DATA STRUCTURES AND ALGORITHMS

Extra reading 1

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2024 - 2025

Content

- The Master method
- Empirical algorithm analysis

Master method

- In Lecture 1 we have talked about how the complexity of recursive functions can be computed. We had to write the recursive formula for the complexity and do some computations.
- An alternative method, which does not imply computations is the *Master method*.
- The *Master method* can be used to compute the time complexity of algorithms having the following general recursive formula:

$$T(n) = a \cdot T(\tfrac{n}{b}) + f(n)$$

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

Master method

- Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function and let $T(n)$ be:

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

- $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.
2. If $f(n) = \Theta(n^{log_b a} lg^k n)$, for some constant $k \geq 0$ then $T(n) = \Theta(n^{log_b a} * lg^{k+1} n)$.
3. If $f(n) = \Omega(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a * f(n/b) \leq c * f(n)$ for some constant $c > 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

- In short, in the master method, the function $f(n)$ is compared to $n^{log_b a}$ and the complexity of the recursive function will be given by the result of the comparison.

- **Obs:** the three cases of the master method do not cover every possible situation. It is possible that none of the three branches can be applied and then an alternative method is needed to compute the complexity.

Examples with the Master method I

- In the following, a few examples of using the master method will be given. We will start directly from the recurrence relation.
- Example 1

$$T(n) = 9T(\frac{n}{3}) + n$$

- In this case $a = 9$ and $b = 3$, $f(n) = n$.
- We need to compare $f(n)$ to $n^{log_3 9} = n^2$.
- Considering $\epsilon = 1$, we can say that $f(n) = O(n^{log_3 9 - \epsilon})$, so the first case of the master method applies and the complexity is $T(n) = \Theta(n^2)$.

Examples with the Master method II

- Example 2

$$T(n) = T(\frac{2n}{3}) + 1$$

- In this case $a = 1$, $b = \frac{2}{3}$ and $f(n) = 1$.
- We need to compare $f(n)$ to $n^{log_{2/3}1} = n^0 = 1$.
- Since $f(n) = \Theta(n^{log_b a} * lg^k n)$, for $k = 0$ the second case of the master method applies and the complexity is $T(n) = \Theta(lgn)$.

Examples with the Master method III

- Example 3

$$T(n) = 3 * T(\frac{n}{4}) + n * lg\ n$$

- In this case $a = 3$, $b = 4$ and $f(n) = n * lg\ n$.
- We need to compare $f(n)$ to $n^{log_4 3} = n^{0.79}$.
- Considering $\epsilon = 0.1$, $f(n) = \Omega(n^{log_b a + \epsilon})$ so the third case of the master method applies and the complexity is $T(n) = \Theta(n * lg\ n)$

Examples with the Master method IV

- Example 4

$$T(n) = 2 * T(\frac{n}{2}) + n * lg\ n$$

- In this case $a = 2$ and $b = 2$ and $f(n) = n * lg\ n$.
- We need to compare $f(n)$ to $n^{log_2 2} = n$.
- Since $f(n) = \Theta(n^{log_b a} * lg^k n)$ for $k = 1$, the second case of the master method applies and the complexity is $T(n) = \Theta(n * lg^2 n)$

Examples with the Master method V

- Example 5

$$T(n) = 8 * T(\frac{n}{2}) + 1$$

- In this case $a = 8$, $b = 2$ and $f(n) = 1$.
- We need to compare $f(n)$ to $n^{log_2 8} = n^3$.
- Since $f(n) = O(n^{log_2 8 - \epsilon})$, for $\epsilon = 1$, we can apply the first case and conclude that the complexity is $T(n) = \Theta(n^3)$.

Empirical Analysis of Algorithms

- During Lecture 1 we have talked about *asymptotic algorithm analysis* (when you use the O, Θ and Ω notations).
- There is also *empirical algorithm analysis*:
  - If you have several algorithms that solve the same problem and you want to know which is the best one, empirical analysis means to implement them all, run them all, measure the run-time and compare them.
  - Obviously, in many situations it is not feasible to implement several versions of an algorithm, and asymptotic analysis can tell us which is the best one, without implementing them
  - Empirical analysis however, might seem more hands-on

Empirical Analysis of Algorithms - Examples I

- In the following two examples are presented.
- Example 1 is a classic problem, which has 4 different solutions, with 4 different complexities. You will find on the slides:
  - the pseudocode for the 4 possible solutions
  - the asymptotic complexity for each solution
  - a table where you have actual run-time measurements (empirical analysis) for the 4 solutions.
- Goal of Example 1 is to show you that we can measure empirically what asymptotic analysis claims.

Empirical Analysis of Algorithms - Examples II

- Example 2 is an empirical measurement of the following claim from Lecture 1:
  - The following piece of code in Python has a different complexity depending on whether *cont* is a *list* or *dict*.

```
if elem in cont:
    print("Found")
```

- Goal of Example 2 is to show you that you can be a better programmer and write more efficient code if you know how containers are implemented.

Example 1 - Problem statement

- *Given an array of positive and negative values, find the maximum sum that can be computed for a subsequence. If the array contains only negative numbers, we assume the maximum sum 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)

Example 1 - First algorithm

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

```
function first (x, n) is:
//x is an array of integer numbers, n is the length of x
    maxSum ← x[1]
    for i ← 1, n execute //beginning of the sequence
        for j ← i, n execute //end of the sequence
        //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 (for loops in the code can be written as sums, when computing complexity):

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

Example 1 - Second algorithm

- In the first algorithm, if, at a given step, we have computed the sum of elements between positions i and j, the next computed 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
```

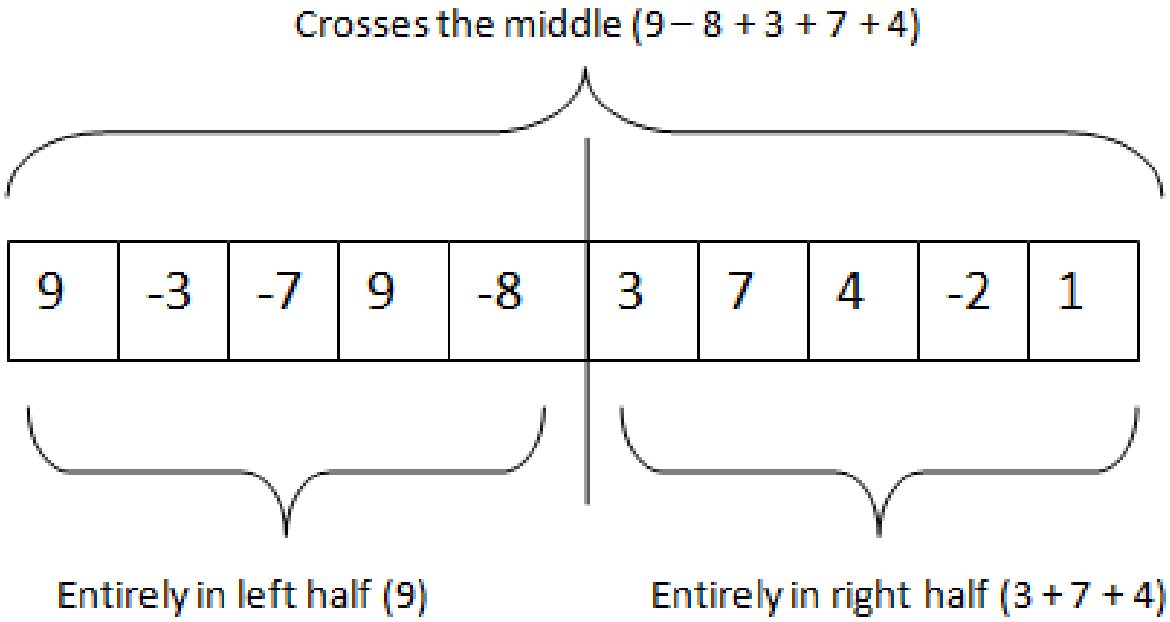
Complexity of the algorithm:

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

Third algorithm I

- The third algorithm 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 one of the following 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 algorithm 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 algorithm 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 algorithm 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 algorithm 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 algorithm 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 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 ← (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 $\leftarrow$ fromInterval(x, 1, n)
end-function

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

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

- In case of a recursive algorithm, complexity computation starts from the recursive formula of the algorithm.
  - The two recursive calls for the left and right half give us the T(n/2) terms, while the *crossMiddle* algorithms has a complexity of $\Theta(n)$

Let $n = 2^k$

Ignoring the parameter x we rewrite the recursive branch:

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

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

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

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

Let's go back to the notation with n.

$$\text{If } n = 2^k \Rightarrow k = log_2 n$$
$$T(n) = n + n * log_2 n \in \Theta(n log_2 n)$$

Fourth algorithm

- 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

- For every input size, 5 random arrays were generated with that specific input size and all four algorithms were run on the inputs.
- Actual running time was measured using Python's default_timer and the average running time (for the 5 arrays) is recorded in the following tables.
- The first table contains executions when the input size was doubled, while the second contains executions when the input size was multiplied by 10.

Input size	**First** $\Theta(n^3)$	**Second** $\Theta(n^2)$	**Third** $\Theta(nlogn)$	**Fourth** $\Theta(n)$
1,000	43.5687	0.194647	0.013558	0.000509
2,000	339.007	0.7634	0.02838	0.00101
4,000	2686.92	3.0412	0.0591	0.002003
10,000	-	18.8334	0.1541	0.00493
20,000	-	76.06477	0.324736	0.01031
40,000	-			

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

Comparison of actual running times I

- From the previous table we can see that complexity and running time are indeed related:
- When the input is doubled:
  - The first algorithm ($\Theta(n^3)$) needs $\approx$ 8 times more time
  - The second algorithm ($\Theta(n^2)$) needs $\approx$ 4 times more time
  - The third algorithm ($\Theta(n * logn)$) needs $\approx$ 2.1 times more time
  - The fourth algorithm ($\Theta(n)$) needs $\approx$ 2 times more time

Input size	**First** $\Theta(n^3)$	**Second** $\Theta(n^2)$	**Third** $\Theta(nlogn)$	**Fourth** $\Theta(n)$
100	0.04509	0.00205	0.00119	0.0000606
1,000	43.5687	0.194647	0.013558	0.000509
10,000	-	18.8334	0.1541	0.00493
100,000	-	-	1.7181	0.04961
1,000,000	-	-	19.695	0.5127
10,000,000	-	-	213.859	5.1284

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

Comparison of actual running times I

- 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 times more time
  - The fourth algorithm needs $\approx$ 10 times more time

Example 2

- Consider the following algorithm (written in Python), which checks how many elements from the list *l* can be found in the container *cont*.

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

Example 2 - Scenario 1

- Consider the following scenario for a given integer number *size*:
  - Generate a random list with *size* with unique elements from the interval [0, size * 2)
  - Add these elements in a container (list or dictionary - value is equal to key for dictionary)
  - Generate another random list with *size* elements from the interval [0, size * 2)
  - Call the *testContainer* function for the container and the second list and measure the execution time for it.
  - Repeat this 5 times every time regenerating the arrays

Example 2

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

Size	Time for list	Time for dictionary	Average count
10	0.00000957	0.00000656	4.8
100	0.0000889	0.0000545	54.4
1000	0.004795	0.000336	487.4
10000	0.43684	0.00345	4956.4
100000	42.489	0.0377	49999.8
1000000	-	0.4828	499850.8
10000000	-	6.1214	5000022.6

Example 2

- We can see that when the size of the list is multiplied by 10:
  - The time for the list increases by up to ≈ 100 times.
  - The time for the dictionary increases by up to ≈ 11 times.
- **Obs:** the average value of *count* tells us, how many successful and how many unsuccessful search did we have. This is important, because in case of a successful search, the search algorithms stop checking. In our case $\approx 50\%$ of the searches is unsuccessful.

Example 2 - Scenario 2

- Consider the following scenario for a given integer number *size*:
  - Generate a random list with *size* with unique elements from the interval [0, size * 2)
  - Add these elements in a container (list or dictionary - value is equal to key for dictionary)
  - Call the *testContainer* function for the container and the list and measure the execution time for it.
  - Repeat this 5 times every time regenerating the arrays
  - **Obs.** Since this time we are searching for the elements from the same list from which we have added them to the container, we will have only successful searches.

Example 2

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

Size	Time for list	Time for dictionary	Average count
100	0.00007218	0.00004096	100
1000	0.003012	0.000352	1000
10000	0.26049	0.00368	10000
100000	38.791	0.042377	100000
1000000	-	0.528	1000000
10000000	-	6.1727	10000000

Example 2

- We can see that we have similar values and similar differences as in case of the first scenario: the algorithm on the list takes a lot longer and in general it takes around 100 times longer if we have an input with 10 times as many elements.
- **Obs:** I have left the column with the Average count in the table to see that indeed all searches are successful.

Example 2 - Scenario 3

- Consider the following scenario for a given integer number *size*:
  - Generate a random list with *size* with unique elements from the interval (0, size * 2)
  - Add these elements in a container (list or dictionary - value is equal to key for dictionary)
  - Create a second list by multiplying the elements of the first with -1.
  - Call the *testContainer* function for the container and the second list and measure the execution time for it.
  - Repeat this 5 times every time regenerating the arrays
  - **Obs.** Since this time we are searching for negative elements in a list containing only positive elements, we will have only unsuccessful searches.

Example 2

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

Size	Time for list	Time for dictionary	Average count
100	0.0001008	0.0000328	0
1000	0.005228	0.0002279	0
10000	0.49816	0.0024156	0
100000	94.209	0.0278	0
200000	542.078	0.05444	0
1000000	-	0.34004	0
10000000	-	4.5806	0

Example 2

- Besides the observations that we have made previously, for this scenario we have another interesting observation: how inefficient the list becomes when we have a lot of unsuccessful searches. The running time for 100,000 elements is almost twice as in case of successful searches. And while small variations in the running time are absolutely normal, this is a very big difference.

Example 2

- The difference between these run-times is given by the difference in how the list and the dictionary are implemented, more exactly the data structure which is used to store the elements.
- We will talk about the details during this semester.
- While you can use these containers without knowing anything about their implementation, you can write more efficient code if you consider implementation details as well.

Source code

- You can find the Python source code for both examples on Ms Teams (folder Lecture 1), if you want to experiment with it. Obviously, do not expect to get the same results as me (your computer might be faster), but you should see similar differences between run-times, when the value of n is incremented.
- The current implementation of Example 2 only measures the time for running the function *testContainer*. If you want to play around with it, include in the timing the part which creates the container (list or dictionary) or measure that one separately as well, to see if there are differences there.