TND004: Lab 1

Joel Paulsson, *joepa811*
William Uddmyr, *wilud321*

April 16, 2020

Exercise 3

Iterative

Time complexity

If we only consider the most costly aspects of our implementation the time complexity $T(n)$ depends on the body of the loop, how many times the loop is executed and the reverse function. The body of the loop consists of one function call to function `even()`. The function makes a simple calculation and returns a `bool` i.e. it's of constant time complexity: $\mathcal{O}(1)$. The `std::reverse` function is equivalent to a while loop and a call to a `swap` function and is therefore linear i.e $\mathcal{O}(n)$. Regarding the numbers of iterations in the `for` loop, it doesn't matter what elements the sequence consists of it will run $n-1$ times anyway, giving a linear time complexity. Hence the total time complexity becomes: $T(n)$=$\mathcal{O}(n)$.

Space complexity

The space complexity $S(n)$ is determined by the most costly parts of the code. In this case it's when allocating memory for the `std::vector` tempVec. The space complexity for the allocation is linear, i.e $S(n) = \mathcal{O}(n)$.

Divide-and-conquer

Time complexity

The time complexity for the recursive function is defined by the most costly aspects of the code, i.e the function calls: The call to the function `even()`, the recursive calls to `TND004::stable_partition()` and the call to `std::rotate()`. `even()` have constant time complexity, $T(n) = \mathcal{O}(1)$. `TND004::stable_partition()` have $T(n) = \mathcal{O}(\frac{n}{2})$ since it's two calls each time for half the sequence. `std::rotate()` have linear time complexity, $T(n) = \mathcal{O}(n)$. We get the following expression:

$$\begin{cases} T(n) = 1 & ,if\, n = 1 \\ T(n) = 3 + 2T(n/2) + \mathcal{O}(n) & ,if\, n > 1 \end{cases}$$

Using the master theorem gives the following:

$$a = 3, b = 2, c = 2$$

$$f(n) = 3 + \mathcal{O}(n) = \mathcal{O}(n) \Rightarrow k = 1$$

We get:

$$T(n) = f(n) + T(n/b)$$

$$b = c^k \Rightarrow k = 1.$$

Hence:

$$\mathcal{O}(n^k \log(n)) \Rightarrow T(n) = \mathcal{O}(n \log(n))$$

Space complexity
The space complexity $S(n)$ is defined by the most costly aspects of the code, i.e when initialising the mid iterator and when the recursive calls are made. Initialising has a space complexity of $S(n) = 1$ and the recursive calls $S(n) = \frac{n}{2}$, since the sequence is split into half before the recursive calls. Resulting in: $S(n) = 1 + S(\frac{n}{2})$ where $n > 1$, otherwise $S(n) = 0$. Applying the master theorem:

$$a = 0, b = 1, c = 2$$

$$f(n) = 1 = \mathcal{O}(1) = \mathcal{O}(n^k) \Rightarrow k = 0$$

Hence:

$$b = c^k \Rightarrow k = 0 \Rightarrow$$

$$\mathcal{O}(n^k \log(n)) \Rightarrow S(n) = \mathcal{O}(\log(n))$$

Conclusion

The time and space complexity for the functions are presented in table 1. Conclusions that can be determined are: The iterative function preform better regarding time complexity, but the recursive function is better regarding space complexity.

Table 1: *Results.*

	Iterative	**Divide-and-conquer**
Time complexity $T(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log(n))$
Space Complexity $S(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log(n))$