Algorithms & Data Structures
Exercise Sheets

Weeks 5-6: Recursion and Array Sorting

Exercises

(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.

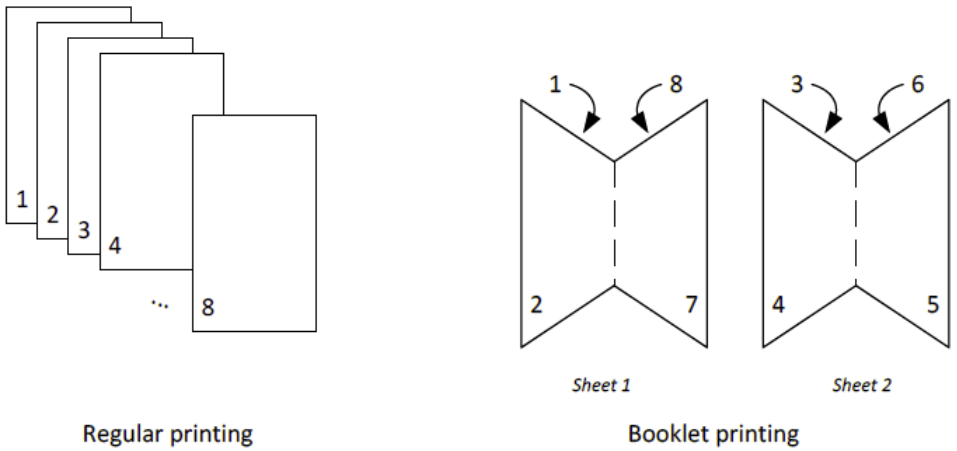
(b) find the maximum and minimum elements in A.

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

(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)*:

****
*****
******
******
*****
****

(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 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
```

(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:

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

(5) (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. Extend the `test_sort.h` file with a template implementation of this algorithm (i.e. it can take a vector with a generic element type), which is known as *selection sort*. Give the best-case and worst-case running times for the resulting algorithm.

(6) (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 $\Theta(N)$ (remember to define what N is). What is the space complexity?

(7) (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-introsort-cs-sorting-w` (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?