Sorting an array
in C

Algorithms and data structures ID1021

Johan Montelius

Fall 2024

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

The implementation is only a couple if lines:

```
// n is the length of array
for (int i = 0; i < n -1; i++) {
  int candidate = ...
  for (int j = i; j < n ; j++) {
     :
  }
```

```
    :
  }
```

This algorithm is called *selection 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 item, we *insert* the next 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 (i.e. insert)
- when inserted at the right place, move the index forward one step.

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 forward. 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 if implement a procedure `swap` that will swap two elements in an array.

```
// n is the length of the array
for (int i = 0; i < n; i++) {
  for (int j = i; j > 0  && ....  ; j--) {
    swap(..., ..., ...);
  }
}
```

Note that we only have to swap elements as long as we find something bigger. As soon as we we find a smaller element we're done and can move forward.

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? You have to think a bit on the last question and then possibly confirm your thoughts in an experiment

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 figure out 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. Think about having two piles of sorted playing cards face up and always selecting the smallest card.

Hmm, how do we sort the two parts? Should we use insertion sort? ... could we use merge sort? The magic of recursion will solve our problems.

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 indices sorted.

```
void sort(int *org, int n) {
  if (n == 0)
     return;
  int *aux = (int*)malloc(n * sizeof(int));
  sort(org, aux, 0, n-1);
}
```

So what should the new sort method look like? We have two arrays, one with the unsorted elements and one that we can use as an auxiliary storage area. If our task is to sort all elements between the two indices then if the `lo` index is not equal to the `hi` index we need to do something.

```
void sort(int *org, int *aux, int lo, int hi) {

  if (lo != hi) {
    int mid =  (lo + hi)/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 (thanks to our recursive call) 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 us we can use the one we get). We first copy all element over to the temporary array and then start the merging.

```
  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 then
      //   move the j'th item to the org array, update j

      // else if j is greate than hi then
      //   move the i'th item to the org array, update i

      // else if the i'th item is smaller than the j'th item,
      //   move the i'th item to the org array, update i

      // else
      //   move the j'th item to the org array, update j

    }
  }
```

Before you copy and past this code, perform the algorithm using a pen and paper. Assume that you have an array that is sorted from 0 to 3 and from 4 to 7. You should then merge the arry given `lo` equal to 0, `mid` equal to 3 and `hi` equal to 7.

```
{2,5,6,9,1,3,7,8}
```

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 is 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. 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.

Another difference between Merge sort and Quick sort is that Merge sort is *stable*. A stable sorting algorithm is an algorithm that does not change an already given order if not required. Let's say you have an array of people identified by their given name and their surname. If we now use Quick sort to sort the array on the first name Anders Andersson will occur before Bengt Andersson. If we now sort the sorted array based on the surname using Quick sort, there is no guarantee that Anders Andersson is before Bengt Andersson since they have the same surname. A stable sorting algorithm will preserve any given order and guarantee that Anders Andersson comes before Bengt Andersson.

give this a try

In the description of the Merge sort algorithm there is a lot of copying of values to-and-fro the auxiliary array. There is one clever trick that we can use that will eliminate much of the copying, easy to implement - harder to explain.

The first thing we do is to create a copy of the original array i.e. copy its content to the auxiliary array. We now use our sort method to sort the elements in the auxiliary array and return the answer in the original array.

The sort method is thus given two identical arrays and should sort the elements of the second array and return the result in the first array. What if we in the recursive step now toggle the arguments so when we do the recursive call we sort the elements in the first array and return the result in the second array. We then call the merge method to merge the sorted results from the second array to the first array. The merge method now does not have to copy anything to an auxiliary array since the elements are already sorted in the auxiliary array - the merger will immediately place the result in the original array and we are done.

To realize that this actually works even though we are repeatedly doing recursive calls to the sort method could be a bit mind-boggling but it works. If you get it right you can run some benchmarks and see if it pays off.