Selection sort and quick sort, including pre- and post-conditions and invariants

FOUNDATIONS OF COMPUTER SCIENCE

Martín Escardó
School of Computer Science, University of Birmingham, UK

Version of November 18, 2010

Contents

The blue pieces of text are clickable links in the electronic version of this document.

1 Summary

There are many sorting algorithms, for example, bubble sort, selection sort, insertion sort, shell sort, quick sort, merge sort, heap sort, to name a few of the most common ones. In this handout:

1. We discuss the selection sort and quick sort algorithms.
2. We write their specifications using pre- and post-conditions, which themselves are written as algorithms. This helps to avoid potential mistakes in our reasoning.
3. We design the algorithms using invariants, as in the previous lectures and handouts.
4. We predict their run time, and compare them for efficiency. Logarithms arise again.
5. We perform experiments to confirm the theoretical run time predictions and to have a feeling of the time magnitudes that arise in practice.

2 Specification of the sorting task

Our sorting algorithms will have the following specification:

```
void sort(int [] a)
{
  // pre-condition: a is not null.

  ... // Suitable code, depending on the chosen algorithm.

  // post-condition: a is sorted in ascending order.

}
```

As always, we try to reason accurately about programs, so that we get programs right without the need of endless testing/debugging followed my modification and fixing of problems. However, even when we try to be careful, as we do, we may make mistakes. In order to double check that our reasoning is good, and gain further confidence, we can write the program specifications, variants, and invariants as algorithms, and check them using assertions. For example, the above can be rewritten as follows:

```
void sort(int [] a)
{
  // pre-condition:
  assert(a != null);

  ... // Suitable code, depending on the chosen algorithm.

  // post-condition:
  assert(isSorted(a));
```

Of course, we then need to write the algorithm for checking the post-condition. This can be done as follows in this example:

```
boolean isSorted(int [] a)
{
    assert(a != null);

    for (int i = 0; i < a.length - 1; i++)
      if (a[i] > a[i+1])
        return false;

    return true;
}
```

The point is that it is easier to check whether an array is sorted than to sort the array, and so we are less likely to make a mistake in the `isSorted` algorithm than in the `sort` algorithm.

Exercise. After reading this handout, go through the algorithms given, and write the invariants as boolean expressions and assert them at appropriate places of the algorithms. I will partially solve the exercise for you, and you have to complete the details.

A program with assertions can be run with and without the "enable assertions" option "`-ea`". Before deploying the program for general use, we enable the option to gain confidence of the correctness of the algorithm, and when we deploy it we disable the option for the sake of efficiency. But one has to be rather careful regarding safety critical systems that may cause injury if they do not operate properly — I am not giving any advice in this direction in this module. I just observe that both enabling and disabling assertions in such systems can be dangerous: if assertions are enabled, then the system may abort when it should be running and controlling something (e.g. air traffic); on the other hand, if assertions are disabled, then the system may issue instructions that cause injury (e.g. laser eye operating equipment and again air traffic).

3 Selection sort

Algorithm: The idea is to find the smallest element, the second smallest element the third smallest element, and so on.

1. Work with bigger and bigger initial sub-arrays.
2. Keep them sorted.
3. Make sure all the elements of the sub-array are smaller than the elements outside the sub-array.
4. Start with the empty sub-array.
5. Repeatedly grow it by finding the smallest element outside the array and swapping it with the first element outside the sub-array.
6. Finish when the sub-array is actually the whole array.

Example. Suppose we want to sort the array

```
72 69 27 92 7 52 65 37 69 28
```

We indicate the initial sub-array with square brackets and the smallest element outside the array with round brackets:

```
[] 72 69 27 92 (7) 52 65 37 69 28 -- find smallest
[] 7 69 27 92 (72) 52 65 37 69 28 -- swap with first outside

[7 ] 69 (27) 92 72 52 65 37 69 28 -- find
[7 ] 27 (69) 92 72 52 65 37 69 28 -- swap

[7 27 ] 69 92 72 52 65 37 69 (28) -- find
[7 27 ] 28 92 72 52 65 37 69 (69) -- swap

[7 27 28 ] 92 72 52 65 (37) 69 69 -- find
[7 27 28 ] 37 72 52 65 (92) 69 69 -- swap

[7 27 28 37 ] 72 (52) 65 92 69 69 -- find
[7 27 28 37 ] 52 (72) 65 92 69 69 -- swap

[7 27 28 37 52 ] 72 (65) 92 69 69 -- find
[7 27 28 37 52 ] 65 (72) 92 69 69 -- swap

[7 27 28 37 52 65 ] 72 92 (69) 69 -- find
[7 27 28 37 52 65 ] 69 92 (72) 69 -- swap
```

```
[7 27 28 37 52 65 69 ] 92 72 (69) -- find
[7 27 28 37 52 65 69 ] 69 72 (92) -- swap

[7 27 28 37 52 65 69 69 ] (72) 92 -- find
[7 27 28 37 52 65 69 69 ] (72) 92 -- swap

[7 27 28 37 52 65 69 69 72 ] (92) -- find
[7 27 28 37 52 65 69 69 72 ] (92) -- swap

[7 27 28 37 52 65 69 69 72 92]    -- Done.
```

A precise definition of the algorithm.

```
void selectionSort(int [] a)
{
  // Pre-condition:
  assert(a != null);

  for (int i = 0; i < a.length; i++)
    {
    // Invariant: the first i elements of the array are sorted
    // and smaller or equal than the other elements.

    int p = i;

    for (int j = i + 1; j < a.length; j++)
      {
        // Invariant: a[p] is the
        // smallest element in the range from i to j - 1.

        if (a[j] < a[p])
        p = j;
      }

    swap(a,i,p);
    }

  // Post-condition:
  assert(isSorted(a));
}
```

Swapping. Can of course be defined as:

```
void swap(int [] a, int i, int j)
{
  int n = a[i];
  a[i] = a[j];
  a[j] = n;
}
```

Exercise. Does the given algorithm work for (non-null) arrays of length zero? Why or why not?

Exercise. Rewrite the above program so that invariants are checked with assertions.

Partial solution. You have to complete the details. First write the invariants as programs. For the outer loop, we need:

```
boolean initialSortedAndSmallerThanOthers(int [] a, int i)
{
  assert(a != null);
  assert(0 <= i && i <= a.length);

  // Return true if and only if the first i elements of the
  // array are sorted and smaller or equal the other elements.

  ... // Fill this in.
}
```

For the inner loop, we need:

```
boolean positionWithSmallest(int a[], int i, int j, int p)
{
  assert(a != null);
  assert(0 <= i && i <= p && p <= j && j <= a.length);

  // Return true if and only if
  // a[p] is the smallest element in the range from i to j-1.

  ... // Fill this in.
}
```

Finally, rewrite the algorithm to use a while-loop rather than a for-loop, and assert the invariants at the appropriate places (read the previous handouts if you have forgotten where the appropriate places are):

```
void selectionSort(int [] a)
{
  // Pre-condition:
  assert(a != null);

  int i = 0;

  assert(initialSortedAndSmallerThanOthers(a,i));

  while (i < a.length)
  {
    assert(initialSortedAndSmallerThanOthers(a,i));

    int j = i+1;
    int p = i;

    assert(positionWithSmallest(a,i,j,p));

    while (j < a.length)
      {
        assert(positionWithSmallest(a,i,j,p));

        if (a[j] < a[p])
           p = j;

        j++;

        assert(positionWithSmallest(a,i,j,p));
      }

    assert(positionWithSmallest(a,i,j,p));

    swap(a,i,p);
    i++;

    assert(initialSortedAndSmallerThanOthers(a,i));
    }

  assert(initialSortedAndSmallerThanOthers(a,i));

  // Post-condition:
  assert(isSorted(a));
}
```

Your task. Fill the dots.

Run-time analysis. We count the number of comparisons as a function of the length n of the given array. We consider sub-arrays of growing length, starting from zero, until the sub-array is the whole array. For each element outside the sub-array, we perform one comparison to find the least element outside the array. So in the first stage we have $n-1$ comparisons to find the smallest element, $n-2$ comparisons to find the second smallest, and so on, and hence the total number of comparisons is

$$(n-1)+(n-2)+(n-3)+\cdots+3+2+1.$$

It can be shown that this number of comparisons is equal to

$$n(n-1)/2.$$

(We proved this in the lecture using an $n \times (n-1)$ grid of dots, where half of the number of dots, namely those on one side of the larger diagonal, is given by $1+2+\cdots(n-1)$. This can also be proved using mathematical *induction*.) Also the run-time is proportional to $n(n-1)/2 \approx n^2/2$. And because we say *proportional*, the division by 2 can be removed, and we can say that the run-time is approximately proportional to n^2. This can be made mathematically rigorous introducing the so-called O- and Θ-notations, which you will learn in due course.

Exercise. Argue that selection sort performs $n-1$ swaps.

4 Quick sort

The quick sort algorithm is more complicated:

1. It uses an auxiliary algorithm called `partition`.
2. It uses recursion, reflecting the fact that this is a divide-and-conquer approach.

Non-empty sub-arrays. Given an array `a`, we wish to consider sub-arrays of `a`, given by valid indices `first <= last`. Such sub-arrays are non-empty.

Empty sub-arrays. We also allow the case `first = last+1`, which describes empty sub-arrays. When `first=0`, we have `last = -1`, which is not a valid index.

Partitioned sub-arrays. We say that a non-empty sub-array of an array `a`, given by `first <= last`, is partitioned at position `p` with `first <= p <= last` if and only if the following conditions hold:

1. For every valid index `i` with `first <= i < p` (if any), we have `a[i] < a[p]`. These are the indices from `first` to `p-1`.
2. For every valid index `i` with `p < i <= last` (if any), we have `a[p] <= a[i]`. These are the indices from `p+1` to `last`.

Checking the partitioning specification with an algorithm.

```
boolean partitioned(int [] a, int first, int p, int last)
{
    assert(a != null);
    assert(0 <= first && first <= p && p <= last && last < a.length);

    int x = a[p];

    // Elements in positions < p are < x:

    for(int i = first; i < p; i++)
        if (a[i] >= x)
            return false;

    // Elements in positions > p are >= x:

    for(int i = p+1; i <= last; i++)
        if (a[i] < x)
            return false;

    // If both tests pass:

    return true;
}
```

Partitioning. The above says when a sub-array is partitioned at a given position. But how do we get an array to be partitioned? We choose an element in any position of the sub-array, and re-arrange the sub-array so that the partitioning condition holds. Typically, the first element is chosen. If another element is desired, then we can swap it with the first element, and hence there is no loss of generality in considering the first element always (see below). The re-arrangement can be performed in many ways,

which give rise to different, but correct, partitionings. We consider an algorithm attributed to Nico Lomuto by *R. Johnsonbaugh and M. Schaefer, Prentice Hall, Pearson Education, 2004, Section 6.2*, in their book *Algorithms*.

```
int partition(int [] a, int first, int last)
{
  // Pre-condition:
  assert(a != null);
  assert(0 <= first && first < last && last < a.length);

  int x = a[first];
  int h = first;

  for (int k = first + 1; k <= last; k++)
    {
    // Invariant: the elements from position first+1 to h are
    // smaller than x, and the elements from h+1 to k-1 are
    // bigger than or equal to x.

    if (a[k] < x)
      {
       h = h + 1;
       swap(a,h,k);
      }
    }

  swap(a, first, h);

  // Post-condition:
  assert(partitioned(a,first,h,last));

  return h;
}
```

Sorting. If `a` is partitioned at `first <= p <= last`, then the sub-array may not be sorted, but the element at position `p` is in the right position when the sub-array is sorted. Therefore, in order to sort the sub-array, it is enough to sort the sub-arrays from `first` to `p-1` and from `p+1` to `last`. This is precisely the idea behind quick sort:

1. To sort a sub-array from `first` to `last`, if the sub-array is empty or has just one element, we are done.

2. Otherwise, partition it at some position `p`.

3. Then sort the sub-arrays from `first` to `p-1` and from `p+1` to `last`.

4. The recursive calls are with strictly smaller sub-arrays, and so they eventually terminate when we reach empty or one-element arrays. (Both cases are possible.)

```
void quickSort(int [] a)
{
    // Pre-condition:
    assert(a != null);

    quickSort(a, 0, a.length - 1);

    // Post-condition:
    assert(isSorted(a));
}

void quickSort(int [] a, int first, int last)
{
  // Similar pre- and post-conditions (exercise).

  if (first < last)
    {
    int p = partition(a, first, last);

    assert(first <= p && p <= last);

    quickSort(a, first, p - 1);
    quickSort(a, p + 1, last);
    }
}
```

Here is a sample run, where elements in square brackets have reached their sorted positions, and where round brackets indicate partitioned sub-arrays:

```
40 48 62 28 31 9 74 88 52 40 89 63 25
(25 28 31 9 < 40 >= 48 74 88 52 40 89 63 62 )
(9 < 25 >= 31 28 ) [40] 48 74 88 52 40 89 63 62
[9] [25] (28 < 31 >= ) [40] 48 74 88 52 40 89 63 62
[9] [25] [28] [31] [40] (40 < 48 >= 88 52 74 89 63 62 )
[9] [25] [28] [31] [40] [40] [48] (62 52 74 63 < 88 >= 89 )
[9] [25] [28] [31] [40] [40] [48] (52 < 62 >= 74 63 ) [88] 89
[9] [25] [28] [31] [40] [40] [48] [52] [62] (63 < 74 >= ) [88] 89
9 25 28 31 40 40 48 52 62 63 74 88 89
```

Recursion tree. In the lecture we discussed that the recursive calls can be pictured as a tree, with the root as the main call, and the leaves indicating the end of the recursion. Please check your notes.

Run-time analysis. The exact run time will depend on the position the pivot element ends up when the array is partitioned.

Best case. If we are lucky, the pivot will end up in the the middle of the array always, in which case we will have even splittings. In this case, we will end up with a fairly balanced tree, which will have depth $\lceil \log n \rceil$. In the first level of the tree, we perform 1 partitioning for an array of length n. In the second level, we perform 2 partitionings for 2 arrays of length $n/2$. In the third level, we perform 4 partitionings for 4 arrays of length $n/4$, and so on. Hence putting together the work at each level, we perform approximately n comparisons at each level, and because there are approximately $\log n$ levels we perform approximately $n \log n$ comparisons.

Worst case. If the array is sorted to begin with, and we sort it with quick sort, then we get the worst case behaviour. This is because the pivot is chosen to be the first element, in which case we get the most uneven splitting, with one empty sub-array and a sub-array with $n - 1$ elements (the remaining element is the pivot). And then we will perform a splitting giving another empty sub-array, and an array of length $n - 2$. Hence we get

$$(n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1 = n(n-1)/2$$

as the depth of the recursion three, and also as the number of comparisons performed by quick sort. A similar uneven splitting occurs when the given array is sorted in reverse order. So quick sort, as it is presented above, doesn't work very efficiently for nearly sorted, or near=ly reversely sorted, arrays. One solution, if such arrays occur in our applications, is to chose not the first element as the pivot, but the element at the middle of the array. Another solution, which is better in all cases, is to use randomized quick sort, discussed below.

Average case. This is much harder to compute, and turns out to be approximately proportional to $n \log n$. There are $n!$ (n factorial) different arrangements of n elements, where the factorial is defined by

$$n! = 1 \times 2 \times 3 \times \cdots \times n.$$

A recursive definition is

$$0! = 1, \qquad (n+1)! = n! \times (n+1).$$

To say that the average run time is proportional to $n \log n$ amounts to saying that if you take the actual run time for each of the $n!$ arrangements, add them up, and divide by $n!$, we get a approximately $n \log n$ multiplied by a constant (which we are deliberately ignoring, but is there). This formula is of course very difficult to compute, but it can be done. Please check the given references. Another remark is that the number of arrays that give rise to the worst case behaviour is very small compared to $n!$. This doesn't mean that they are unlikely: as we have seen, nearly sorted lists are examples that lead to the worst case. To make this unlikely, we consider randomized quick sort.

Exercise. Quick sort performs many more swaps than selection sort, although it performs far fewer comparisons.

Randomized quick sort. To make the *expected time* to be $n \log n$, including "rogue" lists such as sorted, approximately sorted, and reversed arrays, one can use randomized quick sort. The difference is that rather than always choosing the first element of the sub-array as the pivot, we choose a random element of the sub-array.

```
Random randomPivot = null;

void randomizedQuickSort(int [] a)
{
  randomPivot = new Random();
  // Pre-condition:
  assert(a != null);

  randomizedQuickSort(a, 0, a.length - 1);

  // Post-condition:
  assert(isSorted(a));
}

void randomizedQuickSort(int [] a, int first, int last)
{
  // Similar pre- and post-conditions (exercise).

  if (first < last)
    {
     int p = randomizedPartition(a, first, last);
     assert(first <= p && p <= last);
     randomizedQuickSort(a, first, p - 1);
     randomizedQuickSort(a, p + 1, last);
    }
}
```

```
int randomizedPartition(int [] a, int first, int last)
{
  // Reduce the situation to the above partitioning algorithm, by
  // swapping the first element with a random element (in the range
  // from first to last).

  int k = first + randomPivot.nextInt(last - first + 1);
  assert(first <= k && k <= last);

  swap(a,first,k);
  return(partition(a,first,last));
}
```

Median quick sort. The obvious solution to deterministically avoid the uneven splitting would be to always choose the *median* of the sub-array. This is an element that would produce an even partition. However, it is expensive to find the median. It can be done in linear time, which then makes quick sort to be always $n \log n$, but also makes quick sort to be much slower in practice. What one could do in principle is to run the normal partitioning and if it produces a very uneven result we could repartition using the median. The following page is not perfect but has good information:

http://en.wikipedia.org/wiki/Selection_algorithm

5 Experimental results

Can be found in the program `SortTrace.java` available at the module website, which you should experiment with (as we did in the lecture).