DOA Week 5 - 6 Exercises

Recursion & Array Sorting

Software Engineering
Aarhus University

10. October 2023

Studie Nr
Simon Dybdahl Damulog Andersen: 202204902
Max Glintborg Dall: 202208488
Emil Lydersen: 202207059

Contents

Exercise 1 1
Exercise 2 3
Exercise 3 4
Exercise 4 6
Exercise 5 8
Exercise 6 10
Exercise 7 12
Appendix 15

Exercise 1

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.

Following is the implementation for both assignments (a) and (b):

```
#include <iostream>
#include <algorithm>

template <typename T>
bool recursive_search(const T *arr, int size, const T &x)
{
    if (size == 0)
    {
        return false;
    }
    if (arr[size - 1] == x)
    {
        return true;
    }
    return recursive_search(arr, size - 1, x);
}

template <typename T>
void recursive_minmax(const T *arr, int size, T &min, T &max)
{
    if (size == 1)
    {
        min = max = arr[0];
        return;
    }
    recursive_minmax(arr, size - 1, min, max);
    min = std::min(min, arr[size - 1]);
    max = std::max(max, arr[size - 1]);
}

int main()
{
    int arr[] = {3, 5, 2, 8, 1, 9, 4, 7, 6};
    int size = sizeof(arr) / sizeof(arr[0]); // divide the sizes of arr to get real int.

    // search for element 5
    if (recursive_search(arr, size, 5))
    {
        std::cout << "Element 5 found in the array.\n";
    }
    else
    {
        std::cout << "Element 5 not found in the array.\n";
    }

    // find min and max elements
    int min, max;
    recursive_minmax(arr, size, min, max);
    std::cout << "Minimum element: " << min << "\n";
    std::cout << "Maximum element: " << max << "\n";

    return 0;
}
```

For the first assignment the function `recursive_search`, was implemented. The name is pretty self-explanatory, but to put it into simpler terms - the function uses itself to progress through each iteration of steps until it has reached an ending condition, or went through the whole array.

```
Element 5 found in the array.
Minimum element: 1
Maximum element: 9
```

Figure 0.1
Output of the program.

To find the minimum and maximum values the `<algorithm>` library was used as seen in line 26 & 27. The recursion is basically, just a constant decrement of the array. The function recursively divides the array into smaller parts and computes the minimum and maximum values. As it returns from the recursive calls, it updates the minimum and maximum values accordingly, eventually giving you the minimum and maximum values for the entire array.

For the `recursive_search` funcion's worst-case scenario, the function has to traverse the entire array, element by element, until it either finds the target element or reaches the end of the array. Therefore, the worst-case time complexity is O(N), where N is the size of the array.

For the `recursive_minmax` function's worst-case scenario, the function recursively divides the array into halves until it reaches the base case (size == 1). The key insight here is that each recursive call reduces the problem size by half. Therefore, the worst-case time complexity is O(N), where N is the size of the array.

Both algorithms have a worst-case time complexity of O(N), where N is the size of the input array. This means that the time it takes to execute these algorithms grows linearly with the size of the array.

Exercise 2

(2) 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.

Following code is the implementation, made from above assignment's specification:

```cpp
#include <iostream>
using namespace std;

void triangle(int m, int n)
{
    if (m > n)
    {
        return; // Base case: stop recursion when m is greater than n
    }

    for (int i = 0; i < m; i++)
    {
        cout << "*";
    }
    cout << endl;

    triangle(m + 1, n); // Recursively call triangle with m+1

    for (int i = 0; i < m; i++)
    {
        cout << "*";
    }
    cout << endl;
}

int main()
{
    triangle(4, 6);
    return 0;
}
```

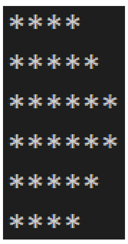

Figure 0.2
Output of above program.

In this Implementation, the "Triangle" function Recursively calls itself with a bigger value (n+1) until it reaches the base case where $m > n$.

Exercise 3

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

Following code is the implementation, made from above assignment's specification:

```cpp
#include <iostream>
#include <set>

void bookletPrint(int startPage, int endPage, int sheetNumber = 1)
{
    if (startPage > endPage)
    {
        return;
    }

    std::set<int> pageSet;

    // Add the pages to the set in the specified order
    pageSet.insert(startPage);
    pageSet.insert(endPage);

    if (startPage != endPage)
    {
        pageSet.insert(endPage - 1);
        pageSet.insert(startPage + 1);
    }

    // Print the sheet details
    std::cout << "Sheet " << sheetNumber << " contains pages ";
    for (int page : pageSet)
    {
        std::cout << page << ", ";
    }
    std::cout << std::endl;

    bookletPrint(startPage + 2, endPage - 2, sheetNumber++);
}

int main()
{
    // Example usage
    bookletPrint(1, 8);
    std::cout << std::endl;
    bookletPrint(1, 40);

    return 0;
}
```

For this exercise the sorting capability of `<set>` was utilized. This ensured that the output would always be sorted, as shown in the example output given in the assignment specification. Since the logic applied, in an other ADT than set, would have given the right output, but in a messy order. Alternatively a sorting algorithm, could have been implemented, but this implementation seemed more elegant.

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

Sheet 1 contains pages 1, 2, 39, 40,
Sheet 1 contains pages 3, 4, 37, 38,
Sheet 1 contains pages 5, 6, 35, 36,
Sheet 1 contains pages 7, 8, 33, 34,
Sheet 1 contains pages 9, 10, 31, 32,
Sheet 1 contains pages 11, 12, 29, 30,
Sheet 1 contains pages 13, 14, 27, 28,
Sheet 1 contains pages 15, 16, 25, 26,
Sheet 1 contains pages 17, 18, 23, 24,
Sheet 1 contains pages 19, 20, 21, 22,
```

Figure 0.3

Output from program, with exercise example and expanded size.

Exercise 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’} };

```cpp
#include <iostream>

const int ROWS = 5;
const int COLS = 5;

bool searchMaze(char maze[ROWS][COLS], int row, int col)
{
    // Check if we have reached the end of the maze
    if (maze[row][col] == 'E')
    {
        return true;
    }

    // Check if the current position is a wall or has already been visited
    if (maze[row][col] == 'X' || maze[row][col] == '.')
    {
        return false;
    }

    // Mark the current position as visited
    maze[row][col] = '.';

    // Recursively search in all four directions
    if (searchMaze(maze, row - 1, col) || // Up
        searchMaze(maze, row + 1, col) || // Down
        searchMaze(maze, row, col - 1) || // Left
        searchMaze(maze, row, col + 1)) // Right
    {
        return true;
    }

    // If no path was found, mark the current position as unvisited
    maze[row][col] = ' ';

    return false;
}
```

Above is the implementation of the recursive maze algorithm, as specified in the assignment’s objectives. The criteria was interpreted as $'\ ' = traversable$, $'X' = wall$, $'E' = exit$. One functionality that was added, was the implementation of the ’.’, such that it was possible to see where the algorithm traverses, which made debugging a whole lot easier, and gave visual confirmation that the algorithm is working as intended.

```
1  int main()
2  {
3      char maze[ROWS][COLS] = {
4          {'X', 'X', 'X', 'X', 'X'},
5          {'X', ' ', ' ', ' ', 'X'},
6          {'X', ' ', 'X', ' ', 'X'},
7          {'X', ' ', 'X', ' ', 'X'},
8          {'X', 'E', 'X', 'X', 'X'}};
9  
10     if (searchMaze(maze, 1, 1)) // zero indexed
11     {
12         std::cout << "Path found!" << std::endl;
13     }
14     else
15     {
16         std::cout << "No path found." << std::endl;
17     }
18 
19     // Print the maze
20     for (int i = 0; i < ROWS; i++)
21     {
22         for (int j = 0; j < COLS; j++)
23         {
24             std::cout << maze[i][j] << " ";
25         }
26         std::cout << std::endl;
27     }
28 
29     return 0;
30 }
```

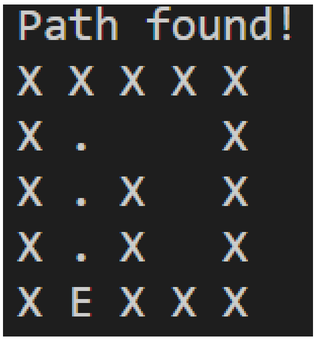

Figure 0.4
Output from above main().

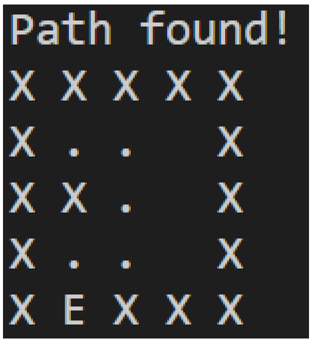

Figure 0.5
Illustrating how algo behaves.

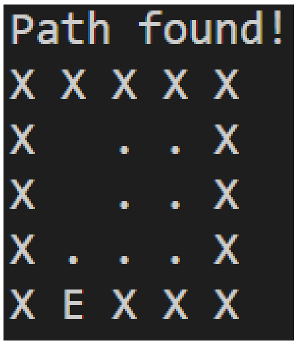

Figure 0.6
Starting (3,3). Algo's behaviour.

The included figures clearly illustrate that the algorithm was build with starting position (1,1)[zero-indexed] in mind. As we see the pretty ineffective route it takes in figure 6, if starting on the opposite corner of the maze. As a closing note, we thought it was more beneficial to illustrate how the algorithm moves, but it will display "No path found!", if a route is not possible, i.e. `searchMaze` returns false, as seen in line 16 of the `int main()` source code.

Exercise 5

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 minus 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

Selection sort is a sorting algorithm that works by taking the min or max element in an array and sorts it accordingly for each of their respective sizes. The algorithm repeatedly selects the minimum (or maximum) element from the unsorted array and swaps it with the first element of the unsorted part. It will repeat this process until the remaining unsorted part of the given array is sorted.

Below is a visual example of how selection sort works:

5	**2**	9	3	6

Look through & find lowest element, then swap it with first index.

2	5	9	**3**	6

This process happens again & again, index +1 for each iteration

2	3	**9**	**5**	**6**

... This process repeats itself until selection sort is finished

2	3	5	6	9

Now that selection sort is a little bit clearer the following code is presented:

```
template <typename AnyType>
void selectionSort(vector<AnyType> &a) // sorts a vector using selection sort
{
    for (int i = 0; i < a.size(); ++i) // for each element in the vector
    {
        int min = i; // assume the current element is the smallest
        for (int j = i + 1; j < a.size(); ++j)
        {
            if (a[j] < a[min]) // if the current element is smaller than the smallest element
            {
                min = j;
            }
        }
        swap(a[i], a[min]); // swap the smallest element with the current element
    }
}
```

In the above mentioned code, the `selectionSort` function is made. For each element in the vector, `selectionSort` will take the current element and swap it with the smallest element. It will keep doing this, until the vector has been sorted.

A for loop is made on line 4, that states for each element in the vector, it will assume the current element is the min element. if the current element is smaller than the smallest element it will swap the smallest element with the current element, thus slowly sorting the vector one element at a time.

To analyse the best & worst case running times, a test small test was conceived, where the best and average case, i.e. an already sorted data set and a randomly sorted data set is benchmarked using the `<chrono>` lib. Three test were taken and we saw a small performance difference, when the set was already pre-sorted. The last test the data set was doubled from $100k \xrightarrow{2x} 200k$.

Test	Sorted(ms)	Random(ms)	diff(%)
1	18741	19155	-2.16
2	18768	19171	-2.10
3	74563	76636	-2.7

Table 0.1
Overblik over tests af running times.

As shown there is a small difference, which can be explained by the fact that there is taken several less swap operations, since the data set is already sorted. The time-complexity is of course still $O(n)^2$, attributed by the nested loop. In conclusion though it is shown how the data set can slightly affect the running time.

Exercise 6

(Week 6) Now consider a strategy to sort an array A of k integers in the range 0, ... , 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 O(N) (remember to define what N is). What is the space complexity?

To better understand counting sort, a visualization was made, where a random unsorted input array goes through the general logic applied with the technique.

Input Array:

4	2	2	8	3	3	1

Count each occurrence, with size of max element(8) + 1:

0	1	2	2	1	0	0	0	1

Accumulate values:

0	1	3	5	6	6	6	6	7

-1 index, such that we get where each value starts in the result array:

0	0	1	3	5	6	6	6	6

Result array: (Highest order of duplicates apply)

1	2	2	3	3	4	8

For the three arrays of size max element+1 it is important to notice that each index represents its given numeric value, from left to right in rising order. The final counting array's values represent the index for each of the original array's values.

Now that counting sort is outlined, the next page will include the source code built from the assignments objectives.

Here is the code for Exercise_6

```
void counting_sort(vector<int> &A) // sorts a vector using counting sort
{
    int max_element = A[0];
    for (int i = 1; i < A.size(); i++) // find the maximum element in the vector
    {
        if (A[i] > max_element)
        {
            max_element = A[i]; // update the maximum element if the current element is larger than the
                maximum element
        }
    }
    vector<int> counting(max_element + 1, 0); // create a vector of size max_element + 1 and initialize
         all elements to 0
    for (int i = 0; i < A.size(); i++)    // count the number of occurrences of each element
    {
        counting[A[i]]++;
    }
    for (int i = 1; i < counting.size(); i++)
    {
        counting[i] += counting[i - 1]; // update the number of occurrences of each element to the number
             of elements less than or equal to it
    }
    vector<int> result(A.size());
    for (int i = A.size() - 1; i >= 0; i--)
    {
        result[counting[A[i]] - 1] = A[i]; // place each element in its correct position in the result
             vector
        counting[A[i]]--;
    }
    A = result;
}
```

In main, the following code is added: `counting_sort(a);` When the code is run, the checksort algorithm starts, meaning that counting_sort function works as intended. The space complexity of the above mentioned code, can be found be analyzing it:

Counting sort has a time complexity of O(n + k) and a space complexity of O(n + k), where n is the number of items to be sorted, and k is the number of distinct values among those items.

The algorithm involves two linear iterations through the input items, one for counting and one for filling the output array, making them both O(n) operations. Additionally, there's a linear iteration through the counts array to build the nextIndex array, which is O(k).

Counting sort uses three additional arrays: counts, nextIndex, and the output array. The counts and nextIndex arrays are both sized based on k (the number of distinct values), and the output array is sized based on n (the number of items). Therefore, the space complexity is O(n + k).

In practice, when the number of distinct values (k) is not significantly different from the number of items (n), counting sort is often referred to as having a simplified linear time and space complexity of O(n). This simplification occurs because the k term becomes proportionally similar to n, and the overall complexity is effectively O(n). Like in the source code.

Exercise 7

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?

For the sub-assignment a), the following code was added to the quick_sort.h file:

```cpp
// at top of file (line 6)
const int useInsertion = 16;

// where quicksort was originally placed (from line 52)
template <typename Comparable>
void quickSort(vector<Comparable> &a, int left, int right)
{
	if (right - left <= useInsertion)
	{
		insertionSort(a);
	}
	else
	{
		int i = partition(a, left, right);
		quickSort(a, left, i - 1);   // Sort small elements
		quickSort(a, i + 1, right); // Sort large elements
	}
}
```

Pretty simple, but this is what was understood from the assignment specification. Regarding the assert, we weren't really sure what was meant, but since there were no issues, it was decided to exclude it.

For the sub assignment b) the performance of the `quick_sort` function was measured with input sizes 100k, 500k & 1m, they were each tested over the range of 16-65536 with each step being x2 the previous step starting from 16.

Input	Time(ms)	useInsertion	Percentage	%/diff from last
1000000	665005	16		
1000000	440729	32	33,73	33,73
1000000	301465	64	54,67	31,60
1000000	238717	128	64,10	20,81
1000000	186150	256	72,01	22,02
1000000	187782	512	71,76	-0,88
1000000	160803	1024	75,82	14,37
1000000	163160	2048	75,46	-1,47
1000000	151387	4096	77,24	7,22
1000000	148062	8192	77,74	2,20
1000000	144949	16348	78,20	2,10
1000000	158516	32768	76,16	-9,36
1000000	142432	65536	78,58	10,15
500000	188409	16		
500000	132907	32	29,46	29,46
500000	96115	64	48,99	27,68
500000	77277	128	58,98	19,60
500000	65861	256	65,04	14,77
500000	72099	512	61,73	-9,47
500000	57844	1024	69,30	19,77
500000	58509	2048	68,95	-1,15
500000	56726	4096	69,89	3,05
500000	63431	8192	66,33	-11,82
500000	57326	16348	69,57	9,62
500000	64502	32768	65,76	-12,52
500000	58212	65536	69,10	9,75
100000	9755	16		
100000	8152	32	16,43	16,43
100000	4954	64	49,22	39,23
100000	5090	128	47,82	-2,75
100000	3664	256	62,44	28,02
100000	5244	512	46,24	-43,12
100000	3567	1024	63,43	31,98
100000	3484	2048	64,28	2,33
100000	4821	4096	50,58	-38,38
100000	5219	8192	46,50	-8,26
100000	7914	16348	18,87	-51,64
100000	4329	32768	55,62	45,30
100000	4586	65536	52,99	-5,94

Figure 0.7

Data collection with different useIteration values.

As can be read in the tables of our data collection a clear trend can be seen in the beginning, where running time rapidly increases as the variable `useInsertion` is increased. This is clear until 256 for 1m & 500k and is less prevalent for 100k, where the rapid increase already slows down at 64. This may be attributed to the fact that we are dealing with a smaller data set, and therefore the work of the insertion algorithm kicks in way earlier, than in the others, which means less data variation in shorter time.

While taking these test the hope was to show that as the `useInsertion` variable increased to a mid range size, that the running time would decrease, as we managed to somewhat show in the data. However as the variable ran into bigger size, the beforehand assessment was that the time would start to rise with $O(n^2)$ as the insertion sort made more and more of the work, but for some reason the result was drastically different and it instead began to become very chaotic in the running times. Our best guess for the cause of this is that it is something to do with where it chooses to place the pivot points in the `quick_sort` function, and this became more and more important as the variable was increased, since it may choose a spot right next to the threshold, or was less, making the times very random as it divides into the smaller subarrays.

Input	Run-time(ms)	diff(%)
100k	20	
500k	121	505
1m	269	122

Table 0.2
Overblik over tests af running times.

The `stl::sort` was obviously way faster than our implementation, in part because the median of medians for the partitions was not implemented in ours, and the fact that many other language under the hood optimizations was made. Some of these methods will probably be covered in the course over the coming weeks. And obviously shows that there is a massive room for improvement in our current level & understanding of algorithms.

Appendix

Included source code in folder: "Week_5_6_RecursionAndArraySorting.zip"

This can also be found on GitHub:

https://github.com/SjioGG/DOA-Exercises/tree/main/Week_5_6_RecursionAndArraySorting