Algorithms & Data Structures
Part 2 (weeks 6–10)
Set 2: Sorting

Konrad Dabrowski
konrad.dabrowski@durham.ac.uk

Online Office Hour:
Mondays 13:30–14:30
See Duo for the Zoom link

Table of Contents

We'll see two (naturally) **iterative** algorithms

- InsertionSort
- SelectionSort

and two (naturally) **recursive** ones

- MergeSort
- QuickSort

The latter based on "divide & conquer" algorithmic paradigm

Very different "flavours" of analyses
Will also see (and analyse) randomised QuickSort

InsertionSort

InsertionSort $(a_1, \ldots, a_n \in \mathbb{R}, n \geq 2)$

```
1: for j = 2 to n do
2:     x = a_j
3:     i = j − 1
4:     while i > 0 and a_i > x do
5:         a_{i+1} = a_i
6:         i = i − 1
7:     end while
8:     a_{i+1} = x
9: end for
```

We know:

- when j has certain value, it inserts the j-th element into already sorted sequence $a_1, \ldots, a_{j-1}$, yielding sorted $a_1, \ldots, a_j$
- can be proved correct by using invariant "after j-th iteration first $j + 1$ elements are in order"
- running time between $n - 1$ and $\frac{n(n-1)}{2}$ – worst case $O(n^2)$

SelectionSort

SelectionSort $(a_1, \ldots, a_n \in \mathbb{R}, n \geq 2)$

```
 1: for i = 1 to n − 1 do
 2:     elem = a_i
 3:     pos = i
 4:     for j = i + 1 to n do
 5:         if a_j < elem then
 6:             elem = a_j
 7:             pos = j
 8:         end if
 9:     end for
10:     swap a_i and a_pos
11: end for
```

SelectionSort

Invariant: after i-th iteration positions $1, \ldots, i$ contain the overall i many smallest elements in order

Not necessarily the first i elements (as it was in InsertionSort)!

In i-th iteration of outer loop, we search the i-th smallest element in remainder (positions $i + 1, \ldots, n$) of input and swap it into position i

- elem keeps track of current idea of **value** i-th smallest element
- pos keeps track of current idea of **position** of i-th smallest element

SelectionSort

Time complexity:

$$\begin{aligned}
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 &= \sum_{i=1}^{n-1} (n-i) \\
&= \left(\sum_{i=1}^{n-1} n \right) - \left(\sum_{i=1}^{n-1} i \right) \\
&= (n-1) \cdot n - \frac{n(n-1)}{2} \\
&= \frac{n(n-1)}{2} \\
&= O(n^2)
\end{aligned}$$

Recursive algorithms

MergeSort

The basic idea is tremendously simple:

- if given sequence of no more than one element then we're done
- otherwise (length$>$1), split sequence into two shorter sequences of equal length, sort them recursively, and merge the two resulting (sorted!!!) sequences

Assumption (WLOG, just for simplification):

- length of top-level input sequence is a power of two (e.g. 2,4,8,16,32,...)
- this allows for nice splitting into equal-sized sub-problems, e.g.
  - 128 becomes 2×64,
  - each 64 becomes 2×32,
  - each 32 becomes 2×16,
  - each 16 becomes 2×8,
  - each 8 becomes 2×4,
  - each 4 becomes 2×2,
  - each 2 becomes 2×1.

list **MergeSort** (list m)

```
1: if length(m) ≤ 1 then
2:     return m
3: end if

4: int middle = length(m) / 2
5: list left, right, leftsorted, rightsorted

6: left = m[1..middle]
7: right = m[middle+1..length(m)]

8: leftsorted = MergeSort(left)
9: rightsorted = MergeSort(right)

10: return Merge(leftsorted, rightsorted)
```

How to actually merge two sorted sequences?

Also simple. In fact, it's probably exactly what you would do yourself.
Suppose two **sorted** sequences given as arguments, say left and right.

Start with initially empty result sequence

If both left and right aren't empty then look at leftmost element from each, say ℓ_1 and r_1
If $\ell_1 < r_1$ then append ℓ_1 to result (and remove it from left), otherwise append r_1 to result (and remove it from right)
If either left or right is empty, append the entire other one to result.
Repeat until both empty.

list **Merge** (list left, list right)

```
1:  list result
2:  while length(left) > 0 or length(right) > 0 do
3:      if length(left) > 0 and length(right) > 0 then
4:          if first(left) ≤ first(right) then
5:              append first(left) to result
6:              left = rest(left)
7:          else
8:              append first(right) to result
9:              right = rest(right)
10:         end if
11:     else if length(left) > 0 then
12:         append left to result
13:         left = empty list
14:     else      // length(right) > 0
15:         append right to result
16:         right = empty list
17:     end if
18: end while
19: return result
```

A few facts

MergeSort...

- is probably the simplest recursive sorting algorithm
- its bad cases are a lot less bad than those of some other sorting algorithms that you know:
  - InsertionSort
  - SelectionSort
- its good cases, however, may be worse than those of some of the above (not all algorithms always take the same number of steps for each input)

- More technically, MergeSort always requires $\approx n \log(n)$ steps to sort any n numbers (will see proof later)
- InsertionSort may get away with $\approx n$ in particularly nice cases, but **will** require $\approx n^2$ for others
- n^2 is a lot worse than $n \log(n)$; e.g. for $n = 100k$: $n^2 = 10$ billion, whereas $n \log_2(n) \approx 1.5$ million
- You'll see lower bound proof later

Thought that's it about sorting?

Oh no, it isn't.
We're going to see QuickSort now.

Another example of a recursive sorting algorithms
Idea not too far away from MergeSort

Good/average case fine, but worst case poor (in fact, like SelectionSort)

QuickSort

Another example of divide & conquer:

- split input into two parts (but somewhat differently now)
- recursively sort them, one after the other
- concatenate the resulting, sorted subsequences

Notice how I say "concatenate" rather than "merge": QuickSort partitions differently from MergeSort in that it'll be guaranteed that any element in the "left" subproblem will be less than any in the "right"

QuickSort

Another example of divide & conquer:

- split input into two parts (but somewhat differently now)
- recursively sort them, one after the other
- **concatenate** the resulting, sorted subsequences

Notice how I say "concatenate" rather than "merge": QuickSort partitions differently from MergeSort in that it'll be guaranteed that any element in the "left" subproblem will be less than any in the "right"

At the beginning of each recursive call, QuickSort picks one element from the current sequence, the **pivot**.
The partitioning (into left and right subsequences) will be done w.r.t. this pivot:

- each element smaller than the pivot goes into the left part,
- each element bigger than the pivot goes into the right part;
- parts may have (very) different sizes.

In some sense,

- MergeSort does the "complicated" part (the merging) after the sorted subsequences come back from recursive calls (because we don't know anything about the relationship of elements in the left and right parts, other than they they each are sorted), and
- QuickSort does its difficult bit prior to recursing. This means that simple concatenation afterwards is OK.

The basic flow is as follows (assuming all elements are pairwise distinct).

QuickSort (int A[1...n], int left, int right)

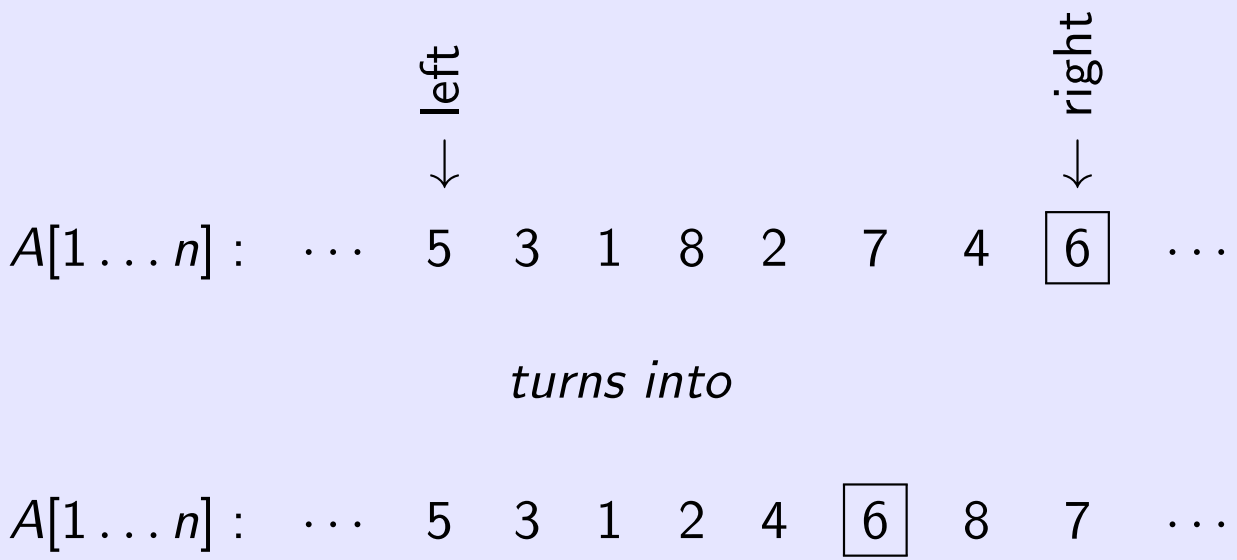
```
 1: if (left < right) then
 2:     // rearrange/partition in place
 3:     // return value "pivot" is index of pivot element
 4:     // in A[] after partitioning
 5:     pivot = Partition (A, left, right)

 6:     // Now:
 7:     // everything in A[left...pivot-1] is smaller than pivot
 8:     // everything in A[pivot+1...right] is bigger than pivot
 9:     QuickSort (A, left, pivot-1)
10:     QuickSort (A, pivot+1, right)
11: end if
```

It's entirely up to the function **Partition()** to select the actual pivot element. There's a vast number of different approaches.

The partitioning function

As I said, much depends on how pivot is actually being selected
Simplest (but certainly not best) version will always pick fixed position within current subsequence, e.g. right-most
In other words, when called as "`partition (A, left, right)`" then it may always pick `A[right]` as pivot:



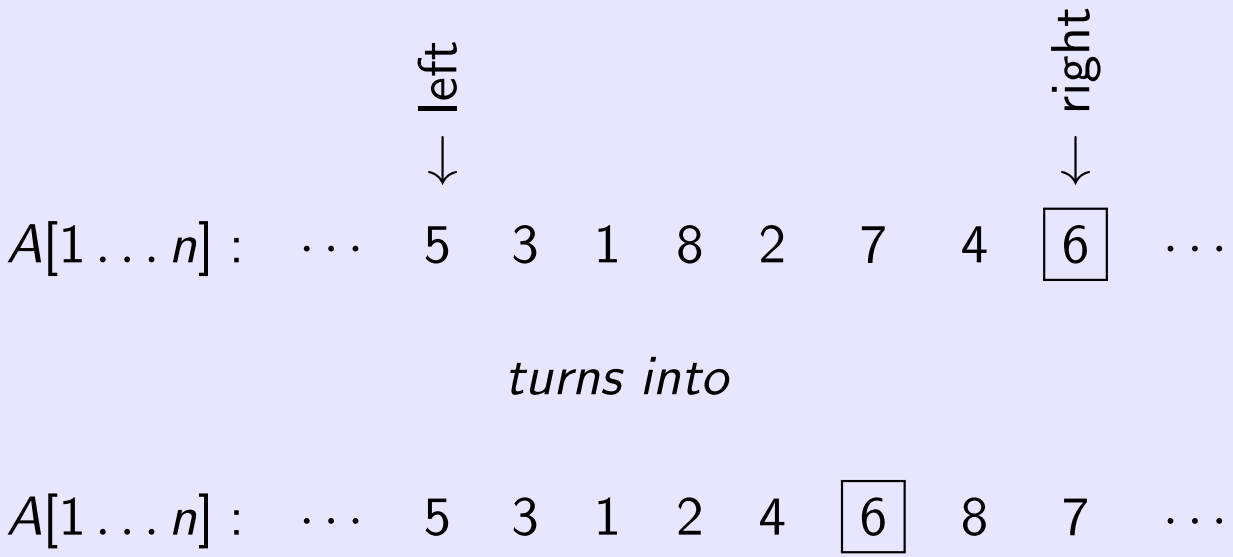
Partition() simply shoves everything smaller than the pivot to the left, and everything bigger to the right.

It does not sort!

Depending on the actual implementation, it may, or may not, rearrange among the smaller, or among the bigger, but strictly speaking, that's not (necessarily) part of the specification.

There may be the issue of *stability* though - it may be desirable to leave the relative order among the smaller (bigger) guys untouched.

Previous example



QuickSort (not partition) would then recurse into

- the "smaller-than-the-pivot" part $(5, 3, 1, 2, 4)$, and then
- the "bigger-than-the-pivot" part $(8, 7)$

If everything goes well (and it will), then after returning from those recursive calls, this part (i.e. A[left...right]) will look like

$$(1, 2, 3, 4, 5)\ 6\ (7, 8)$$

and will therefore be sorted. Such is the power of recursion!

Here's how the `partition` procedure can be implemented (x is the pivot **element** and is here chosen as right-most):

int **Partition** (A[1...n], int left, int right)

```
 1: int x = A[right]
 2: int i = left-1

 3: for j=left to right-1 do
 4:     if A[j] < x then
 5:         i = i+1
 6:         swap A[i] and A[j]
 7:     end if
 8: end for

 9: swap A[i+1] and A[right]
10: return i+1    // pivot position after partitioning
```

Table of Contents

Table of Contents