

## TND004: Data Structures

### Lab 1

#### Goals

- To use recursion versus iteration.
- To use divide-and-conquer algorithm design technique.
- To analyse the time and space complexity of algorithms.
- To compare the efficiency of different algorithms that solve the same problem.

#### Preparation

You must perform the tasks listed below before the start of the lab session *Lab1 HA*. In each *HA* lab session, your lab assistant has the possibility to discuss about three questions with each group. Thus, it is important that you read the lab descriptions and do the indicated preparation steps so that you can make the best use of your time during the lab sessions.

- Download the [files for this exercise](#) from the course website. Then, you can use [CMake](#) to create a project for this lab. For how to use CMake, you can see this [short guide](#). Make sure you have installed CMake.
- Compile, link, and execute the program. The first test fails (“TEST PHASE 1”) and the program ends, since not all functions are yet fully implemented.
- Assertions are used to test your code. See the appendix, for more information about [how assertions can be used to test code](#).
- Review the concepts presented in [lecture 1](#) and [lecture 2](#).
- The exercise in this lab is about the stable partition problem. Read the [description](#) of this problem.
- Do [exercise 1](#).
- Read and understand the [divide-and-conquer algorithm](#) to stable partition a sequence.
- Simulate manually in a piece of paper the execution of the divide-and-conquer algorithm for the following sequences. You don’t need to show this part to your lab assistant. This is just for your own understanding.
  - $S = \langle 3, 3 \rangle$
  - $S = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9 \rangle$
- Read the section “[Presenting lab and deadlines](#)”.

If you have any specific question about the exercises, then send us an e-mail. Be short and concrete, otherwise you won’t get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail’s subject, i.e. “TND004: ...”.

## The stable partition problem

Consider the following boolean function declaration

```
bool p(int i);
```

and a sequence  $S$  of values (e.g. integers). The stable partition of sequence  $S$ , using  $p$ , `stable_partition(S, p);`

is a function that rearranges the elements in  $S$ , in such a way that all the elements for which  $p$  returns `true` precede all those for which it returns `false`, and the relative order of elements within each group is preserved.

For example, assume  $S = \langle 3, 4, 1, 2, 5, 6, 7, 8, 9 \rangle$  and consider the boolean function even to test whether an integer is even.

```
bool even(int i) {  
    return i % 2 == 0;  
}
```

Then, `stable_partition(S, even)` rearranges the items in  $S$  as follows.

$S = \langle 4, 2, 6, 8, 3, 1, 5, 7, 9 \rangle$

Note that  $\langle 2, 4, 6, 8, 3, 1, 5, 7, 9 \rangle$  is not a stable partition of  $S$  because both 2 and 4 are even numbers and they appear in swapped order when compared with the original sequence.

The stable partition problem is used to solve many practical problems (e.g. sorting). Due to its relevance, the C++ standard library contains an implementation of it, [`std::stable\_partition`](#).

In this lab, you are not going to use `std::stable_partition`. Instead, you are going to make your own implementation, using two different algorithms that solve the stable partition problem.

- The first algorithm is iterative (i.e. recursion should not be used) and executes in  $O(n)$  time, for any sequence with  $n > 0$  items. Make sure the underlying constants are as low as possible (remember that an algorithm that executes e.g.  $2n$  steps is preferred to an algorithm that executes  $1000n$  steps).
- The second algorithm uses a divide-and-conquer strategy.

This lab consists of three exercises.

- **Exercise 1:** to create and implement an iterative linear algorithm<sup>1</sup> that solves the stable partition problem of a sequence.
- **Exercise 2:** to implement the divide-and-conquer algorithm described [here](#).
- **Exercise 3:** to analyze the running time and space usage of both algorithms.

### Exercise 1: iterative algorithm

You are requested to design an **iterative linear time algorithm** that creates a stable partition of a given sequence and then implement it in the function `TND004::stable_partition_iterative`.

---

<sup>1</sup> If a sequence has  $n > 0$  items then a linear time algorithm executes  $O(n)$  steps, in the worst-case.

Test whether your code compiles and runs with the main given in `lab1.cpp`. No assertions should fail.

## Exercise 2: divide-and-conquer algorithm for the stable partition

Divide-and-conquer is a common technique to design algorithms. Divide-and-conquer algorithms consist of two parts.

- **Divide:** divide the problem into two or more smaller sub-problems, with exception for the base cases. Each of these smaller sub-problems are then solved recursively.
- **Conquer:** The solutions of the smaller sub-problems are then put together to create a solution for the original problem.

A divide-and-conquer algorithm to solve the stable partition problem is described below. The [figures in the appendix](#) help to illustrate the algorithm.

1. **Divide:** divide the sequence  $S = \langle v_1, \dots, v_n \rangle$ , with  $n > 1$ , in two halves:  $S_L = \langle v_1, \dots, v_{mid-1} \rangle$  and  $S_R = \langle v_{mid}, \dots, v_n \rangle$ . Then, apply stable partition recursively to  $S_L$  and  $S_R$ . Empty sequences or one-item sequences form the base cases.
2. **Conquer:** use the C++ STL algorithm [std::rotate](#) to place the  $S_R$ -block of items with property  $p$  (e.g. to be an even number) just after the  $S_L$ -block of items with the same property.

**Exercise 2** of this lab consists in understanding and then implementing the algorithm described above. You should add your code for this exercise to the auxiliary function `TND004::stable_partition2` in the file `lab1.cpp`. Note that the fourth argument, `std::function<bool(int)> p`, represents a function with an integer argument returning a bool value.

Uncomment the last three lines of the function named `execute`. Test whether your code compiles and runs with the given main. No assertions should fail.

## Exercise 3: algorithms analysis

This exercise requires that you analyze the running time and space usage of both algorithms you have implemented. Use Big-Oh notation and write a **clear motivation** for your analysis.

1. Analyze the time and space complexity of the iterative algorithm.
2. Analyze the time and space complexity of the divide-and-conquer algorithm.
3. Compare both algorithms. In which situations would you prefer to use the iterative algorithm (if any)? In which situations would you prefer to use the divide-and-conquer algorithm (if any)?

When doing algorithm analysis, make sure to report the

- time and space complexity of the C++ standard library functions used in your code;
- time and space complexity of functions implicitly called like destructors; and

---

<sup>2</sup> Note that this function has four arguments.

- the meaning of the arguments of the functions used to express the time and space complexity of the algorithms analysed.

You can find examples of algorithm analysis in the [course compendium of exercises](#) (besides those presented in the lectures), e.g. exercise 14 and exercise 19.d.

Submit a **pdf file** with your written answers to the exercises in this part through Lisam until **April 9<sup>th</sup>, 20:00 sharp**. Remember that handwritten answers can be scanned. Do not forget to indicate the name plus LiU-id of each group member. Unreadable answers will be rejected.

You'll get feedback about this exercise until the end of week 15.

## Presenting lab and deadlines

The exercises in this lab are compulsory and you should demonstrate your code solutions during the lab session *Lab1 RE*. Read the instructions given in the [labs web page](#) and consult the course schedule. We also remind you that your code for the lab exercises cannot be sent by email to the staff.

Necessary requirements for approving your lab are given below.

- Use of global variables is **not** allowed, but global constants are accepted.
- The code must be readable, well-indented, and use good programming practices. Note that complicated functions and over-repeated code make code quite unreadable and prone to bugs.
- Compiler warnings that may affect badly the program execution are not accepted, though the code may pass the given tests. If have used CMake to create the project then the warning level is set, automatically.
  - If you have manually created a project in Visual Studio then you should [set the compiler's warning level](#) to Level14 (“W4”) and make sure the compiler is compiling C++17.
  - If you have manually created a project in Xcode then set the pedantic warning flag of the compiler (go to “*Build Settings*”) and make sure the compiler is compiling C++17.
- The iterative algorithm to solve stable partition problem must execute in linear time.
- The `std::stable_partition` algorithm cannot be used to solve the exercises in this lab.

Note that this lab has strict deadlines.

- Failing to have the code approved for exercises 1 and 2 on *Lab1 RE* session scheduled for your group implies that you cannot be awarded 3p for the labs in the course.
- Failing to submit your solutions for exercise 3 via Lisam until **April 9<sup>th</sup>, 20:00**, implies that you cannot be awarded 3p for the labs in the course. Note that only pdf files are accepted for exercise 3.

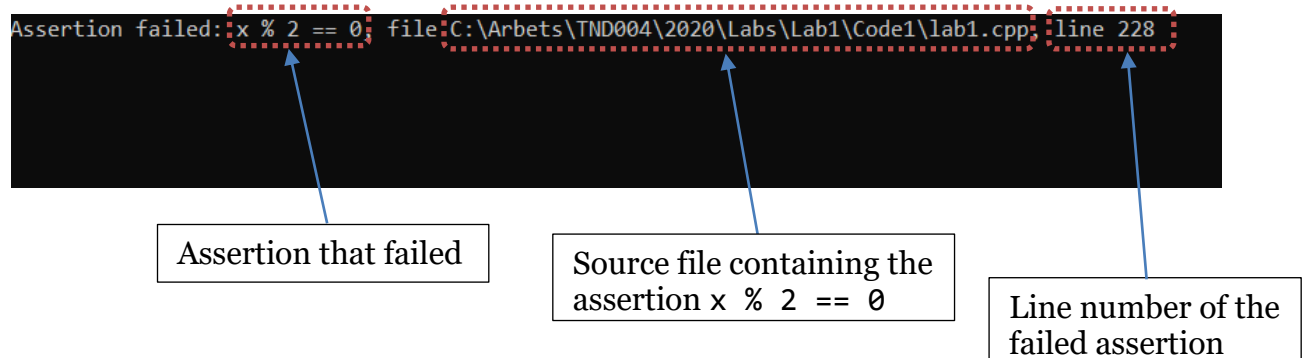
## Appendix

### Testing the code: assertions

In C/C++ programming language, assertions can be used to express that a given condition must be true, at a certain point in the code execution. For instance, consider the following code.

```
int main() {  
    int x{7};  
  
    /* Some code in between and let's say x  
       is accidentally changed to 9 */  
    x = 9;  
  
    // Programmer assumes x is even in rest of the code  
    assert(x % 2 == 0);  
  
    /* Rest of the code */  
}
```

The expression `assert(x % 2 == 0);` tests, during execution time, whether the condition `x % 2 == 0` evaluates to true. Then, if the evaluated condition is not true – an **assertion failure** –, then the program typically crashes and information about the failed assertion is shown. Note that a program stops executing at the first assertion that fails.



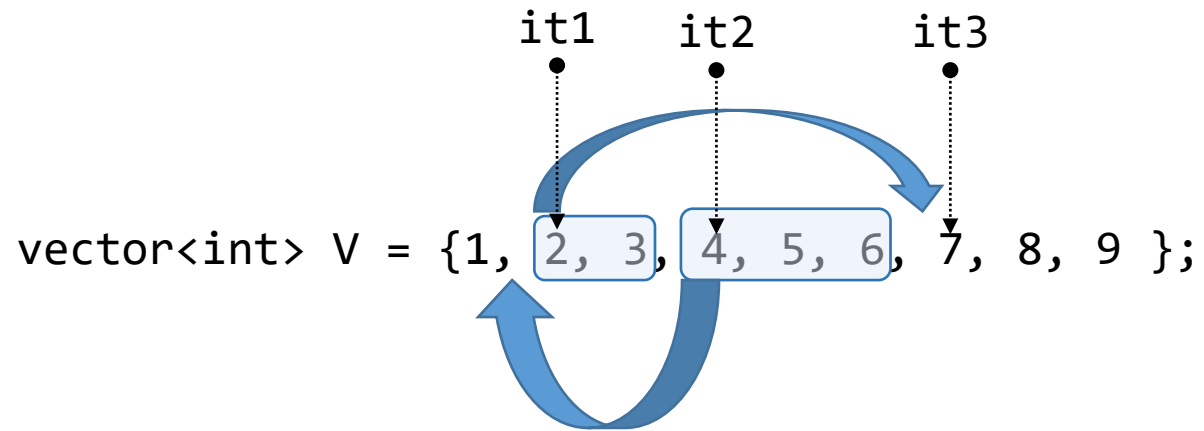
Assertions can be useful to identify bugs in a program and they are used in each test phase of the provided main function. Finally, to use assertions in C/C++ language, it is needed to include the library `cassert.h`.

### Stable partition: a divide-and-conquer algorithm

In the next pages you can find some figures.

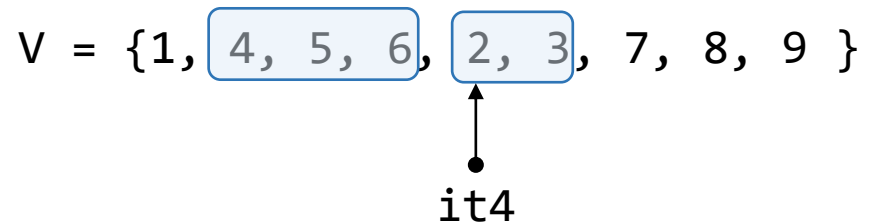
- The first figure illustrates how C++ [`std::rotate`](#) function works. Please, consult the online library documentation for more details. Recall that `std::rotate` is used in the implementation of the divide-and-conquer algorithm described in [exercise 2](#).
- The remaining figures illustrate how the divide-and-conquer algorithm for the stable partition problem of a sequence  $S$  works.

## std::rotate



```
vector<int>::iterator it1 = begin(V) + 1;  
vector<int>::iterator it2 = begin(V) + 3;  
vector<int>::iterator it3 = begin(V) + 6;
```

```
auto it4 = std::rotate(it1, it2, it3);
```




Note: if `it2` and `it3` point to the same item in the sequence then `std::rotate` does not modify `V`


# Divide-and-conquer: stable\_partition

```
void stable_partition(std::vector<int>& S, std::function<bool(int)> p)
{
    //call auxiliar function
    stable_partition(std::begin(S), std::end(S), p);
}

//Divide-and-conquer algorithm: stable-partition the sub-sequence starting at first and ending at last-1
//If there are items with property p then return an iterator to the end of the block containing the items with property p
//If there are no items with property p then return first
vector<int>::iterator stable_partition(std::vector<int>::iterator first, std::vector<int>::iterator last, std::function<bool(int)> p)
{
    //IMPLEMENT
}
```

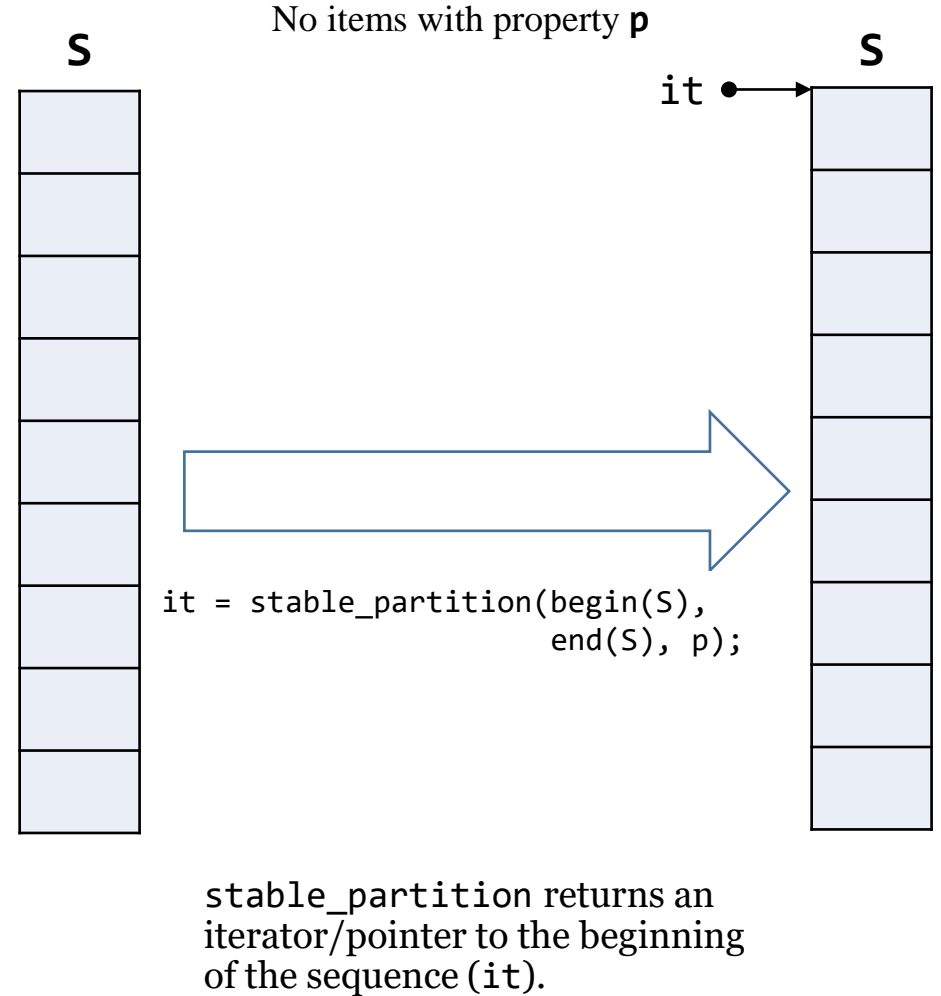
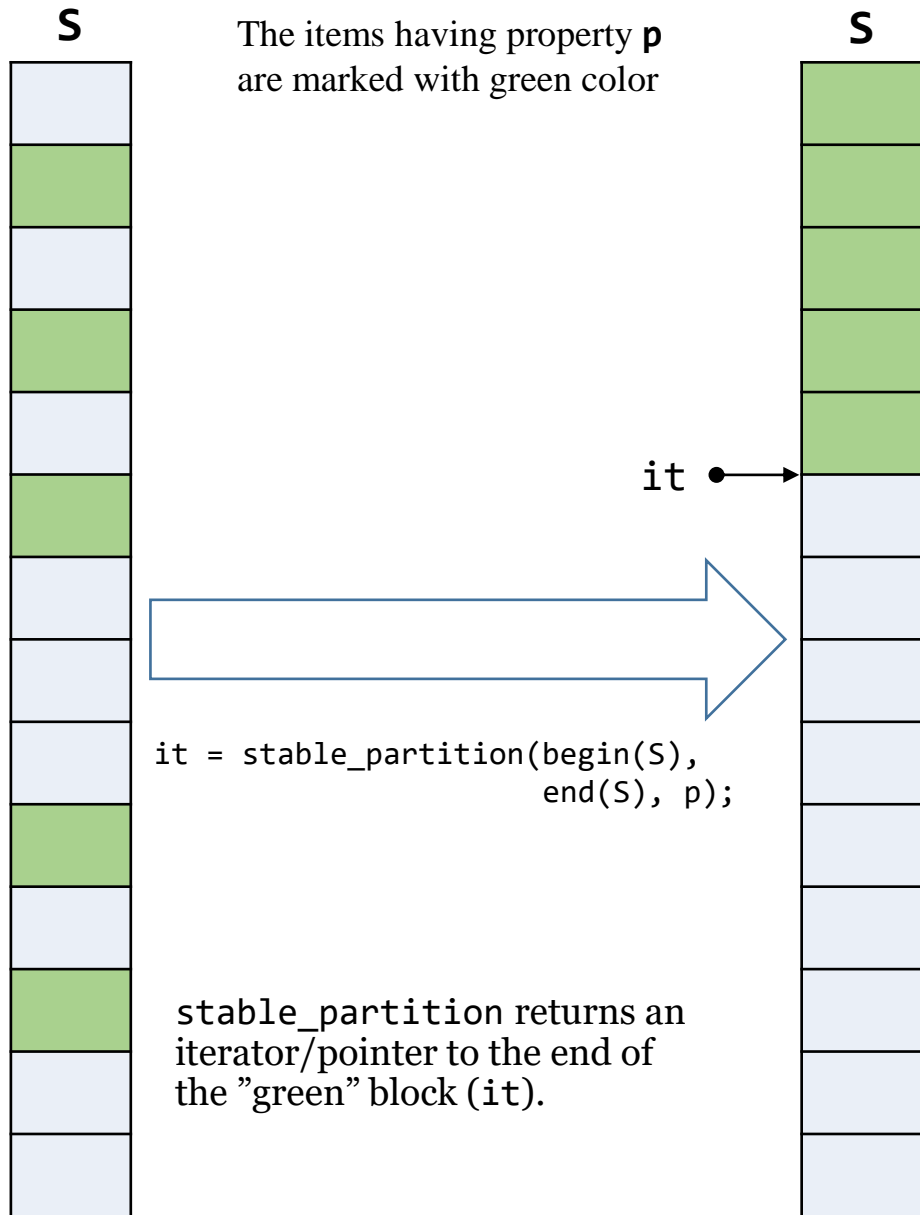


Return an iterator to the end of the block containing the items with property p (same as iterator to the first item in the given range not having property p)



Argument p represents a function with an int argument and returns a bool

# Divide-and-conquer: stable\_partition

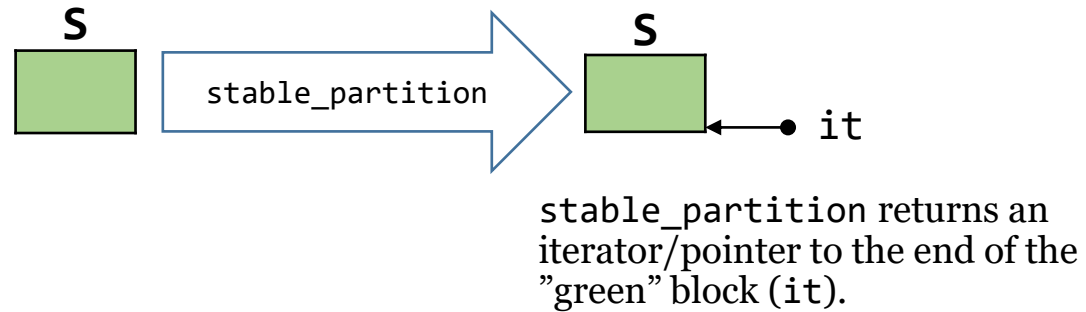




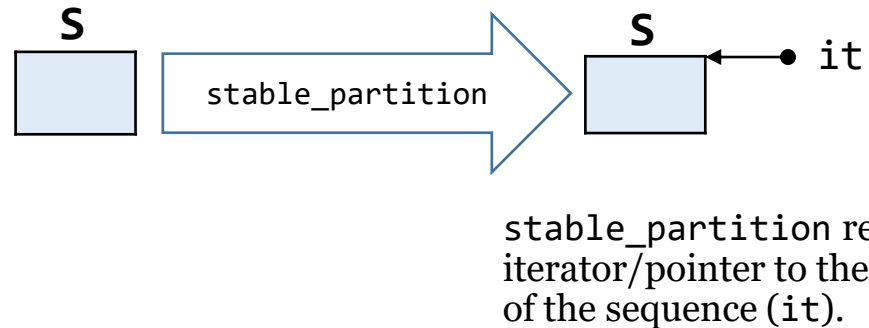
## Base-case: sequence with one item

```
it = stable_partition(begin(S), end(S), p);
```

Item has property **p** (marked with green color)



Item does not have property **p**



# Divide-and-conquer: stable\_partition

