CSCI 470 Fundamentals of Algorithms
Howard University
Instructor: Vijay Chaudhary   Homework 1 [Solutions]

Homework 1 [Solutions]

Please write your solutions in the LaTeX. You may use online compiler such as Overleaf or any other compiler you are comfortable with to write your solutions in the LaTeX.

Due date: Tuesday, Sep 19, 5PM (Extended Deadline)

I will collect your submissions when the class meets on Monday, Sep 18 at 9:10 EDT. Please handover a printed copy of your Homework 0 solutions (preferably written in the LaTeX). Also, please make sure that you have your full name and student ID in your submission.

You can use the LaTeX submission template I have shared along with the homework. There are two .tex files (macros.tex, and main.tex). You can upload the zipped folder directly to Overleaf, and edit main.tex to write your solutions. macros.tex is mostly for macros (predefined commands). The zipped folder also contains Python files. You may want to edit those files in your code editor, mostly to check the correctness of your solution for Problem 4.

Handwritten solutions will also be accepted. Points will be deducted if handwritten solutions are not legible.

Problem 1-1. [5 points] Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in $8n^2$ steps, while merge sort runs in $64n \lg n$ steps. For what range of values of n does insertion sort beat merge sort?

Solution: Enumerating over the values of n, we can show that for $2 \le n < 43$, $8n^2 > 64n \lg n$. Therefore, during this range for the values of n, insertion sort has less running time than the merge sort.

Problem 1-2. Consider the ***searching problem***:

Input: A sequence of n numbers $A = \langle a_1, a_2, ..., a_n \rangle$ and a value v.
Output: An index i such that $v = A[i]$ or the special value NIL if v does not appear in A.

(a) **[10 points]** Write pseudocode for ***linear search***, which scans through the sequence, looking for v.

Solution:

```
LINEAR-SEARCH($A, v$)
1  for $j = 1$ to $A.length$
2      if $A[j] == v$
3          return $j$
4  return NIL
```

(b) **[15 points]** Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties (Initialization, Maintenance, & Termination).

Solution:

Loop invariant: Before the j-th iteration, the subarray $A[1, ...j-1]$ has different elements other than v.

Initialization: Before $j = 1$, the subarray $A[1, ..., j-1]$ is empty. Trivially, v is not in this subarray.

Maintenance: With each iteration, line 2 compares $A[j]$ with v, and returns j, in case the line 2 holds true, terminating the loop. Therefore, as long as the loop continues, before j-th iteration, $A[1, ..., j-1]$ will not have v in it. With the line 2 false, the loop moves to $(j+1)$-th iteration, such that $A[1, ..., j]$ will not contain v in it, maintaining the loop invariant.

Termination: The loop terminates in two cases:

Case 1: If line 2 holds true at j-th iteration, the loop terminates returning j, preserving the loop invariant, where v is not in the subarray $A[1, ..., j-1]$.

Case 2: When the loop exhausts all the items in A, the loop terminates when $j = n+1$, in which case, the subarray $A[1, ..., n+1-1] = A[1, ...n]$. In this case, the function return NIL, and $A[1, ...n]$ does not contain v in it, preserving the loop invariant.

Problem 1-3. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element $A[1]$. Then the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A.

(a) **[10 points]** Write pseudocode for this algorithm, which is known as ***selection sort***.

Solution:

```
SELECTION-SORT($A$)
for $i = 1$ to $A.length - 1$
    currMin = $i$
    for $j = i + 1$ to $A.length$
        if $A[j] < A[\text{currMin}]$
            currMin = $j$
    Swap $A[\text{currMin}]$ and $A[i]$
```

(b) **[5 points]** What loop invariant does this algorithm maintain?

Solution:

Loop invariant: Before each i-th iteration, the subarray $A[1, ..., i-1]$ contains the $i-1$ smallest elements from array A in sorted order.

(c) **[5 points]** Why does it need to run for only the first $n-1$ elements, rather than for all n elements?

Solution: With $n-1$ iterations, $A[1, ..., n-1]$ will have $n-1$ smallest elements in sorted order. Therefore, $A[n]$ will be the largest elements from array A, making $A[1, ...n]$ completely sorted.

(d) **[5 points]** Give the best-case and worst-case running times of selection sort in Θ-notation.

Solution: If we enumerate the iterations of inner for-loop, it runs something like this:

- $n-1$ times
- $n-2$ times
- $n-3$ times
- ...
- 1 times

Therefore the total runtime becomes:

$$\begin{aligned} T(n) &= (n-1) + (n-2) + (n-3) + ... + 1 \\ &= \sum_{i=1}^{n-1} (n-i) \\ &= \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i \\ &= n(n-1) - \frac{(n-1)n}{2} \\ &= \frac{n(n-1)}{2} \\ &= \Theta(n^2) \end{aligned}$$

With any array A, the two loops will run constant amount of times, $\frac{n(n-1)}{2}$, making both the best-case and worst-case $\Theta(n^2)$.

Problem 1-4. [15 points] Using the definitions of $O(n^2)$ and $\Omega(n^2)$, show that $f(n) = 2n^2 + 3n + 5$ is $f(n) = O(n^2)$. Also, show that $f(n) = \Omega(n^2)$. Once you show that $f(n) = O(n^2)$ and $f(n) = \Omega(n^2)$, use the definition of $\Theta(n)$ to show that $f(n) = \Theta(n^2)$.

Solution:

As per Theorem 3.1, if we can show $f(n) = \Theta(n^2)$, it also implies $O(n^2)$ and $\Omega(n^2)$.

Let's show $f(n) = \Theta(n^2)$. As per the definition, we need to find c_1 and c_2 such that

$$0 < c_1 n^2 \le f(n) \le c_2 n^2$$

$$\begin{aligned} &c_1 n^2 \le f(n) \le c_2 n^2 \\ &c_1 n^2 \le 2n^2 + 3n + 5 \le c_2 n^2 \\ &c_1 \le 2 + \frac{3}{n} + \frac{5}{n^2} \le c_2 \end{aligned}$$

For a larger value of n, $c_1 \le 2 \le c_2$ will hold.

Therefore, we pick $c_1 = 1$ and $c_2 = 3$. Now we need to find n_0 for which the inequality above holds.

For $n_0 = 1$, it does not hold, however, for any value $n_0 \ge 4$, the inequality holds. Therefore, for $c_1 = 1$, $c_2 = 3$, and $n_0 \ge 4$, we show that the $f(n) = \Theta(n^2)$.

This implies that $f(n) = O(n^2)$ and $f(n) = \Omega(n^2)$.

Problem 1-5. [5 points] Explain why the statement, "The running time of algorithm A is at least $O(n^2)$." is meaningless.

Solution: If $T(n) = O(n^2)$, then $0 < T(n) \le cn^2$ for some $c > 0$ and $n \ge n_0$. In this case, the lower bound for $T(n)$ can be a fraction of n^2, a linear function or $c_1 \lg n$ for some $c_1 > 0$. Therefore, there is no lower bound of $T(n)$ in terms of n^2 to say that the running time of algorithm A is at least $O(n^2)$, making it meaningless.

Problem 1-6. [10 points] Prove that for any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Solution:

First we show that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

By definition, $f(n) = \Theta(g(n))$ states that there exists c_1 and c_2 such that $0 < c_1 g(n) \le f(n) \le c_2 g(n)$ for $n \ge n_0$.

From the definition itself, we can conclude that $0 < c_1 g(n) \le f(n)$ for $n \ge n_0$, which is the definition of $f(n) = \Omega(g(n))$.

Similarly, $0 < f(n) \le c_2 g(n)$ for $n \ge n_0$, shows that $f(n) = O(g(n))$.

Therefore, $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Now, we need to show that if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \Theta(g(n))$.

From the definitions, $f(n) = \Omega(g(n))$ states that there exist c_3 such that $0 < c_3 g(n) \leq f(n)$ for $n \geq n_0^1$.

Similarly, $f(n) = O(g(n))$ states that there exist c_4 such that $0 < f(n) \leq c_4 g(n)$ for $n \geq n_0^2$

Therefore, we combine these two for $n_0 = \max\{n_0^1, n_0^2\}$, giving us $0 < c_3 g(n) \leq f(n) \leq c_4 g(n)$ for $n \geq n_0$, which is the definition of $f(n) = \Theta(g(n))$.

Problem 1-7. [10 points] Show that the solution of $T(n) = T(n-1) + n$ is $O(n^2)$ using the substitution method.

Solution: In order to show that $T(n) = O(n^2)$, we need to show $T(n) \leq cn^2$.

Let's assume that $T(m) \leq cm^2$, where $m < n$, particularly, $m = n - 1$, yielding
$T(n-1) \leq c(n-1)^2$. Substituting it in the original equation, we get:

$$\begin{aligned} T(n) &\leq c(n-1)^2 + n \\ &= cn^2 - 2cn + 1 + n \\ &= cn^2 + 1 + n - 2cn \\ &= cn^2 + 1 + (1-2c)n \\ &\leq cn^2 \quad \text{for } c \geq 1 \;\&\; n \geq 1 \end{aligned}$$

The substitution shows that we can induct from $n-1$ to n on the inductive hypothesis $T(n) \leq cn^2$, for $c \geq 1$ and $n \geq 1$, resulting on the *exact form* of inductive hypothesis, making the inductive hypothesis true. Therefore, $T(n) \leq cn^2$ for $c \geq 1$ and $n \geq 1$ which implies $T(n) = O(n^2)$.

Problem 1-8. [10 points] Show that the solution of $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$ using substitution method.

Solution: In order to show $T(n) = O(\lg n)$, we need to show $T(n) \leq c \lg n$.
Inductively, we need to show that for $m < n$, $T(m) \Rightarrow T(n)$, where $m = \lceil n/2 \rceil$.

Note that $\lceil n/2 \rceil \leq (n+1)/2$ is a valid inequality because $\lceil n/2 \rceil = (n+1)/2$ when n is odd while $\lceil n/2 \rceil < (n+1)/2$ when n is even.

Let's assume that $T(\lceil n/2 \rceil) \leq c \lg \lceil n/2 \rceil$ is true. Now, we substitute this in the original equation yielding,

$$\begin{aligned} T(n) &\leq c \lg \lceil n/2 \rceil + 1 \\ &\leq c \lg((n+1)/2) + 1 \\ &= c \lg(n+1) - c \lg 2 + 1 \\ &\leq c \lg(n+1) \\ &\leq c \lg(n) \quad \text{for } c \geq 1 \;\; \textit{wrong!!} \end{aligned}$$

$T(n) \le c\lg(n+1) \le c\lg(n)$ is not a valid conclusion because $c\lg(n+1) > c\lg(n)$ for any value of $c > 0$. Therefore, we are stuck at $T(n) \le c\lg(n+1)$.

This is NOT what we want to end up with while using substitution method. Mathematical induction does not work unless we prove the exact form of the inductive hypothesis. Our inductive hypothesis was $T(n) \le c\lg n$, however, we ended up with $T(n) \le c\lg(n+1)$, which is not the *exact form* of the inductive hypothesis.

In order to deal with this difficulty, we guess $T(n) \le c\lg(n-d)$ for $d \ge 0$.

Note that this will still imply that $T(n) = O(\lg n)$ because $T(n) \le c\lg(n-d) \le c\lg n$ for $d \ge 0$ for $T(n) = c\lg(n)$ is a non-decreasing function.

Repeating the same procedure with our new guess, we get,

$$\begin{aligned}
T(n) &\le c\lg(\lceil n/2\rceil - d) + 1 \\
&\le c\lg((n+1)/2 - d) + 1 \\
&= c\lg((n+1-2d)/2) + 1 \\
&= c\lg(n+1-2d) - c\lg 2 + 1 \\
&\le c\lg(n+(1-d)-d) \ \ (\text{c} \ge 1) \\
&\le c\lg(n-d) \quad \text{for } d \ge 1
\end{aligned}$$

This time, we end up with the exact form of hypothesis after the substitution, ensuring that we are correctly inducting from $n/2$ to n for $c \ge 1$ and $d \ge 1$. Note that in this recurrence, the inductive step goes from $n/2$ to n, as that's how we are evaluating n-th value based on $(n/2)$-th value.

Problem 1-9. [15 points] Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$. Use the substitution method to verify your answer.
[If you are using LaTeXto write solution to this, a quick way to show your recursion tree would be to use a picture of a handwritten recursion tree, and the rest of the answer in LaTeX]

Solution: The tree has no branching, and the sequences goes as $n^2, (n/2)^2, (n/4)^2, ...(n/2^d)^2$ till the bottom of the tree. The depth of the tree will be $\lg n + 1$.

Therefore, the runtime function can be summarized as:

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\lg n} n^2 (\frac{1}{4})^i \\
&= n^2 \sum_{i=0}^{\lg n} (\frac{1}{4})^i \\
&\le n^2 \sum_{i=0}^{\infty} (\frac{1}{4})^i \\
&= \frac{4}{3} n^2
\end{aligned}
$$

Therefore, $T(n) \le \frac{4}{3}n^2$, which makes us guess $O(n^2)$.

Now, we need to verify that $T(n) \le cn^2$. Let the inductive hypothesis be $T(m) \le cm^2$, where $m < n$, then, $T(n/2) \le c(n/2)^2$.

$$
\begin{aligned}
T(n) &\le c(n/2)^2 + n^2 \\
&= (c/4 + 1)n^2 \\
&\le cn^2 \quad \text{for } c \ge \frac{4}{3}
\end{aligned}
$$

Problem 1-10. [15 points] Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Use the substitution method to verify your answer.
[If you are using LaTeXto write solution to this, a quick way to show your recursion tree would be to use a picture of a handwritten recursion tree, and the rest of the answer in LaTeX]

Solution: Each subtree will have three branching. The sequence will be:

- $n = (3^0/2^0)n$
- $n/2 + n/2 + n/2 = (3^1/2^1)n$
- $(n/4 + n/4 + n/4) + (n/4 + n/4 + n/4) + (n/4 + n/4 + n/4) = 3/4 * n * 3 = (3^2/2^2)n$
- ...
- $(3/2)^i n$
- ...
- $(3/2)^{\lg n} n$

The depth of the tree will be $\lg n + 1$.

There will be $n^{\lg 3}$ leaves. You can verify this on your own. Consider $n = 8$, for a tree's node branching with 3 children, and each depth, n is divided by 2. There will be 27 leaves, which is $n^{\log_2 3} = 8^{\log_2 3} = 3^{\log_2 8} = 27$.

Considering $T(1)$ costs $\Theta(1)$, for $n^{\lg 3}$ leaves, it will be $\Theta(n^{\lg 3})$.

To summarize this:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\lg n - 1} (3/2)^i n + \Theta(n^{\lg 3}) \\
&= \frac{(3/2)^{\lg n} - 1}{3/2 - 1} n + \Theta(n^{\lg 3}) \\
&= 2((3/2)^{\lg n} - 1)n + \Theta(n^{\lg 3}) \\
&= 2(n^{\lg \frac{3}{2}} - 1)n + \Theta(n^{\lg 3}) \\
&= 2(n^{\lg 3 - \lg 2} - 1)n + \Theta(n^{\lg 3}) \\
&= 2n^{\lg 3 - 1}.n - 2n + \Theta(n^{\lg 3}) \\
&= 2n^{\lg 3} - 2n + \Theta(n^{\lg 3}) \\
&= O(n^{\lg 3})
\end{aligned}$$

We guess $T(n) \le cn^{\lg 3} - dn$,

$$\begin{aligned}
T(n) &\le 3c((\lfloor n/2 \rfloor)^{\lg 3} - d(\lfloor n/2 \rfloor)) + n \\
&\le 3c((n/2)^{\lg 3} - d(n/2)) + n \\
&= \frac{3}{2^{\lg 3}} cn^{\lg 3} - 3d(n/2) + n \\
&= cn^{\lg 3} + (1 - 3d/2)n \\
&\le cn^{\lg 3} - dn + (1 - d/2)n \\
&\le cn^{\lg 3} - dn \quad \text{for } d \ge 2
\end{aligned}$$

Extra Credit

Problem 1-11. [12 points] Use the master method to give tight asymptotic bounds of the following recurrences.

(a) $T(n) = 2T(n/4) + 1$.

Solution: $a = 2$, $b = 4$, and $f(n) = 1$.
$n^{\log_b a} = n^{\log_4 2}$
$f(n) = O(n^{\log_4 2 - \epsilon}) = O(n^{\log_4 2 - 1/2}) = 1$, where $\epsilon = 1/2 > 0$.
This fits into case 1.
Therefore, $T(n) = \Theta(n^{\log_4 2}) = \Theta(\sqrt{n})$.

(b) $T(n) = 2T(n/4) + \sqrt{n}$.

Solution: $a = 2$, $b = 4$, and $f(n) = \sqrt{n}$.
$n^{\log_b a} = n^{\log_4 2}$
$f(n) = \Theta(\sqrt{n}) = \Theta(n^{\log_4 2})$.
This fits into case 2.
Therefore, $T(n) = \Theta(n^{\log_4 2} \lg n) = \Theta(\sqrt{n} \lg n)$.

(c) $T(n) = 2T(n/4) + n^2$.

Solution: $a = 2$, $b = 4$, and $f(n) = n^2$.
$n^{\log_b a} = n^{\log_4 2}$
$f(n) = \Omega(n^2) = \Omega(n^{\log_4 2 + \epsilon}) = \Omega(n^{\log_4 2 + 3/2})$, where $\epsilon = 3/2 > 0$
This fits into case 3.
$af(n/b) = 2f(n/4) = 2(n/4)^2 = \frac{n^2}{8} \leq \frac{1}{8}n^2 = \frac{1}{8}f(n)$, where $c < 1$
Therefore, $T(n) = \Theta(f(n))) = \Theta(n^2)$.

Problem 1-12. Referring back to the searching problem **1.2**, observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of remaining portion of the sequence each time.

(a) [10 points] Write recursive pseudocode for binary search.

Solution:

```
BINARY-SEARCH($A, low, high, v$)
1   if $low \leq high$
2       $mid = high - \lfloor (high - low)/2 \rfloor$
3       if $A[mid] = v$
4           return $mid$
5       else if $A[mid] < v$
6           return BINARY-SEARCH($A, mid + 1, high, v$)
7       else
8           return BINARY-SEARCH($A, low, mid - 1, v$)
9   else
10      return NIL
```

(b) **[10 points]** Show that the worst-case running time of binary search is $\Theta(\lg n)$. Write out the running time function of binary search as a recurrence relation, and use the recurrence to show that worst-case running time.

Solution:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ T(n/2) + \Theta(1) & \text{if } n > 1 \end{cases}$$

The worst case of a binary search occurs when the target we are looking for is at the edge of the array, (or the element is not in the array and have to return NIL).

In case of element being at the edge of the array, we need to exhaust all the mid-values till $low = high$. This situation can also be framed as a situation, when we keep halving the array till we reach an array of size 1. It takes $\log(n)$ steps reach a subproblem where the size of the subarray is 1.

Therefore, the worst-case runtime is $\Theta(\lg n)$.