This lecture handout was prepared by Julian Garcia

# 1 Quicksort

## 1.1 Sorting problem

**Input:** A sequence of $n$ elements $\langle a_1, a_2, ..., a_n \rangle$
**Output:** A permutation of the original sequence, $\langle a'_1, a'_2, ..., a'_n \rangle$, such that $a'_1 \leq a'_2 \leq ... \leq a'_n$

- Sorting tipically brings to mind *arrays* of natural numbers, but we can sort any list of objects provided we have a way to compare objects.
- We want to sort **efficiently**, for instance by minimizing the number of comparisons performed or the memory that the sorting procedure uses.
- Sorting algorithms can be studied quite formally for efficiency and correctnes.
- Sometimes knowing a little bit about the data we expect can be very useful in making a good algorithm choice.

## 1.2 Some quicksort facts

- A prime example of **divide and conquer**: reduce the size of the problem and apply recursively on the smaller parts.
- It is recursive, the algorithm *calls itself*. We should provide an exit door, to ensure it stops. This is typically a subproblem that is trivial to solve (e.g. sort a list of one element, or an empty list).
- Sorts **in place**. No extra memory, auxiliary arrays or other data structures. Sorting in place is key to a proper quicksort implementation.
- Top 10 algorithms of the 20th century (according to SIAM)

## 1.3 General strategy of quicksort

- The input is an array $A[p..r]$, where $p$ denotes is the index of the first element of $A$ and $r$ denotes the index of the last element in $A$.
- We *partition* A, in two smaller sub-arrays. $A[p, q]$ and $A[q+1, r]$. This implies moving elements around in $A$ (**in place**) and coming up with an index $q$.
- The partition given by $q$ is such that the elements in the subarray $A[p, q]$ are smaller or equal than the elements in the subarray $A[q + 1, r]$.
- We sort the subarray $A[p, q]$ using quicksort, and then we sort the subarray $A[q+1, r]$, using quicksort.
- Since quicksort works *in place*, we do not have to combine or stich up the resulting ordered subarrays. We are done.

## 1.4 Hoare partition

Most of the work is actually done by the partition method. There are several ways to perform the partition. Below we implement the original algorithm as proposed by A.C.R Hoare. This is known as the Hoare partition, and is the partition method often found in most textbooks.

```
def partition(an_array, p, r):
    """
    An array, p is the index of the first element of the sub_array we want to partition,
    and r is the index of the last element we want to partition.
    """
    #we choose a pivot x, around which we partition (first element of the subarray)
    x = an_array[p]
    #i is the index going forward
    i = p-1
    #j is the index going backwards
    j = r+1
    while True:
        #we first go backwards decreasing j
        while True:
            j-=1
            #until we find an element <= than the pivot
            if (an_array[j] <= x):
                break
        #then we go forward increasing i
        while True:
            i+=1
            #until we find an element >= than the pivot
            if (an_array[i] >= x):
                break
        if i < j:
            #we exchange the elements at positions i and j
            an_array[i], an_array[j] = an_array[j], an_array[i]
        #we are done when j and i have crossed.
        else:
            return j
```

Having implemented the partition, quicksort is straightforward:

```
def quicksort(an_array, p, r):
    #if the problem is not trivial
    if p < r:
        #divide
        q = partition(an_array, p, r)
        #and conquer
        quicksort(an_array, p, q)
        quicksort(an_array, q+1, r)
```

Let us sort the array we used as an example in class:

```
an_array = [4,1,5,3,7,2,6]
quicksort(an_array, 0, len(an_array)-1)
print an_array
```

```
[1, 2, 3, 4, 5, 6, 7]
```

We can also check the first partition of this array, the one we used as an example in class:

```
#before partition
an_array = [4,1,5,3,7,2,6]
print "Array before the partition A = {} ".format(an_array)
q = partition(an_array, 0, len(an_array)-1)
```

```
#after partition
print "Array after partition A = {}".format(an_array)
print "q = {}".format(q)
print "Left-hand sub-array A[p, q] = {}".format(an_array[0:q+1])
print "Right-hand sub-array A[q+1, r] = {}".format(an_array[q+1:len(an_array)])
```

```
Array before the partition A = [4, 1, 5, 3, 7, 2, 6]
Array after partition A = [2, 1, 3, 5, 7, 4, 6]
q = 2
Left-hand sub-array A[p, q] = [2, 1, 3]
Right-hand sub-array A[q+1, r] = [5, 7, 4, 6]
```

Note that we are not returning any array in any of the functions involved, nor creating any additional sublists. This is because everything happens **in place** by moving elements around in A and determining the index q. To make sure that this actually works let us try it out on a couple of random arrays:

```
import random
random_array = [random.randint(0,100) for r in xrange(10)]
print "Unsorted = {}".format(random_array)
quicksort(random_array, 0, 9)
print "Sorted = {}".format(random_array)
```

```
Unsorted = [72, 29, 50, 14, 79, 82, 52, 88, 20, 23]
Sorted = [14, 20, 23, 29, 50, 52, 72, 79, 82, 88]
```

```
random_array = [random.randint(0,100) for r in xrange(10)]
print "Unsorted = {}".format(random_array)
quicksort(random_array, 0, 9)
print "Sorted = {}".format(random_array)
```

```
Unsorted = [3, 6, 55, 92, 84, 95, 79, 84, 49, 22]
Sorted = [3, 6, 22, 49, 55, 79, 84, 84, 92, 95]
```

This partition that we saw in class is known as the Hoare partition. It is named after C.A.R Hoare, the inventor of quicksort. It dates back to 1962.

### 1.4.1 Intuition about performance.

- The partition is actually a very efficient way to compare the elements in the array. We choose one pivot and compare everything to it, instead of comparing all elements against each other.
- Note that in the **best case**, we will partition in halves all along. This occurs when the pivots happens to fall on the median of the array.
- The **worst case** occurs for a sorted array, in which the partitions will be of size 1 and $n-1$
- If the input is random, we would expect the worst case to be very unlikely. If all the permutations are equali likely, it is reasonable to expect that the pivot will often fall somewhere in the middle of the array.
- For quicksort, the best case is very good and the worst case is not so good, but on average we expect things to be closer to the best case.

## 1.5 Other partitions

### 1.5.1 Lomuto partition

Next to the Hoare partition, another popular way to divide the sorting problem is given by the **Lomuto** partition. We will not cover this in class, but it is important to grasp the main idea. It is also key to understand that there are several ways to perform a *in-place* partition for quicksort. The main difference is that the partition happens around the last element of the array ($x = A[r]$). Partitioning an array $A[p..r]$ will return the pivot $q$. The smaller sub-arrays are $A[p..q-1]$ and $A[q+1..R]$.

```
def lomuto_partition(an_array, p, r):
    #pick the pivot
    x = an_array[r]
    #from p to i elements <= x
    i = p-1
    for j in xrange(p, r):
        if an_array[j] <= x:
            #from i to j elements > x
            i+=1
            #exchange elements between regions
            an_array[i],an_array[j] = an_array[j],an_array[i]
    #put the pivot in place, from position r to position i+1
    an_array[i+1],an_array[r] = an_array[r],an_array[i+1]
    return i+1
```

Note that this implementation is more compact. We redefine the quicksort to work with this partition in the following manner...

```
def quicksort_lomuto(an_array, p, r):
    if p < r:
        q = lomuto_partition(an_array, p, r)
        quicksort_lomuto(an_array, p, q-1)
        quicksort_lomuto(an_array, q+1, r)
```

Let us sort the example we saw in class, now with the Lomuto partition:

```
an_array = [4,1,5,3,7,2,6]
quicksort_lomuto(an_array, 0, len(an_array)-1)
print an_array
```

```
[1, 2, 3, 4, 5, 6, 7]
```

We can also check the first partition, like we did before:

```
#before
an_array = [4,1,5,3,7,2,6]
print "Array before the partition A = {} ".format(an_array)
q = lomuto_partition(an_array, 0, len(an_array)-1)
#after
print "Array after partition A = {}".format(an_array)
print "q = {}".format(q)
print "Left-hand sub-array A[p, q-1] = {}".format(an_array[0:q])
print "A[q] = [{}]".format(an_array[q])
print "Right-hand sub-array A[q+1, r] = {}".format(an_array[q+1:len(an_array)])
```

```
Array before the partition A = [4, 1, 5, 3, 7, 2, 6]
Array after partition A = [4, 1, 5, 3, 2, 6, 7]
q = 5
Left-hand sub-array A[p, q-1] = [4, 1, 5, 3, 2]
A[q] = [6]
Right-hand sub-array A[q+1, r] = [7]
```

Notice that the partitions performed by the Hoare and the Lomuto partition (on our example array) differ substantially, but both comply with the requirement of providing two smaller subproblems with the elements of one subproblem being all smaller than the elements of the other subproblem. We can also try with a random arrays:

```
import random
random_array = [random.randint(0,100) for r in xrange(10)]
print "Unsorted = {}".format(random_array)
quicksort(random_array, 0, 9)
print "Sorted = {}".format(random_array)
```

```
Unsorted = [68, 80, 6, 83, 57, 8, 94, 27, 19, 61]
Sorted = [6, 8, 19, 27, 57, 61, 68, 80, 83, 94]
```

I encourage you to try to implement your own partition methods. Partitions can be more intuitive, for instance, if we work with other data structures (e.g., linked lists instead of plain arrays). This may come at a cost in efficiency. It is difficult to beat the Hoare or the Lomuto partition, but it is hard to resist the invitation to tinker these methods.

## 2 Some literature


- Leiserson, Charles E., Ronald L. Rivest, and Clifford Stein. **Introduction to algorithms**. Edited by Thomas H. Cormen. The MIT press, 2001.
- Sedgewick, Robert. and Wayne, Kevin. **Algorithms**. Pearson Education, 2011.
- Hoare, Charles AR. *Quicksort.* The Computer Journal 5, no. 1 (1962): 10-16.


This lecture notes were generated with IPython notebook.