Sorting an array

Algorithms and data structures ID1021

Johan Montelius

Fall term 2022

Introduction

In this assignment you will explore some search algorithms, all have their pros and cons and it's important to know when to use which. We will in this assignment only work with arrays of a given size and not add any new elements to the data set. For each of the algorithms you should explain the run time as a function of the size of the array.

the not so efficient

Let's start with a simple algorithm that is not very efficient but simple to implement and it works ok if the array is small. The algorithms goes as follows:

start by setting an index to the first position in the array, then

- from to the index position to the end of the array, find the smallest element smaller than the value at the index position,
- swap the two elements and,
- move forward one position in the array.

You will have corner cases such as if no smaller element is found but the implementation is only a couple if lines:

```
for (int i = 0; i < array.length -1; i++) {
  // let's set the first candidate to the index itself
  int cand = i;
  for (int j = i; j < array.length ; j++) {
    // If the element at position j is smaller than the value
    // at the candidate position - then you have a new candidate
    // posistion.
```

```
    }
    // Swap the item at position i with the item at the candidate position.
  }
```

This algorithm is called *section sort* since the algorithm scans the rest of the array and selects the item that is the smallest. What is the run time complexity of this algorithm? Do some benchmarks and verify your assumption.

slightly more complicated

We can twist the algorithm around and instead of selecting the smallest we *insert* an item in the already sorted part of the array. The algorithm goes as follows:

start with the first index in the array,

- if the item at the index position is smaller than the item before move it towards the beginning (insert)
- when inserted at the right place, move the index forward.

When we move an item towards the beginning we of course need to make room for the item so all items that we pass should be moved one step. This is of course a costly operation but as you will see it might be worth the price.

The implementation is a bit trickier but still only a few lines of code:

```
  for (int i = 0; i < array.length; i++) {
    // for each element from i towards 1, swap the item found with the
    // item before it if it is smaller
    for (int j = i; j > 0  && ....  ; j--) {
       :
    }

  }
```

Do some benchmarks and determine how well insertion sort compares to selection sort. Is there a difference, why? Are there any arrays that the insertion algorithm works better or worse for?

completely different

There are plenty of sorting algorithms to choose from but here we will look at the one called *merge sort*. The difference here is that we will work with a additional array where we will store temporary results before producing the

final sorted array. We will also implement the algorithm *recursively* which might be a bit mind boggling if this is the first time you encounter this technique.

The algorithm is quite easy to describe and not very hard to implement but one needs to understand what is going on. It goes like follows:

start with the original array, divide it in two and place a sorted version of the first part in one array and a sorted version of the second part in another array, then *merge* the two arrays into one sorted array.

The only thing wee need to understand is how merging is done and this is quite simple. We will go through the two arrays item by item and select the smallest item found to be the next item in the list. Thing about having two piles of sorted playing cards face up and always selecting the smallest card.

Hmm, how do we sort the two part? Should we use insertion sort? ... could we use merge sort?

Let's start by constructing a temporary array and then call a method with the two arrays. The new method will also take to indices between which it should do the sorting. The method, when done, should have all items between (and including) the two numbers sorted.

```
public static void sort(int[] org) {
  if (org.length == 0)
     return;
  int[] aux = new int[org.length];
  sort(org, aux, 0, org.length -1);
}
```

So what should the new sort method look like? We have two array, one

```
private static void sort(int[] org, int[] aux, int lo, int hi) {

  if (lo != hi) {
    int mid =  lo + (hi-lo)/2;
    // sort the items from lo to mid
    :
    // sort the items from mid+1 to hi
    :
    // merge the two sections using the additional array
    merge(org, aux, lo, mid, hi);
  }
}
```

You can wait a bit with the sort method and first implement the merge method. When we call this method the original array is divided into two

parts (from lo to mid and from mid+1 to hi). Both parts are internally sorted and the task is to merge them into one sorted sequence.

We now make use of the temporary array (we could have created it here but since it is given to we can use the one we get). We first copy all element over to the temporary array and then start the merging.

```
private static void merge(int[] org, int[] aux, int lo, int mid, int hi) {

  // copy all items from lo to hi from org to aux
  for ( ......  ) {
    :
  }

  // let's do the merging

  int i = lo;    // the index in the first part
  int j = mid+1; // the index in the second part

  // for all indices from lo to hi
  for ( int k = lo; k <= hi; k++) {
    // if i is greater than mid,  move the j item to the org array, update j
    :
    // else if j is greate than hi, move the i item to the org array, update i
    :
    // else if the i item is smaller than the j item,
    //         move it to the org array, update i
    :
    // else you can move the j item to the org array, update j
    :
  }
}
```

If you get this right you have all the pieces of the puzzle. The only thing that is missing is fixing the sort method. If you can solve this you have a method that given an original array and a temporary array will sort the original array from lo to hi.

The only thing you need is a method that can sort an array from one index to another... but, is this not what we have? Can we use our own sort method to do the sorting of the two sub-parts?

The recursive thinking could be quite tricky to get around the first time you see it but once you master it you will start to look at all problems with recursive eyes.

Do some benchmarks and see how merge sort compares to selection sort and insertion sort. Can you find a function that roughly explains the execution time?

there are more

Merge sort is a very efficient sorting algorithm with over all good performance. It does need a temporary array to do the merging but apart from that, it has few cons.

There are other sorting algorithms and one of the most used is an algorithm called quick sort. The quick sort algorithm is also a recursive algorithm but here we first rearrange the item of the array so we have all the smaller items to the left and the all higher to the right (of some element). We then sort the two parts recursively and then we're done. The benefit is that we do not need a temporary array but we will have some weird cases where quick sort degenerates to something that is worse than selection sort. Merge sort is more robust and will always perform well.