CS 6363: Design and Analysis of Algorithms - Fall 2019
Homework #2 Solutions
Professor D. T. Huynh

Problem 1: Given an array of integers A[1..n] such that $|A[i] - A[i + 1]| \leq 1$ for all $i = 1, \ldots, n - 1$. (That is the values of adjacent elements differ by at most 1.) Let $x = A[1]$ and $y = A[n]$, and assume that $x < y$. Design an efficient search algorithm to find j such that $A[j] = z$ for a given value z with $x \leq z \leq y$. Estimate the number of comparisons required by your algorithm (it should be O(logn)). Using the decision tree method, show that your algorithm is optimal.

Solution:

Consider the following two conditions:

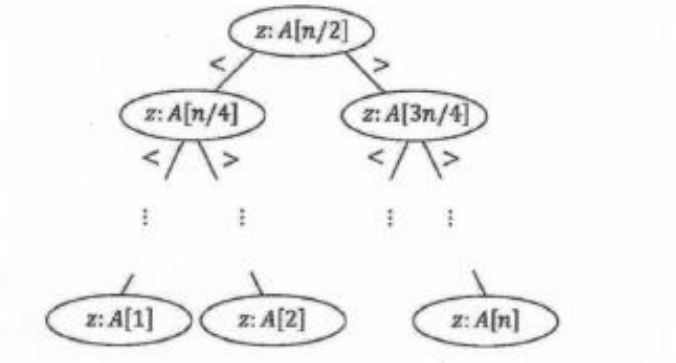
1. $|A[i] - A[i+1]| \leq 1$
2. $x \leq z \leq y$

From the conditions, we know that the array is a continuous function w.r.t. integer.
Therefore, we can apply binary search to find the value z.

Algorithm: b-search (A, i, j)

```
1.  begin
2.      if j = i + 1 and z ≠ A[i] and z ≠ A[j]
3.          then return ∞ //z does not exsit.
4.      k ← ⌊(i+j)/2⌋
5.      if z = A[k] then return k
6.      else if z > A[k]
7.          then b-search(A, k + 1, j)
8.      else
9.          b-search(A, i, k − 1)
10. end
```

We assume that z is a global variable and initial values of i and j are 1 and n respectively.
There are n leaves in the tree and its height is $\log_2 n$.
Thus, we need $O(\log n)$ computations, and this is an optimal time complexity for comparison -based search algorithm.



Problem #2: You have n coins that are supposed to be gold coins of the same weight, but you know that one coin is fake and weighs less than the others. There is available a balance scale so that you can put any number of coins on each side of the scale at one time, and it will tell you whether the two sides weigh the same, or which side weigh less than the other. Outline an efficient algorithm to find the fake coin. Determine the number of weighings as a function of n using the O notation.

Solution:

(Method 1)

1. Divide n coins into two piles of $\lceil \frac{n}{2} \rceil$ coins each, leaving one extra coin apart if n is odd.
2. Put the two piles on the scale. If the piles weigh the same, the coin put aside must be fake; otherwise we can continue with the lighter pile containing the fake coin.

Time Complexity: $T(n) = T(\lceil \frac{n}{2} \rceil) + 1$ for $n > 1$ and $T(1) = 0$ (the worst case). Thus, $T(n) = \Theta(\log_2 n)$

(Method 2)

1. Divide n coins into three piles of $\lceil \frac{n}{3} \rceil$, $\lceil \frac{n}{3} \rceil$ and $n - 2\lceil \frac{n}{3} \rceil$.
2. Put the first two piles on the scale. The lighter one of two should contain the fake coin. If both piles weigh equal, then the third pile should contain the fake coin.
3. After weighing two of the piles, we are left to solve a single problem of the one-third the original size. So, it is a decrease by a factor of 3.

Time Complexity: $T(n) = T(\lceil \frac{n}{3} \rceil) + 1$ for $n > 1$ and $T(1) = 0$. Thus, $T(n) = \Theta(\log_3 n)$

Problem #3: Let A[1..n] be an array of n elements. An element x is called a majority element in A[1..n] if Card({ i | A[i] = x }) > 2n/3. Give an O(n) algorithm that decides whether A[1..n] contains a majority element, and if so finds it.

Solution:

(Solution 1)

Idea: If there exists a majority element that appears more than $2n/3$ times in a given array, then the majority must be the median of the input array of $n > 1$ distinct elements.

Algorithm:

1. Given an unsorted input array $A[1..n]$, we use SELECT algorithm to determine the median x of A.
2. Then we pass over the array and count the number times C that x appears. If $C > 2n/3$, then a majority element exists and it is x. Otherwise, there is no majority element.

Complexity: SELECT algorithm runs in $O(n)$. Counting the number times that a majority element appears takes $O(n)$. Therefore, the algorithm runs in $O(n)$.

(Solution 2)

We will first try to eliminate as many elements as we can from being candidates for majority. It runs out that we can eliminate all but one element. Finding this one candidate is helped by the following observation, which allows us to reduce the problem to a smaller one:

- If $x_i \neq x_j$ and we eliminate both of these elements from the list, then the majority in the original list remains a majority in the new list.

(Notice that the opposite is not true: the list $1, 2, 5, 5, 3, 5$ has no majority, but if we remove 1 and 2, then 5 becomes a new majority.)
So, if we find two unequal cards, we eliminate both, find the majority in the smaller list, and check whether it is a majority in the original list. What if we do not find unequal cards? If we scan the cards and they are all equal, then we have to keep track of only one possible candidate; once we find a vote that is not equal, then we have to check only one candidate. This is the seed of the idea; we now show how to implement it.
The cards are scanned in the order they appear. We use two variables, C (candidate) and M (multiplicity). When we consider x_i, C is the only candidate for majority among $x_1, x_2, \ldots, x_{j-1}$, and M is the number of times C appeared so far excluding the times C was eliminated.

Algorithm: Majority(A, n)
Input: An array $A[1..n]$
Output: Majority (the majority in A if it exists, or -1 otherwise)

```
1.   C ← A[1]
2.   M ← 1
     //first scan; eliminate all but one card C
3.   for i ← 2 to n do
4.       if M = 0 then
5.           C ← A[i]
6.           M ← 1
7.       else
8.           if C = A[i] then M ← M + 1
9.           else M ← M − 1
     //second scan; check whether C is a majority
10.  if M = 0 then Majority ← −1
11.  else
12.      Count ← 0
13.      for i ← 1 to n do
14.          if A[i] = C then Count ← Count + 1
15.      if Count > (2n/3) then Majority ← C
16.      else Majority ← −1
17.  return Majority
```

Complexity: We used $n - 1$ comparisons to find a candidate and $n - 1$ comparisons, in the worst case, to determine whether this candidate is majority. Thus, overall, there are at most $2n - 2$ comparisons. In any case, since there are constant number of other operations per comparison, the overall running time is $O(n)$.

Problem #4: Consider the alphabet Σ = {a, b, c}, and the following multiplication table: (note that this multiplication is *not* associative)

	a	b	c
a	c	b	b
b	a	b	a
c	b	a	a

Design an efficient algorithm that examines an input string $w = w_1 \ldots w_n \in \Sigma^*$, and decides whether or not it is possible to parenthesize w in such a way that the value of the resulting expression is a. For instance, if w = bbbba, then your algorithm should return "Yes" since ((b(bb))(ba)) = a. (Hint. Use dynamic programming!)

Solution:

Let $w_{ij} = w_i w_{i+1} \cdots w_j \in \Sigma^*$ and $V_{ij} \subseteq \Sigma$ denote the set of the possible symbols that could be generated with w_{ij}.
If $a \in V_{1n}$, then algorithm returns "Yes".

Algorithm: Non-Associative Multiplication
Input: String $w_1 w_2 \cdots w_n$, and multiplication table
Output: "Yes" if the value of the resulting expression could be a, "No" otherwise

```
1.  begin
2.      for i ← 1 to n do
3.          V_ij ← {w_i}
4.      for l ← 2 to n do
5.          for i ← 1 to n − l + 1 do
6.              j ← i + l − 1
7.              V_ij ← ∅
8.              for k ← 1 to j − 1 do
9.                  V_ij ← V_ij ∪ {x | x ← yz, ∀y ∈ V_ik, ∀z ∈ V_{k+1,j}}
10.     if a ∈ V_1n then print "Yes"
11.     else print "No"
12. end
```

Complexity: Obviously, the above algorithm has $O(n^3)$ complexity.

Problem 5: Let X, Y, Z be strings over the alphabet Σ. Z is said to be a shuffle of X and Y if Z can be written as $Z = X_1Y_1X_2Y_2 \ldots X_nY_n$, where X_i and Y_j are strings in Σ^* such that $X = X_1X_2 \ldots X_n$ and $Y = Y_1Y_2 \ldots Y_n$. For example, cchocohilaptes is a shuffle of chocolate and chips, but chocochilatspe is not. Devise a dynamic-programming algorithm that determines for X, Y, Z in the input whether Z is a shuffle of X and Y . Analyze the complexity of your algorithm. (Hint. Construct a Boolean table.)

Solution:

Given X, Y and Z, we can ask whether Z is a shuffle of X and Y. For this to be true, the last character of Z must be equal either to the last character X or to the last character of Z, and the remaining characters of Z must be a shuffle of the remaining characters in X and Y. There are also base cases where X or Y is empty. If X is the empty string, then Z must be equal to Y; if Y is empty, then Z must be equal to X.

This lead to an obvious recursive algorithm, shown here with some pseudo code:

bool isShuffle (string X, string Y, string Z):

```
n ← X.length
m ← Y.length
r ← Z.length
if n == 0
    return y == z
string x1 = the first n − 1 characters in X
string y1 = the first m − 1 characters in Y
string z1 = the first r − 1 characters in Z
return [(z[r − 1] == x[n − 1] AND isShuffle(x1, Y, z1))
        OR (z[r − 1] == y[m − 1] AND isShuffle(X, y1, z1))]
```

This algorithm has to potential to call itself twice, and when it calls itself, the size of the problem has only been reduced by one character. This means that there is a potential for exponential growth (although it will not occur for all strings). However, there is only a limited number of different problems, and so we can apply dynamic programming. The dynamic programming algorithm uses a n-by-m array, S, of boolean values, where $S[i][j]$ is true if and only if the first $i + j$ characters of Z are a shuffle of the first i characters of X together with the first j characters of Y. S will hold all the values that would be computed by the recursive isShuffle algorithm for all possible sub-cases, but each sub-case will only be computed once. We are really interested in $S[n][m]$, but we have to fill the array starting with smaller values of indices. The recursive algorithm can be translated fairly easily into this dynamic programming version:

bool isShuffle_nonrecursive (string X, string Y, string Z):

```
n ← X.length
m ← Y.length
r ← Z.length
if r ≠ n + m
    return false //obvious case
boolean S[n][m];
S[0][0] ← true
for i ← 1 to n do
    S[i][0] ← S[i − 1][0] AND (z[i − 1] == x[i − 1])
for j ← 1 to m do
    S[0][j] ← S[0][j − 1] AND (z[j − 1] == y[j − 1])
for i ← 1 to n do
    for j ← 1 to m do
        S[i][j] ← [((z[i + j − 1] == x[i − 1]) AND S[i − 1][j])
                   OR ((z[i + j − 1] == y[i − 1]) AND S[i][j − 1])]
return S[n − 1][m − 1]
```

This algorithm always runs in time $\Theta(nm)$.

Problem #6: Do Problem 15.4-5 on page 397 in [CLRS].

Solution:

Longest **Monotonically Increasing Subsequence** (MIS) of n number:

```
MIS (Num[])
L [1 .. n]              // contain the length of longest monotonically increasing subsequence
Prev [1 .. n]           // previous element
For (i =  1 .. n)
        L[i] = 1;
        Prev[i] = i;
        For(k = 1 .. i-1)
                If(Num[i] > Num[k] && L[i] <  L[k] + 1)
                        L[i] = L[k] + 1;
                        Prev[i] = k;
        End for
End for
```

Then you can trace back the **Monotonically Increasing Subsequence** from array Prev in O(n) time.