Assignment 2

Carlo Theunissen (1353926)
c.f.theunissen@student.tue.nl

September 29, 2018

Exercise 5

(a) The worst case of a decision tree corresponds to the maximum amount of comparisons the algorithm has to do. Because *Insertion Sort* only takes $(n-1)$ comparisons in the best case, which is lower than n^2, the lower bound is not affected by the height of the decision tree.

(b) A search algorithm returns the index of the found element of an array. If there are n elements in an Array; than the algorithm can return n different indexes.

Proof. Let h be the height of a binary tree, l be the number of reachable leaves and n the number of possible outcomes. Since a binary tree of height h has no more than 2^h leaves, we have:
$n \leq l \leq 2^h \implies h \geq \log n$

$$\begin{aligned} h &\geq \log n \\ &= \Omega(\log n) \end{aligned} \tag{1}$$

□

(c)

Proof. We use prove by contradiction.
Assume there exist an algorithm which checks if an array is a Max-Heap in $n-2$ comparisons.
Let A be an array which is a valid Max-Heap.
Our algorithm starts at the end of the array and verifies if the parent is bigger than element we're working with.
After doing $n-2$ comparisons A[2] has not been checked with A[1]. The algorithm will argue the array is a valid Max-Heap, and in this case, it's correct.
Now, let there be an array B which is the same as A, but we switch the first and second element such that B[2] $>$ B[1].
Our algorithm will then argue again that our array is still correct. However, B[2] $>$ B[1]. Hereby we've found a contradiction.
And therefore the assumptions is not correct. □

Exercise 6

Loop Invariant: At the start of i-th iteration of the loop. All elements in the array are sorted according to the last $(i-1)$ digits of their value. Ascending.

Initialization: At the start of the 1-st loop the loop invariant states: 'At the start of the 1-st iteration of the loop, All elements in the array are sorted according to the last $(1-1)$ digits of their value. We have sorted 0 digits so far, and this is what the Loop Invariant expected.

Maintenance: Assume the *Loop Invariant* holds for $i-1$ and assume the intermediate sort is stable. Then we only have to sort the array's value according to their i-th digit. We also make sure we sort using a stable technique. Otherwise, the assumption at the next iteration does not hold. This is indeed what we do in the body of RadixSort. Therefore the *Loop Invariant* holds at the start of the $i+1$ iteration, which is what we needed to prove.

Termination: When the loop terminates at the $d+1$ iteration. The loop invariant states: 'At the start of the $(d+1)$-th iteration of the loop, All elements in the array are sorted according to the last $(d-1+1)$ digits of their value. This is indeed what we expected: All d digits have been sorted. The algorithm is correct.

Exercise 7

The following algorithm counts how many times an element is present in the array and thereafter calculates the new position of each item. It swaps the items until all items are placed correctly, but instead of decreasing the position value, it increases.

```
   ONPLACECOUNTINGSORT(A, k)
   Input: an array A of n numbers and a number k
 1 C = new Array of k length, starting at index 1
 2 for i ← 1 to n do
 3       C[A[i]]++;
 4 total ← 0;
 5 for i ← 1 to k do
 6       copy = C[i];
 7       C[i] = total;
 8       total = total + copy;
 9 for i ← n to 1 do
10       while C[A[i]] < i do
11             oldPos ← A[i];
12             swap A[C[A[i]]] with A[i];
13             C[oldpos]++;
```

This algorithm works because we start at the end of the array, and keep swapping the elements until we've found the element whose position is the same or bigger than the index we're working on.
Our "position array" (C) keeps track of the start position of each item. So when a position is used, it is increased to make sure an element with the same key will be inserted after the last one.
Because the "working index" decreases and the "positions" increases we make sure the algorithm doesn't touch the elements that are already placed at the correct spot.

Line 1: takes O(k+1) = O(k)
Line 2-3: takes O(n) because we're doing n-times a O(1) operation.
Line 4: takes O(1)
Line 5-8: takes O(k) because we're doing k-times a O(3) = O(1) operation
Line 9-13: For each operation the while loop makes, it places at least one element at the correct position. Having only n elements, the while loop will take $\sum_{i=1}^{n} O(1) = O(n)$. For all subsequent calls, the while loop takes O(1). Therefore the for loop costs, s being the number of calls to the while loop when the array has already been sorted, O(n + s) = O(n).

O(k + n + k + n) = O(2n + 2k) = O(n + k)

Exercise 8

The following algorithm uses a "bucket sort" approach to distribute the input into years from 0 to 99. Then it uses a linear search technique to find a duplicate

```
   FINDDUPLICATEYEAR(A)
   Input: an array A of n numbers
   Output: the duplicate value
 1 b ← array of length 100, starting at index 0;
 2 for i ← 0 to 99 do
 3     b[i] = new LinkedList();
 4 for i ← 1 to n do
 5     do insert A[i] into list B[ floor(A[i]) ]
 6 for i ← 0 to 99 do
 7     for j ← 1 to B[i].length do
 8         for x ← j + 1 to B[i].length do
 9             if B[i][x] == B[i][j] then        // ignoring the floating point precision problem
10                 return B[x];
```

Line 1: takes O(100 + 1) = O(1)
Line 2-3: takes O(100) = O(1) because we're doing a O(1) operation 100 times
Line 4-5: takes O(1 * n) = O(n) because we're doing a O(1) operation n times
line 6 - 10 : $\sum_{i=1}^{99} O(n_i^2) = O(99n_i^2) = O(n_i^2)$
line 7 - 10 : $\sum_{j=1}^{n_i} O(n_i) = O(n_i^2)$
line 8 - 10 : $\sum_{j+1<x<n_i} O(1) = O(n_i)$

line 1 - 5 : O (1 + 1 + n) = O(n)

Assumption: Pr {A[j] is in LinkedList B[i]} = $\frac{1}{99}$ for each i
let b = the amount of elements in bucket B[i]
= E [O(n) + $\sum_{i=1}^{99} O(b^2)$]
= O(n + $\sum_{i=1}^{99} E[b^2]$)
= O(n + $\sum_{i=1}^{99} 2 - \frac{1}{99}$]
= O(n)

Loop Invariant: At the start of x-th iteration of the loop. No elements of B[i][j+1:j+x-1] are the same as B[i][j]

Initialization: At the start of the 1-st loop the loop invariant states: 'At the start of 1-th iteration of the loop. No elements of B[i][j+1:j+1-1] are the same as B[i][j]. B[i][j+1:j] is an empty array. B[i][j] $\notin \emptyset$ = B[i][j+1:j]. This in indeed expected, therefore the Loop Invariant holds

Maintenance: Assume the *Loop Invariant* holds for $x - 1$. We have to prove the *Loop Invariant* holds at the start of the $x + 1$ iteration. In the body of the for loop we check if B[i][j+x] is the same as B[i][j]. We have two cases: 1) the condition holds, then we stop executing by returning the found element. 2) the condition does not hold. Then we continue executing and we are sure B[i][j+1:j+x] does not hold an element the same as B[i][j]. At the start of the next iteration, the Loop Invariant holds, what is needed to prove.

Termination: When the loop terminates at the $(B[i].length - j + 1)$ iteration. The loop invariant states: 'At the start of the $(B[i].length - j + 1)$-th iteration of the loop, no elements

of $B[i][j+1 : B[i].length+1-1]$ are the same as B[i][j]. This is indeed what we expected. At the termination the Loop Invariant holds.

Loop Invariant: At the start of j-th iteration of the loop. Each element of B[i][0:j-1] has no duplicate in the array B[i]

Initialization: At the start of the 1-st loop the loop invariant states: 'At the start of 1-th iteration of the loop. Each element of B[i][0:1-1] has no duplicate in the array B[i]. B[i][0:0] is an empty array. There does not exist an item that is both an element of B[i] and an empty array. This in indeed expected, therefore the Loop Invariant holds.

Maintenance: Assume the *Loop Invariant* holds for $j-1$. We have to prove the *Loop Invariant* holds at the start of the $j+1$ iteration. In the body we check if a duplicate can be found in the array $B[i][j+1 : B[i].length]$ we have two cases: 1) there is an duplicate found, then the executing stops by returning the result. 2) there is no duplicate found in the array $B[i][j+1 : B[i].length]$ and because of our assumption there is also no duplicate in $B[i][0 : j]$ therefore the Loop Invariant holds at the start of the next iteration. This is what we needed to prove.

Termination: When the loop terminates at the $(B[i].length+1)$ iteration. The loop invariant states: 'At the start of the $(B[i].length+1)$-th iteration of the loop, no elements of $B[i][0 : B[i].length+1-1]$ have a duplicate in $B[i]$. This is indeed as expected. Therefore the Loop Invariant holds.

Loop Invariant: At the start of i-th iteration of the loop. Each element of the arrays B[0:i-1] has no duplicate in the arrays B[0:i-1]

Initialization: At the start of the 1-st loop the loop invariant states: 'At the start of 1-th iteration of the loop. Each element of the arrays B[0:1-1] has no duplicate in the arrays B[0:1-1]. B[0,0] is an empty array. It is impossible to find a duplicate in an empty array. The Loop Invariant holds

Maintenance: Assume the *Loop Invariant* holds for $i-1$ and all elements with the same floor value are placed in the same bucket. We have to prove the *Loop Invariant* holds at the start of the $i+1$ iteration. In the body we check if there are duplicates in the $B[i]$ array. We have two cases: 1) there is a duplicate found. Then the executing stops by returning the answer. 2) there is no duplicate found in $B[i]$. With the assumption that all numbers with the same floor value are placed in the same bucket. We can be sure there are no other duplicates are in $B[0 : i]$. This what we needed to prove. The Loop Invariant holds

Termination: When the loop terminates at the $(n+1)$ iteration. The loop invariant states: 'At the start of the $(n+1)$-th iteration of the loop, no elements of $B[0 : n+1-1]$ have a duplicate in $B[0 : n+1-1]$. This is indeed as expected. Therefore the Loop Invariant holds.

Exercise 9

A sequence of numbers $\{a_1, a_2, ..., a_n\}$such that $a_1 \leq a_2 \leq ... \leq a_n$ or $a_1 \geq a_2 \geq ... \geq a_n$ will give $\Omega(n^2)$ running time.
QuickSort is a divide-and-conquer algorithm, it divides the input into two partitions with the help of an pivot. All numbers that are smaller or equal to the input are places in one sub array, and the numbers that are bigger in another one. This is repeated until the array has

been sorted. However, when all the numbers are smaller or equal to the pivot or are all bigger than the pivot, they'll end up in one sub array. If this happens all the time, QuickSort loses its "divide" technique and will cost $\Theta(n^2)$. This situation happens with a sorted input.

Exercise 10

The algorithm consist out of multiple sub algorithms. We have PARTITION, FINDMEDIAN-WITHRECURSION, FINDMEDIANWITHSORTING, FINDSMALLESTNUMBERATINDEX and Find-PassedMedianAndFailedMedian.

PARTITION Arranges the array so that all numbers less than g are on one side and the other numbers are on the other side. It returns the index of the second half.

```
PARTITION(A, g)
Input: an array A of n numbers and a number g
Output: the start index of the second half of the partition
1 i = 1
2 for j = 1 to A.length do
3     if A[j] < g then
4         i = i+1
5         exchange A[i] with A[j]
6 return i
```

Uses Recursion and FINDMEDIANWITHSORTING to find the median of the given array. If the array is of length 1, it simply returns the first element.

```
FINDMEDIANWITHRECURSION(A)
Input: an array A of n numbers
Output: the median of the given array
1 if A.length is 1 then
2     return A[1];
3 let medians = new Array(ceil(A.length / 5));
4 for let i ← 1 to medians.length do
5     medians[i] = FindMedianWithSorting(A[(i-1) * 5 + 1 : min(A.length, i*5)]);
6 return FindMedianWithRecursion(medians);
```

Assumes the input array is of length 5 or smaller. Than is uses a sorting algorithm to sort the array.

```
FINDMEDIANWITHSORTING(A)
Input: an array A of n numbers
Output: the median of the given array
1 sort A with a O(n) algorithm
2 return A[floor(A.length/2)]
```

Tries to find the median and divides the input array until it has found the median at the given index.

FINDSMALLESTNUMBERATINDEX(A, $index$)
Input: an array A of n numbers and an $index$
Output: if A would be sorted ascending, the number at the given index

```
1 let partition = Partition(A, FindMedianWithRecursion(A));
2 if index < partition then
3     return FindSmallestNumberAtIndex(A[0 : partition-1], index);
4 if index > partition then
5     return FindSmallestNumberAtIndex(A[partition + 1 : A.length], index - partition - 1)
6 return A[partition];
```

Uses FINDSMALLESTNUMBERATINDEX to find the median of the grades that passed the assigment and those who failed.

FINDPASSEDMEDIANANDFAILEDMEDIAN(A, g)
Input: an array A of n numbers and an g
Output: the median of grades that failed and the median of those who passed

```
1 let split = Partition(A, g);
2 return {passed: FindSmallestNumberAtIndex(A[0 : split-1], (split-1)/2, failed:
   FindSmallestNumberAtIndex(A[split : n], (n-split)/2 }
```

Running Time:

PARTITION
line 1: O(1)
line 2 - 5: $\sum_{i=1}^{n} O(1) = O(n)$

FINDMEDIANWITHSORTING
line 1: O(n)
line 2: O(1)

FINDMEDIANWITHRECURSION
line 1-2: O(1)
line 3: O(n)
line 4: $\sum_{i=1}^{n}$ O(1) = O(n). FindMedianWithRecursion takes O(5) because the input array will never be larger than 5

T(n) = T(n/5) + O(n)

FINDSMALLESTNUMBERATINDEX
line 1-2: O(n) + T(n/5) + O(n)
line 3-5: T(7n/10 + 6)
T(n) = T(7n/10 + 6) + T(n/5) + O(n)
= O(n)

FindPassedMedianAndFailedMedian
line 1: O(n)
line 2: O(n) + O(n)

The total running time of the algorithm is O(n)

Exercise 11

(a) Quadratic probing has the formula: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m$. Let $c_1 = c_2 = 1$, $m = 13$ the formula becomes: $h(k,i) = (h'(k) + i + i^2) \mod 13$

Proof. The formula $h'(k) = k \mod 13$ will never give a number higher than 12. Therefore we only have to show $i + i^2 = a + 13b$ with $0 \leq a < 13$ and $0 \leq i \leq 13$.

$$\begin{aligned} i + i^2 &= a + 13b \\ -a &= 13b - (i + i^2) \\ -a &\leq 13b - (13 + 169) \\ -a &\leq 13b - 182 \\ b &= 14 \\ a &\geq 0 \end{aligned} \tag{2}$$

With this equation we have shown that there exist an a and b for every i □

An other starting number will result in a shift in the sequence. The sequence remains the same, stays between 0 and 13. But will be shifted by the amount of the starting number.

(b) Assume we're using Simple uniform hashing. The probability that an element is hashed to a key value of the hash table is $\frac{1}{m}$. We have mn elements $\frac{1}{m}mn = n$ so n values will hash to the same value.
If we're using the chaining technique then we have to search an element in a linked list of n big. By definition of a linked list, searching will take $\Theta(n)$.