, justify your answer.

Solution: Let A be an $n \times n$ matrix, where n is a large power of two. We split A into four $n/2 \times n/2$ submatrices:

$$A = \begin{pmatrix} U & V \\ X & Y \end{pmatrix}$$

Note that

$$A^2 = \begin{pmatrix} U & V \\ X & Y \end{pmatrix} \begin{pmatrix} U & V \\ X & Y \end{pmatrix} = \begin{pmatrix} U^2 + VX & UV + VY \\ XU + YX & XV + Y^2 \end{pmatrix}.$$

Since matrix multiplication is not commutative, we cannot write this as for 2×2 matrices. Here are two reasons why Justin is wrong:

- By applying Justin's idea, the number of recursive calls is eight.
- In only two recursive calls, we compute the square of a matrix. In the other six recursive calls, we multiply two unrelated matrices.

Question 4: You are given an array $A[1 \ldots n]$ of n numbers. Describe a divide-and-conquer algorithm that returns, in $O(n \log n)$ time, the value

$$\max\{A[j] - A[i] : 1 \leq i < j \leq n\}.$$

You may describe your algorithm in plain English or in pseudocode. Justify the correctness of your algorithm and explain why the running time is $O(n \log n)$. You may use any result that was proven in class.

Solution: Assume that n is an even integer with $n \geq 4$. Consider the solution to the problem, i.e., the two indices i and j, with $1 \leq i < j \leq n$, that maximize $A[j] - A[i]$. There are three possibilities for the locations of i and j:

- $j \leq n/2$. Then $A[j] - A[i]$ is the largest value in the subarray $A[1 \ldots n/2]$.
- $i \geq 1 + n/2$. Then $A[j] - A[i]$ is the largest value in the subarray $A[1 + n/2 \ldots n]$.
- $i \leq n/2$ and $j \geq 1 + n/2$. In this case, $A[i]$ is the smallest number in the subarray $A[1 \ldots n/2]$ and $A[j]$ is the largest number in the subarray $A[1 + n/2 \ldots n]$.

This suggests the following recursive algorithm:

Base case: The base case is when $n = 2$. In this case, we return $A[2] - A[1]$.

Non-base case: Assume that n is a power of two and $n \geq 4$.

Step 1: Compute the smallest number, say $A[i]$ in the subarray $A[1 \ldots n/2]$, and compute the largest number, say $A[j]$, in the subarray $A[1 + n/2 \ldots n]$. Set

$$x = A[j] - A[i].$$

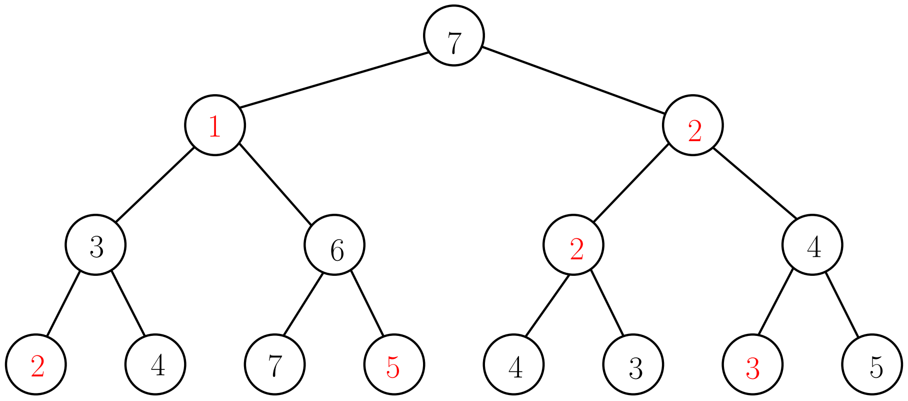
Step 2: Recursively solve the problem for the subarray $A[1 \ldots n/2]$. Let y be the value of the solution.
Step 3: Recursively solve the problem for the subarray $A[1 + n/2 \ldots n]$. Let z be the value of the solution.
Step 4: Return the largest among x, y, and z.

The running time $T(n)$ satisfies the merge-sort recurrence

$$T(n) = n + 2 \cdot T(n/2).$$

We have seen in class that $T(n) = O(n \log n)$.

Question 5: You are given a complete binary tree (i.e., all leaves are at the same level and the bottom level is full). The levels are numbered $0, 1, 2, \ldots, d-1$ and the number n of nodes satisfies $n = 2^d - 1$. Each node u in this tree stores an integer $\mathit{value}(u)$.



A node u is called a *local minimum* if $\mathit{value}(u)$ is less than or equal to the values in its neighboring nodes. In the example above, all local minima are colored red.

Describe a recursive algorithm that returns, in $O(\log n)$ time, a local minimum in this tree.

You may describe your algorithm in plain English or in pseudocode. Justify the correctness of your algorithm and explain why the running time is $O(\log n)$. You may use any result that was proven in class.

Solution: We first observe that there must be at least one local minimum: The smallest value in the tree is a local minimum.

Base case: The base case is when the tree has only one node. In this case, the value in this node is a local minimum; thus, we return it.

Non-base case: Assume that the tree has at least three nodes.

Let r be the root, let u be its left child, and let v be its right child. We consider three cases.

- If $\mathit{value}(r) \leq \mathit{value}(u)$ and $\mathit{value}(r) \leq \mathit{value}(v)$, then the root is a local minimum. In this case, we return the root and terminate.

- If $\mathit{value}(u) < \mathit{value}(r)$: Then any local minimum in the left subtree is also a local minimum in the entire tree. (If u is a local minimum in the left subtree, then, since $\mathit{value}(u) < \mathit{value}(r)$, u is a local minimum in the entire tree.) Thus, we recurse in the left subtree.

- If $value(v) < value(r)$: Then any local minimum in the right subtree is also a local minimum in the entire tree. Thus, we recurse in the rightsubtree.

Since there can be only one recursive call, the running time $T(n)$ satisfies the recurrence

$$T(n) = 1 + T(n/2).$$

We have seen in class that $T(n) = O(\log n)$.

Question 6: You are given a *sorted* array $A[1 \dots n]$ of n *distinct integers.* Describe a recursive algorithm that decides, in $O(\log n)$ time, if there is an index i such that $A[i] = i$. If such an index exists, the algorithm returns one such index. Otherwise, the algorithm returns "No".

You may describe your algorithm in plain English or in pseudocode. Justify the correctness of your algorithm and explain why the running time is $O(\log n)$. You may use any result that was proven in class.

Solution: Assume that n is larger than some small constant c. Let $m = \lceil n/2 \rceil$. There are three possible cases:

- If $A[m] = m$, then we are done.
- Assume $A[m] > m$. Since the entries are integers, we have

  $$A[m] \geq m + 1.$$

  Since the array is sorted and all entries are distinct integers, we have

  $$A[m+1] \geq A[m] + 1 \geq m + 2,$$

  $$A[m+2] \geq A[m+1] + 1 \geq m + 3,$$
  $$A[m+3] \geq A[m+2] + 1 \geq m + 4,$$

  etc. Thus, the index i we are trying to find is not in the subarray $A[m \dots n]$.
- Assume $A[m] < m$. Since the entries are integers, we have

  $$A[m] \leq m - 1.$$

  Since the array is sorted and all entries are distinct integers, we have

  $$A[m-1] \leq A[m] - 1 \leq m - 2,$$

  $$A[m-2] \leq A[m-1] - 1 \leq m - 3,$$
  $$A[m-3] \leq A[m-2] - 1 \leq m - 4,$$

  etc. Thus, the index i we are trying to find is not in the subarray $A[1 \dots m]$.

This suggests the following algorithm:

Base case: If the size of the array is at most c, then we try all values for i.

Non-base case: Assume that $n > c$.

- Check if $A[m] = m$. If this is the case, return the index m.
- If $A[m] > m$, recurse in the subarray $A[1 \dots m-1]$.
- If $A[m] < m$, recurse in the subarray $A[m+1 \dots n]$.

Since there can be only one recursive call, the running time $T(n)$ satisfies the recurrence

$$T(n) = 1 + T(n/2).$$

We have seen in class that $T(n) = O(\log n)$.

Question 7: Let $\mathcal{A}$ be the fastest algorithm that takes as input two n-bit integers x and y, and returns the product xy. Let $T(n)$ be the running time (in terms of bit-operations) of this algorithm.

Let $\mathcal{A}'$ be the fastest algorithm that takes as input one n-bit integer x, and returns the square x^2. Let $T'(n)$ be the running time (in terms of bit-operations) of this algorithm.

- Explain, in at most two sentences, why $T'(n) = O(T(n))$.
- Professor Taylor Swift claims that $T'(n) = o(T(n))$, i.e., $T'(n)$ is asymptotically smaller than $T(n)$. Her reasoning is that the input to algorithm $\mathcal{A}'$ is just one integer, whereas the input to algorithm $\mathcal{A}$ consists of two integers.

  Is Professor Swift's claim correct? As always, justify your answer.

  Hint: $(x+y)^2$.

Solution: We start[1] with the first part. Let $\mathcal{A}''$ be the algorithm that, on input x, runs algorithm $\mathcal{A}$ on input x and $y = x$. The running time $T''(n)$ of $\mathcal{A}''$ is at most $T(n)$. Since[2] $\mathcal{A}'$ is the fastest algorithm for computing x^2, we have $T'(n) \leq T''(n) \leq T(n)$.

For the second part, we observe that

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

Thus, we can compute xy by

- computing $x + y$ in $O(n)$ bit-operations, and then running $\mathcal{A}'$ to compute $(x+y)^2$,

[1]This sentence is not part of the "at most two sentences".

[2]This is the third sentence!

- running $\mathcal{A}'$ to compute x^2,
- running $\mathcal{A}'$ to compute y^2,
- doing two subtractions to compute $(x+y)^2 - x^2 - y^2$ in $O(n)$ bit-opertions,
- removing the rightmost bit from $(x+y)^2 - x^2 - y^2$ in $O(1)$ time.

The total running time is $O(n) + 3 \cdot T'(n)$. Since $T'(n) = \Omega(n)$, the total running time is $O(T'(n))$. Thus, we can multiply two n-bit integers in $O(T'(n))$ bit-operations. Since $\mathcal{A}$ is the fastest algorithm for computing xy, we have $T(n) = O(T'(n))$. Combining this with the first part, we conclude that $T(n) = \Theta(T'(n))$. Sorry Taylor! I hope your boyfriend wins the Super Bowl.

We have been cheating a "little bit". The integer $x+y$ may have $n+1$ bits. Thus, the upper bound on the total running time should be $O(n) + 2 \cdot T'(n) + T'(n+1)$. We assumed above that

$$T'(n+1) = O(T'(n)).$$

There are functions T' for which this is not true. A simple example is $T'(n) = n!$. An example with $n \leq T'(n) \leq n^2$ is

$$T'(n) = n \cdot (\lfloor \log \log n \rfloor)!.$$