Algorithms and Datastructures (HS2024)
Week 5

Rui Zhang

October 21, 2024

Contents

1 Revision Theory

This week, you went through two more algorithms for sorting and some simple datastructures.

1.1 Algo: Quicksort

Quicksort is a sorting algorithm that you will often hear online. It is also a divide and conquer / recursive algorithm. The steps of this algorithm are:

1. choose an element p (also called pivotelement) (for example first or last element)

2. sort all elements less than p to the left of p and all elements greater than p to the right of p.

3. sort the subarrays left of p and right of p recursively, by repeating steps 1, 2, 3 for the smaller subarray.

The exact pseudocode / details (indices, $<$ or $\leq$) are not that relevant. Let us instead look at the runtime.

Notice, that the runtime of this algoithm can be good or absolutely horrendous. Imagine we choose the pivotelement randomly and everytime, we get unlucky and this pivotelement is the largest element of the current subarray. In this case, we will always have to sort all elements to the left and repeat, having only reduced the size of the subarray by just one element. As seen in the lecture notes, this results in a recursive runtime function of

$$T(n) = T(n-1) + cn$$

which is in $O(n^2)$ (try it via induction!).

The best case appears when we choose a pviot element "perfectly" everytime. That would be when the pivot is exactly in the middle. We get the recursive relationship:

$$T(n) \leq 2 \cdot T(n/2) + cn$$

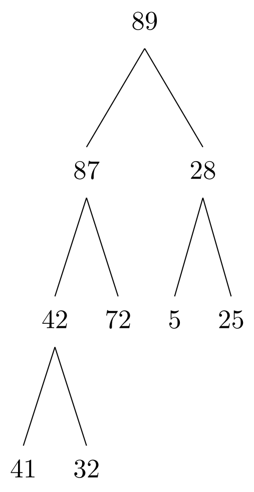
which is in $O(n \log n)$ (again, try it.)

However, if we do choose the pivotelement randomly, then "on average", we expect $O(n \log n)$ (Source: ~~it came to me in a dream~~ 2. semester shenanigans). So quicksort is quick on average, but not always.

1.2 Datastructure: Heap

Datastructures are ways to organize data so that they fulfil a certain purpose well.

For the next algorithm, heapsort, we want to organize data such that we can find the maximum of a set of data fast. We do this by putting all of the elements into a so called "tree".



In a tree, each element / number is a "node". The topmost node (89) is called the "root". Two nodes connected by a so called "edge" (those lines) are in a relation: the node towards the top is the "parent" and the node towards the bottom is the "child". Nodes without any children are "leaves".

The idea behind a (max-) heap is that the data is structured in a tree as above such that the largest element is always at the root. And parents always have larger so called "keys" stored in them (the values you can see in the tree) than their children. Intuitively, we can say that the further up we are, the larger the elements get.

For those curious, storing a heap in memory / in a java program for example often consists of creating an array with a special structure. For details check out the lecture notes or take a look at this useful GeeksForGeeks Article: `https://www.geeksforgeeks.org/introduction-to-heap/`.

1.3 Algo: Heapsort

Using this datastructure, we can now create heapsort, consisting of two steps:

1. transform array into heap
2. find maximum in heap, remove and put into the next largest spot in the array. (n times)

Let us take a closer look at the second step. To do this step efficiently, We remove the root from the tree (since this node will always have the largest key) and replace it with the right-most leaf of this tree. (the one on the bottom right). In the tree before, this would be removing 89 and subsequently putting 32 where 89 was. This is not enough however, since the heap now violates the structure we had imposed before ("the higher up we are, the larger the elements"). The only node doing so is the root, since we have not modified anything else.

We can solve this problem by swapping the newly inserted root with one of its children, namely the larger one (the one with the larger key). If we swapped with the smaller child, then a "smaller" node would be higher up than a "larger" node. We will then repeat this (in our example with 32) until this node reaches a spot where it is larger than both of its children.

The runtime for this is $O(\log n)$. The reason being that the number of "levels" in the tree is $O(\log_2 n)$, which is the maximum number of swaps we will have to do.

For the first step, the insertion, we just put the new node with its respective value at the next available spot for a leaf (in the example above, this would be "next to" 32) and thus a child to 72. Then, to make sure that the structure of the heap is kept, we will again, swap this node with its only parent until its parent-node's key is larger than the this node's value. Again, same as before, we may argue that the maximum number of swaps / runtime is $O(\log n)$

In total, since we will execute insertion and removal n times each, we have a runtime of $O(n \log(n))$ for heapsort.

1.4 Limits of Comparison Based Sorting

You might have noticed now that we often get this runtime of $O(n \log n)$. But somehow, we cannot find an algorithm better than this runtime.

This is because we literally cannot do better if we create algorithms based on comparisons. To understand this, envision a tree consisting of nodes in which each node executes a comparison. That comparison either yields "true" or "false". This tree is obviously also a binary tree and it is special: it is called a decision tree. Imagine a comparison-based sorting algorithm. Then, this algorithm go through such a tree, going through each node and either continuing with its left or right child depending on the result of the comparison.

Now, let each leaf of this tree be the end result which the algorithm returns. For the algorithm to be fully correct, we would need to have at least $n!$ many leaves, since each possible permutation of the array (there are $n!$ many) must be a possible result. This gives us a lower bound for the number of leaves and therefore the number of nodes in this decision tree.

We also have an upper bound though. For each layer k, we have at most 2^k many nodes. For a tree of "height" h, that results in $1 + 2 + 4 + 8 + \cdots + 2^h = 2^{h+1} - 1$ many nodes at most. We thus have the inequality

$$2^{h+1} \geq \#nodes \geq n!$$

resulting in

$$h \geq \log_2 n! - 1 \geq \Omega(n \log(n))$$

where the last inequality comes from the result of a previous exercise sheet. Now, since the height of the tree is at least $\Omega(n \log(n))$, we have that the number of elementary operations is at least just the comparisons made, which is at least the height of the tree, which is at least $\Omega(n \log(n))$. So we cannot do better than this.

Note: This lower bound only applies to *comparison*-based sorting algorithms. Usually, if we have no other assumptions about the given, unsorted array, we will indeed have to use a comparison based algorithm. However, if we know additional information about the array, then we may also apply other sorting algorithms such as bucketsort: `https://en.wikipedia.org/wiki/Bucket_sort`.

1.5 Linked Lists

We have seen one example of a datastructure already: Heaps. Let us take a look at more. First up: Linked Lists.

In programming, we often have linked lists as data structures to have a more modifiable alternative to an array. The linked list consists of multiple nodes with keys and a pointer pointing towards the next node in the list. As opposed to arrays, we may now save the first node of the linked list and everytime we want to add an element, we just create a new node, set its key and point its "next pointer" to the first node that we saved. Then we naturally change the first node to be the new node we created.

1.6 Stack

Imagine a stack just like in real life. As long as you are not a psychopath, you will add new elements to the top of the stack and remove stuff from the top of the stack. This datastructure follows the "LIFO" (Last in first out) principle. I think you get why we call it "LIFO".

We have the operations "push" (adding an element) and "pop" (removing an element).

1.7 Queue

Again, same as in real life, a queue works just as expected (except if you know some friends in the polymensa who are already ahead in the line, in which case you unsuspectingly go up to them and start a conversation, effectively cutting in line in a socially acceptable way).

We have the operations "enqueue" (which adds a new element at the very back of the queue) and "dequeue" (which removes the element which is at the front of the queue, who has been waiting so diligently for their turn (or have cut in line at the polymensa)).

1.8 Priority Queue

What if the polymensa decided to make some more money? Imagine someday (god forbid this actually happens), the head of polymensa announces with joy, that they will now be offering VIP passes with a certain value assigned to the passes, depending on how much we as customers decide to pay for this VIP pass. The higher the value, the higher up we will be in the queue when we wait for our food.

Now, imagine you are the head of polymensa. You are probably already unfathomably rich, but now your three year old daughter is asking for a private jet, and how could you deny such a request? Now, which datastructure that we have already seen can implement this priority queue / act as a substitute?

(Left as an exercise to the reader, who is a proud parent wishing to fulfil the wishes of their lovely child and thus would do anything to figure out the answer to this question, even looking up the solution from the lecture notes if really needed.)

2 Exercises

2.1 Exercise Sheet 5 - Priority List

Again, top is most important, bottom is least important.

1. 5.1), 5.4) Exercises where you have to apply algorithms to datastructures such as heaps are common in exams
2. 5.3) Still a bonus exercise
3. 5.5) Being able to implement datastructures is an essential skill not just for A&D
4. 5.2) this exercise deepens your understanding of the proof pattern used for the the limits of comparison based sorting and searching.

2.2 Exercise Sheet 3 - Feedback

1. 3.1) was not easy. Please take a look at the solutions and get to understand them well.
2. Do not mix up Θ, Ω and Big-O Notations. Proving that some function is in Big-O of something, we need to find an upper bound, so the chain of inequalities is formed with $\leq$ and $=$ only. For Ω, it is a lower bound. That is $\geq$ and $=$. For Θ, you need to show a tight bound, this means that we have to show that both the lower bound Ω and the upper bound Big-O agree in their order. Do not combine $\leq$ and $\geq$ in one chain of inequalities.
3. Do not apply logarithms on fractions. Only do so in equations and inequalities
4. Do not apply any other function on fractions.
5. The only way to modify a fraction without changing its value is to expand. Nothing else.

6. There are two main ways (plus a few others) to prove Θ, Ω and Big-O. one of them is to prove a chain of inequalities and another is compute the limit (plus finding constants, etc.). The point is however: choose one strategy and stick with it. Do not start with proving inequalities, then continue on informally using the logic for limits.

3 Supplementary Exercises

Foreword before the supplementary exercises: In class as well as in these supplementary exercises, I often cover old exam exercises. As eccentric as this might sound, I would like you to try and avoid remembering these exact exercises and their solutions.

The point of exercises is not for you to know every solution, but to develop the correct intuition and to develop the correct way of thinking in order to solve problems. Skills are more important than raw knowledge.

3.1 Sorting Quiz

This exercise question comes from the FS2022 Exam on VIS Comsol: `https://exams.vis.ethz.ch/exams/hwjcsaqj.pdf`.

For each of the following questions, choose if the statement is correct:

- In the worst case, selection sort needs less swaps than insertionsort
- The worst case for bubble sort is when the array is already sorted
- Quicksort is asymptotically faster than bubblesort in the worst case

3.2 Counting Loop Iterations

This exercise question also comes from the same exam.

For the following code snippets, derive an expression for the number of times f is called. Simplify it into the Θ-Notation

1)

```
for j = 1, ..., n do:
  for k = j^2, j^2+1, ..., (j+1)^2 do:
    f()
```

2)

```
for j = 1, ..., n do:
  for k = 1,..., n do:
    l <- 1
    while k + l <= j do:
      f()
      l <- 2*l
```