Introduction

Andres Mendez-Vazquez

September 14, 2020

To my dear Fabi... For her strength and understanding in those rainy days...

Contents

1 Introduction **3**
1.1 The Rise of the Algorithm Engineer . . . 3

2 The absolute definition of an algorithm **3**

3 Going from Bad to Good Algorithms **5**

4 Using Structural Induction for Proving Correctness **7**
4.1 Recursively Defined Sets and Structures . . . 8
4.2 Using Insertion Sort as an Example of Correctness . . . 9
4.3 Exercises about Correctness . . . 11

5 Counting Steps in an Algorithm **12**
5.1 Counting Exercises . . . 13

6 Best, Worst and Average Cases **14**
6.1 Worst Best Exercises . . . 15

References **17**

List of Algorithms

1 Example for the Kolmogorov definition . . . 5
2 Recursive Fibonacci Algorithm . . . 6
3 Iterative Fibonacci Algorithm . . . 8
4 The insertion sort algorithm . . . 10
5 Binary Search algorithm . . . 16
6 Shell Sort algorithm . . . 16

List of Figures

1 Here, we can notice the full recursive tree for the Fibonacci recursion 6
2 The first steps in the recursive definition of full trees . . . 9
3 Examples of binary search trees . . . 14
4 Array elements before the swapping . . . 15

1 Introduction

Why the importance of having good designs when dealing with algorithms? After all, many people assume that you only need to code to be a computer scientist. Thus, it is not necessary to bother with the design itself [1]. This is far from the truth because a computer scientist must be a designer of algorithms. Why? In order to build solid foundations for the software to be created and designed. Therefore, it is necessary for a computer scientist to be able to understand the analysis of algorithms in a deeper and meaningful way. Furthermore, the computer scientist has to have in mind the following process when solving problems:

1. Define and understand the problem at hand.
2. Use the necessary mathematical foundations to express this understanding.
3. Use this to obtain an initial solution.
4. Express this solution in pseudo-code.
5. Move that pseudo-code to a target language.
6. Understand the hardware to improve the code.
7. Finally return to 4, if you are able to find a better solution.

Even though the previous steps express a personal point of view, it is quite interesting to realize that great programmers tend to follow them.

1.1 The Rise of the Algorithm Engineer

It is more, nowadays there is an entire new type of engineer, the algorithm engineer which are skilled computer scientist which have the following tenants [2]:

1. A strong theoretical foundation is vital to computer science.
2. Theory can be enriched by practice.
3. Practice can be enriched by theory.

Finally, they are becoming in this era of high volumes of data in the new champions of any enterprise that wants to be successful.

2 The absolute definition of an algorithm

Although the history of algorithms is as old as the approximation of π [3], there is still a lack of an absolute definition of an algorithm. Moreover, the situation becomes more complex when you take in account:

1. Parallel Algorithms.
2. Distributed Algorithms.
3. Quantum Algorithms.
4. Approximation Algorithms.
5. Heuristics.

Thus, the quest for the absolute definition is still going on [4] in the hopes that one day, we will be able to define what is an algorithm. Nevertheless, manly because we want something simple, the Kolmogorov's definition will be used through this text.

Definition 1. (Kolmogorov) An algorithmic process has the following properties:

1. An algorithmic process splits into steps whose complexity is bounded in advance. This bound is independent of the input and the current state of the computation.
2. Each step consists of a direct and immediate transformation of the current state.
3. This transformation applies only to the active part of the state and does not alter the remainder of the state.
4. The size of the active part is bounded in advance.
5. The process runs until either the next step is impossible or a signal says the solution has been reached.

For example, given (Algorithm 1), we have:

- The state of the algorithm is maintained by the variables A and i.
- The active part is always an element $A[i]$ and i. In addition, the transformation only applies to those parts.
- Clearly each step is bounded in time while executing the instructions inside the loop.
- Finally, the process continues until i becomes 0.

Given the previous example, this definition is actually a really complete one. It allows to obtain a deeper understanding of what an algorithm is.

function *Bounded*

Input: integer n, array of integers $A\,[1..n]$

Output: array $A\,[1..n]$ with elements less than n

1. $i \leftarrow n$
2. while $i > 0$
3. $\quad A\,[i] \leftarrow n - i$
4. $\quad i \leftarrow i - 1$
5. return A

Algorithm 1: Example for the Kolmogorov definition

3 Going from Bad to Good Algorithms

An example of a bad design can be seen at the recursive Fibonacci algorithm [5].

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{if } n > 1 \\ 1 & \text{if } n = 1 \\ 0 & \text{if } n = 0 \end{cases}$$

Each number in the Fibonacci sequence can be calculated using the previous two numbers. Thus, naively, one could use a recursion (Algorithm 2) in a computer to calculate the Fibonacci number n.

Although it is possible to calculate the Fibonacci value, the number of steps increases exponentially with respect to n. This is more poignant while looking at the full partial recursive tree of the Fibonacci recursion (Fig. 1).

function *Recursive Fibonacci*

Input: integer n

Output: the Fibonacci number F_n

1. if $n == 0$
2. return 0
3. if $n == 1$
4. return 1
5. return $fib1\,(n-1) + fib1\,(n-2)$

Algorithm 2: Recursive Fibonacci Algorithm

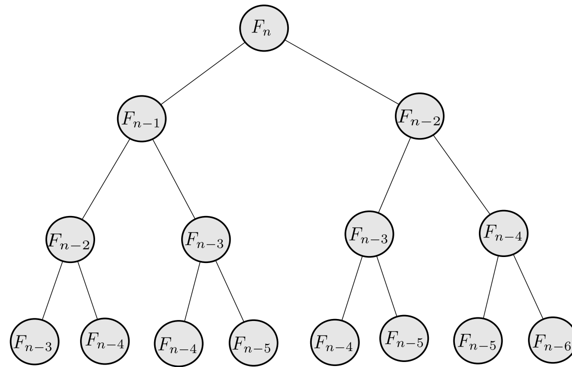

Figure 1: Here, we can notice the full recursive tree for the Fibonacci recursion

Basically, we have a full binary tree, where the leaves are functions receiving 0 or 1 values. Thus, the number of nodes per level is equal to 2^h with $h = 0$ at the root. Then, the total number of nodes in this recursion is equal to the following sequence:

$$1 + 2^1 + 2^2 + ... + 2^n, \tag{1}$$

which is a geometric sum equal to

$$1 + 2^1 + 2^2 + ... + 2^n = \frac{1 - 2^{n+1}}{1 - 2} = 2^{n+1} - 1. \tag{2}$$

Now, given our knowledge of recursive functions, it is possible to represent the work done as:

$$T(n) = T(n-1) + T(n-2) + \text{Some Work}.$$

Here, "Some Work" is equal to the test at the base cases and the addition at the end of the recursion (Algorithm 2). Thus, "Some Work" is equal to 3 steps of work i.e. two tests and one addition. Therefore, given that each node is doing a work of 3 steps, the total work is equal to $3 \times (2^n - 1)$ steps. Additionally, it is possible to prove the following inequality between the recursive algorithm and the Fibonacci number:

$$T(n) \geq F_n. \tag{3}$$

Not only that, Fibonacci numbers grow almost as fast as powers of 2, and in general $F_n \approx 2^{0.694n}$.

Now, while looking at the following sequence of Fibonacci numbers, it is possible to notice something.

$$\begin{aligned} F_2 &= F_1 + F_0 \\ F_3 &= F_2 + F_1 \\ F_4 &= F_3 + F_2 \\ F_5 &= F_4 + F_3 \\ &\vdots \end{aligned}$$

Surprised!!! Yes, this simple example gives a different way to calculate the Fibonacci numbers. One that requires the use of extra memory to avoid the use of recursion. Exactly the same trick used in Dynamic Programming [6]. Thus, if we want to obtain F_n , we can start with F_2, then F_3 and so on. Thus, instead on relaying in a recursive algorithm, we can use an iterative procedure to calculate F_n (Algorithm 3).

In this case, the number of total steps is equal to

$$T(n) = 1 + n + 1 + 2 + n + (n-1) + 1 = 3n + 4. \tag{4}$$

Quite an improvement!!! The final cautionary tale, we need to be careful when designing algorithms!!! Because the slowness in modern machines when switch from one function framework to another function framework. One reason why threads are designed to use the same framework.

4 Using Structural Induction for Proving Correctness

Now, we have a huge problem! How do we prove the correctness of an iterative procedure? After all, we are substituting a recursion with an equivalent iterative procedure for better performance. For this, we can use a tool from discrete

function *Iterative Fibonacci*

Input: integer n

Output: the Fibonacci number F_n

1. if $n == 0$
2.   return 0
3. Create an array $f[0...n]$
4. $f[0] \leftarrow 0$, $f[1] \leftarrow 1$
5. for $i \leftarrow 2, ..., n$
6. $f[i] = f[i-1] + f[i-2]$
7. return $f\,[n]$

Algorithm 3: Iterative Fibonacci Algorithm

mathematics, the structural induction. It happens to be perfect to prove that transformed data, when an algorithm leaves a local section of it, represents a sub-solution of a bigger problem. This is known as "Loop Invariance" which is essential in analysis of algorithms to prove the correctness of the loop structures. Thus, it is necessary to define the concept of recursively defined sets and structures.

4.1 Recursively Defined Sets and Structures

Looking back to our course on discrete math, Do you remember the use of recursion to define sets and structures? If not, please take a look at the following definition:

Definition 2. A recursive definition for a set have two parts:

1. The *basis step* specific an initial collection of the elements in the set.
2. The *recursive step* gives the rules for forming new element from those already known to be in the set.

For example, we have the following recursive definition for the natural numbers $\mathbb{N}$.

1. Basis step: $0 \in \mathbb{N}$.
2. Recursive step: If $n \in \mathbb{N}$, then $n + 1 \in \mathbb{N}$.

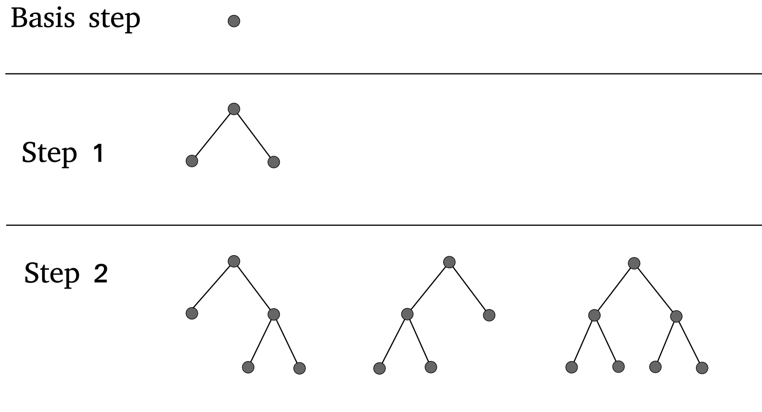

Figure 2: The first steps in the recursive definition of full trees

Furthermore, it is possible to define complex structures using a similar definition [7]. Therefore, we have the necessary elements to prove the correctness of complex algorithms. How you ask yourselves? Very simple, algorithms can be seen as building new recursive structures while following specific restrictions. Actually, the recursive definitions have a series of specific exclusion rules to restrict how new structures can be generated. An example of these recursive structures are the full binary trees.

Definition 3. The set of full binary trees can be defined by:

1. Basis step: There is a full binary tree consisting of only a single vertex r.
2. Recursive step: If T_1 and T_2 are disjoint full binary trees, there is a full binary tree consisting of a root r together with edges connecting the root to each of the roots of the left subtree T_1 and the right subtree T_2.

In (Fig. 2), it is possible to see the first steps in the recursive definition of trees.

4.2 Using Insertion Sort as an Example of Correctness

If we look at (Algorithm 4), it is possible to express the loop invariance as:

- At the end of each loop, the newly created sub-array is always sorted.

Thus, using the insertion sort as an example, an inductive proof of correctness works as follows:

1. Initialization $j = 1$, we know that all inputs of size one are sorted.

function *Insertion Sort*

Input: An array $A\,[1..n]$ of integers

Output: An array of sorted integers $A\,[1..n]$

1. for $j \leftarrow 2$ to length(A)
2. do
3. $key \leftarrow A[j]$
4. ▶Insert $A[j]$ into the sorted sequence $A[1, ..., j-1]\}$
5. $i \leftarrow j-1$
6. while $i > 0$ and $A[i] > key$
7. do
8. $A[i+1] \leftarrow A[i]$
9. $i \leftarrow i-1$
10. $A[i+1] \leftarrow key$

Algorithm 4: The insertion sort algorithm

2. Main body of the loop when, if $A[1, 2, , .., j-1]$ is sorted, then inserting the element in the correct position will keep the sequence sorted. This accomplished by the inner while loop.

3. Once the last number is inserted, the sequence $A[1, 2, ..., n]$ is sorted

Note: The step two is always the most difficult part of the proof of correctness for the algorithm, please be aware of it.

Given this procedure, we can really prove the correctness of really complex algorithms. However, do not get crazy if at the beginning it is quite difficult. After all, as my friends in mathematic always comment, this requires practice... and practice... and practice. Nevertheless, we leave you with some basic exercises where you can begin to practice this fundamental way of proving correctness.

4.3 Exercises about Correctness

1. Prove the correctness of the following algorithm that computes the real value of the polynomial:

$$a[n]\,x^n + a[n-1]\,x^{n-1} + ... + a[1]\,x + a[0]$$

   Input: $n > 0$ integer, an array $a[0...n]$ of real numbers, x a real number

   - $polyval = a[n]$
   - for $i = 1$ to n
   - $\quad polyval = polyval \times x + a[n-i]$
   - return *polyval*

   For this, state the "Loop invariance"

2. Consider the following recursion:

$$E_k = \begin{cases} 0 & \text{if } k = 1 \\ E_{k-1} + k + 1 & \text{if } k \geq 2 \end{cases}$$

   (a) Convert this recursive formulation, by using a similar trick that the one in Fibonacci, to an iterative version.

   (b) Use structural induction to prove the correctness of the algorithm by stating the loop invariance of the iterative algorithm.

5 Counting Steps in an Algorithm

Now, given that we have proved the correctness of our algorithm, we would also like to have a way to measure the number of steps while executing the algorithm. For this we will look back to insertion sort (Algorithm 4). In addition, in order to be able to perform the counting of steps, we can use the following equalities about sequences of numbers (Eq. 5).

$$
\begin{aligned}
\sum_{j=1}^{N} j &= \frac{N(N+1)}{2} \text{ (Arithmetic sum).} \\
\sum_{j=2}^{N} j &= \frac{N(N+1)}{2} - 1 \\
\sum_{j=2}^{N} (j-1) &= \frac{N(N-1)}{2}
\end{aligned} \tag{5}
$$

From here, we can take again a look to the insertion sort algorithm (Algorithm 4) for the analysis of each of step:

1. The external for loop will require $N+1$ steps to finish (Counting the test that fails), if it had been initiated at $j = 1$. In this case, we have that the number of steps is only N **because j starts at 2.** Thus, taking in account the hidden constant that every step has per instruction, we get that line 1 will take $c_1 N$.
2. In line 3, we have that the step is repeated $N-1$ times because we get into the main body of the loop only that many times, thus line 3 will take $c_2 (N-1)$.
3. Similarly in line 5, we have $c_3 (N-1)$ steps.
4. In line 6, each while loop depends on the initial value of $i := j-1$. Therefore, counting the fail together with the worst case scenario (Sequences of numbers like 10, 9, 8, 7,...), we have that the following number of steps is $(1+1)+(2+1)+(3+1)...+(N-1+1) = c_4 \sum_{j=2}^{N} j$.
5. In line 8, we have something similar without the failing test step. Therefore, we have that $1+2+3+...+(N-1) = c_5 \sum_{j=2}^{N} (j-1)$ steps.
6. Similarly in line 9, we have $c_6 \sum_{j=2}^{N} (j-1)$ steps.
7. Finally in line 10, we have again $c_7 (N-1)$ steps.

Using all these values, we have the following sum (Eq.).

$$
\begin{aligned}
T(N) = & c_1 N + c_2(N-1) + ... \\
& C_3(N-1) + c_4 \left(\frac{N(N+1)}{2} - 1 \right) + ... \\
& c_5 \left(\frac{N(N-1)}{2} \right) + c_6 \left(\frac{N(N-1)}{2} \right) + c_7 (N-1)
\end{aligned} \tag{6}
$$

Thus, it is possible to collapse the entire equation into the following quadratic from

$$
T(N) = aN^2 + bN + c \leq (a+b+c)N^2 \tag{7}
$$

The meaning is clear, the number of steps taken by the insertion sort can be bounded by a quadratic function times a constant. This is the beginning of what is better known as asymptotic notation, which was introduced by Paul Bachmann [8] which was popularized in computer science by Donald Knuth in his incredible work “The Art of Computer Programming” [9].

5.1 Counting Exercises

Compute the total number of steps for the following pieces of code

1. Two nested loops

   Input arrays of integers $a[1...n]$ and $b[1...2n]$

   (a) for $i \leftarrow 1$ to n
   (b)   for $j \leftarrow 1$ to $2n$
   (c)   if $mod(j, 2) == 0$
   (d) $a[i] \leftarrow a[i] - b[j]$
   (e) return a

2. A more complex case

   Input arrays of integers $a[1...n]$ and $b[1...2n]$

   (a) for $i \leftarrow 1$ to $n-1$
   (b)   for $j \leftarrow 1$ to n
   (c)   if $a[j] > a[i]$ then do
   (d) $temp \leftarrow a[i]$
   (e) $a[i] \leftarrow a[j]$
   (f) $a[j] \leftarrow temp$
   (g) return a

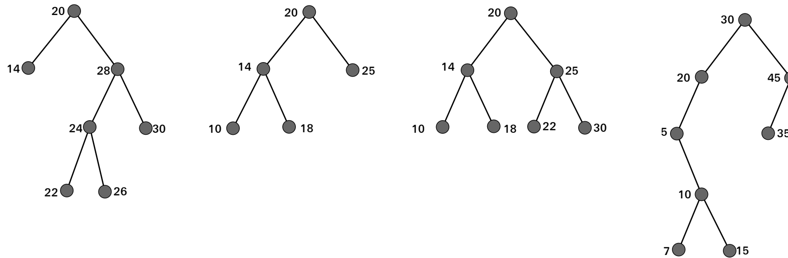

Figure 3: Examples of binary search trees

6 Best, Worst and Average Cases

Here is one of the most important concepts when dealing with the concept of complexity. After all, any well designed algorithm, given the nature of data, can have a wide range of different inputs. For example, given an efficient algorithm to find an element in a binary search tree (Fig. 3) with N nodes, we have several cases:

1. When we have a full tree, thus the worst case of the algorithm is going to be $\log_2 N$ steps.
2. What if you have a degenerated case where all the nodes form a chain? In that case you have N steps.
3. Are there inputs that when you build the tree, it resembles the best case?

Actually, there are and those inputs are known as the average case. The other two first cases are known as the best and worst cases respectively.

Going back to the insertion sort, the worst case is dN^2 steps given a decreasing sequence N elements and the equation (Eq. 7). In addition, the best case of insertion sort is a sequence of increasing N elements making the algorithm to run for dN steps. Even with the average input the insertion sort still has a quadratic term. How do we see this? For this, we need to introduce the idea of inversions in a particular input array. An inversion is an array is a pair of elements $A[i]$ and $A[j]$ such that $i < j$, but $A[j] < A[i]$. For example,

$$A = [0\ 1\ 3\ 2\ 4\ 5\ 7\ 6] \tag{8}$$

In this array there are two inversions: 3 and 2, 7 and 6.

One important property of inversions is that a sorted array has no inversions in it. Why do we care about this? First, the inner loop at insertion sort (Algorithm 4) is in charge of doing the swapping of the elements at the correct positions. Therefore the amount of work done by the insertion sort is coming from two places:

1. The outer loop which grows the sorted array $2, 3, ..., n$ times.

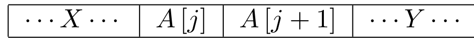

Figure 4: Array elements before the swapping

2. The inner loop that performs the swaps to keep the sorted property.

The outer loop always does n steps of work. The inner loop does an amount of steps that is proportional to the total number of swaps made during the entire runtime of the algorithm. How do we calculate this? We need to see how many swaps are done in total during running time.

Given the inversions in the input, notice that the insertion sort always swaps adjacent elements in the array only if they are an inversion. Now, suppose that we swap $A[j]$ and $A[j+1]$, a question you need to ask yourself is How many inversions are left when the swapping is done? We have several cases and in order to understand them we have the following order on the array elements before the swapping (Fig. 4).

1. Imagine both elements of the inversions are in X or Y. Thus, after swapping of $A[j]$ and $A[j+1]$, the inversions in X or Y are still there.

2. On element is in X or Y and the other element is $A[j]$ or $A[j+1]$. After swapping, the inversion is still there because the relative ordering of the elements has not changed.

3. If one element is $A[j]$ and the other is $A[j+1]$. Then, after swapping the inversion is removed.

Thus, after a swapping, exactly one inversion is removed. Therefore the number of swaps is equal to the number of inversions I. Therefore, the total number of steps done by the insertion sort is equal to $n+I$ steps. Thus, assuming that the input is such that given two elements in the array there is a probability of $\frac{1}{2}$ to have an inversion, we have a way to prove the average case input cost of complexity for the insertion sort. We will see more of this in the following classes, and we will see why the average case is equal to calculate the number of expected steps in an algorithm.

6.1 Worst Best Exercises

1. Given the Binary Search algorithm (Algorithm 5), please give the best and the worst complexity on the number of steps.

2. Given the Shell Sorting algorithm (Algorithm 6), please give the best and the worst complexity on the number of steps.

function *Binary Search*

Input: An array $A[1...n]$ of n elements sorted in nondecreasing order and an element x.

Output: j if $x = A[j]$, $1 \leq j \leq n$ and 0 otherwise

1. $low \leftarrow 1;\ high \leftarrow n;\ j \leftarrow 0$
2. while $(low \leq high)$ and $(j == 0)$
3. $\quad mid \leftarrow \left\lfloor \frac{(low+high)}{2} \right\rfloor$
4. $\quad$ if $x == A[mid]$ then $j \leftarrow mid$
5. $\quad$ if $x < A[mid]$ then $high \leftarrow mid - 1$
6. $\quad$ else $low \leftarrow mid + 1$
7. return j

Algorithm 5: Binary Search algorithm

function *Shell Sort*

Input: An array $A[1...n]$ of n elements and a decreasing integer gap sequence $G[1..m]$.

Output: The array $A[1...n]$ sorted in increasing order.

1. for $g \leftarrow G[1]$ to $G[m]$
2. $\quad$ for $i \leftarrow g$ to n
3. $\qquad temp \leftarrow A[i]$
4. $\qquad j \leftarrow i$
5. $\qquad$ while $g \leq j$ and $A[j-g]$
6. $\qquad\quad A[j] \leftarrow A[j-g]$
7. $\qquad\quad j \leftarrow j - g$
8. $\qquad A[j] \leftarrow temp$

Algorithm 6: Shell Sort algorithm

References

[1] A. Azhar, "Coding is not enough, we need smarter skills," April 2016.

[2] A. Aho, D. S. Johnson, R. M. K. (chair, S. R. Kosaraju, C. C. Mcgeoch, C. H. Papadimitriou, and P. Pevzner, "Emerging opportunities for theoretical computer science," tech. rep., 1996.

[3] J. Arndt and C. Haenel, *Pi-Unleashed.* Springer Berlin Heidelberg, 2000.

[4] A. Blass and Y. Gurevich, "Algorithms: A quest for absolute definitions," *Bulletin of the European Association for Theoretical Computer Science*, 2003.

[5] M. Bóna, *A Walk Through Combinatorics: An Introduction to Enumeration and Graph Theory.* World Scientific Publishing Company, 2 ed., Oct. 2006.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd ed., 2009.

[7] S. S. Epp, *Discrete Mathematics with Applications.* Pacific Grove, CA, USA: Brooks/Cole Publishing Co., 4th ed., 2010.

[8] P. Bachmann, *Die Analytische Zahlentheorie.* 1894.

[9] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms.* Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997.