Algorithms & Datastructures
Laboratory Exercise Sheet 3

Wolfgang Pausch <wolfgang.pausch@uibk.ac.at>
Heiko Studt <heiko.studt@uibk.ac.at>
René Thiemann <rene.thiemann@uibk.ac.at>
Tomas Vitvar <tomas.vitvar@uibk.ac.at>

March 24th, to be discussed on April 13th

Exercise 1) The Master Theorem Consider the recurrence equations

- $T_1(n) = 2 * T(\frac{n}{2}) + n^3$
- $T_2(n) = 2 * T(\frac{n}{2}) + n * log(n)$
- $T_3(n) = 16 * T(\frac{n}{4}) + n^2$
- $T_4(n) = 2 * T(\frac{n}{4}) + \sqrt{n}$

For $i \in \{1, 2, 3, 4\}$, either

- find the solution $T_i(n) = \Theta(\dots)$ using the master theorem
- or show why using the master theorem for T_i is not possible.

Hint: You may assume that $n^\epsilon \notin \mathcal{O}(log(n))$ for all $\epsilon > 0$.

Exercise 1) Solution

- Given is: $T_1(n) = 2 * T(\frac{n}{2}) + n^3$. Thus $a = 2$, $b = 2$ and $f(n) = n^3$. $log_2 2 = 1$ and $n^3 \in \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$. Furthermore, $2 * (\frac{n}{2})^3 = 2 * \frac{n^3}{8} \leq n^3$, thus case 3 of the master theorem may be applied. The solution is thus $T(n) = \Theta(n^3)$.

- Here, $T_2(n) = 2 * T(\frac{n}{2}) + n * log(n)$ is given. Thus $a = 2$, $b = 2$ and $f(n) = n * log(n)$. $log_2 2 = 1$. Obviously, $f(n) = n * log(n) \notin \mathcal{O}(n^{1-\epsilon})$, so case 1 is not applicable. Similarily, $f(n) = n * log(n) \notin \Theta(n^1)$, so case 2 is not applicable either.

  Thus, for having a chance to apply the master theorem using case 3, $f(n) \in \Omega(n^{1+\epsilon})$ for some $\epsilon > 0$ must be true. This means, we have to find a $c > 0$ and a $n_0 \in \mathbb{N}$ such that for all $n \geq n_0$ the following is true: $c * n^{1+\epsilon} \leq n * log(n)$. In other words, $c * n^{\epsilon} \leq log(n)$ would be required. As $n^{\epsilon} \notin \mathcal{O}(log(n))$, case 3 is not applicable either, so the master theorem cannot be applied for $T_2(n)$.

- Given is $T_3(n) = 16 * T(\frac{n}{4}) + n^2$. Here, $a = 16$, $b = 4$ and $f(n) = n^2 = \Theta(n^2)$. $log_4 16 = 2$. Thus case 2 is applicable and $T(n) = \Theta(n^2 * log(n))$.

- Finally, $T_4(n) = 2 * T(\frac{n}{4}) + n^{\frac{1}{2}}$. So, $a = 2$, $b = 4$ and $f(n) = n^{\frac{1}{2}} \in \Theta(n^{\frac{1}{2}})$. Because $log_4 2 = \frac{1}{2}$, case 2 is applicable here as well, so $T(n) = \Theta(n^{\frac{1}{2}} * log(n))$.

Exercise 2) MinMax-Calculation Let A be an array of size n. If you want to find both the smallest and the biggest value inside A, one straightforward way is scanning the algorithm twice, the first time for finding the smallest value, the second time for finding the biggest value. Obviously, this algorithm involves $2n-2$ comparisons between array elements.

1. Implement a divide and conquer algorithm which calculates both the minimum and the maximum of A using less than $2n-2$ comparisons between array elements. Assume that here, n can have any arbitrary value ≥ 0. React to the special case that $n = 0$ in an appropriate way.
2. Test your implementation. Write down (and use for a test) a set of test arrays which tests all relevant cases in your opinion.
3. Now we assume that $n = 2^m$ is a power of two. Prove that your algorithm needs less than $2n-2$ comparisons between array elements. It might be useful to guess the exact number of comparisons and then prove this formally by induction on m.

Hint: You can use the following code skeletons:

MinMaxInfo.java

```
class MinMaxInfo {
    int minValue;
    int maxValue;

    MinMaxInfo(int minValue, int maxValue) {
        this.minValue = minValue;
        this.maxValue = maxValue;
    }
}
```

MinMax.java

```
public class MinMax {
   public static void main(String[] args) {
       int[] testArray = {1,1};
       MinMaxInfo result = calculateMinMax(testArray);
       System.out.println("Minimum is " + result.minValue + "; maximum is "
            + result.maxValue);
   }

   static MinMaxInfo calculateMinMax(int[] values) {
       throw new RuntimeException("add code here");
   }

   static MinMaxInfo calculateMinMax(int[] values, int minIndex, int
      numberOfIndices) {
       throw new RuntimeException("add code here");
   }
}
```

Exercise 2) Solution

1. Treat the array as binary tree. For trees of height 2, one comparison is sufficient since the element which is not the smaller one is always the bigger one and vice versa. For higher trees, minimum must be compared with minimum and maximum with maximum. This in total results in $\frac{3}{2} * n - 2$ comparisons.

   Source code:

MinMaxInfo.java

```java
class MinMaxInfo {
    int minValue;
    int maxValue;

    MinMaxInfo(int minValue, int maxValue) {
        this.minValue = minValue;
        this.maxValue = maxValue;
    }
}
```

MinMax.java

```
public class MinMax {
    public static void main(String[] args) {
        int[] testArray = {1,1};
        MinMaxInfo result = calculateMinMax(testArray);
        System.out.println("Minimum is " + result.minValue + "; maximum
            is " + result.maxValue);
    }

    static MinMaxInfo calculateMinMax(int[] values) {
        return calculateMinMax(values, 0, values.length);
    }

    static MinMaxInfo calculateMinMax(int[] values, int minIndex, int
        numberOfIndices) {
        if (numberOfIndices == 0) {
            throw new RuntimeException("No zero-length arrays supported
                .");
        } else if (numberOfIndices == 1) {
            // called for just one value, so this value is min and max.
            return new MinMaxInfo(values[minIndex], values[minIndex]);
        } else if (numberOfIndices == 2) {
            // called for two values, so the value which is not the
                smaller one is the bigger one
            if (values[minIndex] < values[minIndex + 1]) {
                return new MinMaxInfo(values[minIndex], values[minIndex
                    + 1]);
            } else {
                return new MinMaxInfo(values[minIndex + 1], values[
                    minIndex]);
            }
        } else {
            // called for more than two values, so we need a recursive
                call
            int numberOfIndicesFirstCall = numberOfIndices / 2;
            int minIndexSecondCall = minIndex +
                numberOfIndicesFirstCall;
            int numberOfIndicesSecondCall = numberOfIndices -
                numberOfIndicesFirstCall;
            MinMaxInfo resultFirstCall = calculateMinMax(values,
                minIndex, numberOfIndicesFirstCall);
            MinMaxInfo resultSecondCall = calculateMinMax(values,
                minIndexSecondCall, numberOfIndicesSecondCall);

            int minValue;
            if (resultFirstCall.minValue < resultSecondCall.minValue) {
                minValue = resultFirstCall.minValue;
            } else {
                minValue = resultSecondCall.minValue;
            }

            int maxValue;
            if (resultFirstCall.maxValue > resultSecondCall.maxValue) {
                maxValue = resultFirstCall.maxValue;
            } else {
                maxValue = resultSecondCall.maxValue;
            }
            return new MinMaxInfo(minValue, maxValue);
        }
    }
}
```

2. For example, test using the following arrays:

   - The empty array (we expect some error here, since the minimum of an empty array is undefined)
   - An array of size 1, e.g. (11)
   - An array of size 2, e.g. $(4, 17)$
   - An array of size 3, e.g. $(3, 1, 4)$
   - An array of size 4 with just equal values, e.g. $(4, 4, 4, 4)$
   - Some bigger array, size not a power of two, e.g. $1, 3, 4, 3, 2, 4, 67, 3, 4, 1, 3, 4, 4, 4, 6, 2, 4, 3, 1$

3. Let $n = 2^h$ and $c(h)$ the needed number of key comparisons. We show that the number of comparisons for $c(h) = \frac{3}{2} \cdot 2^h - 2$ for all $h \geq 1$ by induction over h. (And hence, the algorithm needs $\frac{3}{2} \cdot n - 2$ comparisons.)

   - $h = 1$: Then $n = 2^1$, thus $c(1) = 1 = \frac{3}{2} * 2 - 2$.
   - $h \to h + 1$: Then $c(h + 1) = 2 * c(h) + 2 = 2 * (\frac{3}{2} * 2^h - 2) + 2 = \frac{3}{2} * 2^{h+1} - 2$, since we compare the results of two sub-trees t_1 and t_2, i.e. calculate $min_{t_1} < min_{t_2}$ and $max_{t_1} < max_{t_2}$.

Exercise 3) Interpolation search You know interpolation search from the slides.

1. Implement it.
2. You know from the slides that it has worst case complexity $\mathcal{O}(n)$. Find an example where it actually needs such a large number of operations.

Exercise 3) Solution

1. Implementation:

InterpolationSearch.java

```
public class InterpolationSearch {
   static int interpolationSearch(int[] a, int x) {
      int left = 0;
      int right = a.length - 1;
      if (right < 0 || x < a[0] || x > a[right]) {
         return -1;
      }
      while (left <= right) {
         int al = a[left];
         int ar = a[right];
         int middle = left + (int) ((x-al) / (double) (ar - al) * (
             right - left));
         if (x < a[middle]) {
            right = middle - 1;
            if (x > a[right]) {
               return -1;
            }
         } else if (x == a[middle]) {
            return middle;
         } else {
            left = middle + 1;
            if (x < a[left]) {
               return -1;
            }
         }
      }
      return -1;
   }
}
```

2. Consider the array $0, \ldots, 0, n$ of length n and a call of `interpolationSearch(a,1)`.
   The `right`-value will always stay at $n-1$, whereas the `left`-value will be $0, 1, 2, \ldots, n-1$.

Exercise 4) Heap-Sort

1. Let v be a level inside a heap as defined in the lecture. (The root has level 0, the children of the root have level 1, their children 2 and so on....)

   How many elements $n(v)$ can level v have at maximum? Prove your results by induction.

2. Sort the numbers $5, 3, 17, 10, 84, 19, 6, 22, 9$ using HeapSort. Show the heap both as array and as tree after each run of both downHeap and swap.

Exercise 4) Solution

1. Level v can have at most 2^v elements. Proof:
   - $n(0) = 1 = 2^0$ since the root alone is one element.
   - $v \to v+1$: $n(v+1) \leq 2 * n(v) = 2 * 2^v = 2^{v+1}$ since each node can have at most two children.
2. Starting with a heap represented by the array $5, 3, 17, 10, 84, 19, 6, 22, 9$ first do the following downHeap steps:
   - Call downHeap for the 10, now we have $5, 3, 17, 22, 84, 19, 6, 10, 9$.
   - Call it for the 17, now we have $5, 3, 19, 22, 84, 17, 6, 10, 9$.
   - Call it for the 3, now we have $5, 84, 19, 22, 3, 17, 6, 10, 9$.
   - Call it for the 5, now we have $84, 22, 19, 10, 3, 17, 6, 5, 9$.

   Now, perform the following swap and downHeap steps:
   - Swap 84 and 9, now we have $9, 22, 19, 10, 3, 17, 6, 5 \mid 84$. The remaining heap is the array up to the 5. Now call downHeap for the 9, now we have $22, 10, 19, 9, 3, 17, 6, 5 \mid 84$.
   - Swap 22 and 5, now we have $5, 10, 19, 9, 3, 17, 6 \mid 22, 84$. Now call downHeap for the 5, resulting in $19, 10, 17, 9, 3, 5, 6 \mid 22, 84$.
   - Swap 19 and 6, now we have $6, 10, 17, 9, 3, 5 \mid 19, 22, 84$. Call downHeap for the 6, resulting in $17, 10, 6, 9, 3, 5 \mid 19, 22, 84$.
   - Swap 17 and 5, now we have $5, 10, 6, 9, 3 \mid 17, 19, 22, 84$. Call downHeap for the 5, resulting in $10, 9, 6, 5, 3 \mid 17, 19, 22, 84$.
   - Swap 10 and 3, now we have $3, 9, 6, 5 \mid 10, 17, 19, 22, 84$. Call downHeap for the 3, resulting in $9, 5, 6, 3 \mid 10, 17, 19, 22, 84$.
   - Swap 9 and 3, now we have $3, 5, 6 \mid 9, 10, 17, 19, 22, 84$. Call downHeap for the 3, resulting in $6, 5, 3 \mid 9, 10, 17, 19, 22, 84$.
   - Swap 6 and 3, now we have $3, 5 \mid 6, 9, 10, 17, 19, 22, 84$. Call downHeap for the 3, resulting in $5, 3 \mid 6, 9, 10, 17, 19, 22, 84$.
   - Finally swap 5 and 3, resulting in $3, 5, 6, 9, 10, 17, 19, 22, 84$ and we are finished.