National Taiwan University
Department of Electrical Engineering
Algorithms, Fall 2019

November 2, 2019
Handout #12
TAs: Yu-Jie Cai, Chen-Hung Wu

Sample Solutions to Homework #2

1. (10) Modified Problem 7-1(a)

   See Figure 1.

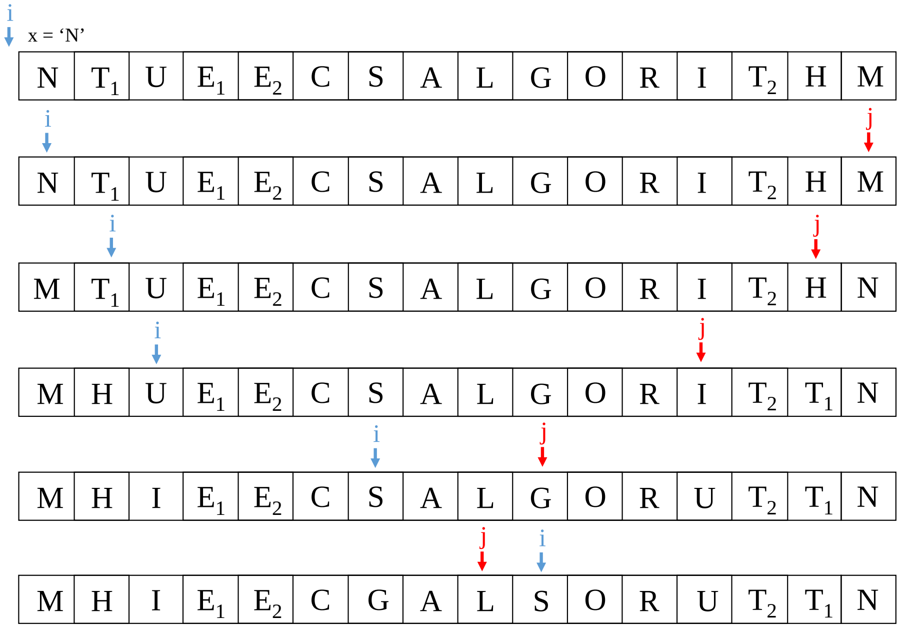

i
j
x = 'N'
N T1 U E1 E2 C S A L G O R I T2 H M
N T1 U E1 E2 C S A L G O R I T2 H M
M T1 U E1 E2 C S A L G O R I T2 H N
M H U E1 E2 C S A L G O R I T2 T1 N
M H I E1 E2 C S A L G O R U T2 T1 N
M H I E1 E2 C G A L S O R U T2 T1 N

Figure 1: The process steps for Problem 1.

2. (10) Modified Exercise 8.2-1

   See Figure 2 at next page.

3. (10) Exercise 8.2-4

   Construct a counting array C used in *counting sort* in $O(n+k)$ time. Since $C[a]$ denotes the number of integers that fall into the range $[0, a]$, it is obvious to see the answer is $C[b] - C[a-1]$, where $C[-1]$ is defined as 0. Therefore, the query then can be answered in $O(1)$ time.

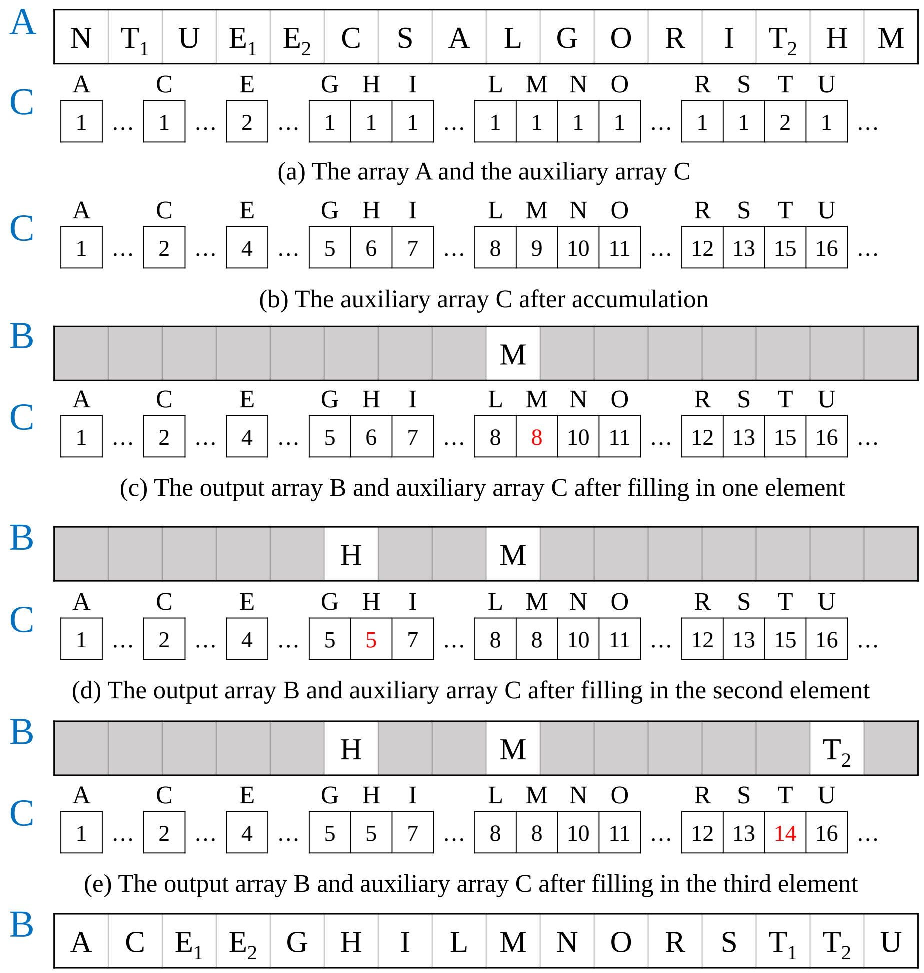
4. (15) Problem 8-4 (a), (b)

   (a) Since there are n red jugs and n blue jugs, it takes worst-case $\Theta(n^2)$ time to compare each red jug with each blue jug.

   (b) The computation of the algorithm can be viewed in terms of a decision tree. Every internal node is labelled with two jugs (red and blue) and has three outgoing edges (red jug smaller, same size, or larger than the blue jug). The leaves are labelled with a unique matching of jugs.

   The height of the decision tree is equal to the worst-case number of comparisons the algorithm has to make to determine the matching. To bound that size, we first compute the number of possible matchings for n red and n blue jugs. If we label both the blue and the red jugs from 1 to n before starting the comparisons, every outcome of the algorithm can be represented as a set:

$$\{(i, \pi(i)) : 1 \leq i \leq n \textit{ and } \pi \textit{ is a permutation on } \{1, ..., n\}\},$$



(a) The array A and the auxiliary array C

(b) The auxiliary array C after accumulation

(c) The output array B and auxiliary array C after filling in one element

(d) The output array B and auxiliary array C after filling in the second element

(e) The output array B and auxiliary array C after filling in the third element

(f) The sorted result

Figure 2: The process steps for Problem 2.

which contains the pairs of red jugs (first component) and blue jugs (second component) that are matched up. Since every permutation π corresponds to a different outcome, there must be exactly $n!$ different results.

Therefore, the height h of the decision tree can be bounded as follows: every tree with a branching factor of 3 (every inner node has at most three children) has at most 3^h leaves, and the decision tree must have at least $n!$ children. This statement follows that

$$3^h \geq n! \geq (n/e)^n \Rightarrow h \geq n \log_3 n - n \log_3 e = \Omega(n \lg n).$$

So any algorithm solving this problem must use $\Omega(n \lg n)$ comparisons.

5. (15) Problem 9.1-1

   First find by the maximum using a *tennis tournament* structure: compare the n elements in pairs, then compare the $n/2$ *winners* in pairs, and so on. (Unpaired elements get a bye to the next round.) Since every element except the winner loses exactly once, this takes $n-1$ comparisons. But now note that the second largest element must be one which loses to the winner, as it could have not been defeated by any other elements. So you need to find the maximum among all the (up to) $\lceil \lg n \rceil$ elements that were defeated by the winner, and finding this maximum can be done in $\lceil \lg n \rceil - 1$. Finally, the total number of comparisons is $n + \lceil \lg n \rceil - 2$.

6. (10) Exercise 9.3-8

This problem can be solved by binary search. Let p be the median of $X[1..n]$, and q be the median of $Y[1..n]$. If $p > q$, then the median of the two arrays is in $X[1..\frac{n}{2}]$ or $Y[\frac{n}{2}..n]$. If $p \leq q$, then the median of the two arrays is in $X[\frac{n}{2}..n]$ or $Y[1..\frac{n}{2}]$. Therefore, we can use recursion to solve this problem. For any instance with two sorted arrays $X[1..n]$ and $Y[1..n]$, we first compare the medians of them, and then follow the mentioned rules to divide the two arrays into half of the original size. After dividing, we can set the new smaller arrays as the new instance and apply the same strategy recursively.

Obviously, this binary-search approach divides the problem size into half size after each recursion. Thus we have the time complexity, $T(n) = T(n/2) + \Theta(1) = O(\lg n)$.

7. (15) Exercise 12.2-1 (a), (b), and (d)

   (a) **O**

   (b) **O**

   (d) **O**

8. (10) Problem 12-2

   One possible algorithm works in the following steps.

   (a) Insert the sequences one by one into the radix tree.

   (b) Traverse the radix tree with pre-order tree walk, and write down sequences visited.

   The correctness of the algorithm can be justified by proving that the pre-order tree walk visits sequences in monotonically increasing order. According to the structure of radix trees and the definition of "lexicographically less than," for any node i in a radix tree, we have

   (The sequence on i) < (any sequence in the left subtree of i) < (any sequence in the right subtree of i.)

   Therefore, the pre-order tree walk does visit sequences in monotonically increasing order.

   The timing bound can be derived as follows. Inserting the sequences takes $\Theta(n)$ time and the pre-order tree walk takes $\Theta(n)$ time. In conclusion, the timing of the algorithm is $\Theta(n)$.

9. (20) Search trees

   - See Figure 3.

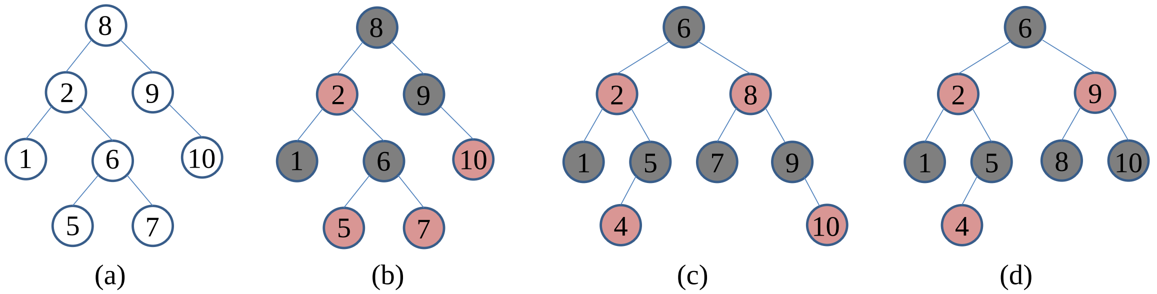


Figure 3: Sample trees for Problem 9.

10. (30) Problem 13-2

    (a) When *RB-Insert* and *RB-Delete* are performed, *T.hb* can only be updated by "*Insertion Fixup*" and "*Deletion Fixup*", respectively. According to the pseudocodes of the two fixup functions, *T.hb* can be maintained by checking whether the root's color is changed or not at the end of the function. Thus, the asymptotic running time is unchanged and no extra storage in the tree nodes is required.

    While descending through T, we can determine the black-height of each node by starting with *T.hb* and subtracting 1 for each black node visited along the path. It can be done in $O(1)$ time per node.

(b) We start from the root, and descend through T_1 along the right child of each node until the black-height of the traversed node is equal to $T_2.hb$. According to the binary-search-tree property, the chosen node is the largest one among the nodes whose black-height is $T_2.hb$.

(c) We create a subtree T_x with root x, left subtree T_y, right subtree T_2 and we replace the original subtree T_y in T_1 with the subtree T_x. The binary-search-tree property holds because $x_1.key \leq x.key \leq x_2.key$ for any $x_1 \in V(T_1)$ and $x_2 \in V(T_2)$.

(d) We should color x red. Because we know that y and the root of T_2 are black and $T_y.hb = T_2.hb$, red-black properties 1, 3, 5 are maintained.The fixup function is applied only when y's parent was red (property 2). The function takes $O(lgn)$ time to enforce properties 2 and 4.

(e) When $T_1.hb \leq T_2.hb$, the following symmetric situations arise. In T_2, we replace the subtree T_y rooted at y using T_x. Note that y is the smallest one among the nodes whose black-height is $T_1.hb$. T_x has root x, left subtree T_1, right subtree T_y. The fixup function is applied similarly when y's parent is red.

(f) According to the analysis of each step above, we conclude that the total time complexity is $O(lgn)$.

11. (30) Dynamic programming implementations

(a) The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 5$, and the sequence of dimensions $< 3, 5, 10, 6, 8, 30 >$ are shown in Figure 4. The minimum number of scalar multiplications to multiply the five matrices is $m[1, 5] = 1194$, and its corresponding parenthesization is $((((A_1A_2)A_3)A_4)A_5)$.

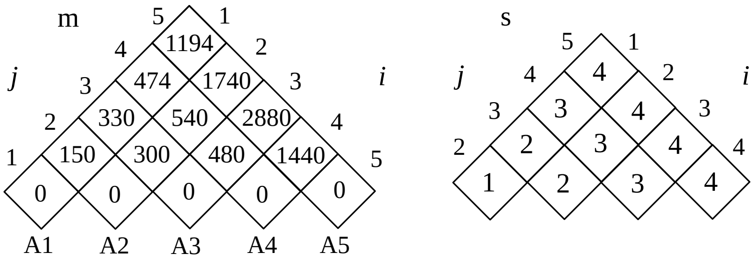


Figure 4: The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 5$ and the sequence of dimensions $< 3, 5, 10, 6, 8, 30 >$.

(b) The table computed by LCS-LENGTH on the sequence $X =< A, B, C, D, A, B >$ and $Y =< B, D, A, C, D, B >$ is shown in Figure 5. The longest common subsequence of X and Y is $< A, C, D, B >$.

(c) The e, w, and *root* tables computed by OPTIMAL-BST for the given probabilities are shown in Figure 6(a), (b) and (c). The lowest expected search cost of any binary search tree for the given probabilities is 2.96, and the corresponding structure is shown in Figure 6(d).

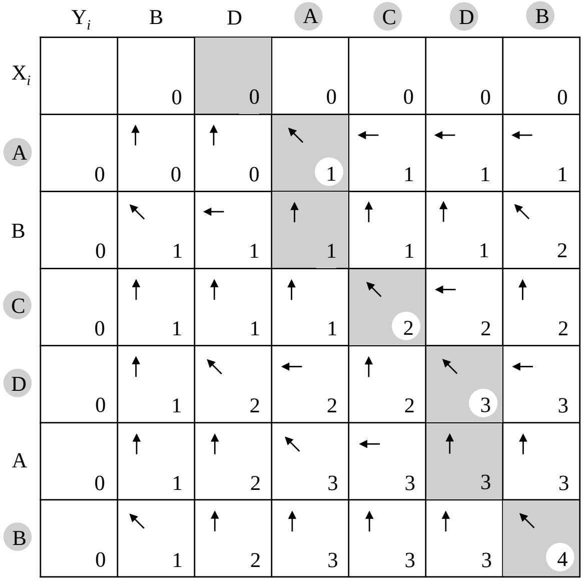
12. (15)

(a) Let $L = 4$, $n = 2$, and the two labelled positions are at d_1 and d_2. If we first cut the wood on d_1 and second on d_2, the total payment is $d_3 + (d_3 - d_1) = 2d_3 - d_1$. On the other hand, if we first cut the wood on d_2 and second on d_1, the total payment will be $d_3 + d_2$.

(b) Suppose $i < m < j$, where i, m, j are integers, then we give the following recurrence of $c(i, j)$:

$$c(i,j) = \begin{cases} 0 & \text{,if } j - i \leq 1, \\ \min_{i<m<j}\{c(i,m) + c(m,j)\} + (d_j - d_i) & \text{,otherwise.} \end{cases}$$

Suppose the optimal solution is achieved when $m = m_0$, and the resuting two sublogs are $(d_{m_0} - d_i)$-length and $(d_j - d_{m_0})$-length. We claim that the sequence of cuts to each sublog is optimal to itself. Otherwise, we can find a better solution to the original log by finding another better solution to its sublogs.



	Y_i	B	D	A	C	D	B
X_i		0	0	0	0	0	0
A	0	↑ 0	↑ 0	↖ 1	← 1	← 1	← 1
B	0	↖ 1	← 1	↑ 1	↑ 1	↑ 1	↖ 2
C	0	↑ 1	↑ 1	↑ 1	↖ 2	← 2	↑ 2
D	0	↑ 1	↖ 2	← 2	↑ 2	↖ 3	← 3
A	0	↑ 1	↑ 2	↖ 3	← 3	↑ 3	↑ 3
B	0	↖ 1	↑ 2	↑ 3	↑ 3	↑ 3	↖ 4

Figure 5: The table computed by LCS-LENGTH on the sequence $X =< A, B, C, D, A, B >$ and $Y =< B, D, A, C, D, B >$.

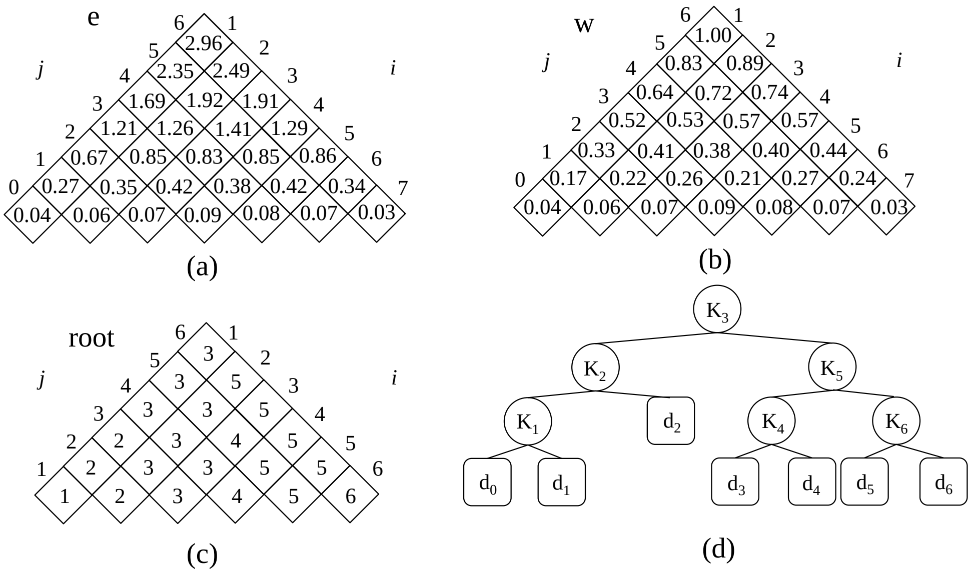


Figure 6: The e, w, and *root* tables computed by OPTIMAL-BST for the given probabilities, and the structure of a binary search tree with the lowest expected search cost.

(c) We show an algorithm for the wood-cutting problem as follows:

```
WOODY-CUT($C, n, d_{1..n}$):
    for $i = 1$ to $n$
        $C[i, i+1] = 0$
    for $i = 1$ to $n - 1$
        $C[i, i+2] = d_{i+2} - d_i$
    for $t = 3$ to $n + 1$
        for $i = 0$ to $n + 1 - t$
            $j = i + t$
            $c_{min} = \infty$
            for $m = i + 1$ to $j - 1$
                if $c_{min} > C[i, m] + C[m, j] + (d_j - d_i)$
                    $c_{min} = C[i, m] + C[m, j] + (d_j - d_i)$
            $C[i, j] = c_{min}$
    return $C$
```

To fill the table C, it takes $(n-1) + 2(n-2) + ... + (n-1)1$ comparisons. Since $(n-1) + 2(n-2) + ... + (n-1)1 = \sum_{i=1}^{n} i(n-i) = n\sum_{i=1}^{n} i + \sum_{i=1}^{n} i^2 = O(n^3)$, we conclude that the running time of the algorithm is $O(n^3)$.

13. (20)

  (a) The optimal row length is 34, and the cell c_1, c_2 and c_4 should be flipped.

  (b) i. Optimal substructure: If the minimum row length through f_j^{α} is through f_{j-1}^{p}/f_{j-1}^{r}, the minimum row length from the starting point to f_{j-1}^{p}/f_{j-1}^{r} must be taken.

   ii. Overlapping subproblems: The minimum row length through f_j^{α} is either the minimum length until f_{j-1}^{p} plus $\phi_{c_{j-1}^{p}, c_j^{\alpha}}$, or the minimum length until f_{j-1}^{r} plus $\phi_{c_{j-1}^{r}, c_j^{\alpha}}$.

  (c)

$$T(f_i^{\alpha}) = \begin{cases} x_i, & \text{if } i = 1, \\ w_{i-1} + \min_{\beta \in \{p,r\}} \{T(f_{i-1}^{\beta}) + \phi_{c_{i-1}^{\beta}, c_i^{\alpha}}\}, & \text{if } i > 1. \end{cases} \tag{1}$$

$$T(f^*) = w_{n_{s_j}} + \min\{T(f_{n_{s_j}}^{p}), T(f_{n_{s_j}}^{r})\} \tag{2}$$

  (d) Both the time and space complexity of the following dynamic programming algorithm are $\Theta(n)$.

```
**Minimum-Row-Length**($f, w, \phi, n$)
$T_p[1] = 0$
$T_r[1] = 0$
**for** $i = 2$ to $n$
    **if** $T_p[i-1] + \phi_{c_{i-1}^{p}, c_i^{p}} \leq T_r[i-1] + \phi_{c_{i-1}^{r}, c_i^{p}}$
        $T_p[i] = T_p[i-1] + w_{i-1} + \phi_{c_{i-1}^{p}, c_i^{p}}$
    **else**
        $T_p[i] = T_r[i-1] + w_{i-1} + \phi_{c_{i-1}^{r}, c_i^{p}}$
    **if** $T_p[i-1] + \phi_{c_{i-1}^{p}, c_i^{r}} \leq T_r[i-1] + \phi_{c_{i-1}^{r}, c_i^{r}}$
        $T_r[i] = T_p[i-1] + w_{i-1} + \phi_{c_{i-1}^{p}, c_i^{r}}$
    **else**
        $T_r[i] = T_r[i-1] + w_{i-1} + \phi_{c_{i-1}^{r}, c_i^{r}}$
$T^* = w_n + \min\{T_p[n], T_r[n]\}$.
```

14. (40) DIY.