

DATA STRUCTURES AND ALGORITHMS

LECTURE 2

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2019 - 2020

- Course Organization
- Abstract Data Types and Data Structures
- Pseudocode
- Algorithm Analysis
 - O - notation
 - Ω - notation
 - Θ - notation
 - Best Case, Worst Case, Average Case

Today

- Algorithm analysis
- Dynamic Array
- Iterators

- The function $T(n) = 2n^2 + \frac{3}{2}n + 1000$ belongs to:

- | | | |
|-------------|------------------|------------------|
| a) $O(1)$ | f) $\Omega(1)$ | k) $\Theta(1)$ |
| b) $O(n)$ | g) $\Omega(n)$ | l) $\Theta(n)$ |
| c) $O(n^2)$ | h) $\Omega(n^2)$ | m) $\Theta(n^2)$ |
| d) $O(n^3)$ | i) $\Omega(n^3)$ | n) $\Theta(n^3)$ |
| e) $O(2^n)$ | j) $\Omega(2^n)$ | o) $\Theta(2^n)$ |

- Assume that the inputs of size n of a certain algorithm are of the following types:
 - **Type 1:** for each input of this type, the algorithm takes time $\Theta(n^4)$. The probability that the input is of this type is $\frac{1}{n^3}$.
 - **Type 2:** for each input of this type, the algorithm takes time $\Theta(n^3)$. The probability that the input is of this type is $\frac{1}{n}$.
 - **Type 3:** for each input of this type, the algorithm takes time $\Theta(n)$. The probability that the input is of this type is $1 - \frac{1}{n^3} - \frac{1}{n}$.
- What is the best case, worst case and average case of this algorithm?

Empirical analysis

- In order to see empirically how much the number of steps taken by an algorithm can influence its running time, we will consider 4 different implementations for the same problem:

Empirical analysis

- In order to see empirically how much the number of steps taken by an algorithm can influence its running time, we will consider 4 different implementations for the same problem:
- *Given an array of positive and negative values, find the maximum sum that can be computed for a subsequence. If a sequence contains only negative elements its maximum subsequence sum is considered to be 0.*
- For the sequence $[-2, 11, -4, 13, -5, -2]$ the answer is 20 ($11 - 4 + 13$)
- For the sequence $[4, -3, 5, -2, -1, 2, 6, -2]$ the answer is 11 ($4 - 3 + 5 - 2 - 1 + 2 + 6$)
- For the sequence $[9, -3, -7, 9, -8, 3, 7, 4, -2, 1]$ the answer is 15 ($9 - 8 + 3 + 7 + 4$)

First version

- The first algorithm will simply compute the sum of elements between any pair of valid positions in the array.

function first (x , n) **is**:

// x is an array of integer numbers, n is the length of x

maxSum \leftarrow 0

for $i \leftarrow 1$, n **execute**

for $j \leftarrow i$, n **execute**

//compute the sum of elements between i and j

 currentSum \leftarrow 0

for $k \leftarrow i$, j **execute**

 currentSum \leftarrow currentSum + $x[k]$

end-for

if currentSum > maxSum **then**

 maxSum \leftarrow currentSum

end-if

end-for

end-for

first \leftarrow maxSum

end-function

Complexity of the algorithm:

$$T(x, n) = \sum_{i=1}^n \sum_{j=i}^n \sum_{k=i}^j 1 = \dots = \Theta(n^3)$$

Second version

- If, at a given step, we have computed the sum of elements between positions i and j , the next sum will be between i and $j + 1$ (except for the case when j was the last element of the sequence).
- If we have the sum of numbers between indexes i and j we can compute the sum of numbers between indexes i and $j + 1$ by simply adding the element $x[j + 1]$. We do not need to recompute the whole sum.
- So we can eliminate the third (innermost) loop.

function second (x , n) **is**:

// x is an array of integer numbers, n is the length of x

maxSum \leftarrow 0

for $i \leftarrow 1, n$ **execute**

currentSum \leftarrow 0

for $j \leftarrow i, n$ **execute**

currentSum \leftarrow currentSum + $x[j]$

if currentSum > maxSum **then**

maxSum \leftarrow currentSum

end-if

end-for

end-for

second \leftarrow maxSum

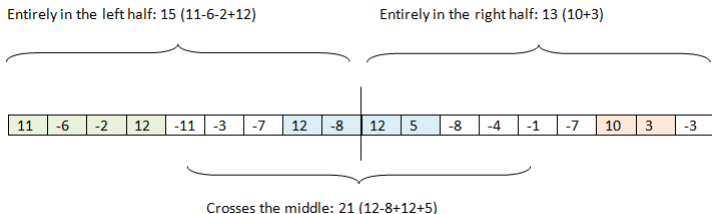
end-function

Complexity of the algorithm:

$$T(x, n) = \sum_{i=1}^n \sum_{j=i}^n 1 = \dots = \Theta(n^2)$$

Third version I

Third version II



- The maximum subsequence sum for the two halves can be computed recursively.
- How do we compute the maximum subsequence sum that crosses the middle?

- We will compute the maximum sum on the left (for a subsequence that ends with the middle element)
 - For the example above the possible subsequence sums are:
 - -8 (indexes 9 to 9)
 - 4 (indexes 8 to 9)
 - -3 (indexes 7 to 9)
 - -6 (indexes 6 to 9)
 - -17 (indexes 5 to 9)
 - -5 (indexes 4 to 9)
 - -7 (indexes 3 to 9)
 - -13 (indexes 2 to 9)
 - -2 (indexes 1 to 9)
 - We will take the maximum (which is 4)

- We will compute the maximum sum on the right (for a subsequence that starts immediately after the middle element)
 - For the example above the possible subsequence sums are:
 - 12 (indexes 10 to 10)
 - 17 (indexes 10 to 11)
 - 9 (indexes 10 to 12)
 - 5 (indexes 10 to 13)
 - 4 (indexes 10 to 14)
 - -3 (indexes 10 to 15)
 - 7 (indexes 10 to 16)
 - 10 (indexes 10 to 17)
 - 7 (indexes 10 to 18)
 - We will take the maximum (which is 17)
- We will add the two maximums (21)

- When we have the three values (maximum subsequence sum for the left half, maximum subsequence sum for the right half, maximum subsequence sum crossing the middle) we simply pick the maximum.

- We divide the implementation of the third version in three separate algorithms:
 - One that computes the maximum subsequence sum crossing the middle - *crossMiddle*
 - One that computes the maximum subsequence sum between positions [left, right] - *fromInterval*
 - The main one, that calls *fromInterval* for the whole sequence - *third*

function crossMiddle(x , left, right) **is:**

// x is an array of integer numbers

//left and right are the boundaries of the subsequence

middle \leftarrow (left + right) / 2

leftSum \leftarrow 0

maxLeftSum \leftarrow 0

for $i \leftarrow$ middle, left, -1 **execute**

leftSum \leftarrow leftSum + $x[i]$

if leftSum > maxLeftSum **then**

maxLeftSum \leftarrow leftSum

end-if

end-for

//continued on the next slide...

```

//we do similarly for the right side
rightSum  $\leftarrow$  0
maxRightSum  $\leftarrow$  0
for i  $\leftarrow$  middle+1, right execute
    rightSum  $\leftarrow$  rightSum + x[i]
    if rightSum > maxRightSum then
        maxRightSum  $\leftarrow$  rightSum
    end-if
end-for
crossMiddle  $\leftarrow$  maxLeftSum + maxRightSum
end-function

```

- Complexity:

```

//we do similarly for the right side
rightSum  $\leftarrow$  0
maxRightSum  $\leftarrow$  0
for i  $\leftarrow$  middle+1, right execute
    rightSum  $\leftarrow$  rightSum + x[i]
    if rightSum > maxRightSum then
        maxRightSum  $\leftarrow$  rightSum
    end-if
end-for
crossMiddle  $\leftarrow$  maxLeftSum + maxRightSum
end-function

```

- Complexity: $\Theta(n)$ - where n is *right* – *left*


```

function fromInterval(x, left, right) is:
  //x is an array of integer numbers
  //left and right are the boundaries of the subsequence
  if left = right then
    fromInterval  $\leftarrow$  x[left]
  end-if
  middle  $\leftarrow$  (left + right) / 2
  justLeft  $\leftarrow$  fromInterval(x, left, middle)
  justRight  $\leftarrow$  fromInterval(x, middle+1, right)
  across  $\leftarrow$  crossMiddle(x, left, right)
  fromInterval  $\leftarrow$  @maximum of justLeft, justRight, across
end-function

```

function third (x , n) **is:**

// x is an array of integer numbers, n is the length of x

third \leftarrow fromInterval(x , 1, n)

end-function

Complexity of the solution (fromInterval is the main function):

$$T(x, n) = \begin{cases} 1, & \text{if } n = 1 \\ 2 \cdot T(x, \frac{n}{2}) + n, & \text{otherwise} \end{cases}$$

- In case of a recursive algorithm, complexity computation starts from the recursive formula of the algorithm.

Let $n = 2^k$

Ignoring the parameter x we rewrite the recursive branch:

$$T(2^k) = 2 \cdot T(2^{k-1}) + 2^k$$

$$2 \cdot T(2^{k-1}) = 2^2 \cdot T(2^{k-2}) + 2^k$$

$$2^2 \cdot T(2^{k-2}) = 2^3 \cdot T(2^{k-3}) + 2^k$$

...

$$2^{k-1} \cdot T(2) = 2^k \cdot T(1) + 2^k$$

$$+$$

$$T(2^k) = 2^k \cdot T(1) + k \cdot 2^k$$

$T(1) = 1$ (base case from the recursive formula)

$$T(2^k) = 2^k + k \cdot 2^k$$

Let's go back to the notation with n .

$$\text{If } n = 2^k \Rightarrow k = \log_2 n$$

$$T(n) = n + n \cdot \log_2 n \in \Theta(n \log_2 n)$$

Fourth version

- Actually, it is enough to go through the sequence only once, if we observe the following:
 - The subsequence with the maximum sum will never begin with a negative number (if the first element is negative, by dropping it, the sum will be bigger)
 - The subsequence with the maximum sum will never start with a subsequence with total negative sum (if the first k elements have a negative sum, by dropping all of them, the sum will be bigger)
 - We can just start adding the numbers, but when the sum gets negative, drop it, and start over from 0.

```
function fourth (x, n) is:  
  //x is an array of integer numbers, n is the length of x  
  maxSum  $\leftarrow$  0  
  currentSum  $\leftarrow$  0  
  for i  $\leftarrow$  1, n execute  
    currentSum  $\leftarrow$  currentSum + x[i]  
    if currentSum > maxSum then  
      maxSum  $\leftarrow$  currentSum  
    end-if  
    if currentSum < 0 then  
      currentSum  $\leftarrow$  0  
    end-if  
  end-for  
  fourth  $\leftarrow$  maxSum  
end-function
```

Complexity of the algorithm:

$$T(x, n) = \sum_{i=1}^n 1 = \dots = \Theta(n)$$

Comparison of actual running times

Input size	First $\Theta(n^3)$	Second $\Theta(n^2)$	Third $\Theta(n \log n)$	Fourth $\Theta(n)$
10	0.00005	0.00001	0.00002	0.00000
100	0.01700	0.00054	0.00023	0.00002
1,000	16.09249	0.05921	0.00259	0.00013
10,000	-	6.23230	0.03582	0.00137
100,000	-	743.66702	0.37982	0.01511
1,000,000	-	-	4.51991	0.16043
10,000,000	-	-	48.91452	1.66028

Table: Comparison of running times measured with Python's `default_timer()`

Comparison of actual running times

- From the previous table we can see that complexity and running time are indeed related:
- When the input is 10 times bigger:
 - The first algorithm needs ≈ 1000 times more time
 - The second algorithm needs ≈ 100 times more time
 - The third algorithm needs ≈ 11 -13 times more time
 - The fourth algorithm needs ≈ 10 times more time

Example 2

- Consider the following algorithm (written in Python):

```
def testContainer(container, l):  
    """  
    container is a container with integer numbers  
    l is a list with integer numbers  
    """  
    count = 0  
    for elem in l:  
        if elem in container:  
            count += 1  
    return count
```

- The above function counts how many elements from the list *l* can be found in the container

Example 2

- Consider the following scenario for a given integer number *size*:
 - Generate a random list with *size* with unique elements from the interval $[0, \text{size} * 2)$
 - Add these elements in a container (list or dictionary - value is equal to key for dictionary)
 - Generate another random list with *size* unique elements from the interval $[0, \text{size} * 2)$
 - Call the *testContainer* function for the container and the second list and measure the execution time for it.

Example 2

- Execution times (for executing *size* times the *in* operation):

Size	Time for list	Time for dictionary
10	0.0000057	0.0000049
100	0.000124	0.0000069
1000	0.0141	0.000266
10000	1.652	0.00151
100000	183.102	0.0157
1000000	-	0.253
10000000	-	3.759

Algorithm Analysis for Recursive Functions

- How can we compute the time complexity of a recursive algorithm?

Recursive Binary Search

```
function BinarySearchR (array, elem, start, end) is:  
  //array - an ordered array of integer numbers  
  //elem - the element we are searching for  
  //start - the beginning of the interval in which we search (inclusive)  
  //end - the end of the interval in which we search (inclusive)  
  if start > end then  
    BinarySearchR  $\leftarrow$  False  
  end-if  
  middle  $\leftarrow$  (start + end) / 2  
  if array[middle] = elem then  
    BinarySeachR  $\leftarrow$  True  
  else if elem < array[middle] then  
    BinarySearchR  $\leftarrow$  BinarySearchR(array, elem, start, middle-1)  
  else  
    BinarySearchR  $\leftarrow$  BinarySearchR(array, elem, middle+1, end)  
  end-if  
end-function
```

Recursive Binary Search

- The first call to the *BinarySearchR* algorithm for an ordered array of nr elements is:

```
BinarySearchR(array, elem, 1, nr)
```

- How do we compute the complexity of the *BinarySearchR* algorithm?

Recursive Binary Search

- We will denote the length of the sequence that we are checking at every iteration by n (so $n = end - start$)
- We need to write the recursive formula of the solution

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ T(\frac{n}{2}) + 1, & \text{otherwise} \end{cases}$$

- The *master method* can be used to compute the time complexity of algorithms having the following general recursive formula:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

- where $a \geq 1$, $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

- Advantage of the master method: we can determine the time complexity of a recursive algorithm without further computations.
- Disadvantage of the master method: we need to memorize the three cases of the method and there are some situations when none of these cases can be applied.

Computing the time complexity without the master method

- If we do not want to memorize the cases for the master method we can compute the time complexity in the following way:
- Recall, the recursive formula for BinarySearchR was:

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ T(\frac{n}{2}) + 1, & \text{otherwise} \end{cases}$$

Time complexity for BinarySearchR

- We suppose that $n = 2^k$ and rewrite the second branch of the recursive formula:

$$T(2^k) = T(2^{k-1}) + 1$$

- Now, we write what the value of $T(2^{k-1})$ is (based on the recursive formula)

$$T(2^{k-1}) = T(2^{k-2}) + 1$$

- Next, we add what the value of $T(2^{k-2})$ is (based on the recursive formula)

$$T(2^{k-2}) = T(2^{k-3}) + 1$$

Time complexity for BinarySearchR

- The last value that can be written is the value of $T(2^1)$

$$T(2^1) = T(2^0) + 1$$

Time complexity for BinarySearchR

- Now, we write all these equations together and add them (and we will see that many terms can be simplified, because they appear on the left hand side of an equation and the right hand side of another equation):

$$T(2^k) = T(2^{k-1}) + 1$$

$$T(2^{k-1}) = T(2^{k-2}) + 1$$

$$T(2^{k-2}) = T(2^{k-3}) + 1$$

...

$$T(2^1) = T(2^0) + 1$$

$$\begin{array}{r} \hline T(2^k) = T(2^0) + 1 + 1 + 1 + \dots + 1 = 1 + k \end{array}$$

- Obs:** For non-recursive functions adding a +1 or not, does not influence the result. In case of recursive functions it is important to have another term besides the recursive one.

Time complexity for BinarySearchR

- We started from the notation $n = 2^k$.
- We want to go back to the notation that uses n . If $n = 2^k \Rightarrow k = \log_2 n$

$$T(2^k) = 1 + k$$

$$T(n) = 1 + \log_2 n \in \Theta(\log_2 n)$$

Time complexity for BinarySearchR

- We started from the notation $n = 2^k$.
- We want to go back to the notation that uses n . If $n = 2^k \Rightarrow k = \log_2 n$

$$T(2^k) = 1 + k$$

$$T(n) = 1 + \log_2 n \in \Theta(\log_2 n)$$

- Actually, if we look at the code from *BinarySearchR*, we can observe that it has a best case (element can be found at the first iteration), so final complexity is $O(\log_2 n)$

Another example

- Let's consider the following pseudocode and compute the time complexity of the algorithm:

```
subalgorithm operation( $n, i$ ) is:  
//  $n$  and  $i$  are integer numbers,  $n$  is positive  
  if  $n > 1$  then  
     $i \leftarrow 2 * i$   
     $m \leftarrow n/2$   
    operation( $m, i-2$ )  
    operation( $m, i-1$ )  
    operation( $m, i+2$ )  
    operation( $m, i+1$ )  
  else  
    write  $i$   
  end-if  
end-subalgorithm
```

- The first step is to write the recursive formula:

$$T(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ 4 \cdot T(\frac{n}{2}) + 1, & \text{otherwise} \end{cases}$$

- We suppose that $n = 2^k$.

$$T(2^k) = 4 \cdot T(2^{k-1}) + 1$$

- This time we need the value of $4 \cdot T(2^{k-1})$

$$\begin{aligned} T(2^{k-1}) &= 4 \cdot T(2^{k-2}) + 1 \Rightarrow \\ 4 \cdot T(2^{k-1}) &= 4^2 \cdot T(2^{k-2}) + 4 \end{aligned}$$

- And the value of $4^2 \cdot T(2^{k-2})$

$$4^2 \cdot T(2^{k-2}) = 4^3 \cdot T(2^{k-3}) + 4^2$$

- The last value we can compute is $4^{k-1} \cdot T(2^1)$

$$4^{k-1} \cdot T(2^1) = 4^k \cdot T(2^0) + 4^{k-1}$$

- We write all the equations and add them:

$$\begin{array}{rcl}
 T(2^k) & = & 4 \cdot T(2^{k-1}) + 1 \\
 4 \cdot T(2^{k-1}) & = & 4^2 \cdot T(2^{k-2}) + 4 \\
 4^2 \cdot T(2^{k-2}) & = & 4^3 \cdot T(2^{k-3}) + 4^2 \\
 & \dots & \\
 4^{k-1} \cdot T(2^1) & = & 4^k \cdot T(2^0) + 4^{k-1} \\
 \hline
 T(2^k) & = & 4^k \cdot T(1) + 4^0 + 4^1 + 4^2 + \dots + 4^{k-1}
 \end{array}$$

- $T(1)$ is 1 (first case from recursive formula)

$$T(2^k) = 4^0 + 4^1 + 4^2 + \dots + 4^{k-1} + 4^k$$

$$\sum_{i=0}^n p^i = \frac{p^{n+1} - 1}{p - 1}$$

$$T(2^k) = \frac{4^{k+1} - 1}{4 - 1} = \frac{4^k \cdot 4 - 1}{3} = \frac{(2^k)^2 \cdot 4 - 1}{3}$$

- We started from $n = 2^k$. Let's change back to n

$$T(n) = \frac{4n^2 - 1}{3} \in \Theta(n^2)$$

Records

- A *record* (or *struct*) is a static data structure.
- It represents the reunion of a fixed number of components (which can have different types) that form a logical unit together.
- We call the components of a record *fields*.
- For example, we can have a record to denote a *Person* formed of fields for *name*, *date of birth*, *address*, etc.

Person:

name: String
dob: String
address: String
etc.

- An array is one of the simplest and most basic data structures.
- An array can hold a fixed number of elements of the same type and these elements occupy a contiguous memory block.
- Arrays are often used as a basis for other (more complex) data structures.

- When a new array is created we have to specify two things:
 - The type of the elements in the array
 - The maximum number of elements that can be stored in the array (*capacity* of the array)
- The memory occupied by the array will be the capacity times the size of one element.
- The array itself is memorized by the address of the first element.

Arrays - Example 1

- An array of *boolean* values (addresses of the elements are displayed in base 16 and base 10)

Size of boolean: 1

Address of array: 00EFF760

Address of element from position 0: 00EFF760 15726432

Address of element from position 1: 00EFF761 15726433

Address of element from position 2: 00EFF762 15726434

Address of element from position 3: 00EFF763 15726435

Address of element from position 4: 00EFF764 15726436

Address of element from position 5: 00EFF765 15726437

Address of element from position 6: 00EFF766 15726438

Address of element from position 7: 00EFF767 15726439

- Can you guess the address of the element from position 8?

Arrays - Example 2

- An array of *integer* values (integer values occupy 4 bytes)

```
Size of int: 4
```

```
Address of array: 00D9FE6C
```

```
Address of element from position 0: 00D9FE6C 14286444
```

```
Address of element from position 1: 00D9FE70 14286448
```

```
Address of element from position 2: 00D9FE74 14286452
```

```
Address of element from position 3: 00D9FE78 14286456
```

```
Address of element from position 4: 00D9FE7C 14286460
```

```
Address of element from position 5: 00D9FE80 14286464
```

```
Address of element from position 6: 00D9FE84 14286468
```

```
Address of element from position 7: 00D9FE88 14286472
```

- Can you guess the address of the element from position 8?

Arrays - Example 3

- An array of *fraction* record values (the fraction record is composed of two integers)

Size of fraction: 8

Address of array: 007BF97C

Address of element from position 0: 007BF97C 8124796

Address of element from position 1: 007BF984 8124804

Address of element from position 2: 007BF98C 8124812

Address of element from position 3: 007BF994 8124820

Address of element from position 4: 007BF99C 8124828

Address of element from position 5: 007BF9A4 8124836

Address of element from position 6: 007BF9AC 8124844

Address of element from position 7: 007BF9B4 8124852

- Can you guess the address of the element from position 8?

- The main **advantage** of arrays is that any element of the array can be accessed in constant time ($\Theta(1)$), because the address of the element can simply be computed (considering that the first element is at position 0):

Address of i^{th} element = address of array + $i * \text{size of an element}$

- The above formula works even if we consider that the first element is at position 1, but then we need to use $i - 1$ instead of i .

- An array is a static structure: once the *capacity* of the array is specified, you cannot add or delete slots from it (you can add and delete elements from the slots, but the number of slots, the capacity, remains the same)
- This leads to an important **disadvantage**: we need to know/estimate from the beginning the number of elements:
 - if the capacity is too small: we cannot store every element we want to
 - if the capacity is too big: we waste memory

Dynamic Array

- There are arrays whose size can grow or shrink, depending on the number of elements that need to be stored in the array: they are called *dynamic arrays* (or *dynamic vectors*).
- Dynamic arrays are still arrays, the elements are still kept at contiguous memory locations and we still have the advantage of being able to compute the address of every element in $\Theta(1)$ time.

Dynamic Array - Representation

- In general, for a Dynamic Array we need the following fields:
 - *cap* - denotes the number of slots allocated for the array (its capacity)
 - *nrElem* - denotes the actual number of elements stored in the array
 - *elems* - denotes the actual array with *capacity* slots for TElems allocated

DynamicArray:

cap: Integer

nrElem: Integer

elems: TElem[]

Dynamic Array - Resize

- When the value of *nrElem* equals the value of *capacity*, we say that the array is full. If more elements need to be added, the *capacity* of the array is increased (usually doubled) and the array is *resized*.
- During the *resize* operation a new, bigger array is allocated and the existing elements are copied from the old array to the new one.
- Optionally, *resize* can be performed after delete operations as well: if the dynamic array becomes "too empty", a resize operation can be performed to shrink its size (to avoid occupying unused memory).

Dynamic Array - DS vs. ADT

- Dynamic Array is a data structure:
 - It describes how data is actually stored in the computer (in a single contiguous memory block) and how it can be accessed and processed
 - It can be used as representation to implement different abstract data types
- However, Dynamic Array is so frequently used that in most programming languages it exists as a separate container as well.
 - The Dynamic Array is not really an ADT, since it has one single possible implementation, but we still can treat it as an ADT, and discuss its interface.

- **Domain** of ADT DynamicArray

$$\mathcal{DA} = \{\mathbf{da} \mid da = (cap, nrElem, e_1 e_2 e_3 \dots e_{nrElem}), cap, nrElem \in N, nrElem \leq cap, e_i \text{ is of type TElem}\}$$

Dynamic Array - Interface II

- What operations should we have for a *DynamicArray*?

Dynamic Array - Interface III

- **init**(*da*, *cp*)
 - **description:** creates a new, empty DynamicArray with initial capacity *cp* (constructor)
 - **pre:** $cp \in \mathbb{N}^*$
 - **post:** $da \in \mathcal{DA}$, $da.cap = cp$, $da.nrElem = 0$
 - **throws:** an exception if *cp* is zero or negative

Dynamic Array - Interface IV

- `destroy(da)`
 - **description:** destroys a DynamicArray (destructor)
 - **pre:** $da \in \mathcal{DA}$
 - **post:** da was destroyed (the memory occupied by the dynamic array was freed)

Dynamic Array - Interface V

- `size(da)`
 - **description:** returns the size (number of elements) of the `DynamicArray`
 - **pre:** $da \in \mathcal{DA}$
 - **post:** `size` \leftarrow the size of da (the number of elements)

Dynamic Array - Interface VI

- `getElement(da, i)`
 - **description:** returns the element from a position from the `DynamicArray`
 - **pre:** $da \in \mathcal{DA}$, $1 \leq i \leq da.nrElem$
 - **post:** $getElement \leftarrow e$, $e \in TElem$, $e = da.e_i$ (the element from position i)
 - **throws:** an exception if i is not a valid position

Dynamic Array - Interface VII

- `setElement(da, i, e)`
 - **description:** changes the element from a position to another value
 - **pre:** $da \in \mathcal{DA}$, $1 \leq i \leq da.nrElem$, $e \in TElem$
 - **post:** $da' \in \mathcal{DA}$, $da'.e_i = e$ (the i^{th} element from da' becomes e), $setElement \leftarrow e_{old}$, $e_{old} \in TElem$, $e_{old} \leftarrow da.e_i$ (returns the old value from position i)
 - **throws:** an exception if i is not a valid position

Dynamic Array - Interface VIII

- **addToEnd**(da, e)
 - **description:** adds an element to the end of a DynamicArray. If the array is full, its capacity will be increased
 - **pre:** $da \in \mathcal{DA}$, $e \in TElem$
 - **post:** $da' \in \mathcal{DA}$, $da'.nrElem = da.nrElem + 1$; $da'.e_{da'.nrElem} = e$ ($da.cap = da.nrElem \Rightarrow da'.cap \leftarrow da.cap * 2$)

Dynamic Array - Interface IX

- **addToPosition**(da, i, e)
 - **description:** adds an element to a given position in the DynamicArray. If the array is full, its capacity will be increased
 - **pre:** $da \in \mathcal{DA}$, $1 \leq i \leq da.nrElem + 1$, $e \in TElem$
 - **post:** $da' \in \mathcal{DA}$, $da'.nrElem = da.nrElem + 1$, $da'.e_j = da.e_{j-1} \forall j = da'.nrElem, da'.nrElem - 1, \dots, i + 1$, $da'.e_i = e$, $da'.e_j = da.e_j \forall j = i - 1, \dots, 1$ ($da.cap = da.nrElem \Rightarrow da'.cap \leftarrow da.cap * 2$)
 - **throws:** an exception if i is not a valid position ($da.nrElem+1$ is a valid position when adding a new element)

Dynamic Array - Interface X

- `deleteFromPosition(da, i)`
 - **description:** deletes an element from a given position from the DynamicArray. Returns the deleted element
 - **pre:** $da \in \mathcal{DA}$, $1 \leq i \leq da.nrElem$
 - **post:** $deleteFromPosition \leftarrow e$,
 $e \in TElem$, $e = da.e_i$, $da' \in \mathcal{DA}$, $da'.nrElem = da.nrElem - 1$,
 $da'.e_j = da.e_{j+1} \forall i \leq j \leq da'.nrElem$,
 $da'.e_j = da.e_j \forall 1 \leq j < i$
 - **throws:** an exception if i is not a valid position

Dynamic Array - Interface XI

- `iterator(da, it)`
 - **description:** returns an iterator for the DynamicArray
 - **pre:** $da \in \mathcal{DA}$
 - **post:** $it \in \mathcal{I}$, it is an iterator over da , the current element from it refers to the first element from da , or, if da is empty, it is invalid

Dynamic Array - Interface XII

- Other possible operations:
 - Delete all elements from the Dynamic Array (make it empty)
 - Verify if the Dynamic Array is empty
 - Delete an element (given as element, not as position)
 - Check if an element appears in the Dynamic Array
 - Remove the element from the end of the Dynamic Array
 - etc.

Dynamic Array - Implementation

- Most operations from the interface of the Dynamic Array are very simple to implement.
- In the following we will discuss the implementation of two operations: *addToEnd*, *addToPosition*.
- For the implementation we are going to use the representation discussed earlier:

DynamicArray:

cap: Integer

nrElem: Integer

elems: TElem[]

Dynamic Array - addToEnd - Case 1

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 6

- Add the element 49 to the end of the dynamic array

Dynamic Array - addToEnd - Case 1

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 6

- Add the element 49 to the end of the dynamic array

51	32	19	31	47	95	49			
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 7

Dynamic Array - addToEnd - Case 2

51	32	19	31	47	95
1	2	3	4	5	6

- capacity (cap): 6
- nrElem: 6

- Add the element 49 to the end of the dynamic array

Dynamic Array - addToEnd - Case 2

51	32	19	31	47	95
1	2	3	4	5	6

- capacity (cap): 6
- nrElem: 6

- Add the element 49 to the end of the dynamic array

51	32	19	31	47	95
↓	↓	↓	↓	↓	↓

51	32	19	31	47	95	49					
1	2	3	4	5	6	7	8	9	10	11	12

- capacity (cap): **12**
- nrElem: **7**

Dynamic Array - addToEnd

subalgorithm addToEnd (da, e) **is:**

if da.nrElem = da.cap **then**

//the dynamic array is full. We need to resize it

da.cap \leftarrow da.cap * 2

newElems \leftarrow @ an array with da.cap empty slots

//we need to copy existing elements into newElems

for index \leftarrow 1, da.nrElem **execute**

newElems[index] \leftarrow da.elems[index]

end-for

//we need to replace the old element array with the new one

//depending on the prog. lang., we may need to free the old elems array

da.elems \leftarrow newElems

end-if

//now we certainly have space for the element e

da.nrElem \leftarrow da.nrElem + 1

da.elems[da.nrElem] \leftarrow e

end-subalgorithm

- What is the complexity of *addToEnd*?

Dynamic Array - addToPosition

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 6

- **Add the element 49 to position 3**

Dynamic Array - addToPosition

51	32	19	31	47	95				
1	2	3	4	5	6	7	8	9	10

- capacity (cap): 10
- nrElem: 6

- Add the element 49 to position 3

51	32	49	19	31	47	95			
1	2	3	4	5	6	7	8	9	10

Diagram illustrating the shift of elements to the right to make space for the new element 49 at position 3. Arrows show the movement of elements from positions 4 to 7 to positions 5 to 8 respectively.

- capacity (cap): 10
- nrElem: 7

- Add the element 49 to position 3

subalgorithm addToPosition (da, i, e) **is**:

if $i > 0$ **and** $i \leq \text{da.nrElem} + 1$ **then**

if $\text{da.nrElem} = \text{da.cap}$ **then** *//the dynamic array is full. We need to resize it*

$\text{da.cap} \leftarrow \text{da.cap} * 2$

$\text{newElems} \leftarrow$ @ an array with da.cap empty slots

for $\text{index} \leftarrow 1, \text{da.nrElem}$ **execute**

$\text{newElems}[\text{index}] \leftarrow \text{da.elems}[\text{index}]$

end-for

$\text{da.elems} \leftarrow \text{newElems}$

end-if *//now we certainly have space for the element e*

$\text{da.nrElem} \leftarrow \text{da.nrElem} + 1$

for $\text{index} \leftarrow \text{da.nrElem}, i+1, -1$ **execute** *//move the elements to the right*

$\text{da.elems}[\text{index}] \leftarrow \text{da.elems}[\text{index}-1]$

end-for

$\text{da.elems}[i] \leftarrow e$

else

@throw exception

end-if

end-subalgorithm

● What is the complexity of *addToPosition*?



Dynamic Array

- Observations:
 - While it is not mandatory to double the capacity, it is important to define the new capacity as a product of the old one with a constant number greater than 1 (just adding one new slot, or a constant number of new slots is not OK - you will see later why).

Dynamic Array

- Observations:
 - While it is not mandatory to double the capacity, it is important to define the new capacity as a product of the old one with a constant number greater than 1 (just adding one new slot, or a constant number of new slots is not OK - you will see later why).
- How do dynamic arrays in other programming languages grow at resize?
 - C++ - multiply by 1.5 (initially 1, then 2, 3, 4, 6, 9, 13, etc.)
 - Java - multiply by 1.5 (initially 10, then 15, 22, 33, etc.)
 - Python - multiply by ≈ 1.125 (0, 4, 8, 16, 25, 35, 46, 58, etc.)
 - C# - multiply by 2 (initially 0, 4, 8, 16, 32, etc.)

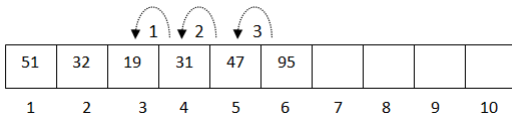
- After a resize operation the elements of the Dynamic Array will still occupy a contiguous memory zone, but it will be a different one than before.

Dynamic Array

```
Address of the Dynamic Array structure: 00D3FE00 13893120
Length is: 3 si capacitate: 3
Address of array from DA: 0039E568 3794280
    Address of element from position 0 0039E568 3794280
    Address of element from position 1 0039E56C 3794284
    Address of element from position 2 0039E570 3794288
-----
Address of the Dynamic Array structure: 00D3FE00 13893120
Length is: 6 si capacitate: 6
Address of array from DA: 003A0100 3801344
    Address of element from position 0 003A0100 3801344
    Address of element from position 1 003A0104 3801348
    Address of element from position 2 003A0108 3801352
    Address of element from position 3 003A010C 3801356
    Address of element from position 4 003A0110 3801360
    Address of element from position 5 003A0114 3801364
-----
Address of the Dynamic Array structure: 00D3FE00 13893120
Length is: 8 si capacitate: 12
Address of array from DA: 00396210 3760656
    Address of element from position 0 00396210 3760656
    Address of element from position 1 00396214 3760660
    Address of element from position 2 00396218 3760664
    Address of element from position 3 0039621C 3760668
    Address of element from position 4 00396220 3760672
    Address of element from position 5 00396224 3760676
    Address of element from position 6 00396228 3760680
    Address of element from position 7 0039622C 3760684
```

Dynamic Array - delete operation

- To delete an element from a given position i , the elements after position i need to be moved one position to the left (element from position j is moved to position $j-1$).



- capacity (cap): 10
- nrElem: 5

- Delete the element from position 3

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
 - size -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
 - size - $\Theta(1)$
 - getElement -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
 - size - $\Theta(1)$
 - getElement - $\Theta(1)$
 - setElement -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
 - size - $\Theta(1)$
 - getElement - $\Theta(1)$
 - setElement - $\Theta(1)$
 - iterator -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
 - size - $\Theta(1)$
 - getElement - $\Theta(1)$
 - setElement - $\Theta(1)$
 - iterator - $\Theta(1)$
 - addToPosition -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
 - size - $\Theta(1)$
 - getElement - $\Theta(1)$
 - setElement - $\Theta(1)$
 - iterator - $\Theta(1)$
 - addToPosition - $O(n)$
 - deleteFromEnd -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
 - size - $\Theta(1)$
 - getElement - $\Theta(1)$
 - setElement - $\Theta(1)$
 - iterator - $\Theta(1)$
 - addToPosition - $O(n)$
 - deleteFromEnd - $\Theta(1)$
 - deleteFromPosition -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
 - size - $\Theta(1)$
 - getElement - $\Theta(1)$
 - setElement - $\Theta(1)$
 - iterator - $\Theta(1)$
 - addToPosition - $O(n)$
 - deleteFromEnd - $\Theta(1)$
 - deleteFromPosition - $O(n)$
 - addToEnd -

Dynamic Array - Complexity of operations

- Usually, we can discuss the complexity of an operation for an ADT only after we have chosen the representation. Since the ADT Dynamic Array can be represented in a single way, we can discuss the complexity of its operations:
 - size - $\Theta(1)$
 - getElement - $\Theta(1)$
 - setElement - $\Theta(1)$
 - iterator - $\Theta(1)$
 - addToPosition - $O(n)$
 - deleteFromEnd - $\Theta(1)$
 - deleteFromPosition - $O(n)$
 - addToEnd - $\Theta(1)$ *amortized*

Amortized analysis

- In *asymptotic* time complexity analysis we consider one single run of an algorithm.
 - *addToEnd* should have complexity $O(n)$ - when we have to resize the array, we need to move every existing element, so the number of instructions is proportional to the length of the array.
 - Consequently, a sequence of n calls to the *addToEnd* operation would have complexity $O(n^2)$.

Amortized analysis

- In *asymptotic* time complexity analysis we consider one single run of an algorithm.
 - *addToEnd* should have complexity $O(n)$ - when we have to resize the array, we need to move every existing element, so the number of instructions is proportional to the length of the array.
 - Consequently, a sequence of n calls to the *addToEnd* operation would have complexity $O(n^2)$.
- In *amortized* time complexity analysis we consider a sequence of operations and compute the average time for these operations.
 - In amortized time complexity analysis we will consider the total complexity of n calls to the *addToEnd* operation and divide this by n , to get the *amortized* complexity of the algorithm.

Amortized analysis

- We can observe that if we consider a sequence of n operations, we rarely have to resize the array
- Consider c_i the cost (\approx number of instructions) for the i^{th} call to *addToEnd*
- Considering that we double the capacity at each resize operation, at the i th operation we perform a resize if $i-1$ is a power of 2. So, the cost of operation i , c_i , is:

$$c_i = \begin{cases} i, & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

- Cost of n operations is:

$$\sum_{i=1}^n c_i \leq n + \sum_{j=0}^{\lceil \log_2 n \rceil} 2^j < n + 2n = 3n$$

- The sum contains at most n values of 1 (this is where the n term comes from) and at most (integer part of) $\log_2 n$ terms of the form 2^j .
- Since the total cost of n operations is $3n$, we can say that the cost of one operation is 3, which is constant.

Amortized analysis

- While the worst case time complexity of *addToEnd* is still $O(n)$, the amortized complexity is $\Theta(1)$.
- The amortized complexity is no longer valid, if the *resize* operation just adds a constant number of new slots.
- In case of the *addToPosition* operation, both the worst case and the amortized complexity of the operation is $O(n)$ - even if *resize* is performed rarely, we need to move elements to empty the position where we put the new element.

- In order to avoid having a Dynamic Array with too many empty slots, we can resize the array after deletion as well, if the array becomes "too empty".
- How empty should the array become before resize? Which of the following two strategies do you think is better? Why?
 - Wait until the table is only half full ($\text{da.nrElem} \approx \text{da.cap}/2$) and resize it to the half of its capacity
 - Wait until the table is only a quarter full ($\text{da.nrElem} \approx \text{da.cap}/4$) and resize it to the half of its capacity