Department of Informatics
Prof. Dr. Anton Dignös
Binzmühlestrasse 14
8050 Zurich
Phone: +41 44 635 4333
Email: adignoes@ifi.uzh.ch

University of Zurich[UZH]

Informatics II — Midterm 1
Spring 2021 — 29.03.2021

Name: ______________________ Matriculation number: ______________

Advice

You have 60 minutes to complete and submit the midterm exam of Informatics II. This is an open book exam.

Submit your solution in one of the following ways:

1. You can print the pdf file, use the available whitespace to fill in your solution, scan your solution, and upload the pdf file to EPIS.
2. You can use blank white paper for your solutions, scan the sheets, and upload the pdf file to EPIS. Put your name and matriculation number on every sheet. State all task numbers clearly.
3. You can use a tablet and pen (iPad, Surface, etc) to fill in your solution directly into the pdf file and upload the completed pdf file to EPIS.
4. You can use a text editor to answer the questions and submit the document as pdf.

Notes:

- If you do not have scanner it is possible to take pictures of your solution with your phone. We recommend Microsoft Office Lens or Camscanner. Create a pdf file that includes all pictures and submit a single pdf file.
- There is no extra time for scanning and submission. The allotted time already includes the time for scanning and submission.
- Only submissions through EPIS are accepted.

Signature:

Correction slot — Please do not fill out the part below

Exercise	1	2	3	Total
Points Achieved				
Maximum Points	12	14	14	40

Exercise 1

For a positive integer i, $d(i)$ is the sum all digits of i plus i. For example, $d(75) = 7+5+75 = 87$. Given a positive integer i and a positive integer n, the *D-Numbers* is the sequence of n numbers such that $d(i)$, $d(d(i))$, $d(d(d(i)))$, ... $d(d(d(...)))$. In the *D-Numbers* for the integer i, the first number is $d(i)$; the second number is $d(d(i))$; the $n-th$ number is $d(d(d(...)))$.

a) [1 points] Let $i = 81$ and $n = 4$. Write down the *D-Numbers* for i.

90 99 117 126

b) [2 points] If i is always less than 1000, what is the asymptotic complexity of `void DNumbers(int i, int n)`? Explain.

$O(n)$. As i is always less than 1000, $d(i)$ is in constant time. The `void DNumbers(int i, int n)` calls $d(i)$ function n times, and the overall time complexity is $O(n)$.

c) [9 points] Write the C function `void DNumbers(int i, int n)` that prints the *D-Numbers* for i.

```
int d(int i) {
  int sum = i;
  while (i != 0) {
    sum = sum + i % 10;
    i = i / 10;
  }
  return sum;
}

void Dnumbers(int i, int n) {
  int k, d_i;
  for (k = 1; k <= n; k++) {
    i = d(i);
    printf("%d ", i);
  }
  printf("\n");
}
```

code/task01.c

Exercise 2

Consider the C function `WhatDoesItdo` shown below, where $n \geq 2$.

```
int WhatDoesItdo(int n) {
  int i;
  for (i = 1; i< n; i++){
    if (i * (i + 1) == n){
        return True;
    }
  }
  return False;
}
```

a) [1 point] What does `WhatDoesItdo(int n)` return for $n = 25$ and $n = 6$, respectively?

`WhatDoesItdo(25)` = False
`WhatDoesItdo(6)` = True

b) [2 points] What does `WhatDoesItdo(int n)` do? The function `WhatDoesItdo(int n)` checks whether n is the mutiplication of two consecutive positive integers.

c) [5 points]Write invariants for Initialization, Maintenance and Termination to prove the correctness of the loop.

(a) Loop Invariant: If the integer $i * (i + 1) = n$ in range($1...n - 1$) then the functions returned True.

(b) Initialization: i starting from 1 holds because if $1 * 2 = n$ exists it can only be in range($1...n$).

(c) Maintenance: Each i checks if $i * (i + 1)$ is equal to n, if it is then True is returned. Else i is increased by 1 and the same invariant holds for range($i + 1...n - 1$).

(d) Termination: if the integer $i * (i + 1) = n$ in range ($1...n$) then the function returns True, else the loop terminates with returning False.

(e) If loop terminates without returning True, the False is returned to indicate that n is not the multiplication of two consecutive postive integers.

d) [2 points] What is the asymptotic complexity of the function? Explain.
$O(n)$. The dominating term of the function is the linear loop up to n, therefore the complexity of the algorithm is $O(n)$.

e) [4 points] Provide better solutions in terms of time complexity. Describe your solutions and show time complexity of your solutions.

Solution1: The function could be run faster if we use a Binary Search approach. This would reduce the asymptotic complexity to $O(\log n)$.

Solution2: The loop only needs to iterate as long as $i^2 < n$ holds. This would reduce the asymptotic complexity to $O(\sqrt{n})$.

Solutions like iterating up to $n/2$ are not accepted, since they are not correct (example: $n = 2$) and the asymptotic complexity is still $O(n)$.

Name: Matriculation number:

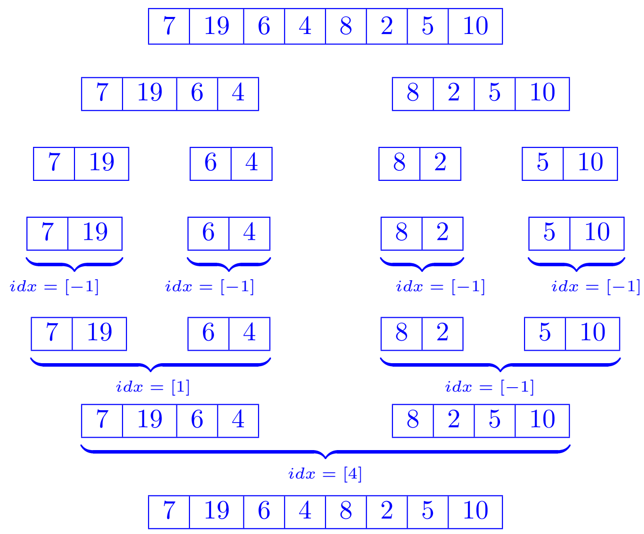
Exercise 3

Divide and Conquer

Consider an array `A[0...N-1]` consisting of N positive integers. Use the divide and conquer approach to find the index of `smallest apex element` in an array. An apex element is an element which is not smaller than its neighbours i.e. `A[i]` is an apex element if $A[i-1] \leq A[i] \geq A[i+1]$. For corner elements, you need to consider only one neighbour.

a) [1 point] What are the apex elements in array `A = [7,19,6,4,8,2,5,10]`.
   19,8,10

b) [2 points] Draw a tree that illustrates the divide and conquer approach of determining the smallest apex element in array `A = [7,19,6,4,8,2,5,10]`.



c) [11 points] Write a C code to determine the index of `smallest apex element` for any input array of size N using **divide and conquer approach**. The complexity of your algorithm should be $O(N)$.

```c
/*
* Function recursively only finds valid minimum apex elements, merges and checks the two middle elements, and checks corner cases only at the highest level.
*/

int findApex(int A[], int low, int high, int n)
{
  printf("low,high: %d,%d \n", low,high);
  int smlstapex = -1;
  if(high-low > 1)
  {
    /*
    * DIVIDE IF SIZE IS LARGER THAN THREE
    */
    int mid = (low + high) / 2;
    int lAIdx = findApex(A, low, mid, n);
    int rAIdx = findApex(A, mid+1, high, n);
    int check = mid;
    printf("low, mid, high, lAIdx, rAidx: %d, %d, %d, %d, %d \n", low,mid,high,lAIdx,rAIdx );
    /*
    * MERGE AND CHECK MID BOUNDARY ELEMENTS
    */

    // check if mid is an apex candidate
    if(A[check] >= A[check-1] && A[check] >= A[check+1])
    {
      if(lAIdx == -1 || A[lAIdx] > A[check])
      {
        lAIdx = check;
      }
    }
    printf("low, mid, high, lAIdx, rAidx: %d, %d, %d, %d, %d \n", low,mid,high,lAIdx,rAIdx );
    // check if mid+1 is an apex candidate
    check = mid + 1;
    if(high > mid+1 && A[check] >= A[check-1] && A[check] >= A[check+1])
    {
      if(rAIdx == -1 || A[rAIdx] > A[check])
      {
        rAIdx = check;
      }
    }
    if(lAIdx == -1)
      smlstapex = rAIdx;
    else if(rAIdx == -1)
      smlstapex = lAIdx;
    else
    {
```

```c
      if(A[lAIdx] < A[rAIdx])
        smlstapex = lAIdx;
      else
        smlstapex = rAIdx;
    }
    printf("low, mid, high, lAIdx, rAidx: %d, %d, %d, %d, %d \n", low,mid,high,lAIdx,rAIdx );

  }
  /*
  * CHECK CORNER CASES
  */
  if(high-low+1 == n)
  {
    // check corners
    if(A[low] >= A[low+1])
    {
      if(smlstapex == -1 || A[low] < A[smlstapex])
        smlstapex = low;
    }
    if(A[high] >= A[high-1])
    {
      if(smlstapex == -1 || A[high] < A[smlstapex])
        smlstapex = high;
    }
  }
  printf("smlst: %d \n",smlstapex);
```
code/Apex.c