DATA STRUCTURES AND ALGORITHMS

LECTURE 2

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2018 - 2019

In Lecture 1...

- Course Organization
- Abstract Data Types and Data Structures
- Pseudocode
- Algorithm Analysis
  - O-notation

Today

1. Algorithm Analysis
2. Arrays

Office hours

- **Office hours**: Friday, between 14-16 in room 440 in the FSEGA building
- Participation at office hours should be announced by mail

O-notation - Reminder I

O-notation

For a given function $g(n)$ we denote by $O(g(n))$ the set of functions:

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 s. t. $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0\}$

- The O-notation provides an *asymptotic upper bound* for a function: for all values of n (to the right of n_0) the value of the function $f(n)$ is on or below $c \cdot g(n)$.
- We will use the notation $f(n) = O(g(n))$ or $f(n) \in O(g(n))$.

O-notation - Reminder II

- Graphical representation:

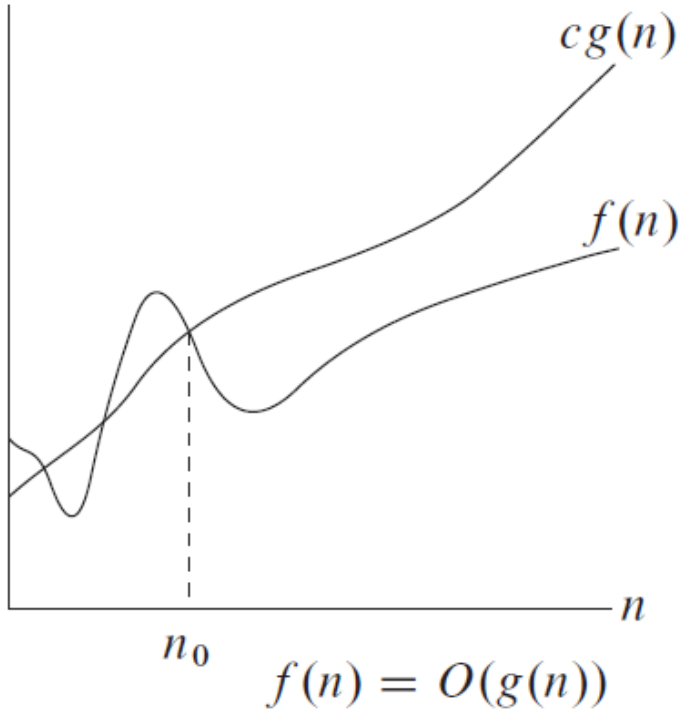


Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

O-notation - Reminder III

Alternative definition

$$f(n) \in O(g(n)) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

is either 0 or a constant (but not ∞).

- Consider, for example, $T(n) = n^2 + 2n + 2$:
  - $T(n) = O(n^2)$ because $T(n) \leq c * n^2$ for c = 2 and n $\geq$ 3
  - $T(n) = O(n^3)$ because

$$\lim_{n\to\infty} \frac{T(n)}{n^3} = 0$$

Ω-notation I

Ω-notation

For a given function $g(n)$ we denote by $\Omega(g(n))$ the set of functions:

$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0 s. t.
$0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0\}$

- The Ω-notation provides an *asymptotic lower bound* for a function: for all values of n (to the right of n_0) the value of the function $f(n)$ is on or above $c \cdot g(n)$.
- We will use the notation $f(n) = \Omega(g(n))$ or $f(n) \in \Omega(g(n))$.

Ω-notation II

- Graphical representation:

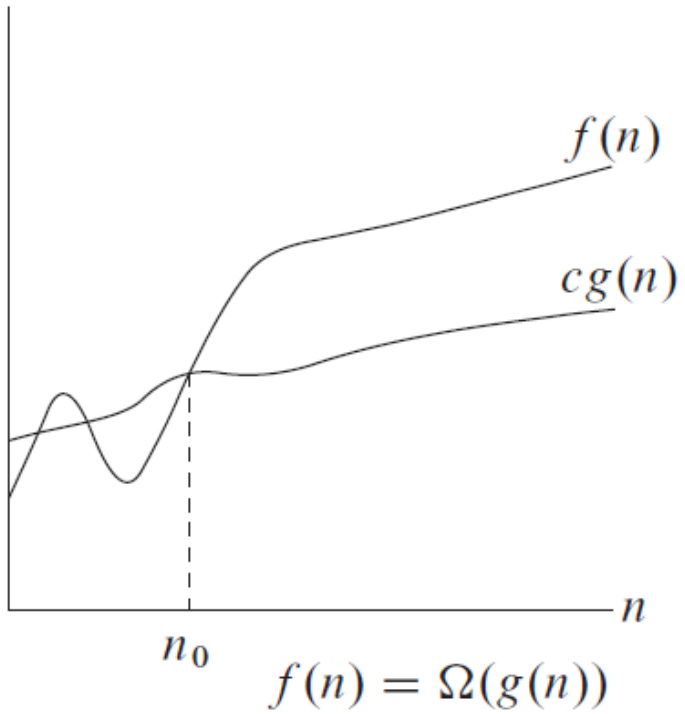


Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

Ω-notation III

Alternative definition

$$f(n) \in \Omega(g(n)) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

is ∞ or a nonzero constant.

- Consider, for example, $T(n) = n^2 + 2n + 2$:
  - $T(n) = \Omega(n^2)$ because $T(n) \geq c * n^2$ for c = 0.5 and n $\geq$ 1
  - $T(n) = \Omega(n)$ because

$$\lim_{n\to\infty} \frac{T(n)}{n} = \infty$$

Θ-notation I

Θ-notation

For a given function $g(n)$ we denote by $\Theta(g(n))$ the set of functions:

$\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 s. t. $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0\}$

- The Θ-notation provides an *asymptotically tight bound* for a function: for all values of n (to the right of n_0) the value of the function $f(n)$ is between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$.
- We will use the notation $f(n) = \Theta(g(n))$ or $f(n) \in \Theta(g(n))$.

Θ-notation II

- Graphical representation:

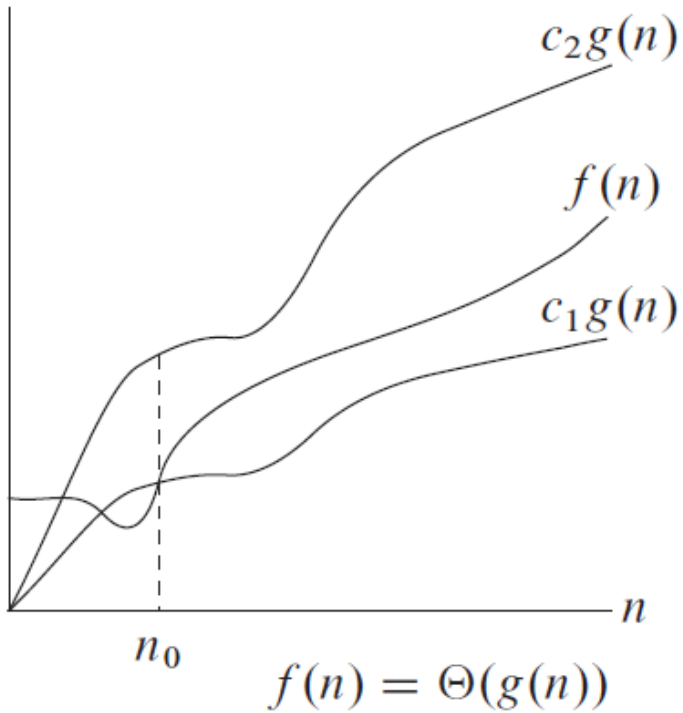


Figure taken from Corman et. al: Introduction to algorithms, MIT Press, 2009

Θ-notation III

Alternative definition

$$f(n) \in \Theta(g(n)) \text{ if } \lim_{n\to\infty} \frac{f(n)}{g(n)}$$

is a nonzero constant (and not ∞).

- Consider, for example, $T(n) = n^2 + 2n + 2$:
  - $T(n) = \Theta(n^2)$ because $c_1 * n^2 \leq T(n) \leq c_2 * n^2$ for $c_1 = 0.5$, $c_2 = 2$ and n ≥ 3.
  - $T(n) = \Theta(n^2)$ because

$$\lim_{n\to\infty} \frac{T(n)}{n^2} = 1$$

Best Case, Worst Case, Average Case I

? Think about an algorithm that finds the sum of all even numbers in an array. How many steps does the algorithm take for an array of length n?

? Think about an algorithm that finds the first occurrence of a number in an array. How many steps does the algorithm take for an array of length n?

Best Case, Worst Case, Average Case II

- For the second problem the number of steps taken by the algorithm does not depend just on the length of the array, it depends on the exact values from the array as well.
- For an array of fixed length n, execution of the algorithm can stop after:
  - verifying the first number - if it is the one we are looking for
  - verifying the first two numbers - if the first is not the one we are looking for, but the second is
  - ...
  - verifying all n numbers - first $n - 1$ are not the one we are looking for, but the last is, or none of the numbers is the one we are looking for

Best Case, Worst Case, Average Case III

- For such algorithms we will consider three cases:
  - Best - Case - the best possible case, where the number of steps taken by the algorithm is the minimum that is possible
  - Worst - Case - the worst possible case, where the number of steps taken by the algorithm is the maximum that is possible
  - Average - Case - the average of all possible cases.
- Best and Worst case complexity is usually computed by inspecting the code. For our example we have:
  - Best case: $\Theta(1)$ - just the first number is checked, no matter how large the array is.
  - Worst case: $\Theta(n)$ - we have to check all the numbers

Best Case, Worst Case, Average Case IV

- For computing the average case complexity we have a formula:

$$\sum_{I \in D} P(I) \cdot E(I)$$

- where:
  - D is the domain of the problem, the set of every possible input that can be given to the algorithm.
  - I is one input data
  - $P(I)$ is the probability that we will have I as an input
  - $E(I)$ is the number of operations performed by the algorithm for input I

Best Case, Worst Case, Average Case V

- For our example D would be the set of all possible arrays with length n
- Every I would represent a subset of D:
  - One I represents all the arrays where the first number is the one we are looking for
  - One I represents all the arrays where the first number is not the one we are looking for, but the second is
  - ...
  - One I represents all the arrays where the first $n-1$ elements are not the one we are looking for but the last one is
  - One I represents all the arrays which do not contain the element we are looking for

Best Case, Worst Case, Average Case VI

- P(I) is usually considered equal for every I, in our case: $\frac{1}{n+1}$

$$T(n) = \frac{1}{n+1}\sum_{i=1}^{n} i + \frac{n}{n+1} = \frac{n \cdot (n+1)}{2 \cdot (n+1)} + \frac{n}{n+1} \in \Theta(n)$$

Best Case, Worst Case, Average Case VII

- When we have best case, worst case and average case complexity, we will report the maximum one (which is the worst case), but if the three values are different, the total complexity is reported with the O-notation.

- For our example we have:
  - Best case: $\Theta(1)$
  - Worst case: $\Theta(n)$
  - Average case: $\Theta(n)$
  - Total (overall) complexity: $O(n)$

Empirical Analysis of Algorithms

- *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 solution

- The first solution 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 ← 0
    for i ← 1, n execute
        for j ← i, n execute
        //compute the sum of elements between i and j
            currentSum ← 0
            for k ← i, j execute
                currentSum ← currentSum + x[k]
            end-for
            if currentSum > maxSum then
                maxSum ← currentSum
            end-if
        end-for
    end-for
    first ← maxSum
end-function
```

Complexity of the algorithm:

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

Second solution

- 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 ← 0
    for i ← 1, n execute
        currentSum ← 0
        for j ← i, n execute
            currentSum ← currentSum + x[j]
            if currentSum > maxSum then
                maxSum ← currentSum
            end-if
        end-for
    end-for
    second ← maxSum
end-function
```

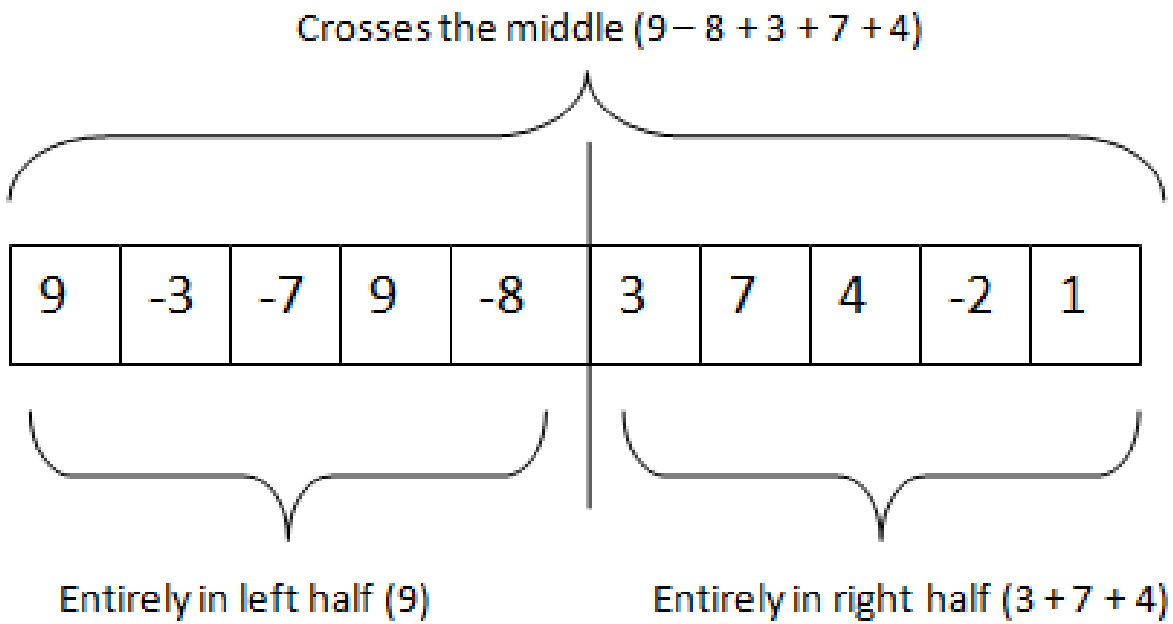
Complexity of the algorithm:

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

Third solution I

- The third solution uses the *Divide-and-Conquer* strategy. We can use this strategy if we notice that for an array of length n the subsequence with the maximum sum can be in three places:
  - Entirely in the left half
  - Entirely in the right half
  - Part of it in the left half and part of it in the right half (in this case it must include the middle elements)

Third solution 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?

Third solution III

- 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 5 to 5)
    - 1 (indexes 4 to 5)
    - -6 (indexes 3 to 5)
    - -9 (indexes 2 to 5)
    - 0 (indexes 1 to 5)
  - We will take the maximum (which is 1)

Third solution IV

- 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:
    - 3 (indexes 6 to 6)
    - 10 (indexes 6 to 7)
    - 14 (indexes 6 to 8)
    - 12 (indexes 6 to 9)
    - 13 (indexes 6 to 10)
  - We will take the maximum (which is 14)
- We will add the two maximums (15)

Third solution V

- 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.

Third solution VI

- We divide the implementation of the third algorithm in three separate algorithms:
  - One that computes the maximum subsequence sum crossing the middle - *crossMiddle*
  - One that computes the maximum subsequence sum between position [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 ← (left + right) / 2
    leftSum ← 0
    maxLeftSum ← 0
    for i ← middle, left, -1 execute
        leftSum ← leftSum + x[i]
        if leftSum > maxLeftSum then
            maxLeftSum ← leftSum
        end-if
    end-for
//continued on the next slide...
```

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

```
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 ← x[left]
   end-if
   middle ← (left + right) / 2
   justLeft ← fromInterval(x, left, middle)
   justRight ← fromInterval(x, middle+1, right)
   across ← crossMiddle(x, left, right)
   fromInterval ← @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 ← 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:

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

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

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

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

Let's go back to the notation with n.

If $n = 2^k \Rightarrow k = log_2 n$

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

Fourth solution

- 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 greater)
  - 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 greater)
  - 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 ← 0
    currentSum ← 0
    for i ← 1, n execute
        currentSum ← currentSum + x[i]
        if currentSum > maxSum then
            maxSum ← currentSum
        end-if
        if currentSum < 0 then
            currentSum ← 0
        end-if
    end-for
    fourth ← maxSum
end-function
```

Complexity of the algorithm:

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

Comparison of actual running times

Input size	**First** $\Theta(n^3)$	**Second** $\Theta(n^2)$	**Third** $\Theta(nlogn)$	**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 $\approx$ 1000 times more time
  - The second algorithm needs $\approx$ 100 times more time
  - The third algorithm needs $\approx$ 11-13 times more time
  - The fourth algorithm needs $\approx$ 10 times more time

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 ← False
    end-if
    middle ← (start + end) / 2
    if array[middle] = elem then
        BinarySeachR ← True
    else if elem < array[middle] then
        BinarySearchR ← BinarySearchR(array, elem, start, middle-1)
    else
        BinarySearchR ← BinarySearchR(array, elem, middle+1, end)
    end-if
end-function
```

Recursive Binary Search

- Initial call to the *BinarySearchR* algorithm for an ordered array of *nr* elements:

```
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

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

Master method

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

$$T(n) = a * T(\frac{n}{b}) + f(n)$$

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

Master method

- 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):

$$T(2^k) = T(2^{k-1}) + 1$$
$$T(2^{k-1}) = T(2^{k-2}) + 1$$
$$T(2^{k-2}) = T(2^{k-3}) + 1$$
$$\ldots$$
$$T(2^1) = T(2^0) + 1$$
$$\text{---------------------------} +$$
$$T(2^k) = T(2^0) + 1 + 1 + 1 + \ldots + 1 = 1 + k$$

- **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 ← 2 * i
        m ← 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 $4 \cdot T(2^{k-1})$

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

- 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:

$$T(2^k) = 4 \cdot T(2^{k-1}) + 1$$
$$4 \cdot T(2^{k-1}) = 4^2 \cdot T(2^{k-2}) + 4$$
$$4^2 \cdot T(2^{k-2}) = 4^3 \cdot T(2^{k-3}) + 4^2$$
$$\dots$$
$$4^{k-1} \cdot T(2^1) = 4^k \cdot T(2^0) + 4^{k-1}$$
$$\text{---------------------------------------} +$$
$$T(2^k) = 4^k \cdot T(1) + 4^0 + 4^1 + 4^2 + \dots + 4^{k-1}$$

- $T(1)$ 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.

Arrays

- 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 representation for other (more complex) data structures.

Arrays

- 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 (boolean values occupy one byte)
- Obs: Address of elements is 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?

Arrays

- 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 * size of an element

- **Obs:** If array indexing starts from 1, the above formula is still valid, but with $i - 1$ instead of i

Arrays

- 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)
  - *len* - denotes the actual number of elements stored in the array
  - *elems* - denotes the actual array with *capacity* slots for TElems allocated

DynamicArray:
cap: Integer
len: Integer
elems: TElem[]

Dynamic Array - Resize

- When the value of *len* 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).