DATA STRUCTURES AND ALGORITHMS

Extra reading - Empirical algorithm analysis

Lect. PhD. Oneț-Marian Zsuzsanna

Babeș - Bolyai University
Computer Science and Mathematics Faculty

2020 - 2021

- 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.

- 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 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)

- 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 ← 0
   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):

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

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.

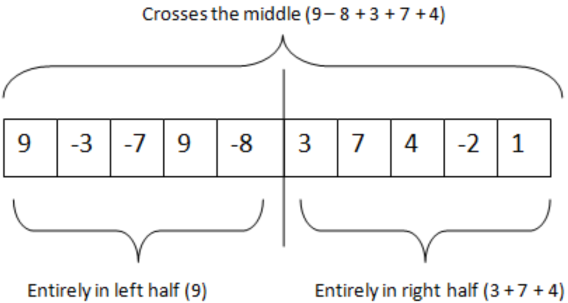
```
function second (x, n) is:
//x is an array of integer numbers, n is the length of x
   maxSum ← 0
   for i ← 1, n execute
      currentSum ← 0
      for j ← i, n execute
         currentSum ← currentSum + x[j]
         if currentSum > maxSum then
            maxSum ← currentSum
         end-if
 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



- 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 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.

- 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 ← 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:

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

$$\overline{\hspace{6cm}} +$$

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

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

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

Let's go back to the notation with n.

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:

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

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 in seconds 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

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, 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, size * 2)
  - Call the *testContainer* function for the container and the second list and measure the execution time for it.

Example 2

- Execution times in seconds (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

- 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.
- If you are curious, see whether the dictionary solution is faster if you include in the measurement the time needed to create the dict from the elements of the list. This is currently done outside of timing.