Tutorial 13

Exercise 1: Sometimes it is necessary to work in algorithms with arrays where it is not known in advance, how many elements should be stored to them during a computation. In this case, it can be reasonable to use a kind of an array whose size can be dynamically changed during the computation — this datatype is usually called *vector*.

A typical implementation of this datatype is such that it allocates an array, which is a little bit bigger than necessary, and keeps information not only about the array and its length but also about the number of elements of the array, which are currently used. When an additional element or elements should be added, the cells that were unused so far are used for them, and only the corresponding index is incremented. Only in the case when the array is completely filled, a new bigger array is allocated, and the content of the original array is copied to it.

For simplicity, we will consider only operation Append, which appends one new element to the end of the array. This operation is described in Algorithm 1. Variable *arr* is the allocated array, variable *allocated* represents the length of this allocated array, and variable *len* represents the number of cells, which are actually used. (It is assumed that the invariant $allocated \geq len$.) For simplicity, consider this three variables (*arr*, *allocated*, *len*) to be global variables. All other variables are local.

Algorithm 1: Adding an element to the end of a vector

```
Append(x):
    if allocated ≤ len then
        s := New-Size(allocated)
        if s < len + 1 then
            s := len + 1
        newarr := Malloc(s)
        Copy(newarr, arr, len)
        Free(arr)
        arr := newarr
        allocated := s
    arr[len] := x
    len := len + 1
```

Several subprocedures are used in procedure Append:

- Malloc(*size*) — it allocates an array of *size* elements (for simplicity, we do not consider here the treating of the situation when this allocation fails),
- Free(*arr*) — it frees the memory used for array *arr*,
- Copy(*dst*, *src*, *cnt*) — it copies *cnt* elements from array *src* to array *dst*

Fot these three subroutines, you can assume that their time complexity is $O(n)$, where n is the number of elements of the given array, resp. the number of elements that must be copied (in subroutine Copy).

Function New-Size determines the size of the newly allocated array depending on the size of the previously allocated array.

Let us consider two possible implemetations of function New-Size:

a) function New-Size(m) returns value $m + 1$,
b) function New-Size(m) returns value $2 * m$.

Consider now an algorithm that starts with an empty array and in a cycle consecutively adds to the array n elements using the procedure Append. (Let us say for simplicity that at the beginning, variables *allocated* and *len* have value 0, and array *arr* contains 0 elements.) What is the complexity of the algorithm for each of the above mentioned variants of function New-Size?

(You can assume that if we do not count the time spent in procedure Append, the time of processing each individual added element is $O(1)$.)

Exercise 2: A sequence of elements can be represented in a computer memory by many different data structures. Examples of such data structures are:

a) an array
b) a singly linked list where we have a pointer to the first element of the list
c) a singly linked list where we have pointers to the first and the last element of the list
d) a doubly linked list where we have pointers to the first and the last element of the list

Recall how these data structures look and how to work with them.

Determine as precisely as possible the time complexity of the following operations on these data structures (assume that n is the total number of elements stored in a given data structure).

1. reading the value of an element of position i where i can be an arbitrary number from 0 to $n - 1$ (assume that elements are numbered from zero),
2. reading the value of the first element (i.e., the element of position 0),
3. reading the value of the last element (i.e., the element of position $n - 1$),
4. adding one element to the beginning (and shifting all other elements by one position),
5. adding an element to the end,
6. adding an element before a specified element (and shifting all following elements by one position),
7. adding an element after a specified element (and shifting all following elements by one position),
8. deleting a specified element from the sequence.

Remark: In points 6, 7, and 8, assume that the position of the element, before or after a new element is added, or which is removed, is specified by an index in the case of an array, and by a pointer in the case of a list.

For simplicity, assume that increasing the size of an array by one element can done in time $O(1)$.

Exercise 3: Let us say that we have n elements, which are stored in an array A, and we would like to perform some operation on all subsets of these n elements.

One possibility, how to generate these subsets, is to use a recursive algorithm. An example of such algorithm is Algorithm 2.

It is assumed that A and B are global arrays and n is a global variable containing as a value the size of these arrays. Array A contains the given elements and its content is not changed during a computation. The algorithm writes the subsets to array B, and the processing of individual subsets is done by procedure PROCESS. Procedure PROCESS obtains as a parameter a number ℓ, which specifies the number of elements in the given subset, and the elements of this subset are stored in the array B as elements $B[0]$, $B[1]$, ..., $B[\ell - 1]$. Variables k and ℓ, which are parameters of procedure SUBSETS, are local in this procedure. At the beginning, the procedure SUBSETS is called with zero values of both arguments, i.e., SUBSETS$(0, 0)$.

Algorithm 2: Generating subsets

```
SUBSETS (k, ℓ):
    if k ≥ n then
        PROCESS (ℓ)
        return
    SUBSETS (k + 1, ℓ)
    B[ℓ] := A[k]
    SUBSETS (k + 1, ℓ + 1)
```

Determine as precisely as possible the time and space complexity of this algorithm. (Assume that the time and space complexity of procedure PROCESS is $O(n)$.)

Exercise 4: Consider the following two variants of Euclid's algorithm for computing the greatest common divisor, described by Algorithms 3 and 4.

Determine the time complexity of both these algorithms, where take the total number of bits of numbers a and b as the size of an input. (For simplicity, you can assume that each arithmetic operation is performed in time $O(1)$.)

Algorithm 3: Euclid's algorithm — an inefficient version

```
EUCLID (a, b):
    if b = 0 then
        return a
    else if a ≥ b then
        return EUCLID(b, a − b)
    else
        return EUCLID(b − a, a)
```

Algorithm 4: Euclid's algorithm — a more efficient variant

```
EUCLID (a, b):
    while b ≠ 0 do
        c := a mod b
        a := b
        b := c
    return a
```

Exercise 5: Recall for each of the following problems what are their inputs and what are the questions. For each of these problems propose an algorithm solving the given problem. Determine the computational complexity of your algorithms.

a) SAT
b) 3-SAT
c) Independent set (IS)
d) Clique (CLIQUE)
e) Vertex cover (VC)
f) Hamiltonian cycle (HC)
g) Hamiltonian circuit (HK)
h) Travelling salesman problem (TSP)
i) Coloring of (nodes of) a graph with k colors
j) SUBSET-SUM

Exercise 6: Determine the time complexities of the following subroutines as accurately as possible. Express the time complexities using asymptotic notation Θ.

Remark: Consider n to be the size of an input. You can assume that values of all variables are already stored in memory.

a) Algorithm 5 — subroutine PROC-A

Algorithm 5:

```
PROC-A (A, b, n):
    for i := 1 to n * n do
        for j := 1 to i do
            A[i][j] := A[i][j] + b[j]
```

b) Algorithm 6 — subroutine PROC-B

c) Algorithm 7 — subroutine PROC-C

Algorithm 6:

```
PROC-B (R, d, n):
    x := 0
    for i := 1 to n do
        j := i * i
        while j > 0 do
            if d[j] < R[i][j] then
                R[i][j] := x − 1
                x := d[j]
            j := j − 1
```

Algorithm 7:

```
PROC-C (Q, n):
    i := 1
    while i < n do
        Q[i] := Q[i] + i
        i := i + i
```

Algorithm 8:

```
PROC-D (E, S, n):
    i := 1;  j := 1
    while i < n do
        E[i][j] := E[i][j] mod S[i]
        i := i + j
        j := j + 1
```

d) Algorithm 8 — subroutine PROC-D

e) Algorithm 9 — subroutine PROC-E

Algorithm 9:

```
PROC-E (A, B, n):
    s := 1
    while s ≤ n do
        i := 0
        while i < n do
            A[i] := A[B[i]] * s
            i := i + s
        s := s * 2
```

f) Algorithm 10 — subroutine PROC-F

Algorithm 10:

```
PROC-F (A, n):
    s := 1
    while s ≤ n do
        i := 0
        while i < n do
            A[i] := A[i] + s
            i := i + s
        s := s + 1
```