Fundamental Algorithms
CSCI-GA.1170-001/Summer 2017

Solution to Homework 2

**Problem 1.** (2 points) For the following recurrences, obtain a solution using the master method (or explain why not applicable), then give a substitution proof for the upper bound:

(a) \( T(n) = 4T\left(\frac{n}{3}\right) + n \). (CLRS 4.3-7)
(b) \( T(n) = 4T\left(\frac{n}{2}\right) + n^2 \). (CLRS 4.3-8)
(c) \( T(n) = 4T\left(\frac{n}{2}\right) + n^2 \lg n \). (CLRS 4.5-4)

**Solution:**

(a) The recurrence \( T(n) = 4T\left(\frac{n}{3}\right) + n \) is of the form \( T(n) = aT\left(\frac{n}{b}\right) + f(n) \), and \( n = O(n^{\log_b a}) \), so we can apply case 1 of the master method to obtain \( T(n) = \Theta(n^{\log_b a}) \).

Let us verify the upper bound \( T(n) = O(n^{\log_b a}) \) by substitution. (By convention, we assume \( T(0) = \Theta(1) \) or \( T(1) = \Theta(1) \), as appropriate, and omit the base cases in this and other inductive proofs.)

With **inductive hypothesis**:

\[
T(k) \leq ck^{\log_b a} \quad \text{for all } 0 < k < n,
\]

Let us perform the **inductive step** by showing:

\[
T(n) \leq cn^{\log_b a}.
\]  
(1.1)

By definition of \( T(n) \) and then by inductive hypothesis:

\[
T(n) = 4T\left(\frac{n}{3}\right) + n,
\]

\[
T(n) \leq 4\left(c\left(\frac{n}{3}\right)^{\log_b a}\right) + n,
\]

\[
T(n) \leq cn^{\log_b a} + n.
\]

Showing (1.1) is thus equivalent to showing:

\[
cn^{\log_b a} + n \leq cn^{\log_b a},
\]

\[
n \leq 0.
\]

Which contradicts the inductive hypothesis, as well as our fundamental assumptions about the meaning and domain of \( T(n) \).

Let us formulate a stronger **inductive hypothesis** by including a lower-order term:

\[
T(k) \leq c_1k^{\log_b a} - c_2k \quad \text{for all } 0 < k < n.
\]
And perform the inductive step by showing:

\[ T(n) \leq c_1 n^{\log_3 4} - c_2 n. \]  

(1.2)

By definition of \( T(n) \) and then by inductive hypothesis:

\[ T(n) = 4T\left(\frac{n}{3}\right) + n, \]

\[ T(n) \leq 4 \left( c_1 \left(\frac{n}{3}\right)^{\log_3 4} - c_2 \frac{n}{3} \right) + n, \]

\[ T(n) \leq c_1 n^{\log_3 4} - \frac{4}{3} c_2 n + n. \]

Showing (1.2) is thus equivalent to showing:

\[ c_1 n^{\log_3 4} - \frac{4}{3} c_2 n + n \leq c_1 n^{\log_3 4} - c_2 n, \]

\[ -\frac{4}{3} c_2 n + n \leq -c_2 n, \]

\[ -\frac{4}{3} c_2 + 1 \leq -c_2, \]

\[ c_2 \geq 3. \]

Which does not contradict any of our assumptions and thus completes the inductive step and the proof.

(b) The recurrence \( T(n) = 4T\left(\frac{n}{2}\right) + n^2 \) is of the form \( T(n) = aT\left(\frac{n}{b}\right) + f(n) \), and \( n^2 = \Theta(n^2) \), so we can apply case 2 of the master method to obtain \( T(n) = \Theta(n^2 \log n) \).

Let us verify the upper bound \( T(n) = O(n^2 \log n) \) by substitution.

With inductive hypothesis:

\[ T(k) \leq ck^2 \log k \text{ for all } 0 < k < n. \]

Let us perform the inductive step by showing:

\[ T(n) \leq cn^2 \log n. \]  

(1.3)

By definition of \( T(n) \) and then by inductive hypothesis:

\[ T(n) = 4T\left(\frac{n}{2}\right) + n^2, \]

\[ T(n) \leq 4 \left( c \left(\frac{n}{2}\right)^2 \log \frac{n}{2} \right) + n^2, \]

\[ T(n) \leq cn^2 \log \frac{n}{2} + n^2, \]

\[ T(n) \leq cn^2 \log n - cn^2 + n^2. \]
Showing (1.3) is thus equivalent to showing:

\[ cn^2 \log n - cn^2 + n^2 \leq cn^2 \log n, \]
\[ -cn^2 + n^2 \leq 0, \]
\[ c \geq 1. \]

Which does not contradict any of our assumptions and thus completes the inductive step and the proof.

(c) While \( T(n) = 4T\left(\frac{n}{2}\right) + n^2 \log n \) is of the form \( T(n) = aT\left(\frac{n}{b}\right) + f(n) \), neither of the master method cases apply here, as the difference between \( f(n) = n^2 \log n \) and \( n^{\log_b a} = n^2 \) is less than polynomial. While case 3 might seem applicable, as indeed:

\[ n^2 \log n = \Omega(n^2). \]

We can see that:

\[ n^2 \log n \neq \Omega(n^{2+\epsilon}). \]

By looking at a ratio:

\[ \frac{n^2 \log n}{n^{2+\epsilon}} = \frac{n^2 \log n}{n^{2+\epsilon}} = \frac{\log n}{n^{\epsilon}}. \]

And observing that \( n^\epsilon \) grows faster than \( \log n \) for any \( \epsilon > 0 \).

There is a definition of master method that does, in fact, cover this case – see CLRS exercise 4.6-2 or https://en.wikipedia.org/wiki/Master_theorem – but we haven’t discussed it in class.

**Problem 2.** (2 points) Find the exact solution of the recurrence:

\[ T(n) = \begin{cases} 
  c & \text{if } n = 0, \\
  aT(n-1) + k & \text{if } n > 0. 
\end{cases} \]

**Solution:** Computing the first few values:

\[ T(0) = c \]
\[ T(1) = aT(0) + k = ac + k \]
\[ T(2) = aT(1) + k = a(ac + k) + k = a^2c + ak + k \]
\[ T(3) = aT(2) + k = a(a^2c + ak + k) + k = a^3c + a^2k + ak + k \]

Seems to suggest:

\[ T(n) = a^n c + \sum_{i=0}^{n-1} a^i k. \]  \hspace{1cm} (1.4)
For $a = 1$ we have:

$$T(n) = c + \sum_{i=0}^{n-1} k = c + nk. \quad (1.5)$$

That we can prove by induction from the base case:

$$T(0) = c + 0 \times k = c.$$

Taking the inductive hypothesis to be that for some $n > 0$:

$$T(n) = c + nk.$$

And performing inductive step to show that (1.5) holds for $n + 1$:

$$T(n + 1) = aT(n) + k$$
$$= T(n) + k$$
$$= c + nk + k$$
$$= c + (n + 1)k.$$

For $a \neq 1$, applying the formula for geometric series to (1.4):

$$T(n) = a^n c + k \frac{1-a^n}{1-a}. \quad (1.6)$$

Let us prove (1.6) by induction from the base case:

$$T(0) = a^0 c + k \frac{1-a^0}{1-a} = c.$$

Taking the inductive hypothesis to be that for some $n > 0$:

$$T(n) = a^n c + k \frac{1-a^n}{1-a}.$$

And performing the inductive step to show that (1.6) holds for $n + 1$:

$$T(n + 1) = a^{n+1} c + k \frac{1-a^{n+1}}{1-a}.$$

By definition of $T(n)$ and then by inductive hypothesis:

$$T(n + 1) = aT(n) + k$$
$$= a \left( a^n c + k \frac{1-a^n}{1-a} \right) + k.$$

Simplifying:

$$T(n + 1) = a^{n+1} c + ka \frac{1-a^n}{1-a} + k$$
$$= a^{n+1} c + k \frac{a(1-a^n) + (1-a)}{1-a}$$
$$= a^{n+1} c + k \frac{1-a^{n+1}}{1-a}.$$
With the base case and inductive step in place for both (1.5) and (1.6), we conclude that (1.4) holds for all natural numbers \( n \) and all values of \( c, a, \) and \( k \).

**Problem 3.** (2 points) Give pseudocode for iterative binary search. Prove correctness of your algorithm using a loop invariant, state the worst-case running time.

**Solution:**

• Iterative binary search in Python:

```python
    def iterative_binary_search(a, key):
        lo = 0
        hi = len(a)
        while hi - lo > 0:
            mid = lo + (hi - lo) / 2
            if key == a[mid]:
                return mid
            elif key < a[mid]:
                hi = mid
            elif key > a[mid]:
                lo = mid + 1
        return -1
```

• We prove the algorithm correct using the following two-part loop invariant:

At the start of each iteration of the while loop on lines 4-11:

1. \( lo = 0 \) or \( key > a[lo - 1] \).
2. \( hi = \text{len}(a) \) or \( key < a[hi] \).

*Initialization*: Before the first iteration, \( lo = 0, hi = \text{len}(a) \) and both invariants hold.

*Maintenance*: The loop body has three possible executions:

1. \( key = a[mid] \). The loop exits early, so there is no next iteration and no need to demonstrate maintenance.
2. \( key < a[mid] \). Observe that after line 9, \( hi = mid \) and the entry condition becomes equivalent to \( key < a[hi] \) (invariant 2). Neither \( key \) nor \( lo \) change, so invariant 1 remains true as well.
3. \( key > a[mid] \). Observe that after line 11, \( lo = mid + 1 \) and the entry condition becomes equivalent to \( key > a[lo - 1] \) (invariant 1). Neither \( key \) nor \( hi \) change, so invariant 2 remains true as well.

*Termination*: We first show that the loop terminates. On entry, \( hi - lo > 0 \) and there are three possible executions:

1. \( key = a[mid] \). The loop terminates immediately.
2. \(key < a[mid]\). The adjusted range \([lo,mid]\) is strictly smaller than the original range \([lo,hi]\):

\[
\begin{align*}
    mid - lo &< hi - lo, \\
    mid &< hi, \\
    lo + \left\lfloor \frac{hi - lo}{2} \right\rfloor &< hi, \\
    \frac{hi - lo}{2} &< hi - lo.
\end{align*}
\]

For integer \(hi\) and \(lo\), and \(hi - lo > 0\), even \(hi - lo\) gives:

\[
\left\lfloor \frac{hi - lo}{2} \right\rfloor = \frac{hi - lo}{2},
\]
\[
\frac{hi - lo}{2} < hi - lo,
\]
\[
hi - lo > 0.
\]

Odd \(hi - lo\) gives:

\[
\left\lfloor \frac{hi - lo}{2} \right\rfloor = \frac{hi - lo - 1}{2},
\]
\[
\frac{hi - lo - 1}{2} < hi - lo,
\]
\[
hi - lo > -1.
\]

Both of which hold by the loop condition \(hi - lo > 0\).

3. \(key > a[mid]\). By analogous argument, the adjusted range \([mid + 1,hi]\) is strictly smaller than the original range \([lo,hi]\).

As on every iteration the loop either terminates immediately or executes over a strictly smaller range, we conclude that the loop terminates.

We now need to show that the invariants hold for both termination conditions:

1. If the loop terminates due to \(\text{return mid}\) at line 7, neither of \(lo, hi,\) or \(key\) have been updated since the start of the iteration, so both invariants hold by \textit{maintenance}. With both invariants holding and with entry condition \(key = a[mid]\), we conclude that \(\text{return mid}\) correctly returns the index of \(key\).

2. If the loop terminates due to \(hi - lo \leq 0\) at line 4, neither of \(lo, hi,\) or \(key\) have been updated since the start of the iteration, so both invariants hold by \textit{maintenance}. The range \([lo,hi]\), however, now includes zero or negative number of elements and thus cannot include \(key\). With both invariants holding and with no possibility of \(key\) being in \([lo,hi]\), we conclude that \(\text{return -1}\) correctly returns the special "not found" value.

With both invariants holding in all cases, and having considered both "key found" and "key not found" scenarios, we conclude that the algorithm is correct.
• The worst-case running time can be observed when key is not present in a and the algorithm terminates due to $hi - lo \leq 0$. Each iteration of the loop reduces the range from $[lo, hi)$ to either $[lo, mid]$ or $[mid + 1, hi)$ and thus reduces the number of elements in the range to either $mid - lo$ or $hi - mid - 1$. By definition of mid:

$$mid - lo = lo + \left\lfloor \frac{hi - lo}{2} \right\rfloor - lo = \left\lfloor \frac{hi - lo}{2} \right\rfloor,$$

$$hi - mid - 1 = hi - lo - \left\lfloor \frac{hi - lo}{2} \right\rfloor - 1.$$

Considering even $hi - lo$:

$$mid - lo = \frac{hi - lo}{2},$$

$$hi - mid - 1 = hi - lo - \frac{hi - lo}{2} - 1 = \frac{hi - lo}{2} - 1.$$

Thus, $hi - lo$, initially equal to $len(a)$, is reduced roughly in half each iteration, leading to the worst-case running time of $\Theta(\lg n)$ with $n = len(a)$.

Problem 4. (2 points) Give pseudocode for recursive binary search. Formulate and solve a recurrence describing the worst-case running time.

Solution:

• Recursive binary search in Python:

```python
1 def recursive_binary_search(a, key, lo=0, hi=len(a)):
2     if hi - lo <= 0:
3         return -1
4     mid = lo + (hi - lo) / 2
5     if key == a[mid]:
6         return mid
7     elif key < a[mid]:
8         return recursive_binary_search(a, key, lo, mid)
9     elif key > a[mid]:
10        return recursive_binary_search(a, key, mid + 1, hi)
```

• Observing that going from $[lo, hi)$ to either $[lo, mid]$ or $[mid + 1, hi)$ approximately halves the range size, and that only a constant number of operations are executed outside recursive calls, we can express the worst-case running time as a recurrence (omitting the base case by convention):

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1).$$

By case 2 of the master method, $T(n) = \Theta(\lg n)$. 
Problem 5. (3 points) Let \( A = (a_1, a_2, ..., a_n) \) be a sequence of \( n \) distinct numbers. A pair \((i, j)\) is called an inversion of \( A \) if \( i < j \) and \( a_i > a_j \). Give pseudocode for a modification of merge sort to count the number of inversions of the sequence. Formulate and solve a recurrence describing the worst-case running time.

Solution:

• Considering an execution of \( \text{merge}(\text{left}, \text{right}) \) with \( i \) and \( j \) keeping track of the current element in \( \text{left} \) and \( \text{right} \) respectively, we observe that taking element \( \text{right}[j] \) before exhausting all elements in \( \text{left} \), indicates that \( \text{right}[j] \) forms inversions with all \( \text{len(left)} - i \) yet unprocessed elements in \( \text{left} \). We then extend a typical implementation of \( \text{mergesort} \) with code to:

1. Count the number of inversions \( \text{right}[j] \) forms with elements in \( \text{left} \) (line 25).
2. Return the total number of inversions detected by this invocation of \( \text{merge} \) (line 30).
3. Return the total number of inversions detected in left subarray, right subarray, and inversions crossing the boundary between two subarrays (lines 6-8 and 10).

Resulting in the following Python code:

```python
def mergesort(a):
    if len(a) < 2:
        return a, 0

    mid = len(a) / 2
    left, inv_left = mergesort(a[:mid])
    right, inv_right = mergesort(a[mid:])
    merged, inv_split = merge(left, right)
    return merged, inv_left + inv_right + inv_split

def merge(left, right):
    merged = []
    i = 0
    j = 0
    inv = 0

    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            inv += 1
            j += 1
    return merged, inv
```


merged.extend(left[i:])
merged.extend(right[j:])

return merged, inv

• Observing that we recurse on both \(a[0, mid]\) and \(a[mid, len(a)]\), each approximately half of \(len(a)\), and that only a linear amount of work is done outside recursive calls, we can formulate a recurrence for the worst-case running time:

\[
T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n).
\]

By case 2 of the master method, \(T(n) = \Theta(n \log n)\).