Assignment 2 solutions

Responsible TA: Claudi Lleyda Moltó

Task 1 - Asymptotic growth rate

Express the following functions using Theta notation (i.e., find the "simplest" function g such that $f(n) = \Theta(g(n))$ for each of the functions). Justify your answer.

1. $f(n) = n^2 - 10n + 20$
2. $f(n) = 3n + k\log_2(n)$, where $k > 0$
3. $f(n) = (n+k)^2 2^{n+k}$, where $k > 0$
4. $f(n) = n(\log_2(n) + \log_3(n) + \log_4(n))$
5. $f(n) = \sqrt[3]{kn} + 4$, where $k > 0$
6. $f(n) = 6 * 2^n + 2 * 6^n$
7. $f(n) = n^2 + n^k \log(n)$, where $k > 0$

Solution. In page 38 of the course book we find the following theorem

> Let f and g be two functions such that
>
> $$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$
>
> exists and is equal to some number $c > 0$. Then $f(n) = \Theta(g(n))$.

We can use this result to solve this problem.

1. We take $g(n) = n^2$ and we calculate

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n^2 - 10n + 20}{n^2} \\ &= \lim_{n\to\infty} \left(\frac{n^2}{n^2} - \frac{10n}{n^2} + \frac{20}{n^2}\right) \\ &= 1 - 0 + 0 = 1\end{aligned}$$

   and therefore $f(n) = \Theta(n^2)$.

2. We take $g(n) = n$ and we calculate

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{3n + k\log_2(n)}{n} \\ &= \lim_{n\to\infty} \left(\frac{3n}{n} + \frac{k\log_2(n)}{n} \right) \\ &= 3 + 0 = 3\end{aligned}$$

and therefore $f(n) = \Theta(n)$.

3. We take $g(n) = n^2 2^n$ and we calculate

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{(n+k)^2 2^{n+k}}{n^2 2^n} \\ &= \lim_{n\to\infty} \frac{(n+k)^2}{n^2} \frac{2^{n+k}}{2^n} \\ &= \lim_{n\to\infty} \frac{n^2 + 2nk + k^2}{n^2} \frac{2^n 2^k}{2^n} \\ &= \lim_{n\to\infty} \left(\frac{n^2}{n^2} + \frac{2nk}{n^2} + \frac{k^2}{n^2} \right) \frac{2^n 2^k}{2^n} \\ &= \lim_{n\to\infty} \left(\frac{n^2}{n^2} + \frac{2nk}{n^2} + \frac{k^2}{n^2} \right) \lim_{n\to\infty} \frac{2^n 2^k}{2^n} \\ &= (1 + 0 + 0) \lim_{n\to\infty} \frac{2^n 2^k}{2^n} \\ &= 2^k \lim_{n\to\infty} \frac{2^n}{2^n} = 2^k\end{aligned}$$

and we know that $2^k > 0$. Therefore $f(n) = \Theta(n^2 2^n)$.

4. We take $g(n) = n\log(n)$, where $\log(n)$ is the natural logarithm, and we calculate

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n(\log_2(n) + \log_3(n) + \log_4(n))}{n\log(n)} \\ &= \lim_{n\to\infty} \left(\frac{n\log_2(n)}{n\log(n)} + \frac{n\log_3(n)}{n\log(n)} + \frac{n\log_4(n)}{n\log(n)} \right) \\ &= \lim_{n\to\infty} \left(\frac{\log_2(n)}{\log(n)} + \frac{\log_3(n)}{\log(n)} + \frac{\log_4(n)}{\log(n)} \right) \\ &= \lim_{n\to\infty} \left(\frac{1}{\log(2)} + \frac{1}{\log(3)} + \frac{1}{\log(4)} \right) \approx 3\end{aligned}$$

and therefore $f(n) = \Theta(n\log(n))$.

5. We take $g(n) = \sqrt[3]{n}$ and we calculate

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{\sqrt[3]{kn}+4}{\sqrt[3]{n}} \\ &= \lim_{n\to\infty} \left(\frac{\sqrt[3]{kn}}{\sqrt[3]{n}} + \frac{4}{\sqrt[3]{n}} \right) \\ &= \lim_{n\to\infty} \sqrt[3]{\frac{kn}{n}} + \lim_{n\to\infty} \frac{4}{\sqrt[3]{n}} \\ &= \sqrt[3]{k \lim_{n\to\infty} \frac{n}{n}} + 0 \\ &= \sqrt[3]{k}\end{aligned}$$

and since $k > 0$, we know that $\sqrt[3]{k} > 0$. Therefore $f(n) = \Theta(\sqrt[3]{n})$.

6. We take $g(n) = 6^n$ and we calculate

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{6 * 2^n + 2 * 6^n}{6^n} \\ &= \lim_{n\to\infty} \left(\frac{6 * 2^n}{6^n} + \frac{2 * 6^n}{6^n} \right) \\ &= \lim_{n\to\infty} \left(6\left(\frac{2}{6}\right)^n + 2\left(\frac{6}{6}\right)^n \right) \\ &= \lim_{n\to\infty} \left(6\left(\frac{1}{3}\right)^n + 2 * 1^n \right) \\ &= 0 + 2 = 2\end{aligned}$$

and therefore $f(n) = \Theta(6^n)$.

7. We take

$$g(n) = \begin{cases} n^2 & k < 2 \\ n^k \log(n) & k \geq 2 \end{cases}$$

and we calculate, if $k < 2$,

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n^2 + n^k \log(n)}{n^2} \\ &= \lim_{n\to\infty} \left(\frac{n^2}{n^2} + \frac{n^k \log(n)}{n^2} \right) \\ &= 0 + 1 = 1\end{aligned}$$

and if $k \geq 2$,

$$\begin{aligned}\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{n^2 + n^k \log(n)}{n^k \log(n)} \\ &= \lim_{n\to\infty} \left(\frac{n^2}{n^k \log(n)} + \frac{n^k \log(n)}{n^k \log(n)} \right) \\ &= \lim_{n\to\infty} \left(\frac{1}{n^{k-2} \log(n)} + 1 \right) \\ &= 1 + 0 = 1\end{aligned}$$

and therefore $f(n) = \Theta(g(n))$, that is $f(n) = \Theta(n^2)$ if $k < 2$ and $f(n) = \Theta(n^k \log(n))$ if $k \geq 2$.

The choices of $g(n)$ are likely to vary from person to person, so these are more like guidelines. $\Diamond$

Task 2 - Runtime analysis

Analyse the runtime complexity of the following algorithm, providing the time complexity in big-O notation. Justify your answer.

```
function FindMaxSubarraySum(Data array; Integer low, high)
    if low = high then
        return array[low]
    end if
    mid ← (low+high)/2
    leftMax ← FindMaxSubarraySum(array, low, mid)
    rightMax ← FindMaxSubarraySum(array, mid + 1, high)
    crossMax ← FindMaxCrossSubarraySum(array, low, mid, high)
    return max(leftMax, rightMax, crossMax)
end function

function FindMaxCrossSubarraySum(Data array; Integer low, mid, high)
    leftSum ← −∞
    sum ← 0
    for iter from mid to low do
        sum ← sum + array[iter]
        if sum > leftSum then
            leftSum ← sum
        end if
    end for
    rightSum ← −∞
    sum ← 0
    for iter from mid + 1 to high do
        sum ← sum + array[iter]
        if sum > rightSum then
            rightSum ← sum
        end if
    end for
    return leftSum + rightSum
end function
```

Solution. Let n be the number of elements in the array.

The FindMaxCrossSubarraySum function performs two functionally identical loops, with a combined n iterations in the worst case scenario, and the complexity of each iteration of these loops is $O(1)$. Therefore, we get

$$\text{FindMaxCrossSubarraySum} = O(n).$$

The FindMaxSubarraySum function follows a divide-and-conquer approach on the array. Each call of the function calls itself twice, each with one half of the input array, and FindMaxCrossSubarraySum once with the whole input array, which we saw is $O(n)$.

Therefore, at every recursion level FindMaxSubarraySum will perform $O(n)$ operations.

To calculate how many recursion levels FindMaxSubarraySum will perform, we note that at every recursion level the array size is reduced by $\frac{1}{2}$. Therefore, at level k we will have an array of size $\frac{n}{2^k}$. The base case for FindMaxSubarraySum is when the array has size 1, so to find at which level this occurs, we must solve for k in

$$1 = \frac{n}{2^k},$$

which gives us $k = \log_2(n)$.

Altogether, we have deduced that FindMaxSubarraySum performs a maximum of $k = \log_2(n)$ recursion levels, each of which performs n operations. Therefore

$$\text{FindMaxSubarraySum} = O\big(n \log(n)\big). \qquad \Diamond$$

Task 3 - Priority queues

Write a simple pseudocode implementation of the `enqueue` and `dequeue` operations for a priority queue using a stack as base type. What is the big-O running time of each operation? Justify your answer.

Solution. There are multiple ways to implement these operations within the constraints, depending mainly on how the data is stored. The `dequeue` operation only specifies that the elements must be retrieved in order, and different approaches may have different big-O running times for both operations. It is not necessary for this exercise to give the most performant implementation; as long as a solution is technically correct, with the accompanying analysis, it should be fine.

Additionally, there is no need to worry about the correctness edge cases, such as the priority queue being empty when dequeuing. These are details beyond the point of the exercise and do not affect the overall big-O running time. We will assume there is always elements to be dequeued from the priority queue, and other similar convenience assumptions that might occur in other approaches.

That being said, this is one possible implementation where the elements are stored in order in the stack.

```
1:  function Enqueue(Stack stack; Object object)
2:      temp_stack ← NewStack()
3:      while Peek(stack) > object do
4:          temp_object ← Pop(stack)
5:          Push(temp_stack, temp_object)
6:      end while
7:      Push(stack, object)
8:      while ¬IsEmpty(temp_stack) do
9:          temp_object ← Pop(temp_stack)
10:         Push(stack, temp_object)
11:     end while
12: end function

13: function Dequeue(Stack stack)
14:     return Pop(stack)
15: end function
```

Let's first check that the implementation is correct.

Note how in the loop in line 3 we are filtering out elements bigger than the element we want to insert, to find it's position in the queue.

The filtered out elements must be stored somewhere temporarily, and for this we use a temporary stack. Conveniently for us, when retrieving the elements back from the stack, in the loop in line 8, these are retrieved in the correct order (that is, the last in is first out), simplifying the interactions with the temporary stack.

Therefore, the object to be queued is inserted at its correct position, and the elements that were removed are placed back in the original order.

For the dequeue operation, we only need to pop the top element in the stack, since the stack is already sorted.

Note for the dequeue operation simply calls the pop function from the stack. Therefore, the big-O running time of Dequeue will be the same as Pop, that is

$$\textsc{Dequeue} = O(\textsc{Pop}).$$

Let n be the number of elements in the priority queue. The enqueue operation has two independent loops, each of which will perform n iterations in the worst case scenario. In both loops, every iteration performs one call to Pop and another to Push. Therefore, both loops have a big-O running time of $O\big(n(\textsc{Pop} + \textsc{Push})\big)$.

Lastly, each call of Enqueue will call NewStack and Push a single time each. Therefore, its total big-O running time will be

$$\textsc{Enqueue} = O\big(\textsc{NewStack} + \textsc{Push} + n(\textsc{Pop} + \textsc{Push})\big).$$

To complete the exercise we must now find out the big-O running time of the stack operations, which we can consult anywhere. Luckily, we find that these are all $O(1)$, meaning we can simplify our results to

$$\textsc{Dequeue} = O(1) \qquad \text{and} \qquad \textsc{Enqueue} = O(n). \qquad \lozenge$$