Analysis of Algorithm

CS313E - Elements of Software Design

Kia Teymourian

05/11/2022

Agenda

1. What is an Algorithm?

2. Growth of Functions

The word *"Algorithm"*

- The origin of the word *"Algorithm"* is from the name of the scientist "Muhammad ibn Musa al-Khwarizmi" (c. 780-850)
- Al-Khwarizmi wrote book on algebra (The Compendious Book on Calculation by Completion and Balancing, c. 813–833) presented the first systematic solution of linear and quadratic equations.
- Many of the sciences that we learn today have their roots back in ancient research in the middle east.

[1]

[1] Statue of al-Khwārizmī carrying an astrolabe in Amir Kabir University, Tehran, Iran. Image from Wikipedia

What is an Algorithm?

- An algorithm is a defined computation procedure that takes a set of input values and produces a set of output values. It can be also seen as a computation steps that converts or maps an input to an output so that we have an input-to-output relation.
- An algorithm can be seen as a higher-level mathematical abstraction of a computer program that does specific computation procedure to solve a specific problem.

Example Algorithm

A frequent problem in computing is sorting.
We mostly have sequence of numbers or elements that we want to sort them based on specific order.
In this case, we have an input data and an output data.

- **Input** is a sequence of numbers $\langle a_1, a_2, \ldots, a_3 \rangle$
- **Output** is an ordered sequence of numbers or a permutation that is reordered based on ascending or descending order.
- For input $\langle a'_1, a'_2, \ldots, a'_3 \rangle$ we get $a'_1 \leq a'_2 \leq \cdots \leq a'_3$

Sorting Example: input of $\langle 78, 23, 32, 89, 45, 67 \rangle$,
a sorting algorithm will output
$\langle 23, 32, 45, 67, 78, 89 \rangle$

Correctness of Algorithm

We say an algorithm is correct when for every input it generates the correct outputs. It is also said that algorithm halts with the correct result or solves the given computation problem.

- Incorrect algorithm might generate wrong results for some input instances.
- Sometimes, we also use incorrect algorithm when we can control their error rates and know in how many cases (percentage) their generate error results.

Example an Algorithm - Insertion Sort

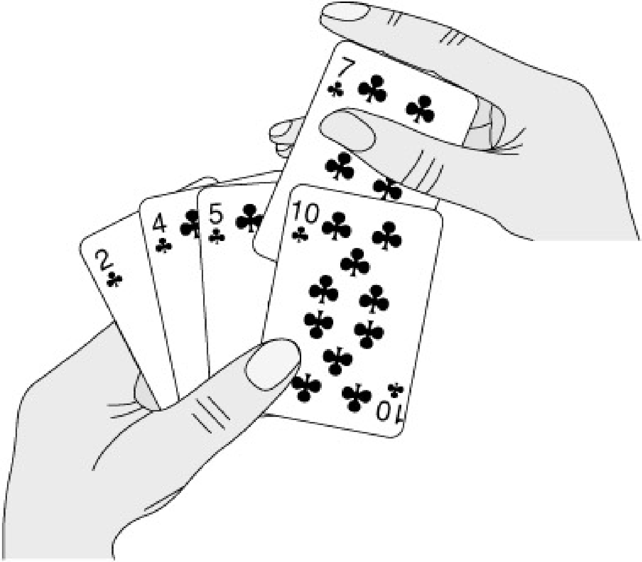


Figure: Sorting a hand of cards using insertion sort.

Example an Algorithm - Insertion Sort

Insertion sort is an efficient algorithm for sorting a small number of elements.

Example: Sort a hand of playing cards

What we do:

- Start with an empty left hand (Cards face down on the table)
- Pick up one card at a time from the table and insert it into the correct position in hand.
- We compare it with each of the cards already in the hand, from right to left, to find the correct position for it.

The same concept can be used in a computer to sort data elements.

Example - Insertion Sort

```
 My Input Array is [5,2,4,6,1,3]
# Key is (2)
# We swap 2 for 5
# Current State is:  [2, 5, (4), 6, 1, 3]
(key is 4)
We swap 4 for 5
# Current State is:  [2, 4, 5, (6), 1, 3]

# Current State is:  [2, 4, 5, 6, (1), 3]
# We swap 1 for 6
# We swap 6 for 5
# We swap 5 for 4
# We swap 4 for 2

# Current State is:  [1, 2, 4, 5, 6, (3)]
# We swap 3 for 6
# We swap 6 for 5
# We swap 5 for 4
# Current State is:  [1, 2, 3, 4, 5, 6]
 Final Output is: [1, 2, 3, 4, 5, 6]
```

Insertion Sort

Algorithm 1 Insertion Sort Algorithm

for $j \leftarrow 2$ to $A.lenght$ **do**
    $key = A[j]$
    // Insert $A[j]$ into the sorted Sequence $A[1..j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$ **do**
        $A[i+1] = A[i]$
        i = i - 1
    end while
    $A[i+1] = key$
end for

Cost Calculation of Insertion Sort Algorithm

Algorithm 2 Cost Calculation of Insertion Sort Algorithm

1: **for** $j \leftarrow 2$ to $A.lenght$ **do** $\quad \triangleright\ C_1 \times n$
2: $\quad key = A[j]$ $\quad \triangleright\ C_2 \times n - 1$
3: $\quad$ // Insert $A[j]$ into the sorted Sequence $A[1..j-1]$
4: $\quad i = j - 1$ $\quad \triangleright\ C_4 \times n - 1$
5: $\quad$ **while** $i > 0$ and $A[i] > key$ **do** $\quad \triangleright\ C_5 \times \sum_{j=2}^{n} t_j$
6: $\quad\quad A[i+1] = A[i]$ $\quad \triangleright\ C_6 \times \sum_{j=2}^{n} (t_j - 1)$
7: $\quad\quad$ i = i - 1 $\quad \triangleright\ C_7 \times \sum_{j=2}^{n} (t_j - 1)$
8: $\quad$ **end while**
9: $\quad A[i+1] = key$ $\quad \triangleright\ C_9 \times (n - 1)$
10: **end for**

Let t_j denote the number of times the while loop test in line 5 is executed for that value of j.

Cost Calculation of Insertion Sort Algorithm

Cost Algorithm Line	Number of Executions
C_1	n
C_2	$n-1$
C_4	$n-1$
C_5	$\sum_{j=2}^{n} t_j$
C_6	$\sum_{j=2}^{n}(t_j-1)$
C_7	$\sum_{j=2}^{n}(t_j-1)$
C_9	$(n-1)$

Cost Calculation of Insertion Sort Algorithm

Cost Algorithm Line	Number of Executions
C_1	n
C_2	$n-1$
C_4	$n-1$
C_5	$\sum_{j=2}^{n} t_j$
C_6	$\sum_{j=2}^{n}(t_j-1)$
C_7	$\sum_{j=2}^{n}(t_j-1)$
C_9	$(n-1)$

The overall costs are:

$$T(n) = C_1 \times n + C_2 \times (n-1) + C_4 \times (n-1) +$$

$$C_5 \times \sum_{j=2}^{n} t_j + C_6 \times \sum_{j=2}^{n}(t_j-1) +$$

$$C_7 \times \sum_{j=2}^{n}(t_j-1) + C_9 \times (n-1)$$

Best-Case Running Time

- When that in sorting, **the best case happens when the array is already sorted.**
- In insertion sort our passing through the array includes no swapping of the array elements because they are already sorted.
- For iterations $j = 2, \ldots, n$ we have $A[j] \leq key$ so that we do not need to swap the elements.

Best-Case Running Time

$$T(n) = C_1 \times n + C_2 \times (n-1) + C_4 \times (n-1) + C_5 \times \sum_{j=2}^{n} t_j + C_6 \times \sum_{j=2}^{n} (t_j - 1) +$$
$$C_7 \times \sum_{j=2}^{n} (t_j - 1) + C_9 \times (n-1)$$

Thus, $t_j = 1$ for $j = 2, \ldots, n$, we will have the best-case running time of:

$$\begin{aligned} T(n) &= C_1 \times n + C_2 \times (n-1) + C_4 \times (n-1) + C_5 \times (n-1) + C_9 \times (n-1) \\ &= (C_1 + C_2 + C_4 + C_5 + C_9) \times n - (C_2 + C_4 + C_5 + C_9) \end{aligned}$$

The above can be written as $\boldsymbol{T(n) = a \times n + b}$ where we have:

- $a = (C_1 + C_2 + C_4 + C_5 + C_9)$
- $b = -(C_2 + C_4 + C_5 + C_9)$

We can express this running time as $a \times n + b$ for constants a and b.
It is for us a thus a linear function of n and we talk about a linear time for the best case.

Worst-Case Running Time

- The worst case happens when the array is sorted in reverse sorted order and we need to move all of the element of the array.

$$T(n) = C_1 \times n + C_2 \times (n-1) + C_4 \times (n-1) + C_5 \times \sum_{j=2}^{n} t_j + C_6 \times \sum_{j=2}^{n} (t_j - 1) +$$
$$C_7 \times \sum_{j=2}^{n} (t_j - 1) + C_9 \times (n-1)$$

We need to compute the entire reverse-sorted sub-array $A[1..j-1]$, and $t_j = j$ for $j = 2, \ldots, n$ so that we have

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

and

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

Worst-Case Running Time

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

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5\left(\frac{n(n+1)}{2} - 1\right)$$

$$C_6\left(\frac{n(n-1)}{2}\right) + C_7\left(\frac{n(n-1)}{2}\right) + C_9(n-1)$$

$$= \left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}\right)n^2 + \left(C_1 + C_2 + C_4 + \frac{C_5}{2} - \frac{C_6}{2} - \frac{C_7}{2} + C_9\right)n$$

$$-(C_1 + C_4 + C_5 + C_9)$$

- We can express this as: $\boldsymbol{an^2 + bn + c}$ for constants a, b and c. This is a **Quadratic Function of n**.
- Later, we say that in such cases, we can suppress constant factors.

Insertion Sort in Python Program

```
def insertionSort(a):
## for every element in our array
for j in range(1, len(a)):
    key = a[j]
    i = j

    while i > 0 and a[i-1] > key:
        print("We swap {} for {}".format(a[i], a[i-1]))
        a[i] = a[i-1]
        i -= 1

    a[i] = key
    print("Current State is: ", a)

return a
```

Listing 1: Insertion Sort in Python Program

Code is Shared here:
https://colab.research.google.com/drive/1XY3IBytYUW7NVvwAk5gax7IynEwQVBa7?usp=sharing

Growth of Functions

Growth of Functions

- The running time that we are interested to know about it in an algorithm analysis is named "rate of growth" or "Order of growth".
- We are interested to know how a function $f(n)$ growth with sufficiently large number of n.

Growth of Functions - Python Example

Let us have a look at growth of functions:

- $f(x) = x$
- $f(x) = 100x^{2.1} + 50$
- $f(x) = x^{3.5} - 2^{10}$

Python Code Shared here: `https://colab.research.google.com/drive/1Th23RUUaMKbM3CHab6RgVV2RFwTT1M_9?usp=sharing`

Big O-Notation, Asymptotic Upper Bound

Big O-notation provides an upper bound for a function to within a constant factor.

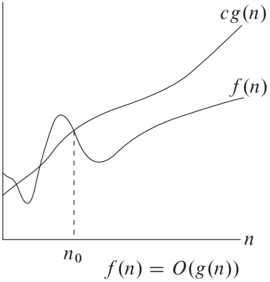
We write $f(n) = O(g(n))$ which means function f grows no faster than function g.

- We can write $f(n) = O(g(n))$ if there are positive constants n_0 **and** c such that at and to the right of n_0 , the value of $f(n)$ is always on or below $c \times g(n)$.
- O notation provides an asymptotic upper bound of a function.

We can formally define the Big O notation as following:

$$O(g(n)) = \{f(n) \text{ there exists constants } c, n_0 > 0 \text{ such that } 0 \leq f(n) \leq c \times g(n) \text{ for all } n \geq n_0\}$$

Big O-Notation, Asymptotic Upper Bound



$$O(g(n)) = \{f(n) \text{ there exists constants } c, n_0 > 0 \text{ such that } 0 \le f(n) \le c \times g(n) \text{ for all } n \ge n_0\}$$

Note: As you can see it does not matter what the function does before n_0 or how the flow if the function is, we are interested to know about the growth of the function for sufficiently large n.

Example - A Polynomial Function (1)

Let us claim that the $f(n) = a_k n^k + a_{k-1} + \cdots + a_1 n + a_0$ is upper bounded by a function $g(n) = n^k$,
$f(n) = O(n^k)$
Note: In a polynomial $a_k, \ldots, a_1, a_0$ are named coefficients.
Proof: Let us consider the following n_0 and c
$n_0 = 1$ and $c = |a_k| + |a_{k-1}| + \cdots + |a_1| + |a_0|$ the sum of the absolute values of all coefficients.
We have to illustrate that $\forall n \geq 1, f(n) \leq c \times n^k$

Example - A Polynomial Function (2)

We have to illustrate that $\forall n \geq 1, f(n) \leq c \times n^k$
In this case, we have for all $n \geq 1$, we can consider the coefficients as absolute values and use n bigger than one, then we can get the following:

$$f(n) \leq |a_k|n^k + |a_{k-1}|n^{(k-1)} + \cdots + |a_1|n + |a_0|$$

- ▷ We get the above because if we use the absolute values we turn some of the negative values into positive values so that the actual $f(n)$ value will always be smaller than the multiplication of all coefficients to the n values when n is bigger than one.
- ▷ We can make the above bigger when we multiply it to the a larger number n^k which makes it bigger.

$$f(n) \leq |a_k|n^k + |a_{k-1}|n^k + \cdots + |a_1|n^k + |a_0|n^k$$
$$f(n) \leq c \times (n^k)$$

And this is the multiplication of the above $c = |a_k| + |a_{k-1}| + \cdots + |a_1| + |a_0|$ to the n^k. So we have proved that the $f(n)$ is always smaller equal to a constant c multiply to n^k

Example of Big O

$$f(n) = 4n^2 - 8n + 4 = O(n^2)$$

because for $c = 4, 4n^2 > f(n)$ when $n > 1$;

$$f(n) = 4n^2 - 8n + 4 = O(n^3)$$

because for $c = 1, n^3 > f(n)$ when $n > 1$;

Example of Big O

$$f(n) = 4n^2 - 8n + 4 = O(n^2)$$

because for $c = 4, 4n^2 > f(n)$ when $n > 1$;

$$f(n) = 4n^2 - 8n + 4 = O(n^3)$$

because for $c = 1, n^3 > f(n)$ when $n > 1$;

$$f(n) = 4n^2 - 8n + 4 \neq O(n)$$

because for $c > 0, cn < f(n)$ when $n > (c+8)/4$

since $4n > c + 8 \Rightarrow 4n - 8 > c \Rightarrow 4n^2 - 8n > cn$

$\Rightarrow 4n^2 - 8n > cn - 4$

$\Rightarrow 4n^2 - 8n + 4 = f(n) > cn$;

Big O using limit definition

The definitions of the various asymptotic notations are closely related to the definition of a limit.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} \neq \infty \implies f = O(g)$$

Knowledge about limits can be helpful in working out asymptotic relationships. In particular, recall L'Hospital's Rule

$$\lim_{n \to \infty} f(n) = \infty \text{ and } \lim_{n \to \infty} g(n) = \infty$$
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{f'(n)}{g'(n)}$$

Order of Growth of Functions

$$1 < log(n) < \sqrt{n} < n < nlog(n) < n^2 < n^3 < \cdots < 2^n < 3^n < \cdots < n^n$$

Function Name	**Big O**
Constant	$O(c)$
Logarithmic	$O(log(n))$
Linear	$O(n)$
Log Linear	$O(nlog(n))$
Quadratic	$O(n^2)$
Cubic	$O(n^3)$
Exponential	$O(2^n)$

Every computer scientist knows two rules of thumb about asymptotics: logarithms grow more slowly than polynomials and polynomials grow more slowly than exponentials.

Readings from CLRS Book (Introduction to Algorithms, 3rd Edition)

- Chapters 1, and 2
- Sec. 2.2 - Introduction
- Sec. 1.2 - Analysis of insertion sort
- Sec. 1.2 - Growth of Functions
- Sec. 3.1 - Asymptotic notation
- Sec. 3.1 and 3.2 Standard notations and common functions