Algorithms & Data Structures

Dr Michele Sevegnani

Joao Almeida-Domingues[*]

University of Glasgow

January 14th, 2020 – March 25th, 2020

Contents

[*]2334590D@student.gla.ac.uk

These lecture notes were collated by me from a mixture of sources , the two main sources being the lecture notes provided by the lecturer and the content presented in-lecture. All other referenced material (if used) can be found in the *Bibliography* and *References* sections.

The primary goal of these notes is to function as a succinct but comprehensive revision aid, hence if you came by them via a search engine , please note that they're not intended to be a reflection of the quality of the materials referenced or the content lectured.

Lastly, with regards to formatting, the pdf doc was typeset in LaTeX, using a modified version of Stefano Maggiolo's class

1 Analysis Techniques

running time , experimental analysis , theoretical analysis , primitive operations , growth rate of running time , Big-Oh

1.1 definition. Data Structure is a systematic way of organising and accessing data

1.2 definition. Algorithm is a step-by-step procedure for performing some task in a finite amount of data

1.3 definition. Running Time is the amount of time it takes an algorithm to execute in full

The main criteria used to compare different algorithms are running time and memory usage. In general the running time of an algorithm increases with the input size, and may vary depending on the hardware and software environments on which it is run. When comparing algorithms one aims to control those variables and express a relation between their run times and inputs via some function

1.1 *Experimental Analysis vs Theoretical Analysis*

One way to study the efficiency of an algorithm is to run an empirical study. The dependent and independent variables are set, several trials are run a with appropriate inputs and then a statistical analysis is carried out on the output of the experiments.

Drawbacks

Though the experiment itself is relatively straightforward to run (once the algorithm is implemented), the analysis can be quite quite complicated to perform. Especially given that it can be hard to know which inputs are appropriate to use and , more importantly , given that fully implementing a complex algorithm is hard work and time-consuming. Hence, if possible a higher-level analysis is performed, if an algorithm can be deemed to be inferior by application of theoretical methods then no experimental analysis is required

So, when developing theoretical methods of analysis one's goal is to overcome the drawbacks mentioned above in order to achieve the following:

1. System Independence
2. Input Coverage
3. High-Level Description

1.4 definition. Primitive Operations are *low-level* instructions with constant execution time (e.g. variable assignment, function call)

In order to express running time as a function of input size we use primitive operations, which are identifiable from abstract implementations (like pseudocode) and taken to have a constant time of completion. In this way, one

can compare the total number t of ops for a given implementation, since by assuming a constant time for all ops one can assume that the total running time will be *proportional* to t

Ideally the average of all possible inputs would be used to characterize a given algorithm. However,finding the average often involves finding an appropriate distribution for the sample inputs. Hence, we characterize it instead in terms of its *worst case* input which is far easier to identify.

1.5 remark. Another advantage is that minimising for the worst case by definition implies that we're optimising the running time for all other possible inputs

1.2 *Common Functions*

This relation between input and running time is often expressed in terms of one of the following 7 functions

Constant

The simplest function of them all is the constant function which simply assigns a given constant c to any input n . It is particularly useful, since it allows us to express the *number of steps* needed to perform a basic operation

$$f(n) = c, \forall n \in \text{ input set}$$

1.6 remark. The essential constant functions if $g(n) = 1$ given that we can express any other constant function in the form $g(n)f(n)$

Logarithm

The logarithmic function, in particular $\log_2$, pops up all the time. A logarithmic runtime is characterized by larger differences in runtime for smaller inputs, with a significant decrease for larger inputs.

1.7 definition. ceiling (of x) the smallest integer greater than x

1.8 notation. $\lceil x \rceil$

We can think of the ceiling function as an approximation of x. We can use it in a similar manner to approximate any given algorithm. By definition $\log_b(x)$ is just the power to which b has to be raised to give x. Hence, we define $\lceil \log_b x \rceil$ as the smallest number for which b has to be raised so that it includes x. So, we can divide repeatedly divide x by b until we get a number less or equal to 1

1.9 example. $\lceil \log_2 12 \rceil = 4$ since $2^3 < 12 < 2^4$

Linear

The linear function assigns the input to itself. It is useful to characterize single basic operations on n elements

$$f(n) = n$$

N-Log-N

For any given input n it assigns n times $\log(n)$

$$f(n) = n\log(n)$$

1.3 *Quadratic*

The quadratic function appears primarily in algorithms with nested loops, since the inner loop performs an operation n times and the outer loop will repeat each loop a *linear* number of times

$$f(n) = n^2$$

Polynomials

A more general class which subsumes the quadratic ($d = 2$), linear ($d = 1$) and constant ($d = 0$) functions, where d is the degree of the polynomial which corresponds to the largest exponent in the polynomial expression. In general, polynomials with lower degrees have better running times

$$f(n) = \sum_{i=0}^{d} a_i n^i$$

Exponential

Exponential time polynomials are generally the worst-case running time for an algorithm, since they grow very rapidly. As an example, take a loop which doubles the number of operations it performs with every iteration. Then, the total number of operations it performs will be 2^n

1.4 *Growth Rates*

If we take $f(n)$ to be the function which gives the total number of operations in the *worst-case*, and a z to be the time taken by the fastest and slowest primitive op , respectively. Then, the worst-case running time $T(n)$ for the algorithm is bounded by the two linear functions $af(n); zf(n)$, i.e

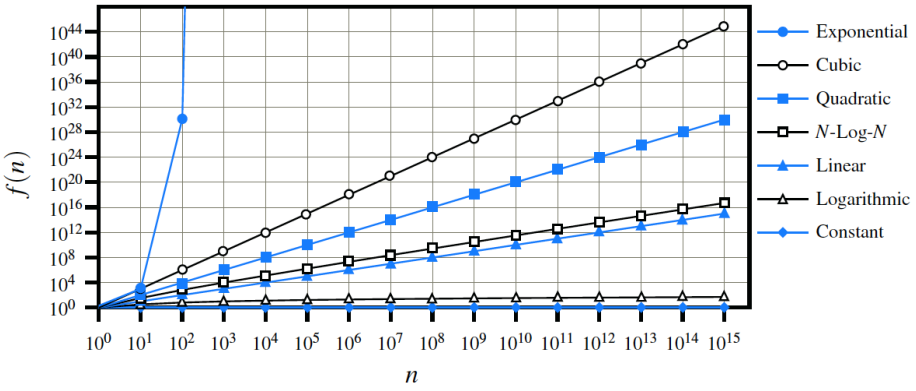
$$af(n) \leq T(n) \leq bf(n)$$

Note that the growth rate of the worst-case running time is an intrinsic property of the algorithm which is not affected by by hardware or software environments. These factors affect $T(n)$ by a constant factor. Hence, when

running an asymptotic analysis one can **disregard** constant and lower-degree terms

1.10 remark. Ideally we want operations on data structures to run in times proportional to the constant or log functions, and algorithms to run in linear or n-log-n

1.11 definition. Asymptotic Analysis analysing how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases with the bound



1.5 *Big-Oh Notation*

1.12 definition. Big-Oh Notation we say that *"$f(n)$ is big-Oh of $g(n)$"* if $\exists c \in \mathbb{R} s.t f(n) \leq cg(n)$, for $n \geq n_0$

1.13 notation. $f(n) is O\left(g(n)\right)$

In essence we make use of the fact that growth run time is not affected by constant factors, and encode this into function notation. In effect bounding the function after a given input size n_0 by another function ; i.e $f(x)$ is strictly less than or equal to $g(x)$ up to to a constant factor and in the *asymptotic sense*. Hence, we can use $g(x)$ to approximate/characterize $f(x)$

We want this relation to be expressed in the simplest terms possible. Obviously it is easy to find such a function if one aims for the moon, i.e clearly $3n^2 + 2$ is $O(n^{10})$ this is however not very useful. Instead, simplifying the first expression by getting read of constant factors , we see that we can be sure that it is for sure $O(n^2) < O(n^{10})$, which is therefore a better approximation

1.14 remark. We want the *simplest* expression of the class of bounding functions (n^2 Vs $4n^2$) and the *smallest* possible class (n^2 Vs n^{10})

1.15 proposition. *For any polynomial $p(n)$ of degree d , $p(n) \in O\left(n^d\right)$*

1.16 proposition. $T_1(n) \in O(f(n)), T_2(n) \in O(g(n)) \implies T_1(n) + T_2(n) \in O(max(f(n), g(n)))$

1.17 proposition. $T_1(n) \in O(f(n))$ *and* $T_2(n) \in O(g(n)) \implies T_1(n)T_2(n) = O(f(n)g(n))$

Proof. $T_1T_2 = klf(n)g(n)$, for constants k, l . Hence, ignoring the constants we have the expected result

1.18 proposition. $T(n) = (\log n)^k \implies T(n) = O(n)$

Proof.

$$T(n) \in O(f(n)) \iff \lim_{n\to\infty}(T(n)/f(n)) = 0)$$

By *L'Hopital's*,

$$lim_{n\to\infty}(f(n)/g(n)) = lim_{n\to\infty}(f\prime(n)/g\prime(n))$$

Take $f(n) = (\log n)^{k+1}$ and $g(n) = n$. Then, by induction on k ,

$$(\log n)^1 = \log n \in O(n) \text{ and } (\log n)^k \in O(n)$$

Hence,

$$\lim_{n\to\infty}\left(\frac{(\log n)^{k+1}}{n}\right) = 0$$

$$\lim_{n\to\infty}\left(\frac{(\log n)^{k+1}}{n}\right) = \lim_{n\to\infty}\left(\frac{(k+1)(\log n)^k}{n}\right) = 0$$

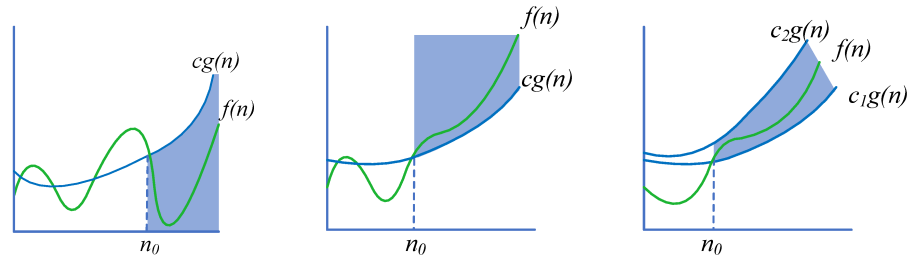
Therefore, the result follows by induction

1.6 *Big-Omega & Big-Theta*

1.19 definition. $\Omega(n)\ f(n) \geq cg(n)$

1.20 definition. $\Theta(n)\ c\prime g(n) \leq f(n) \leq c\prime\prime g(n)$

Similarly to $O(n)$, these provide upper and upper+lower bounds respectively



(a) Big-O Notation (b) Big-Omega Notation (c) Big-Theta Notation

Figure 1: Asymptotic Notation `shorturl.at/xAFQ1`

1.7 *Little-oh*

1.21 definition. $o(g(n))$ $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

We say that $f(x)$ is $o(g(x))$ if $g(x)$ grows much faster than $f(x)$, i.e. if $f(n)$ is $O(g(n))$ and $f(n)$ is not $\Omega(g(n))$

1.8 *Computing Running Times*

The following, follows from the propositions above

1. **Loops :** at most the running time of its contents times the number of iterations

2. **Nested Loops :** it should be analysed inside out. Take the runtime of the expression and multiply it by the product of of the sizes of all the loops

3. **Consecutive Statements :** add

4. **If-then-else :** at most the time of the test condition with the maximum of the running times of the two branches

1.22 example.

```
sum = 0;
for (i=1; i<=n; i++)
        sum += n;
```

See more examples here

Analysis of Insertion-Sort

1.23 example. Insertion-Sort

Algorithm 1: Insertion-Sort

Data: Array of integers : A
Result: Permutation of A such that A[0] $\leq$ A[1] $\leq \ldots$ A[n-1]

```
1  Insert(A)
2  for j = 1 to n − 1 do
                                                        /* n */
3      key := A[j]                                      /* n − 1 */
4
5      i := j-1                                         /* n − 1 */
6
7      while i ≥ 0 and A[i] > key do
                                                        /* Σ_1^{n−1} t_j */
8          A[i+1] := A[i]                               /* Σ_1^{n−1} (t_j − 1) */
9
10         i := i-1                                     /* Σ_1^{n−1} (t_j − 1) */
11
12     end
13     A[i+1] := key                                    /* n − 1 */
14
15 end
```

Hence, summing the individual operations

$$T(n) = n + (n-1) + (n-1) + \sum_{1}^{n-1} t_j + \sum_{1}^{n-1} (t_j - 1) + \sum_{1}^{n-1} (t_j - 1) + (n-1)$$

Best Case: A is already sorted , which means that the while loop is never executed and $t_j = 1$. Hence,

$$T(n) = n + (n-1) + (n-1) + (n-1) + 0 + 0 + (n-1) = O(n)$$

Worst Case: At every iteration we shift j elements : $t_j = j$. So,

$$\sum_{1}^{n-1} t_j = \sum_{1}^{n-1} j = \frac{n(n-1)}{2} = O(n^2)$$

$$\sum_{1}^{n-1} t_j - 1 = \sum_{1}^{n-1} (j-1) = \frac{n(n-1)}{2} - (n-1) = O(n^2)$$

Hence,

$$T(n) = n + (n-1) + (n-1) + O(n^2) + O(n^2) + O(n^2) + (n-1) = O(^2)$$

2 Recursive Algorithms

recursion traces , linear recursion , binary recursion , tail recursion , recursion trees , incremental algorithms , divide-and-conquer algorithms , merge-sort

2.1 definition. Recursive Function is a function which refers to itself in its definition

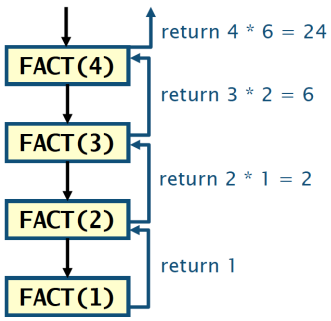
A classical example of a recursive faction is the factorial function, which calls itself always with an argument decreased by one in each call, until the *base case* 1 is reached.

All recursive definitions share these 2 properties, i.e they all have a base case which tells us when to stop calling, and they must reduce the size of the data set with each call

2.1 *Recursion Trace*

A recursion trace allows us to visually trace the execution of recursive algorithms. We draw a box for each call with an arrow pointing downwards connected to the next call, and an arrow pointing upwards with the returning value of that call

2.2 example.



2.2 *Linear Recursion*

2.3 definition. Linear Recursion at most one recursive call with each iteration

2.4 remark. The space required to keep track of the recursive stack grows linearly with n

2.5 remark. Useful when we view a problem in terms of a first/last element plus the remaining set with the same structure (e.g sum array in haskell : `sum(l) = head(l) + sum(tail(l))`)

2.3 *Tail Recursion*

Though useful for designing short and elegant algorithms, recursive can have a high memory cost, since the state of each recursive call must be stored prior to being returned. This costs can sometimes be diminished by using a recursive operation and the very last of the algorithm

2.6 definition. Tail Recursion is linear and the recursive call is its very last operation

In tail recursive implementations, some variable is modified with each call. There's no need to wait until the last call, since the updated variable is immediately passed to the next recursive call. Hence, the state is passed onto the call chain instead of being saved in memory

Recursive ↔ Iterative

A common idiom is converting to/from tail recursive to iterative algorithms. This is achieved by iterating through recursive calls, rather than calling them explicitily

2.7 example. [4] **Recursive**

```
function recsum(x) {
    if (x === 1) {
        return x;
    } else {
        return x + recsum(x - 1);
    }
}

recsum(5)
5 + recsum(4)
5 + (4 + recsum(3))
5 + (4 + (3 + recsum(2)))
5 + (4 + (3 + (2 + recsum(1))))
5 + (4 + (3 + (2 + 1)))
15
```

Tail Recursive

```
function tailrecsum(x, running_total = 0) {
    if (x === 0) {
        return running_total;
    } else {
        return tailrecsum(x - 1, running_total + x);
    }
}
```

```
tailrecsum(5, 0)
tailrecsum(4, 5)
tailrecsum(3, 9)
tailrecsum(2, 12)
tailrecsum(1, 14)
```

Iterative

```
function tailrecsum(x, running_total = 0):
    while x >= 0:
        running_total += x
        x -= 1
    return running_total
```

2.4 *Binary Recursion*

2.8 definition. Binary Recursion when an algorithm makes two recursive calls

2.9 example. Fibonacci : FIB(n-1) + FIB(n-2)

2.10 remark. Highly inefficient. Note how the same computations are made several times over, in different calls. In the recursion tree there are $O(2^n)$ recursive calls, i.e exponential complexity

recall memoization technique from 1P

3 Algorithm Design Paradigms

3.1 *Incremental*

The solution of a problem is built one element at a time

3.1 example.
Sort sub array ; pick unvisited node ; add to sorted subarray

```
INSERTION–SORT(A)
    for j = 1 to n−1
        key := A[j]
        i := j−1
    while i >= 0 and A[i] > key
        A[i+1] := A[i]
        i := i−1
        A[i+1] := key
```

3.2 *Divide-and-Conquer*

Recursive algorithm, with 3 steps

1. Divide into smaller subproblems equivalent to the original
2. Conquer by solving subproblems recursively

3. Combine the solutions to the subproblems to give the general solution

3.2 example.

Merge-Sort (1) split the array into two subarrays of size $\frac{n}{2}$; (2) scan two smaller arrays simultaneously ; (3) add minimum to sorted array, incrementing the pointer of the array from where the value was removed; when fully scanned copy the remaining sub array to main array

4 Efficient Sorting

recursion , divide-and-conquer , merge , Merge-sort , Quick-Sort , Stable Solution , recurrence equations

4.1 definition. divide-and-conquer an algorithm design paradigm based on multi-branched recursion; the problem is recursively broken down into simpler problems until one is simple enough to be solved directly. The solutions to each sub-problem are then merged back together

4.1 *Merge Algorithm*

4.2 definition. Sentinel a number which is much larger than any other number in the array. It can be represented by `Integer.MAX_VALUE`

4.3 remark. A more general definition sees sentinel as any value which is interpreted as a condition to terminate the algorithm

4.4 definition. Stable an algorithm which preserves the input order of repeated elements

4.5 remark. note that this is only relevant if the repeated elements are somehow distinguishable (e.g deck of cards , (rank,value))

4.6 definition. Key for data being recorded as a tuple of values, the key is the value used to sort (e.g (name, surname) , K=S)

4.7 definition. Merging Algorithm an algorithm which takes two sorted sequences and returns a single combined sequence [5]

Overview

The merge algorithm takes an unsorted array and 3 indices, it then splits the array into two subarrays and copies each subarray into memory. Finally, it iterates over every element of the original array, for each iteration it compares the current elements of the 2 subarrays against each other, replacing the appropriate value into the original array and increasing the index of the subarray by 1

Formal Definition

Algorithm 2: Merge

```
   Data: Array A , indices p, q, r
   Result: sorted subarray A[p..r]
 1 Merge(A,p,q,r)
 2 n_1 := q - p + 1                    /* initialize subarrays' size */
 3
 4 n_2 := r - q
 5 L[o..n_1] := A[p..q]                          /* copy to subarrays */
 6
 7 R[n_2..r] := A[q+1..r]
 8 L[n_1] := ∞   /* add the sentinel values to end of subarrays */
 9
10 R[n_2] := ∞
11 i,j := 0
12 for k=p to r do
13 |
14 |  if L[i] ≤ R[j] then
15 |  |
16 |  |  A[k] := L[i]
17 |  |  i := i + 1
18 |  else
19 |  |
20 |  |  A[k] := R[j]
21 |  |  j := j + 1
22 end
```

Properties

- **Running Time :** $O(n)$, since the initialization of the L, R subarrays takes $O(n)$ and the loop is executed n times and contains only constant-time operations
- **Stable**
- **Memory Requirements :** $O(n)$ to store L, R

4.2 *Merge-Sort*

4.8 remark. See informal definition in previous section

Formal Definition

Algorithm 3: Merge-Sort Algorithm

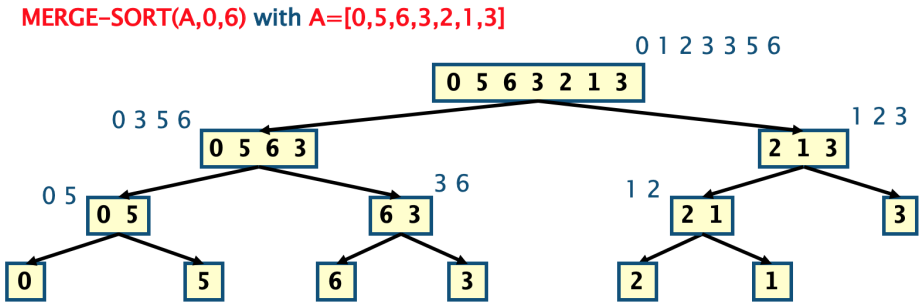
Data: Array A , indices p, r
Result: sorted array $A[p..r]$

```
1 Merge-Sort(A,p,r)
2 if p ¡ r then
3 |   q := (p + r) / 2
4 |   Merge-Sort (A,p,q)
5 |   Merge-Sort(A,q+1,r)
6 |   Merge(A,p,q,r)
```

Execution & Properties

4.9 remark. to sort an array A with n elements, we would call `MERGE-SORT(A,0,n-1)`

Note how the algorithm gives rise to a binary recursive tree, since we split each subarray into 2 with each iteration until we hit the stopping condition (single element array). So, the first subarray is entirely sorted, before the second `MERGE-SORT` call. The two *half-subarrays* returned from each call are then merged by calling `MERGE`. The process is then repeated in the right branch



- **Stable**
- **Memory** $O(n)$
- **Running Time** $O(n \log n)$

Improvements

In-place Merge

Bottom-up Iterative

`IS` for small n

4.3 *Recurrence Equations*

4.10 definition. Recurrence Equation describe the overall running time of a problem of size n in terms of the running time on smaller inputs

Recursive algorithms can be described via *recurrence equations*. Take $t(n)$ to denote the *worst-case* running time of `MERGE-SORT(n)`, then we can characterize $t(n)$ by means of an equation where $t(n)$ is recursively expressed in terms of itself.

If `MERGE-SORT(n)` has running time $T(n)$ then, each recursive call will `MERGE-SORT(n/2)` will run on $T(n/2)$ time. Hence, we can define $T(n)$ recursively as follows

$$T(n) = \begin{cases} b & n \leq 1 \quad \text{(base case)} \\ 2T(n/2) + cn \end{cases}$$

Iterative Method

Though correct, a more informative definition will be one that does not involve $T(n)$ itself. From its *closed-form* characterization, we can define it in *big-Oh* terms. We do this, by iterative substitution on the RHS of the recurrence relation until the base case is reached

Continuing with the expression above, if we substitute n by $n/2$ in the RHS we essentially double the children arrays, and we get

$$\begin{aligned} T(n) &= 2(2T((n/2)/2) + c(n/2)) + cn \\ &= 2^2 t(n/2^2) + 2(cn/2) + cn \\ &= 2^2 t(n/2^2) + 2cn \end{aligned}$$

If we repeat the process, assuming that n is relatively large, then we find the general pattern

$$t(n) = 2^i t(n/2^i) + icn$$

In order to determine the base case, we look at the original definition, and observe that we need to find $t(n)$, for $n \leq 1$. Given our general argument $\frac{n}{2^i}$ we need $2^i = n \iff i = \log n$. Hence, substituting i by $\log n$ gives

$$\begin{aligned} t(n) &= 2^{\log n} t(n/2^{\log n}) + cn \log n \\ &= nt(1) + cn \log n \\ &= nb + cn \log n \end{aligned}$$

Lastly, since b, c are constants, we have shown that $t(n)$ is $O(n \log n)$

Master Method

There is a general form of recurrence relation that arises in the analysis of the

divide-and-conquer algorithms

$$T(n) = aT(n/b) + f(n) \qquad a, b \geq 1$$

Then, for $f(n) = \Theta(n^c)$, the solution will be one of the following 3 cases

1. $c < \log_b a$ then $T(n) = \Theta\left(n^{\log_b a}\right)$
2. $c = \log_b a$ then $T(n) = \Theta(n \leq \log n)$
3. $c > \log_{b2} a$ then $T(n) = \Theta(f(n))$

For `MERGE-SORT`, we have $a = 2, b = 2, f(n) = \Theta(n), c = 1$. Hence, the solution is given by case 2 , $\Theta(n^c \log n) = \Theta(n \log n)$

Tree Method

4.11 remark. See lecture 5 slides 52-60

4.4 *Quick-Sort*

Overview

The `Quick-Sort` algorithm is also of the *divide-and-conquer* type, the main difference from the previous one being that the bulk of the work happens in the *divide* stage

4.12 remark. The following partition method *Lomuto* is not as efficient as *Hoares'*

- **Divide :**

  given an array A and an index q we partition A into $A[lo..q-1]$ to $A[q+1..hi]$ with the former containing all elements smaller than $A[q]$ and the latter all the ones which are greater than $A[q]$

- **Conquer :**

  sort by recurring ; the initial call will branch out on one of the main subarrays and the second on the other

4.13 remark. No merge stage required, since each subarray will be sorted *in-place*

Informal Definition

- **Partition**

  lo, hi are passed on as arguments and represent the first and last index of A

  1. set i index to lo , which will keep track of the *less-than* subarray
  2. set the pivot to $A[hi]$
  3. set j index to lo, which will keep track of the *greater-equal* subarray
  4. scan the array from j to $hi - 1$

(a) compare the current element $A[j]$ with the pivot p
- $A[j] < p$: `i++` ;$A[i] \leftrightarrow A[j]$
- $A[j] \geq p$: keep scanning (i.e `j++`)

5. in order to create the L, R partitions we swap p with $A[i]$ (i.e the first element of R)
6. Lastly, we return i, the pivot index in order to be used as hi, lo index for the recurrence calls

- **Recur**

1. check that the base case has not been reached $lo < hi$
2. set p to be the return value of `PARTITION`
3. call `quick-sort` on L, i.e. call it on $A[lo..p-1]$
4. call `quick-sort` on R, i.e call it on $A[p+1..hi]$

4.14 remark. Note that on each call we exclude p from the subarrays, since it is already sorted. This is why there is no need for merging

Formal Definition

Algorithm 4: PARTITION

Data: Array A , indices lo, hi
Result: pivot index i

```
1  PARTITION(A,lo, hi)
2  pivot := A[hi]
3  i := lo
4  for j = lo to hi do
5      if A[j] < pivot then
6          SWAP(A[j], A[i])
7          i++
8      SWAP(A[i] , A[hi])
9  end
10 return i
```

Algorithm 5: Quick-Sort

Data: Array A, indices lo, hi
Result: sorted array $A[lo..hi]$

```
1  QUICKSORT(A,lo, hi)
2  if lo < hi then
3      p := PARTITION(A , lo, hi)
4      QUICKSORT(A , lo , p − 1)
5      QUICKSORT(A , p + 1, hi)
```

Properties

- **Not Stable**

- **Memory** 0 ; *in-place*
- **Running-Time** cost of partitioning is $O(n)$

  Best the pivot is in the middle and so each subarray is exactly $n/2$. Hence, $O(n \log n$

  Average approximately $O(n \log n)$

  Worst already sorted, which leads to a hugely unbalanced split $(0, n-1)$. $T(n) = T(n-1) + T(0) + O(n) = O(n^2)$

Possible Improvements

Switch to `INSERTION-SORT`

note that in the worst-case scenario it is much faster $O(n)$. So when calling `QUICKSORT` on a subarray with fewer than k elements, return without sorting, when the top-level call returns run `IS`

Tail Call Optimisation

only one recursive call ; hard to implement, but good compilers do it automatically

Iterative with Stack

3-Way `Quick-Sort`

split the initial array into 3 subarrays, the new one including the elements equal to the pivot

based on the dutch national flag algorithm

References

CS2 Software Design & Data Structures **calculating˙program˙running˙time**

CS2 Software Design & Data Structures. URL: https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/AnalProgram.html.

Divide-and-conquer algorithm **divideConquerWiki**

Divide-and-conquer algorithm. Jan. 2020. URL: https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm.

Goodrich et al.: Data Structures and Algorithms in Java, 6th Edition **goodrich˙tamassia˙2014**

Michael Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java, 6th Edition*. John Wiley & Sons, 2014.

Hochstein: What is tail recursion? **hochsteinSO**

Lorin HochsteinLorin Hochstein. *What is tail recursion?* Aug. 1958. URL: https://stackoverflow.com/questions/33923/what-is-tail-recursion.

Merging Algorithms **merge**

Merging Algorithms. URL: http://www.cs.rpi.edu/~musser/gp/algorithm-concepts/merge-algorithms-screen.pdf.