CSCE-629 Homework 3

Wang, Han-Yi
wanghy917@email.tamu.edu

September 4, 2020

Exercise 15.1-3

Algorithm 1 BOTTOM-UP-CUT-ROD-WITH-COST(p, n, c)

```
1 let r[0...n] be a new array
2 r[0] = 0
3 for j = 1 to n
4     q = p[j]
5     for i = 1 to j − 1
6         q = max(q, p[i] + r[j − i] − c)
7     r[j] = q
8 return r[n]
```

We made the following changes to the BOTTOM-UP-CUT-ROD algorithm.

- If there is no cut, the revenue is set to $p[j]$ (line 4)
- The inner loop (line 5-6) iterates from i to $j-1$ because "no-cut"" is considered at line 4
- The revenue cost needs subtract the cost c (line 6)

We show the updated version in the above BOTTOM-UP-CUT-ROD-WITH-COST algorithm, and the text with blue color depicts the difference comparing with the original one.

Exercise 15.3-2

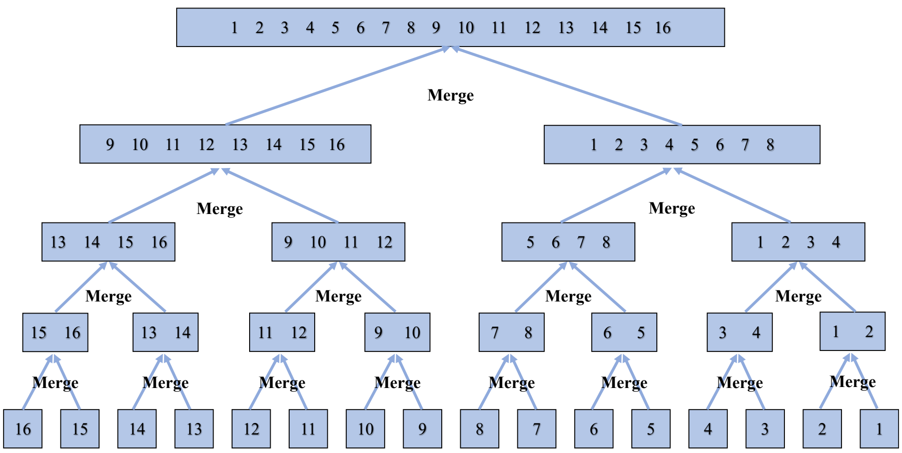

Figure 1: The recursion tree of merge sort.

Figure 1 shows the recursion tree of the merge sort. For two elements $A[i]$ and $A[j]$, $i \neq j$, they only compare with each other one time. After merging into the same subarray, the elements only compare to the elements in other subarrays later. For example, “16” only compares with “15” once in the first merge step, and it never compares with “15” again. Therefore, memorization fails to speed up the merge sort, a good divide- and-conquer algorithm, even if we memorize each pair of elements’ comparison results.

Exercise 15.4-5

Algorithm 2 LONGEST-INCREASING-SUB-SEQUENCE(A)

```
1  let DP be a empty array
2  for i = 1 to A.length
3      found = false
4      for j = 1 to DP.length
5          if DP[j] > A[i]
6              found = true
7              DP[j] = A[i]
8              break
9      if found == false
10         append A[i] onto the end of DP
11 return DP
```

We assume all elements in A are unique and adopt the above LONGEST-INCREASING-SUB-SEQUENCE. The array A is the input, and the array DP is the returned answer. For each item $A[i]$, we search for the first larger item in DP and replace it. If all items are smaller, we append $A[i]$ to the end of DP. Therefore, DP is always monotonically increased, and the length is maximized when returning from the algorithm.

The outer loop runs n (line 2-10) times, and the inner loop (line 4-8) runs at most $n - 1$ times, so the time complexity is $O(n^2)$.

Problem 15-5

a. The EDIT-DISTANCE algorithm (page 3) transforms the input X into Y with six operations. The notations are described as below:

- m and n represent the length of X and Y
- $x[i]$ and $y[j]$ represent the i_{th} and the j_{th} elements of X and Y,
- C is a cost table, for example, $C[$“*copy*”$]$ means the cost of the copy operation
- $dp[i, j]$ stores the minimum edit distance of transforming from $x[0..i]$ to $y[0..j]$

Algorithm 3 EDIT-DISTANCE(X, Y, m, n, C)

```
1  let dp[0..m, 0..n] be a new table
2  let op[0..m, 0..n] be a new table
3  dp[0, 0] = 0, op[0, 0] = “”
4  for i = 1 to m
5      dp[i, 0] = i × C[“delete”]
6      op[i, 0] = “delete”
7  for j = 1 to n
8      dp[0, j] = j × C[“insert”]
9      op[0, j] = “insert”
10 dp[0, n] = dp[0, n] + C[”kill”]
11 for i = 1 to m
12     for j = 1 to n
13         let cur be a new hash table
14         cur[“replace”] = dp[i − 1, j − 1] + C[“replace”]
15         cur[“delete”  ] = dp[i − 1, j] + C[“delete”  ]
16         cur[“insert”  ] = dp[i, j − 1] + C[“insert”  ]
17         if  x[i] == y[j]
18             cur[“copy”] = dp[i − 1, j − 1] + C[“copy”]
19         if i > 1 and j > 1 and x[i] == y[j − 1] and x[i − 1] == y[j]
20             cur[“twiddle”] = dp[i − 1, j − 1] + C[“twiddle”]
21         dp[i, j] = ∞
22         for key, value in cur
23             if  value < dp[i, j]
24                 dp[i, j] = value
25                 op[i, j] = key
26     if i < m
27         dp[i, n] = dp[i, n] + C[“kill”]
28 dist = ∞
29 idx = 0
30 for i = 0 to m
31     if dp[i, n] < dist
32         dist = dp[i, n]
33         idx = i
34 return dist, idx, op
```

The EDIT-DISTANCE algorithm returns the minimum edit distance *dist*, the index of the latest operation *idx*, and the optimal operation table *op* for each steps. The time complexity of the algorithm is $\theta(mn)$ since the outer loop (line 11-27) runs m times, and the inner loop (line 12-25) runs n times. The space requirement is $\theta((m+1)\times(n+1)) = \theta(mn)$ because both the table *dp* and *op* require $(m+1)\times(n+1)$ space.

We use PRINT-EDIT-DISTANCE to print the result. We print the sequence by tracing backward from the end index *idx* over the stored optimal operation table *op*.

Algorithm 4 PRINT-EDIT-DISTANCE(op, m, n, idx)

```
let seq be an empty array                          ▷ index start from 1
i = idx
j = n
if idx < m
    append "kill" to seq
while  i ≥ 0 and j ≥ 0
    if i == 0 and j == 0
        break
    append op[i, j] onto the end of seq
    if op[i, j] == "twiddle"
        i = i − 2
        j = j − 2
    else if op[i, j] == "insert"
        j = j − 1
    else if op[i, j] == "delete"
        i = i − 1
    else                                           ▷ copy or replace
        i = i − 1
        j = j − 1
for i = sequence.length downto 1
    print seq[i]
return
```

b. We could cast the problem of optimal alignment as an edit distance problem:

1. Given two DNA sequences, we regard them as two input string to the EDIT-DISTANCE
2. We assign the cost of each operation as the below Table 1
3. We apply the EDIT-DISTANCE algorithm
4. The negative of the returned minimum edit distance is the score of the sequence alignment

operation	**cost**
copy	-1
replace	$+1$
delete	$+2$
insert	$+2$
twiddle	$+\infty$
kill	$+\infty$

Table 1: Cost of operations for DNA sequence alignment