CS 6363: Design and Analysis of Algorithms - Fall 2019
Homework #1 Solutions
Professor D. T. Huynh

Problem #1. Do Problem# 2-4 (Inversions), p. 41 [Text]. (15 pt)

Solution

a. The five inversions are (1, 5), (2, 5), (3, 4), (3, 5) and (4, 5). (*Remember that inversions are specified by indices rather than by values in the array.)*

b. The array <n, n-1, n-2, ..., 2, 1> has the most number of inversions.
# of inversions = $\binom{n}{2} = \frac{n*(n-1)}{2}$

c. Suppose that the array A starts out with an inversion (k, j). Then k < j and A[k] > A[j]. At the time that the outer **for** loop of lines 1-8 sets key ←A[j], the value that started in A[k] is still somewhere to the left of A[j]. That is, it is in A[i], where $1 \le i < j$, and so the inversion has become (i, j). Some iteration of the **while** loop of lines 5-7 moves A[i] one position to the right. Line 8 will eventually drop key to the left of this element, thus eliminating this inversion. Because line 5 moves only elements that are less than the key, it moves only elements that correspond to inversions. In other words, each iteration of the while loop of lines 5-7 corresponds to the elimination of one inversion.

d. We follow the hint and modify merge sort to count the number of inversions in (n lg n) time.

To start, let us define a **merge-inversion** as a situation within the execution of merge sort in which the MERGE procedure, after copying A[p..q] to L and A[q+1..r] to R, has values x in L and y in R such that x > y. Consider an inversion (I, j), and let x = A[i] and y = A[j], so that I < j and x > y. We claim that if we were to run merge sort, there would be exactly one **merge-inversion** involving x and y. To see why, observe that the only way in which array elements change their position is within the MERGE procedure. Moreover, since MERGE keeps elements within L in the same relative order to each other, and correspondingly for R, the only way in which two elements can change their relative ordering to each other is for the greater one to appear in L and the lesser one to appear in R. Thus, there is at least one **merge-inversion** involving x and y. To see that there is exactly one **merge-inversion**, observe that after any call of MERGE that involves both x and y, they are in the same sorted subarray and will therefore both appear in L or both appear in R in any given call thereafter. Thus, we have proven the claim.

We have shown that every inversion implies one **merge-inversion**. In fact, the correspondence between inversion and **merge-inversion** is one-to-one. Suppose we have a **merge-inversion** involving values x and y, where x was originally A[i] and y was originally A[j]. Since we have a

merge-inversion, x > y. And since x is in L and y is in R, x must be within a subarray preceding the subarray containing y. Therefore x started out in a position I preceding y's original position j, and so (i, j) is an inversion.

Having shown a one-to-one correspondence between inversions and **merge-inversion**, it suffices for us to count **merge-inversions**.
Consider a **merge-inversion** involving y in R. Let x be the smallest value in L that is greater than y. At some point during the merging process, z and y will be the "exposed" values in L and R, i.e., we will have z = L[i] and y = R[j] in line 13 of MERGE. At that time, there will be **merge-inversions** involving y and L[i], L[i+1], L[i+2], ..., L[n_1], and these $n_1 - i + 1$ **merge-inversions** will be the only ones involving y. Therefore, we need to detect the first time that z and y become exposed during the MERGE procedure and add the value of $n_1 - i+1$ at that time to our total count of **merge-inversions**.

COUNT-INVERSIONS(A, p, r)

```
Inversions ← 0
if p < r
        then q ← ⌊p + r⌋ = 2c
                inversions ← inversions + COUNT-INVERSIONS(A, p, q)
                inversions ← inversions + COUNT-INVERSIONS(A, q + 1, r)
                inversions ← inversions +MERGE-INVERSIONS(A, p, q, r)
return inversions
```

MERGE-INVERSIONS(A, p, q, r)

```
n1 ← q - p + 1
n2 ← r - q
create arrays L[1..n1 + 1] and R[1..n2 + 1]
for i ← 1 to n1
        do L[i] ← A[p + i - 1]
for I ← 1 to n2
        do R[j] ← A[p + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
inversions ← 0
counted ← FALSE
for k ← p to r
        do if counted = FALSE and R[j] < L[i]
                then inversions ← inversions + n1 - i + 1
                        counted ← TRUE
        if L[i] ≤ R[j]
                then A[k] ← L[i]
```

```
            i ← i + 1
        else A[k] ← R[j]
            j ← j + 1
            counted ← FALSE
return inversions
```

The initial call is COUNT-INVERSIONS(A, 1, n).

In MERGE-INVERSIONS, the boolean variable counted indicates whether we have counted the merge-inversions involving R[j]. We count them the first time that both R[j] is exposed and a value greater than R[j] becomes exposed in the L array. We set counted to FALSE upon each time that a new value becomes exposed in R. We do not have to worry about merge-inversions involving the sentinel 1 in R, since no value in L will be greater than ∞.

Since we have added only a constant amount of additional work to each procedure call and to each iteration of the last for loop of the merging procedure, the total running time of the above pseudocode is the same as for merge sort: θ (n lg n).

Problem #2. Compare the following pairs of functions $f(n)$, $g(n)$ in terms of order of magnitude. In each case determine whether $f(n) = O(g(n))$, $f(n) = \Omega(g(n))$, and/or $f(n) = \Theta(g(n))$: (Give a brief justification for your answers.) (10pt)

Solutions:

1. $f(n) = 0.1n^{1.005} + \log\log n, g(n) = 100n + (\log n)^5$

$f(n) = \Theta(n^{1.005})$

$g(n) = \Theta(n)$

Hence, $f(n) = \Omega\big(g(n)\big)$

2. $f(n) = (\log\log n)^{\log\log n}$, $g(n) = n^{1.5}/\log n$

Let $m = \log n$, then $n = 2^m$

$f(n) = (\log m)^{\log m}$

$g(n) = (2^{1.5})^m/m$

Then, $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \lim\limits_{m\to\infty} \left(\frac{m(\log m)^{\log m}}{(2^{1.5})^m}\right) = 0$

Hence, $f(n) = O\big(g(n)\big)$

3. $f(n) = n^6 2^{n+4}, g(n) = 4^n$

$\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = \lim\limits_{n\to\infty} \frac{n^6 2^{n+4}}{4^n} = \lim\limits_{n\to\infty} \left(\frac{1}{2}\right)^n 2^4 n^5 = 0$

Hence, $f(n) = O\big(g(n)\big)$

Problem #3.

1. Show that $c^n = o((\log \log \log n)^n)$ for any constant c
2. Show that $\sum_{i=1}^{n} i^k$ is $\Theta(n^{k+1})$.

Solution:

1. $\lim_{n\to\infty} \frac{c^n}{(\log\log n)^n} = \lim_{n\to\infty} \left(\frac{c}{\log\log n}\right)^n = 0$

   Hence, $c^n = o((\log \log n)^n)$

2. $\sum_{i=1}^{n} i^k = 1^k + 2^k + \cdots + (\frac{n}{2})^k + \cdots + n^k$ (n times)

   $\Rightarrow (\frac{n}{2})^k + \cdots + n^k < \sum_{i=1}^{n} i^k < n\,(n^k)$

   $\Rightarrow (\frac{n}{2})(\frac{n}{2})^k < \sum_{i=1}^{n} i^k < n\,(n^k)$

   $\Rightarrow (\frac{n}{2})^{k+1} < \sum_{i=1}^{n} i^k < n^{k+1}$

3. $\Rightarrow \sum_{i=1}^{n} i^k = \Theta(n^{k+1})$.

10pt

Problem #4. Do Problem# 7.4-2, p. 184 [Text]. (10pt)

We'll use the substitution method to show that the best-case running time is $\Omega(n \lg n)$. Let $T(n)$ be the best-case time for the procedure QUICKSORT on an input of size n. We have the recurrence

$$T(n) = \min_{1 \le q \le n-1} (T(q) + T(n - q - 1)) + \Theta(n).$$

Suppose that $T(n) \ge c(n \lg n + 2n)$ for some constant c. Substituting this guess into the recurrence gives

$$\begin{aligned}
T(n) &\ge \min_{1 \le q \le n-1} (cq \lg q + 2cq + c(n - q - 1) \lg(n - q - 1) + 2c(n - q - 1)) + \Theta(n) \\
&= \frac{cn}{2} \lg(n/2) + cn + c(n/2 - 1) \lg(n/2 - 1) + cn - 2c + \Theta(n) \\
&\ge (cn/2) \lg n - cn/2 + c(n/2 - 1)(\lg n - 2) + 2cn - 2c\Theta(n) \\
&= (cn/2) \lg n - cn/2 + (cn/2) \lg n - cn - \lg n + 2 + 2cn - 2c\Theta(n) \\
&= cn \lg n + cn/2 - \lg n + 2 - 2c + \Theta(n)
\end{aligned}$$

Taking a derivative with respect to q shows that the minimum is obtained when $q = n/2$. Taking c large enough to dominate the $-\lg n + 2 - 2c + \Theta(n)$ term makes this greater than $cn \lg n$, proving the bound.

Problem #5 Do Problem# 4.3-8, p. 87 [Text] (Replace n_2 in recurrence by n.) (10pt)

Solution:

There is a typo in the ebook, $T(n) = 4\,T(\frac{n}{2}) + n^2$, however, in the published book, the equation is correct.

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Apply master method with:

a = 4; b = 2' f(n) = n

Master method Case 1: **T(n) =θ(n)**

Substitutions:

Guess: $T(n) \leq cn^2$ is true for some n_0

With n is larger than $2n_0$:

$T(n) = 4T(\frac{n}{2}) + n$

$T(n) \leq 4[c(\frac{n}{2})^2] + n$

$T(n) \leq cn^2 + n$

Since $n > 0$, we fail to prove that $T(n) \leq cn^2$

New guess: $T(n) \leq cn^2 - bn$ is true for some n_0

$T(n) = 4T(\frac{n}{2}) + n$

$T(n) \leq 4[c(\frac{n}{2})^2 - b\frac{n}{2}] + n$

$T(n) \leq cn^2 - bn + n(1-b)$

$T(n) \leq cn^2 - bn$ when $b \geq 1$

$\Rightarrow T(n) = O(n^2)$

Similarly, we can prove that:

$\exists c'$ and $b' < 1$ so that $T(n) \leq c'n^2 - b'n$

$\Rightarrow T(n) = \Omega(n^2)$

$\Rightarrow T(n) = \Theta(n^2)$

Problem #6 Do Problem # 4.3-9, p. 88 [Text] (10pt) note: $n = 2^m$

Solution:

Let $m = \log n$, then $n = 2^m$

$T(n) = 3T(\sqrt{n}) + \log n$

So $T(2^m) = 3T\left(2^{\frac{m}{2}}\right) + m$, Let $T(2^m) = S(m)$

$S(m) = 3S\left(\frac{m}{2}\right) + m$

$S(m) = \Theta(m^{\log_2 3})$

$T(n) = T(2^m) = S(m) = \Theta\left(m^{\log_2 3}\right) = \Theta\left((\log n)^{\log_2 3}\right)$

Problem #7. Solve the following recurrences using the method of iteration: (You may use asymtotic notations.) (15pt)

Solution:

1. $T(n) = T(n-3) + n$, where $T(0) = 1$

$T(n) = T(n-3) + n$

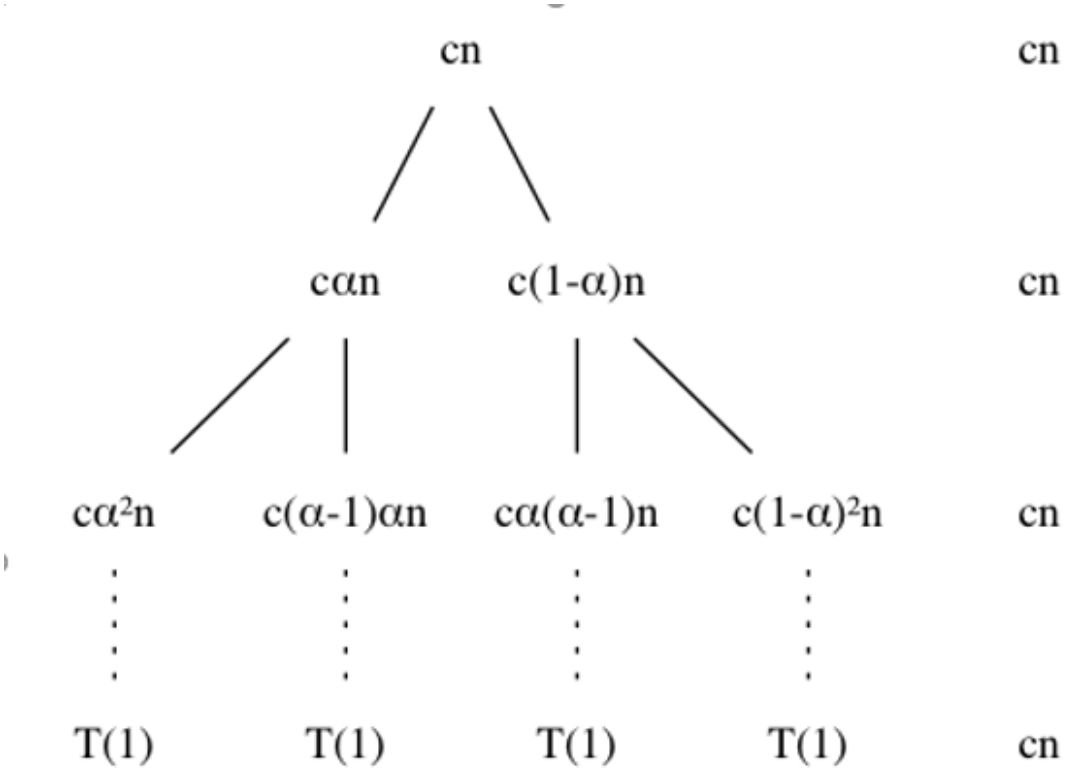
$$= [T(n-6) + (n-3)] + n$$
$$= [T(n-9) + (n-6)] + n + (n-3)$$
$$= [T(n-12) + (n-9)] + n + (n-3) + (n-6)$$
$$= \cdots$$
$$= [T(0) + 3] + n + (n-3) + (n-6) + \cdots + 6$$
$$= 1 + (3 + 6 + \cdots + n)$$
$$= 1 + \frac{n(n+3)}{6}$$
$$= \Theta(n^2)$$

2. $T(n) = 6T(n/4) + n^2$, where $n = 4^k$ and $T(1) = 1$

$$T(n) = n^2 + 6T\left(\frac{n}{4}\right)$$
$$= n^2 + 6\left[\left(\frac{n}{4}\right)^2 + 6T\left(\frac{n}{4^2}\right)\right]$$
$$= n^2 + 6\left(\frac{n}{4}\right)^2 + 6^2\left[\left(\frac{n}{4^2}\right)^2 + 6T\left(\frac{n}{4^3}\right)\right]$$
$$= n^2 + 6\left(\frac{n}{4}\right)^2 + 6^2\left(\frac{n}{4^2}\right)^2 + 6^3\left[\left(\frac{n}{4^3}\right)^2 + 6T\left(\frac{n}{4^4}\right)\right]$$
$$= n^2 + \left(\frac{6}{16}\right)n^2 + \left(\frac{6}{16}\right)^2 n^2 + \cdots + \left(\frac{6}{16}\right)^{k-1} n^2 + 6^k T(1)$$
$$= n^2\left(\frac{1 - \left(\frac{6}{16}\right)^k}{1 - \left(\frac{6}{16}\right)}\right) + 6^k$$

Where $k = \log_4 n$, we have $6^k = 6^{\log_4 n} = n^{\log_4 6}$. Hence, $T(n) = \Theta(n^2) + \Theta\left(n^{\log_4 6}\right) = \Theta(n^2)$

Problem #8. Do Problem# 4.4-9, p. 93 [Text] (10pt)



Thus, $T(n) = \sum_{i=0}^{\log_{\frac{1}{\alpha}} n} cn + \Theta(n) = cn \log_{\frac{1}{\alpha}} n + \Theta(n) = \Theta\big(nlog(n)\big)$

Problem #9. Do Problem# 6.5-9, p.166 [Text] (10pt)

Solution:

We will use heap of size k to merge sorted lists.

A[1..k]: an array for the heap.

B[1..n]: output array.

Algorithm:

```
1.  for i = 1 to k do
2.        A[i] ← the first element of ith list
3.  Build-Heap(A)
4.  for i = 1 to n do
5.        B[i] ← A[1]
6.        A[1] ← the next element of the list from which A[1] came
7.        Heapify(A)
8.  end
```

The running time line 1-3 is O(log k) and 4-8 is O(nlog k)