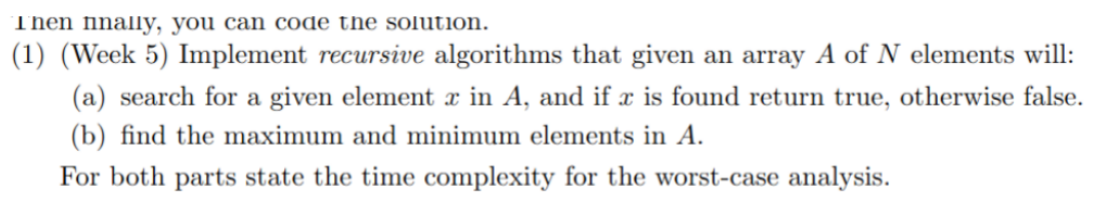
Assignment 3



Then finally, you can code the solution.

(1) (Week 5) Implement *recursive* algorithms that given an array A of N elements will:

(a) search for a given element x in A, and if x is found return true, otherwise false.

(b) find the maximum and minimum elements in A.

For both parts state the time complexity for the worst-case analysis.

How we implemented the code

We used a vector to implement the code since it has some useful functions we could use to make the code implementation easier(i.e.empty(), size()), while the vector still retains the functionality of an array.

More about what each function does can be read inside of the code file.

Time complexity analysis

a) Contains recursive function

For a), we have a function containing a recursive function which checks each vector until it either finds x or reaches the end.

In general for each work per step, we do one comparison A[idx]==x. Time complexity for thay is O(1).

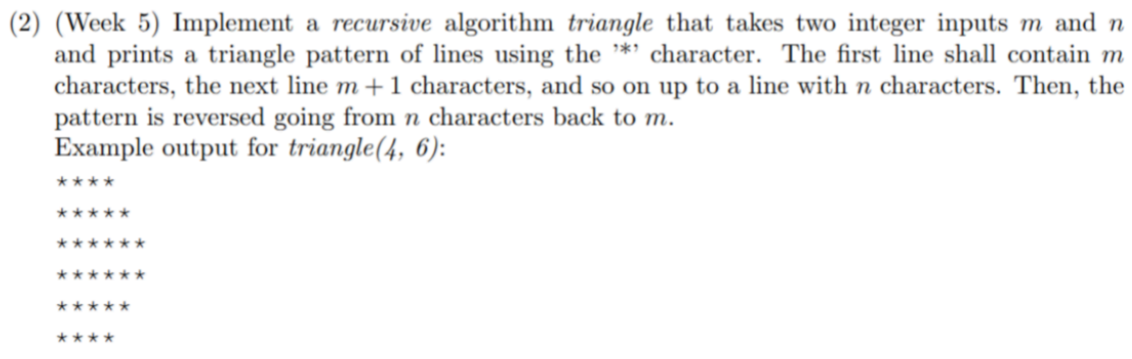
In the worst case we have to look through n elements (n=A.size()), therefore the time complexity of the contains recursive function is O(n).

b) Minmax_recursive.

The function traverses the vector once, carrying current_min and current_max, and updates them as needed if we find something smaller or larger than current.

In general for each work per step, we have two comparisons (A[idx]<min, A[idx]>max).

Regardless of values, we must inspect every element once, so we get time complexity of O(n).



(2) (Week 5) Implement a *recursive* algorithm *triangle* that takes two integer inputs m and n and prints a triangle pattern of lines using the '*' character. The first line shall contain m characters, the next line $m + 1$ characters, and so on up to a line with n characters. Then, the pattern is reversed going from n characters back to m.
Example output for *triangle(4, 6)*:

```
****
*****
******
******
*****
****
```

Solution is implemented in Exercise2.cpp.

Our implementations consists of two functions:

print_star():
This helper function prints k stars on one line via a for-loop. When it has printed k stars, it prints a newline; very simple.

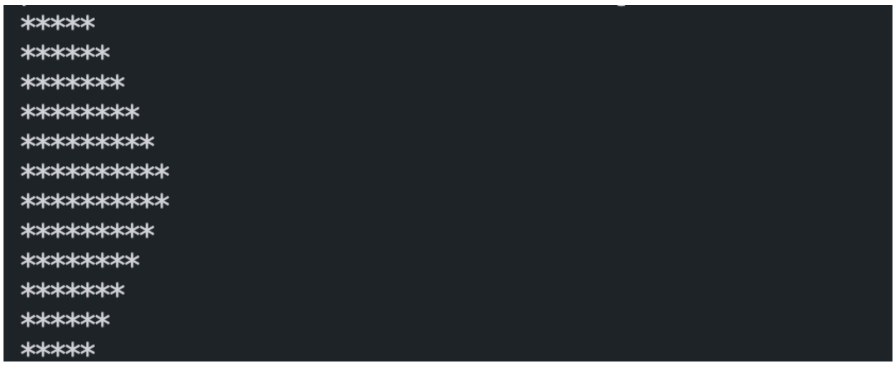
triangle():
The triangle function makes use of recursion. It takes a non-negative integer m and n, so that the first line prints m stars, the next line m +1 up to a line with two n stars. The pattern is then reversed going from n characters back to m. To print stars, we call our helper function print_star().

Base case:
Integers must be non-negative and if n = m, we print 2 lines of n stars.

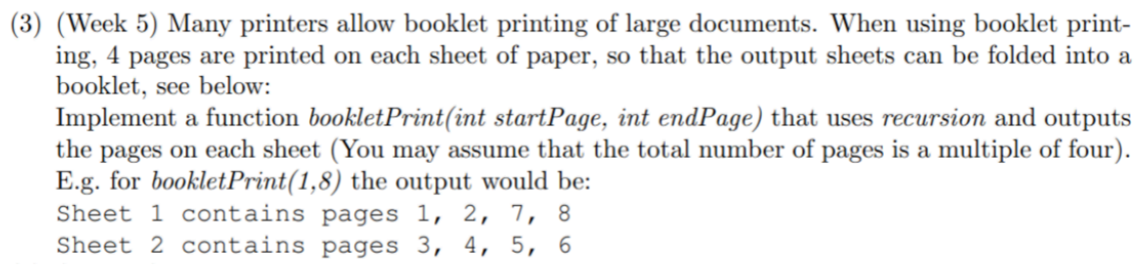
Recursive step:
Print m stars and call function again, once base case is hit, recurse back and print n-1 stars until m is reached.

The implementation is tested for various values of n and m (also negative) and works as intended.

Output example triangle(5, 10):



```
*****
******
*******
********
*********
**********
**********
*********
********
*******
******
*****
```



(3) (Week 5) Many printers allow booklet printing of large documents. When using booklet printing, 4 pages are printed on each sheet of paper, so that the output sheets can be folded into a booklet, see below:
Implement a function *bookletPrint(int startPage, int endPage)* that uses *recursion* and outputs the pages on each sheet (You may assume that the total number of pages is a multiple of four).
E.g. for *bookletPrint(1,8)* the output would be:

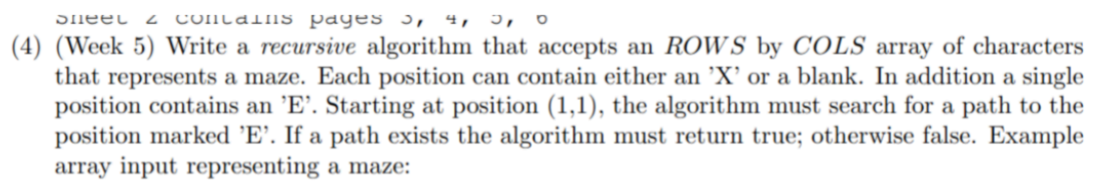
```
Sheet 1 contains pages 1, 2, 7, 8
Sheet 2 contains pages 3, 4, 5, 6
```

The code for this exercise can be found in the file Exercise3.cpp.

We interpret the task as printing a booklet using recursion. The idea is that a booklet is printed in such a way that each sheet shows the outermost two pages on the front and back. This means that in every recursive step, we print 4 pages (2 on the front, 2 on the back). Because of this, the total number of pages in the booklet must always be a multiple of 4 - otherwise, the printing cannot be done correctly.

To organize the solution, we split the implementation into three functions:

1. **printSheetLine**
   - This function has the simple task of printing one line of output (i.e., a single sheet of the booklet).
2. **bookletHelper**
   - This is the recursive function.
   - On each recursive call, it prints the current outermost 4 pages (the first and last two), and then it moves inward by updating the page numbers.
   - The recursion continues until all pages have been printed.
3. **bookletPrint**
   - This is the main function.
   - It checks that the input values for the start and end pages are valid, and ensures that the total number of pages is a multiple of 4.
   - If the input is valid, it calls the recursive bookletHelper function to carry out the printing.


Sheet 2 contains pages 3, 4, 5, 6

(4) (Week 5) Write a *recursive* algorithm that accepts an $ROWS$ by $COLS$ array of characters that represents a maze. Each position can contain either an 'X' or a blank. In addition a single position contains an 'E'. Starting at position (1,1), the algorithm must search for a path to the position marked 'E'. If a path exists the algorithm must return true; otherwise false. Example array input representing a maze:

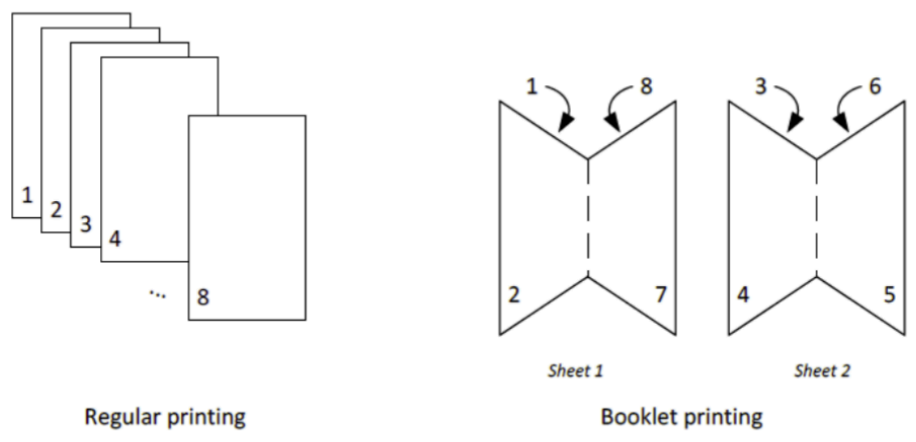
The code for this exercise can be found in the file Exercise4.cpp.

'X' = wall (blocked cell)
' ' (space) = free path
'E' = exit
Start position = (1,1) (top-left corner in the example, but actually second cell since index 0,0 is a wall).

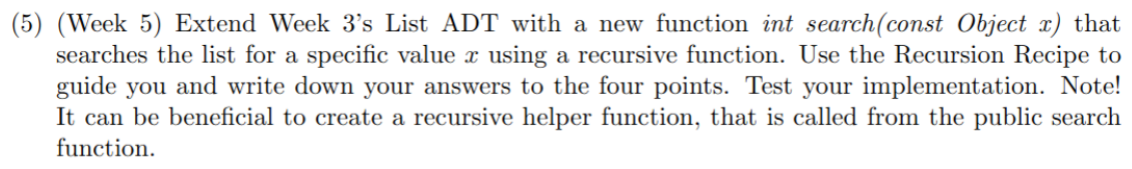
Problem to solve in this code is:
We must check if there exists any path from the starting cell (1,1) to the exit 'E'. The movement is usually up, down, left, right (not diagonals). The algorithm must be recursive.

Key ideas we implement in our code:
For this implementation we decided to make use a vector instead of an array.
If this cell is 'E' → return true (exit found).
If the cell is invalid ('X', out of bounds, or already visited) → return false.
Otherwise:

- Mark cell as visited (to avoid infinite loops).
- Recursively try moving up, down, left, right.
- If any recursive call returns true → propagate true upward.
- If all fail → return false.



```
char maze[ROWS][COLS] = {
    {'X','X','X','X','X'},
    {'X',' ',' ',' ','X'},
    {'X',' ','X',' ','X'},
    {'X',' ','X',' ','X'},
    {'X','E','X','X','X'}
};
```



(5) (Week 5) Extend Week 3's List ADT with a new function *int search(const Object x)* that searches the list for a specific value x using a recursive function. Use the Recursion Recipe to guide you and write down your answers to the four points. Test your implementation. Note! It can be beneficial to create a recursive helper function, that is called from the public search function.

The code for this exercise can be found in the folder Exercise5.

We included list ADT files (list.h and list.tpp) for this exercise and in the bottom of list.tpp we implemented two new functions:

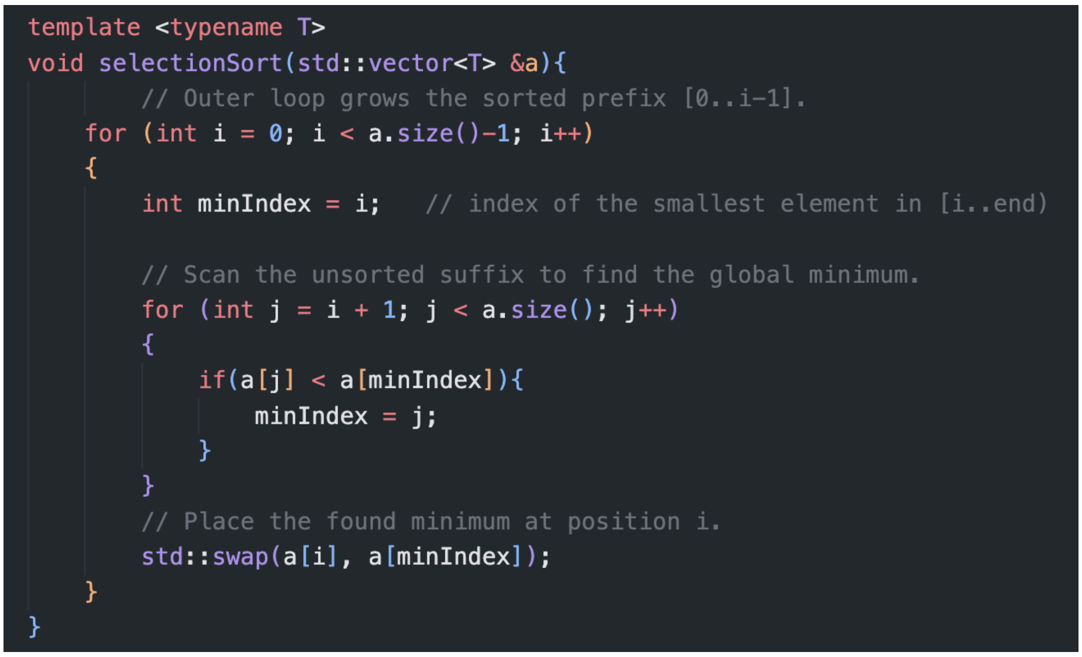
- search:
- search_from: (the recursive helper function)

Following our implementation of these functions, we tested their functionality in Exercise 5.cpp

(6) (Week 6) Consider the following strategy for sorting N numbers in an array A: find the smallest element of A and exchange it with the element in $A[0]$. Then find the second smallest element of A and exchange it with $A[1]$. Continue in this manner for the first $N-1$ elements of A. This algorithm is called the *selection sort*. Follow the give code examples of insertion sort (in particular look at `insertion_sort.h`, `sort.h` and `test_sort.cpp`) and create a new template implementation of the selection sort algorithm (i.e. it can received a vector with a generic element type just like the insertion sort). Don't forget to test the algorithm thoroughly and Give the best-case and worst-case running times for the resulting algorithm.

The implementation can be found in the file Exercise6.cpp.

We have expanded upon the given code examples of insertion sort and created a new generic template implementation of the selection sort algorithm:



```cpp
template <typename T>
void selectionSort(std::vector<T> &a){
        // Outer loop grows the sorted prefix [0..i-1].
    for (int i = 0; i < a.size()-1; i++)
    {
        int minIndex = i;   // index of the smallest element in [i..end)

        // Scan the unsorted suffix to find the global minimum.
        for (int j = i + 1; j < a.size(); j++)
        {
            if(a[j] < a[minIndex]){
                minIndex = j;
            }
        }
        // Place the found minimum at position i.
        std::swap(a[i], a[minIndex]);
    }
}
```

In our selectionSort algorithm, we repeatedly find the minimum element in the unsorted suffix [i..n-1] and swap it with the element at position i. To print the vector, we have implemented the simple print_vec function, which prints each element in the vector.

We have tested the algorithm for integers, chars, negative integers and strings, which can be found in our main() function.

Running-time:

- **Comparisons:** Scans the whole suffix each pass.
    - Exact: $\sum_{i=0}^{N-2}(N-1-i) = \frac{N(N-1)}{2}$.
    - ⇒ Best = Worst = O(N^2) time for comparisons.

- **Swaps (and data movement):**
    - Best case (already sorted): 0 swaps (no self-swaps).

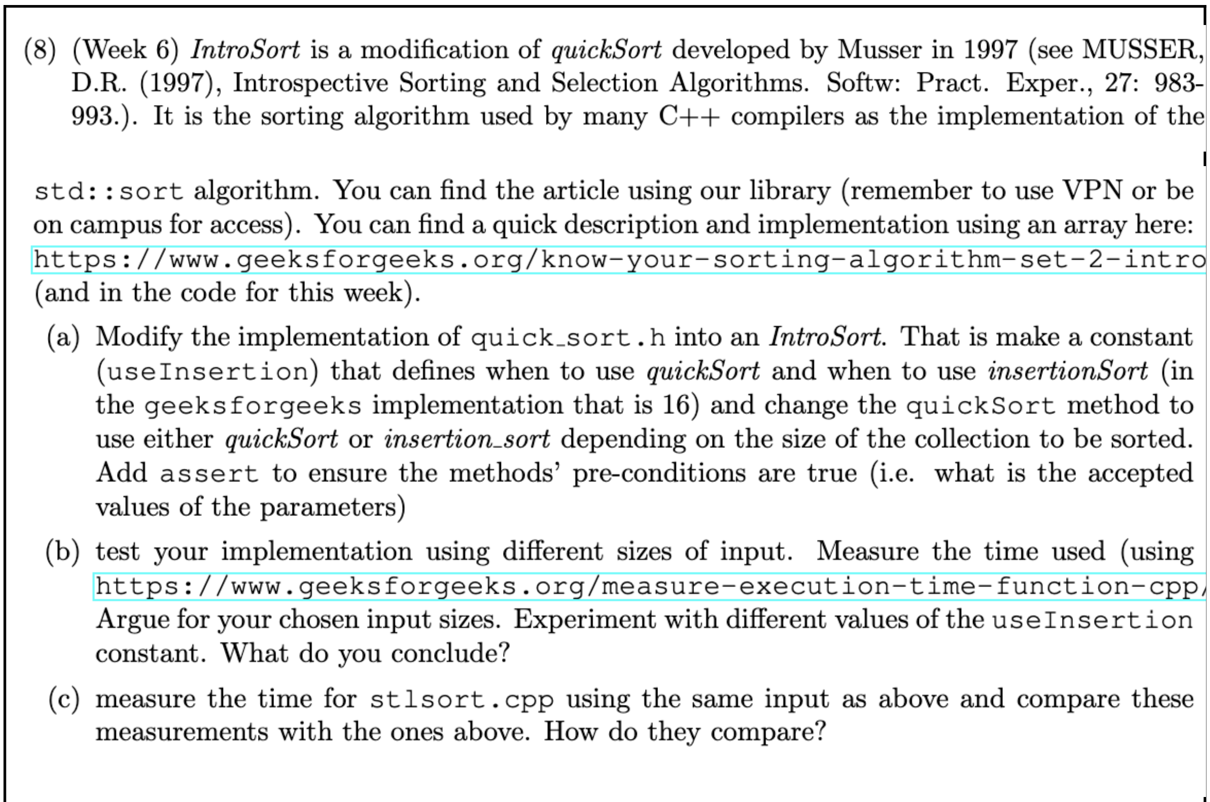
- Worst case: $\boxed{N-1}$ swaps (one per outer pass).

Stability: Not stable because of O(N^2)

(7) (Week 6) Now consider a strategy to sort an array A of k integers in the range $\{0,\ldots,k\}$, called *counting sort*. You can find a description at https://en.wikipedia.org/wiki/Counting_sort. Implement the algorithm in C++. The algorithm takes a vector of int's. Argue that your implementation runs in time $\mathcal{O}(N)$ (remember to define what N is). What is the space complexity?

N = Number of items in the array (input size in elements).
K = Largest key value (range: values are in [0..K])

Worst case: we scan the data a few times (find max, count, place, copy) and walk the value range once (prefix). That's "items" + "range," so time grows with both. Space also grows with both: one output array (size N) and one count array (size up to K). **Conclusion:** T(n) worst case = **O(N+K)**, space worst case = **O(N+K)**.



(8) (Week 6) *IntroSort* is a modification of *quickSort* developed by Musser in 1997 (see MUSSER, D.R. (1997), Introspective Sorting and Selection Algorithms. Softw: Pract. Exper., 27: 983-993.). It is the sorting algorithm used by many C++ compilers as the implementation of the

`std::sort` algorithm. You can find the article using our library (remember to use VPN or be on campus for access). You can find a quick description and implementation using an array here: `https://www.geeksforgeeks.org/know-your-sorting-algorithm-set-2-intro` (and in the code for this week).

(a) Modify the implementation of `quick_sort.h` into an *IntroSort*. That is make a constant (`useInsertion`) that defines when to use *quickSort* and when to use *insertionSort* (in the `geeksforgeeks` implementation that is 16) and change the `quickSort` method to use either *quickSort* or *insertion_sort* depending on the size of the collection to be sorted. Add `assert` to ensure the methods' pre-conditions are true (i.e. what is the accepted values of the parameters)

(b) test your implementation using different sizes of input. Measure the time used (using `https://www.geeksforgeeks.org/measure-execution-time-function-cpp/` Argue for your chosen input sizes. Experiment with different values of the `useInsertion` constant. What do you conclude?

(c) measure the time for `stlsort.cpp` using the same input as above and compare these measurements with the ones above. How do they compare?

a)

The code for this exercise can be found in the folder Exercise8.

Here we have included the provided header files quick_sort.h and insertion_sort.h. We have then implemented the header file intro_sort.h, which makes use of the methods in both quick_sort.h and insertion_sort.h.

Implementation of intro_sort

The implementation of intro_sort combines the efficiency of quicksort with the simplicity of insertion sort. For larger subarrays, the function applies the partitioning method from quick_sort.h to divide the array around a pivot and then recursively sorts the two halves. For smaller subarrays, below a chosen cutoff "kInsertionCutoff" which in this example is 16 (research has found this to be the most effective), the function switches to the insertion sort method from insertion_sort.h, which is faster on small inputs due to lower overhead.

The algorithm also includes base cases to stop the recursion. If the subarray has zero or one element, it is already sorted, and the function simply returns. If the subarray size is below the cutoff, insertion sort is used directly without further partitioning. This ensures that the algorithm does not recurse indefinitely and that small sections are handled efficiently.

b)
The code for this exercise can be found in the folder Exercise8 -> file test_intro_sort.cpp

The “useInsertion constant” determines when the program changes from using quicksort to insertion sort. Quicksort is a sorting method that is efficient for an array with many elements, while insertion is more efficient for smaller arrays. The theoretical best value for changing sorting algorithms in an array is the value 16, that means when it only has 16 elements left to sort, it changes from quicksort to the insertion sort method. In our tests we choose values near the insertion constant, to determine time it takes sort in nanoseconds. As the constant gets larger the more of the array gets sorted via insertion method and therefore takes exceptionally longer.

useInsertion value	Input value	Time taken
16	10	1.291 nanoseconds
	16	1.958 nanoseconds
	20	2.000 nanoseconds
	250	29.125 nanoseconds
	256	28.084 nanoseconds
	260	27.083 nanoseconds
	1000	108.291 nanoseconds
	1024	116.209 nanoseconds
	1100	115.333 nanoseconds
256	10	1.458 nanoseconds
	16	1.833 nanoseconds
	20	3.083 nanoseconds
	250	291.500 nanoseconds
	256	298.666 nanoseconds
	260	238.667 nanoseconds
	1000	730.000 nanoseconds
	1024	969.208 nanoseconds
	1100	873.583 nanoseconds
1024	10	1.416 nanoseconds
	16	1.958 nanoseconds
	20	3.000 nanoseconds

	250	281.667 nanoseconds
	256	296.792 nanoseconds
	260	914.167 nanoseconds
	1000	4.531.167 nanoseconds
	1024	12.731.833 nanoseconds
	1100	3.991.584 nanoseconds

c)
Below is a table showing the running times measured using the chrono library and method from geekbygeeks.com (mentioned in assignment text source). In the second column we have noted down the time for stlsort and to the right the time for insert_sort:

Input value	Time taken (stl)	Time taken (insert)
10	375 nanoseconds	1.291 nanoseconds
16	667 nanoseconds	1.958 nanoseconds
20	500 nanoseconds	2.000 nanoseconds
250	6875 nanoseconds	29.125 nanoseconds
256	6875 nanoseconds	28.084 nanoseconds
260	6833 nanoseconds	27.083 nanoseconds
1000	26.708 nanoseconds	108.291 nanoseconds
1024	26.833 nanoseconds	116.209 nanoseconds
1100	29.167 nanoseconds	115.333 nanoseconds

Looking at our tables, we notice that running STL sort for the same input values as we did for intro sort, we notice a trend that STL sort is about 4 times as fast as intro sort, making it highly efficient.