Assignment 3

Irene Avezzu' - 20142 - IAvezzu@unibz.it
Worked with: Massimo Marcon and Ivana Nworah Bortot

1. A Mysterious Procedure

1. What effect does the call MYSTERY(A, 1, A.length) have on an array A? Explain your answer.

MYSTERY is a method that behave similarly to the Bubble Sort method. In the base case the property P that holds in the segment of length 1 or 2 is that A[l] <= A[r]. I the inductive case the transitive property ensueres that if a<=b and b<=c then a<=c (a, b and c are the tree segment of the array on which we recall the method in line 7,8 and 9).

2. What is the asymptotic running time of MYSTERY? Explain your answer.

By applying the Master Theorem on this algorithm, we define the running time as T(n) = a*T(n/b) + c

- a (branching factor) = 3, for each call of the method we have three recursive calls
- n/b (size of subproblem) = each time we divide the problem in half the size + 1
- $\theta(n^c)$ (cost of the inductive case) = θ(1) so c=0 because we have constant time

Since f(n) = θ(n^ $\log_b a$) we can apply the second case of the Master theorem which define T(n) as $\theta(n^{\log b\,(a)} * \log n) = \theta(\log n)$

2. Matching Pairs

1. Describe in words your idea for an algorithm for this task.

With the worst algorithm we would have to compare each element with all the other elements in the array ($\theta(n^2)$), we can slightly improve (θ(f) with f(n) = $\frac{n(n-1)}{2}$) it but the asymptotic running time remains quadratic. To improve the running time we can start by sorting the array (θ(n*logn)) and then working on the sorted array. With a sorted array we could scan (i:=1 to A.length) the array (θ(n)) and then do the binary search on (s-A[i]) (θ(logn)).

2. Write this algorithm in pseudocode or Java. Hint: an algorithm that you know from the lecture may be useful.

```
boolean hasMatchingPair(int [] A, int s)
    mergeSort(A) //start by sorting
    for i:=1 to A.length do //n iterations
        dif = s - A[i] //value we're looking for with binary search
        difIndex = binarySearch(A, dif)
        if difIndex!= i && difIndex>= 1
            then return true
    return false
```

3. Evaluate the asymptotic worst-case running time of your algorithm (and explain your answer).

The worst running time for this algorithm is when there is no couple that respect the criteria, so we have to check all the elements.

We can divide the running time of this algorithm in three blocks:

- Sorting (merge/insertion/selection sort): θ(n*logn)
- Scanning: θ(n)
- Binary search: θ(log n)

Each time we scan we iterate the binary search so we can say that the number of steps required for the searching process are θ(n*logn).

We must then combine the sorting and searching running time by summing them together, we now have θ(2n*logn) = θ(n*logn) which is a good improvement from the original θ(n^2).

4. Explain why your algorithm is correct. That is, show that whenever your algorithm returns true, there are two values in A that add up to s, and that if your algorithm returns false, there are no such values. Hint: choose the right loop invariant.

After sorting the array, we scan and search for the missing value. The value we look for (*dif*) represent the difference between the input sum (*s*) and the value we're currently scanning (*A[i]*). *difIndex* represent the result of the binary search, it's positive when the element is found and negative when it's not. If it has a positive value and it is different from the index of the element we're currently scanning (*i*) that means that there exists (*difIndex >= 1*) one element which is in a different position from the one we're currently scanning (*difIndex != i*) whose sum with the currently scanned element (*A[i] + A[difIndex]*) is equal to the goal input-value (*s*)

5. Implement the method hasMatchingPair of the class Assignment3 in the Codeboard project for this assignment.

3. Stable Production

1. Write in pseudocode a brute-force algorithm that takes an array A of positive integers as input, and returns the maximal stable production for A. What is its worst-case asymptotic running time?

For each production, we need to find the left and right smaller productions, then multiply its value for the module of resulting subarray to find its msp. The biggest msp represents the solution.

The time complexity can be expressed in θ(n^2)

Pseudo-code:

```
int stableProduction(int[] A)
    int msp
    for i:=1 to A.length do
        int lS, rS
        // Find left smaller value
        for l=i-1 downto 1 do
            if A[l] < A[i] then lS := l
        // Find right smaller value
        for r=i+1 to A.length do
            if A[r] < A[i] then rS := r
        subArrayLength := rS - lS + 1
        localMsp := A[i] * subArrayLength
        if localMsp > msp then msp := localMsp
    return msp
```

2. Develop a more efficient approach, using the divide-and-conquer paradigm:

(a) write your algorithm in Java or pseudocode, and explain why it is correct,
(b) evaluate its worst-case asymptotic running time (explain your answer),
(c) implement this algorithm as the method stableProduction of Class Assignment3 in the Codeboard project for this assignment.

(a)Pseudo-code:

```
int stableProduction(int[] A)
   return int stableProduction(A, 1, A.length)

int stableProduction(int[] A, int l, int r)
   if l == r then return A[l]
   middle := (l+r) / 2
   stableProductionLeft := stableProduction(A, l, middle)
   stableProductionRight := stableProduction(A, middle+1, r)
   stableProductionMiddle := stableProductionMiddle(A, middle, l, r)
   maximumStableProduction := maximum(stableProductionLeft, stableProductionRight)
   maximumStableProduction := maximum(maximumStableProduction, stableProductionMiddle)
   return maximumStableProduction

int stableProductionMiddle(int[] A, int middle, int l, int r)
   maximumStableProduction := 0
   middleLeft := middle
   middleRight := middle+1
   currentMinimumProduction := minimum(A[middleLeft], A[middleRight])

   while middleLeft >= l && middleRight <= r do
      if A[middleLeft] < currentMinimumProduction then currentMinimumProduction := A[middleLeft]
      if A[middleRight] < currentMinimumProduction then currentMinimumProduction := A[middleRight]

      localMaximumStableProduction := (middleRight-middleLeft + 1) * currentMinimumProduction
      maximumStableProduction := maximum(maximumStableProduction, localMaximumStableProduction)

      if middleLeft == l then middleRight++
      else if middleRight == r then middleLeft--
      else
         if (A[middleLeft-1] >= A[middleRight+1]) middleLeft--
         else middleRight++

   return maximumStableProduction;
```

This algorithm conceptually divide a sub-array in two smaller segments assuming it works on them. So it calculates the right and left msp and their combination. This algorithm enters the base case when working on array of size 1 when it returns the element itself.

(b) The worst-case time complexity for this algorithm is for:

- a=2
- n/b=n/2
- $\theta(n^c)=\theta(n^1)$

$\log_b a = \log_2 2 = 1$ (=c)

$T(n) = \theta(n^c * \log n) = \theta(n * \log n)$