National Taiwan University
Department of Electrical Engineering
Algorithms, Fall 2016

Handout #6
October 19, 2016
TA: Zhi-Wen Lin and Yen-Chun Liu

Sample Solutions to Homework #1

1. (10)

   (a) See Figure 1.

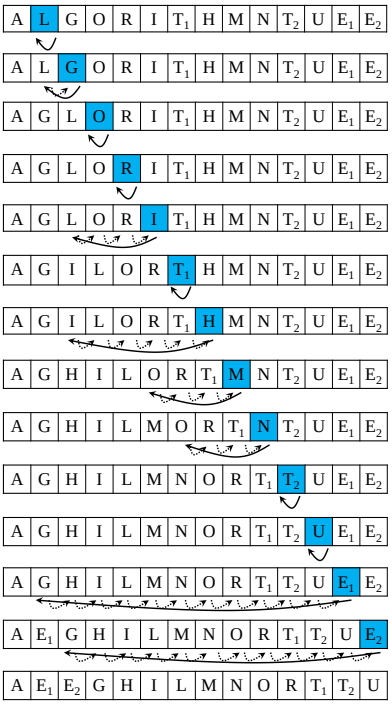

Figure 1: The process steps for Problem 1(a).

   (b) See Figure 2.

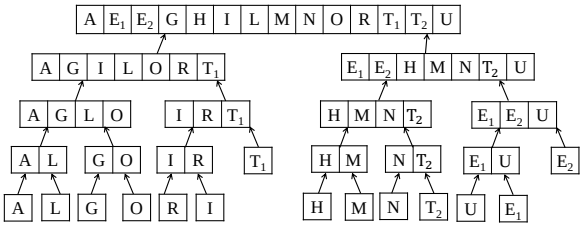

Figure 2: The process steps for Problem 1(b).

2. (20)

   (a) – n: $length[A]$.
   – t_j: # of times that the inequality in line 5 holds at iteration j.

– **Pseudocode:**

SelectionSort(A)	cost	times
1. **for** $j \leftarrow 1$ **to** $length[A] - 1$ **do**	c_1	n
2. $key \leftarrow A[j]$;	c_2	$n-1$
3. $key\_pos \leftarrow j$;	c_3	$n-1$
4. **for** $i \leftarrow j+1$ **to** $length[A]$ **do**	c_4	$\sum_{j=1}^{n-1}(n-j+1)$
5. **if** $A[i] < key$	c_5	$\sum_{j=1}^{n-1}(n-j)$
6. **then** $key \leftarrow A[i]$;	c_6	$\sum_{j=1}^{n-1} t_j$
7. $key\_pos \leftarrow i$;	c_7	$\sum_{j=1}^{n-1} t_j$
8. $A[key\_pos] \leftarrow A[j]$;	c_8	$n-1$
9. $A[j] \leftarrow key$;	c_9	$n-1$

(b) **Loop invariant:** After performing the jth iteration of the external **for** loop the subarray $A[1..j]$ contains the j smallest numbers in A, sorted in non-decreasing order.

– **Initialization:** The first iteration finds the smallest number in A and puts it in $A[1]$. Obviously the subarray is sorted, since it contains only one number.

– **Maintenance:** We must show that if the loop invariant was true at the end of iteration j, it will still hold at the end of iteration $j+1$. Thus, we can assume that at the end of iteration j the subarray $A[1..j]$ contained the j smallest numbers in A sorted in non-decreasing order. The next iteration finds the smallest number in $A[j+1..n]$ and puts it in $A[j+1]$. By our assumption, this number is equal to or greater than any of the numbers in $A[1..j]$. Therefore, the new subarray $A[1..j+1]$ is also sorted in non-decreasing order.

– **Termination:** After the last iteration j equals $n-1$, the subarray $A[1..j]$ contains the $n-1$ smallest numbers, sorted in non-decreasing order. By the same argument as before, the entire array must be sorted in non-decreasing order.

(c) Because every time we select out the smallest number from the subarray $A[j..n]$, the nth smallest number is the last one in the array. Besides, the nth smallest number is also the largest one in the array. Therefore, we do not need to change its place at all, implying that we need to run for only the first $n-1$ elements.

(d) – $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\sum_{j=1}^{n-1}(n-j+1) + c_5\sum_{j=1}^{n-1}(n-j) + c_6\sum_{j=1}^{n-1} t_j + c_7\sum_{j=1}^{n-1} t_j + c_8(n-1) + c_9(n-1)$

– **Best case:** If the array is already sorted, all t_j's are 0.
Quadratic: $T(n) = (\frac{c_4+c_5}{2})n^2 + (c_1+c_2+c_3+c_8+c_9+\frac{c_4-c_5}{2})n - (c_2+c_3+c_4+c_8+c_9) = \Theta(n^2)$

– **Worst case:** If the array is in reverse sorted order, $t_j = j, \forall j$.
Quadratic: $T(n) = (\frac{c_4+c_5+c_6+c_7}{2})n^2 + (c_1+c_2+c_3+c_8+c_9+\frac{c_4-c_5-c_6-c_7}{2})n - (c_2+c_3+c_4+c_8+c_9) = \Theta(n^2)$

3. (10) Use merge sort to sort the n elements in $\Theta(n \lg n)$ time. Then, for each element x_1, use binary search to find if there exists x_2 such that $x_1 + x_2 = x$. The total run time$= \Theta(n \lg n) + n \cdot \Theta(\lg n) = \Theta(n \lg n)$.

4. (20)

(a) The running time of this code fragment is $\Theta(n)$.

(b) Pseudocode for the native polynomial-evaluation algorithm:

```
NATIVE POLYNOMIAL-EVALUATION
1   y = 0
2   for i = 0 to n
3       z = 1
4       for j = 0 to i
5           z = z · x
6       y = y + a_i · z
```

The running time of this algorithm is $\Theta(n^2)$. This is much slower than Honor's rule.

(c) **Initialization:** Before the first iteration, $i = n$, and $y = \sum_{k=0}^{-1} a_{k+n+1}x^k = 0$. The loop invariant holds.

Maintenance: Assume that the loop invariant holds when $i = m$, and $y = \sum_{k=0}^{n-(m+1)} a_{k+m+1}x^k$ at line 2. At line 3, $y = a_m + x \cdot \sum_{k=0}^{n-(m+1)} a_{k+m+1}x^k = a_m + \sum_{k=1}^{n-m} a_{k+m}x^k = \sum_{k=0}^{n-(m-1+1)} a_{k+(m-1)+1}x^k$. The loop invariant still holds when $i = m - 1$.

Termination: After $i = 0$ iteration, $i = -1 \Rightarrow y = \sum_{k=0}^{n} a_k x^k$.

(d) By the proof in (C), we can conclude that the given code fragment can evaluate a polynomial characterized by the coefficients $a_0, a_1, ..., a_n$.

5. (10)

The worst-case time complexity of the given algorithm is $O(x)$. However, if there exists an integer k such that $k = \sqrt{x}$, this algorithm will terminate without running the loop $\frac{x}{2}$ times. Therefore, the lowerbound of the running time of the algorithm is $\Omega(\sqrt{x})$.

However, this algorithm does not run in polynomial time. Since the data in our computers are stored in bits, the maximum number x is not a proper input size that denoting the number of bits. Assume the proper input size of x is s

$$s = \lg x \Rightarrow x = 2^s$$

Thus, the original time complexity $\Omega(\sqrt{x})$ can be rewritten as $O(2^{\frac{s}{2}})$, which is obviously not a polynomial-time complexity.

6. (10) Because $f(n)$ and $g(n)$ are asymptotically positive, we have the following inequalities:

$$0 < f(n),\ \forall\ n \geq n_1$$
$$0 < g(n),\ \forall\ n \geq n_2$$

Let $n_3 = \max\{n_1, n_2\}$. We have the following inequalities:

$$0 < f(n),\ \forall\ n \geq n_3 \tag{1}$$

$$0 < g(n),\ \forall\ n \geq n_3 \tag{2}$$

Let $h(n) = \max\{f(n), g(n)\}$. We can derive the following inequality by Equations 1 and 2:

$$0 < h(n),\ \forall\ n \geq n_3 \tag{3}$$

Thus $h(n)$ is also asymptotically positive.

Moreover, by the definition of $h(n)$ and Equations 1 and 2, we have the following inequality:

$$h(n) = max\{f(n), g(n)\} \leq f(n) + g(n),\ \forall\ n \geq n_3 \tag{4}$$

Therefore, $\max\{f(n), g(n)\} = O(f(n) + g(n))$

To prove the Ω relation, we first derive an euation:

$$h(n) = max\{f(n), g(n)\} = \frac{f(n) + g(n)}{2} + \frac{|f(n) - g(n)|}{2} \geq \frac{f(n) + g(n)}{2} \tag{5}$$

The reason is that there are only three conditions of $f(n)$ and $g(n)$:

$$f(n) < g(n)$$
$$f(n) > g(n)$$
$$f(n) = g(n)$$

By Equation 5, we can see that $\max\{f(n), g(n)\} = \Omega(f(n) + g(n))$. Therefore, by showing the O and Ω relations, $\max\{f(n), g(n)\} = \Theta(f(n) + g(n))$.

7. (10)
For $n \geq |a|$, we have $0 \leq n + a \leq 2n$ and hence $(n + a)^b \leq (2n)^b = 2^b \cdot n^b$. Therefore, $(n + a)^b = O(n^b)$.

For $n \geq 2 \cdot |a|$, we have $n + a \geq n/2$ and $(n + a)^b \geq \frac{n^b}{2^b}$. Therefore, $(n + a)^b = \Omega(n^b)$. We conclude that $(n + a)^b = \Theta(n^b)$.

8. (20)
$2^{2^{n+1}} > e^n > n \cdot 2^n > 2^n > (3/2)^n > n^{\lg \lg n} >$
$(\lg n)! > n^2 = 4^{\lg n} > \lg(n!) > n = 2^{\lg n} > \sqrt{2}^{\lg n} >$
$\lg^2 n > \sqrt{\lg n} > \ln \ln n > \lg^*(\lg n) > \lg(\lg^* n) > n^{1/\lg n} = 1$

9. (20)

   (b) **False.** A counterexample: Let $f(n) = 1$ and $g(n) = n$, and $1 + n \neq O(1)$

   (d) **False.** $f(n) = O(g(n))$ does not imply $2^{f(n)} = O(2^{g(n)})$. If $f(n) = 2n$ and $g(n) = n$ we have that $2n \leq 2 \cdot n$ but not $2^{2n} \leq c2^n$ for any constant c.

   (g) **False.** Let $f(n) = 2^n$. The following statement is always invalid: $\exists c1, c2, n : \forall n \geq n_0 : 0 \leq c_1 \cdot 2^{n/2} \leq 2^n \leq c_2 \cdot 2^{n/2}$.

   (h) **True.** By Problem 6, we have that $f(n) + o(f(n)) = \Theta(\max\{f(n), o(f(n))\})$. Therefore, $f(n) + o(f(n)) = \Theta(\max\{f(n), o(f(n))\}) = \Theta(f(n))$.

10. (10)

```
MAX-SUBARRAY-LINEAR(A)
1   n = A.length
2   maxSum = −∞
3   endingHereSum = −∞
4   for j = 1 to n
5       endingHereHigh = j
6       if endingHereSum > 0
7           endingHereSum = endingHereSum + A[j]
8       else
9           endingHereLow = j
10          endingHereSum = A[j]
11      if endingHereSum > maxSum
12          maxSum = endingHereSum
13          low = endingHereLow
14          high = endingHereHigh
15  return (low, high, maxSum)
```

The variables are intended as follows:

- low and high demarcate a maximum subarray found so far.
- *maxSum* gives the sum of the values in a maximum subarray found so far.
- *endingHereLow* and *endingHereHigh* demarcate a maximum subarray ending at index j . Since the high end of any subarray ending at index j must be j , every iteration of the for loop automatically sets $endingHereHigh = j$.
- *endingHereSum* gives the sum of the values in a maximum subarray ending at index j.

The first test within the **for** loop determines whether a maximum subarray ending at index j contains just $A[j]$. As we enter an iteration of the loop, *endingHereSum* has the sum of the values in a maximum subarray ending at $j - 1$. If $endingHereSum + A[j] > A[j]$, then we extend the maximum subarray ending at index $j - 1$ to include index j. (The test in the if statement just subtracts out $A[j]$ from both

sides.) Otherwise, we start a new subarray at index j , so both its low and high ends have the value j and its sum is $A[j]$. Once we know the maximum subarray ending at index j , we test to see whether it has a greater sum than the maximum subarray found so far, ending at any position less than or equal to j . If it does, then we update low, high, and $maxSum$ appropriately. Since each iteration of the **for** loop takes constant time, and the loop makes n iterations, the running time of MAX-SUBARRAY-LINEAR is $\theta(n)$.

11. (10)

Suppose we want to compute $C = A \times B$, where A and B are two $n \times n$ matrices. Let x be $2^{\lceil \lg n \rceil}$ and y be $2^{\lfloor \lg n \rfloor}$, we will show that the time complexities of both $T(x)$ and $T(y)$ are the same and thus the same as $T(n)$. We use matrices of dimensions $x \times x$ for easier explanation (we fill the missing rows and columns with zeros). Similar to Strassen's algorithm, we divide each of A, B, and C into 2×2 matrices and apply 2×2 matrix multiplication to multiply A and B. We can derive the recurrence

$$T(x) = 3 \cdot T(x/2) + 10 \cdot (x^2).$$

Thus, the time complexity of $T(x)$ is $\Theta(n^2)$ by applying the master method. Similar method can be applied and the time complexity is the same for $T(y)$. As a result, the time complexity of $T(n)$ is $\Theta(n^2)$.

12. (20)

(a)(10) Substitution proof with the assumption $T(n) \leq cn^{\log_3 4}$:

$$\begin{aligned} T(n) &= 4T(n/3) + n \\ &\leq 4c(n/3)^{\log_3 4} + n \\ &= 4c(n^{\log_3 4}/3^{\log_3 4}) + n \\ &= 4c(n^{\log_3 4}/4) + n \\ &= cn^{\log_3 4} + n \\ &\not\leq cn^{\log_3 4} \end{aligned}$$

The substitution proof with the assumption $T(n) \leq cn^{\log_3 4}$ fails.

Assuming the lower-order term to be subtracted is dn, then the substitution proof with assumption $T(n) \leq cn^{\log_3 4} - dn$ will be

$$\begin{aligned} T(n) &= 4T(n/3) + n \\ &\leq 4c(n/3)^{\log_3 4} - (4/3)dn + n \\ &= cn^{\log_3 4} + (1 - (4/3)d)n \end{aligned}$$

By choosing $d \geq 3$ and $c \geq 4$ (assume $T(1) = 1$), the substitution proof with the assumption $T(n) \leq cn^{\log_3 4} - dn$ works.

(b)(10) Let $m = \lg n$, and then $n = 2^m$. Thus, we have $T(2^m) = 3T(2^{m/2}) + m$.
Rename $S(m) = T(2^m)$ to produce the new recurrence $S(m) = 3S(m/2) + m$.
Apply the master theorem, and we know that $S(m) = \theta(m^{lg3})$.
Thus, $T(n) = T(2^m) = S(m) = \theta(m^{lg3}) = \theta(lg^{lg3}n)$.

13. (10)
Let $x = \frac{1}{\alpha}$ and $y = \frac{1}{1-\alpha}$. Construct the recurrence tree as Figure 3. We can obviously see that $T(n) = \Theta(nlgn)$.

14. (10)
Apply the master theorem:
$n^{\log_b a} = n^2$, $f(n) = n^2 \lg n$, and we cannot find $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$. That means we cannot apply the master method to solve this recurrence.

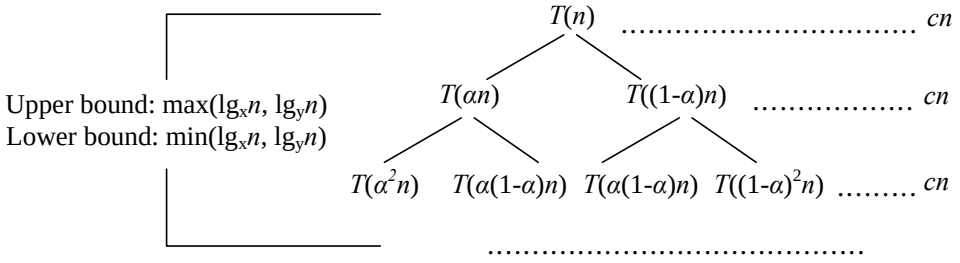

Figure 3: The recursion tree for Problem 13.

Solve by iteration:

$$\begin{aligned}
T(n) &= 4T(\frac{n}{2}) + n^2 \lg n \\
&= 16T(\frac{n}{4}) + n^2 \lg n + n^2 \lg \frac{n}{2} \\
&= O(\sum_{k=0}^{\lg n}(n^2 lg \frac{n}{2^k})) \\
&= O(\sum_{k=0}^{\lg n}(n^2 \lg n - kn^2)) \\
&= O(n^2 \lg n(\lg n + 1) - \frac{n^2}{2} \lg n(\lg n + 1)) \\
&= O(\frac{n^2}{2} \lg n(\lg n + 1))) \\
&= O(n^2 \lg^2 n)
\end{aligned}$$

15. (15)

    (d) $T(n) = 7T(n/3) + n^2 = \Theta(n^2)$. Using master theorem, $f(n) = n^2 = \Omega(n^{\log_b a + \varepsilon}) = \Omega(n^{1.77+\varepsilon})$ for some const $\varepsilon > 0$, and $af(n/b) = 7n^2/9 \leq cf(n) = cn^2$ holds for any $7/9 \leq c < 1$. Thus, case 3 of master theorem applies, and $T(n) = \Theta(n^2)$.

    (e) $T(n) = 7T(n/2) + n^2 = \Theta(n^{\lg 7})$. Since $2 < \lg 7 < 3$, we have that $n^2 = O(n^{\log_2 7 - \varepsilon})$ for some const $\varepsilon > 0$. Thus, case 1 of the master theorem applies, and $T(n) = \Theta(n^{\lg 7})$.

    (f) Apply the master theorem: $n^{\log_b a} = n^{\frac{1}{2}}$, $f(n) = n^{\frac{1}{2}}$, and thus $T(n) = \Theta(n^{\frac{1}{2}} \lg n)$.

16. (15)

    (a) Apply the master theorem: $n^{\log_b a} = n^{\log_3 4} = n^{1.261\ldots}$, $f(n) = n \lg n$, and $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$. Thus, $T(n) = \Theta(n^{\log_3 4})$.

    (e) Using the iteration method, we get (if n is a power of 2)

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

    (f) Drawing the recursion tree, we get

$$\begin{aligned}
T(n) &= n + \frac{7}{8}n + \frac{49}{64}n + \ldots + (\frac{7}{8})^{\lg n} n \\
&= \Theta(n).
\end{aligned}$$

17. (20)

(a)(5) $(1,2)$, $(2,3)$, $(2,4)$, $(3,4)$.

(b)(10) The following is the pseudo code:

```
Num_inversion(A, p, r)
1   if p < r then
2       q = ⌊(p+r)/2⌋
3       n_1 =Num_inversion(A, p, q)
4       n_2 =Num_inversion(A, q, r)
5       n_3 =Merge_inv(A, p, q, r)
6       return n_1 + n_2 + n_3
7   else return 0
```

```
Merge_inv(A, p, q, r)
1   n_1 = q − p + 1
2   n_2 = r − q
3   let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays
4   for i = 1 to n_1
5       L[i] = A[p + i − 1]
6   for j = 1 to n_2
7       R[j] = A[q + j]
8   L[n_1 + 1] = ∞
9   R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  c = 0
13  for k = p to r
14      if L[i] ≤ R[j]
15          A[k] = L[i]
16          i = i + 1
17      else
18          A[k] = R[j]
19          c = c + n_1 + 1 − i
20          j = j + 1
21  return c
```

(c)(5) Because the differences between the new code and MergeSort do not introduce additional loops, the time complexity is unchanged. Therefore, by the same method for MergeSort, the recurrence of the running time for the new algorithm is the same as MergeSort and its time complexity is $\Theta n \lg n$, where n is the number of elements.

18. (40) DIY.