Informatics II, Spring 2024, Solution Exercise 5

Publication of exercise: March 18, 2024
Publication of solution: March 25, 2024
Exercise classes: April 8 - April 12, 2024

Learning Goal

- Understanding heaps and how algorithms can operate on them.
- Knowing how to implement quick sort and where this algorithm has difficulties.
- Being able to apply learned algorithms to unknown problems.

Task 1 [Medium]

1.1: In the lecture you learned what the criteria for a heap are. Use this knowledge to write the function `controlMaxHeap(int array[], int currentIndex, int length)` in C. The goal of this function is to return -1 if the array fulfills all the criteria that a max heap must fulfill. Otherwise the index of an element not fulfilling the property should be returned.

Examples:

- $[5, 5, 3, 2, 1, 3]$ should return -1, since no property is violated.
- $[5, 5, 3, 2, 1, 4]$ should return 2, because 4 can't be a child of 3. Since 2 is the index of the element with value 3 the number 2 should be returned.
- $[5, 5, 3, 2, 1, 6]$ should return 2, even if there is a conflict between the number 5 at index 0 and the number 6 at index 5, the direct violation of a property is the same as in the previous example.

1.2: Your task is to implement the function `heapify(int array[], int currentIndex, int length)` in C. The function should restore the heap property for the desired index and propagate downwards if necessary.

Example:

- When calling heapify with the array [4, 10, 3, 5, 1] of length 5 and the index 0, the function should rearrange the array to [10, 5, 3, 4, 1].

1.3: Suppose we have an array in which exactly one element does not fulfill the heap property. Write the function `fixHeap(int array[], int length)` in C which ensures that the array is a heap again. Use the algorithms from the previous sub tasks to find the element that causes difficulties and fix it.

```c
#include <stdio.h>

int leftChild(int index) {
    return index * 2 + 1;
}

int rightChild(int index) {
    return index * 2 + 2;
}

void swap(int array[], int index1, int index2) {
    int tmp = array[index2];
    array[index2] = array[index1];
    array[index1] = tmp;
}

int controlMaxHeap(int array[], int currentIndex, int length) {
    int leftIndex = leftChild(currentIndex);
    if (leftIndex < length) {
        if (array[currentIndex] < array[leftIndex]) {
            return currentIndex;
        }
        int leftViolation = controlMaxHeap(array, leftIndex, length);
        if (leftViolation != -1) {
            return leftViolation;
        }
    }
    int rightIndex = rightChild(currentIndex);
    if (rightIndex < length) {
        if (array[currentIndex] < array[rightIndex]) {
            return currentIndex;
        }
        int rightViolation = controlMaxHeap(array, rightIndex, length);
        if (rightViolation != -1) {
            return rightViolation;
        }
    }
    return -1;
}

void heapify(int array[], int currentIndex, int length) {
    int leftIndex = leftChild(currentIndex);
    int rightIndex = rightChild(currentIndex);
    int indexLargestElement = currentIndex;
    if (leftIndex < length) {
        if (array[leftIndex] > array[indexLargestElement]) {
            indexLargestElement = leftIndex;
        }
    }
    if (rightIndex < length) {
        if (array[rightIndex] > array[indexLargestElement]) {
            indexLargestElement = rightIndex;
        }
    }
    if (indexLargestElement != currentIndex) {
        swap(array, indexLargestElement, currentIndex);
```

```
57          heapify(array, indexLargestElement, length);
58      }
59  }
60
61  void fixHeap(int array[], int length) {
62      int problemIndex = controlMaxHeap(array, 0, length);
63      while (problemIndex != -1) {
64          heapify(array, problemIndex, length);
65          problemIndex = controlMaxHeap(array, 0, length);
66      }
67      // The loop is necessary to cover cases such as [5, 5, 3, 2, 1, 6].
68      // You could also make this part a bit more efficient by comparing the element directly with the parent node.
69      // However, for the sake of simplicity, this optimization has not been done.
70  }
71
72  int main() {
73      int array[] = {5, 5, 3, 2, 1, 4};
74      int length = sizeof(array) / sizeof(array[0]);
75      fixHeap(array, length);
76      for (int i = 0; i < length; i++) {
77          printf("%d, ", array[i]);
78      }
79      return 0;
80  }
```

Task 2 [Medium]

2.1: In the lecture you learned about the hoare partition. Write the function `hoarePartition(int array[], int leftIndex, int rightIndex)` in the manner of the learned algorithm. Also use the rightmost element as the pivot element.

Example:

- $[2, 6, 4, 1, 5, 3]$ should return 2 and the array should be [2, 3, 1, 4, 5, 6].

2.2: Use the previously programmed partition to implement the function `quickSort(int array[], int leftIndex, int rightIndex)`.

```
1  #include "stdio.h"
2
3  int counter = 0;
4
5  void swap(int array[], int index1, int index2) {
6      int tmp = array[index2];
7      array[index2] = array[index1];
8      array[index1] = tmp;
9  }
10
11 int hoarePartition(int array[], int leftIndex, int rightIndex) {
12     int i = leftIndex - 1;
13     int j = rightIndex + 1;
14     int pivot = array[rightIndex];
15     while (1) {
16         do {
17             counter++;
18             j--;
19         } while (array[j] > pivot);
20         do {
21             counter++;
22             i++;
23         } while (array[i] < pivot);
24         if (i >=j) {
25             return i;
26         }
27         swap(array, i, j);
28     }
29 }
30
31 void quicksort(int array[], int leftIndex, int rightIndex) {
32     if (leftIndex < rightIndex) {
33         int m = hoarePartition(array, leftIndex, rightIndex);
34         quicksort(array, leftIndex, m - 1);
35         quicksort(array, m, rightIndex);
36     }
37 }
38
39 int main() {
40     int array[] = {6, 8, 4, 5, 3, 7, 2, 1, 0, 9};
41     int n = sizeof(array) / sizeof(array[0]);
42     quicksort(array, 0, n - 1);
43     printf("Counter: %d \n", counter);
44     for (int i = 0; i < n; i++) {
```

```
45          printf("%d, ", array[i]);
46      }
47      return 0;
48  }
```

2.3: Modify your quick sort program so that you can determine how often the innermost loops of the hoare partition have been run. Add up the iterations of both loops across all function calls. To do this, you can use a global variable, add an argument to the function with a pointer to the counter (if you already know pointers) or come up with a method of your own. Run this modified algorithm with the following two arrays.

- $[6, 8, 4, 5, 3, 7, 2, 1, 0, 9]$
- $[1, 9, 0, 6, 3, 8, 7, 2, 4, 5]$

What do you notice regarding the counter and how do you explain what you observe?

In this task you should notice that the first array takes longer than the second. The exact number of passes depends somewhat on the implementation, but the first array should take about 61 and the second about 51 iterations. This is the case because the quick sort cannot guarantee a runtime of $O(n \log n)$. In the worst case, the quick sort requires a runtime of $O(n^2)$. Since the first array is relatively close to such a worst case, this array requires more iterations.

2.4: Try to modify the array from the last task by rearranging the elements such that the count of the innermost loops is as high as possible. For which input does the quick sort take the longest?

Quick sort using hoare partition and the rightmost element as the pivot can have problems in cases where the input array is already sorted or nearly sorted. In the worst-case scenario, the pivot will be the largest or smallest element. This results in one of the partitions being empty (or nearly empty) and the other containing almost all the elements.

2.5 Bonus: Do you have any ideas on how to reduce the risk of such a worst-case scenario?

There are several strategies to optimize quick sort like using a hybrid approach and combining quick sort with other sorting algorithms. However, one of the simplest, yet efficient methods is to change the way the pivot is choosen. The aim is to ensure that an already sorted array is not our worst case and that the pivot element divides the array as fairly as possible into two equally sized subarrays. To name two of many possibilities:

- Random pivot: Instead of always choosing a fixed element (like the first, middle, or last) as the pivot, randomly selecting an element can significantly reduce the chances of experiencing the worst-case runtime.
- Median-of-three: Another approach is to take the first, middle, and last element, determine the median of these three and take it as our pivot. This helps in avoiding the worst-case and gives us the opportunity to split the problem more evenly. However, this approach already requires some computations to determine the pivot.

Task 3 [Hard]

The median is the middle value in a set of numbers when they are arranged in numerical order, or the average of the two middle values if there is an even number of elements.

Write a C function `findMedian(int array[], int n)` that takes an array of integers and its length as parameters and returns the median of the array as a float. Your function should handle both even and odd-length arrays. However, you can assume that no number appears twice in the array.

Example:

- For the array [5, 2, 8, 3, 7], the function should return 5.0
- For the array [5, 2, 8, 3, 7, 4], the function should return 4.5

Note: The easiest way to solve this task is to sort the array first and then access the center. However, you can also find the median faster. For this it helps to consider whether the whole array actually has to be sorted. Try to find such a solution that reduces unnecessary sorting effort.

```c
#include "stdio.h"

void swap(int array[], int index1, int index2) {
    int tmp = array[index2];
    array[index2] = array[index1];
    array[index1] = tmp;
}

int hoarePartition(int array[], int leftIndex, int rightIndex) {
    int i = leftIndex - 1;
    int j = rightIndex + 1;
    int pivot = array[rightIndex];
    while (1) {
        do {
            j--;
        } while (array[j] > pivot);
        do {
            i++;
        } while (array[i] < pivot);
        if (i >=j) {
            return i;
        }
        swap(array, i, j);
    }
}

int medianRecursive(int array[], int leftIndex, int rightIndex, int medianIndex) {
    if (leftIndex >=rightIndex) {
        return array[leftIndex];
    }
    int m = hoarePartition(array, leftIndex, rightIndex);
    // Sort only the part in which the median will end up
    if (medianIndex >=m) {
        return medianRecursive(array, m, rightIndex, medianIndex);
    } else {
        return medianRecursive(array, leftIndex, m - 1, medianIndex);
    }
}
```

```
39
40 float median(int array[], int length) {
41     if (length == 0) {
42         return 0;
43     }
44     if (length % 2 == 0) {
45         // Guarantees that the small element from which the median is composed is placed at it's correct position.
46         int smallerMedianElement = medianRecursive(array, 0, length - 1, (length - 1) / 2);
47         /*
48          Since we do not run the complete sorting algorithm, there is no guarantee that the larger element of the median
49          is already in the correct position. To find this element, we could run the same algorithm again, except that
50          we search for the element one to the right. Alternatively, we can make use of the fact that we now search the
51          next bigger element of our previously found element and the fact that this element must be on the right-hand side.
52          */
53         int largerMedianElement = array[(length / 2)];
54         for (int i = (length / 2) + 1; i < length; i++) {
55             if (largerMedianElement > array[i]) {
56                 largerMedianElement = array[i];
57             }
58         }
59         return (float) (smallerMedianElement + largerMedianElement) / 2;
60     } else {
61         return (float) medianRecursive(array, 0, length - 1, length / 2);
62     }
63 }
64
65 int main() {
66     int array[] = {5, 4, 6, 3, 1, 2, 7, 8, 9, 10};
67     int n = sizeof(array) / sizeof(array[0]);
68     printf("%f", median(array, n));
69     return 0;
70 }
```