Analysis of algorithms and computational complexity

Pramesh Kumar

IIT Delhi

January 4, 2024

Outline

Definition (Problem). A problem P is a question to be answered for a given input.

Example(s).

1. The sorting problem is, given a sequence of n natural numbers $\langle a_1, ..., a_n \rangle$, to find a permutation (reordering) $\langle a_1^{'}, ..., a_n^{'} \rangle$ of input sequence such that $a_1^{'} \leq a_2^{'} \leq ... \leq a_n^{'}$.
2. The $s - t$ shortest path problem is, given a directed graph $G(N, A)$, link costs $c : A \mapsto \mathbb{R}$, a source $s \in N$ and a destination $t \in N$, to find a path from source s to destination t with minimum cost.

Definition (Instance). An instance of the problem is the problem specified with specific input data.

Example(s). sort$(\langle 54, 78, 21, 87, 5 \rangle)$ is an instance of the sorting problem.

Definition (Algorithm). An algorithm is a step-by-step procedure that takes an input and produces an output.

Remark. An algorithm is said to be correct, if for every instance, it produces the correct output.

Definition (Data structure). A way to store and organize data in order to facilitate access and modifications.

Outline

Why algorithms matter?

Computer A	Computer B
-Fast -Executes 10 billion instructions per sec -Implement insertion sort -Complexity $= 2n^2$ -Time $= \frac{2\times(10^7)^2}{10^{10}} \approx$ 5.5 hrs	-Slow -Executes 10 million instruction per sec -Implement merge sort -Complexity $= 50nlogn$ -Time $= \frac{50\times10^7 log 10^7}{10^7} \approx$ 20 min.

Table: Source: CLRS

Outline

Insertion sort algorithm

Algorithm 1 An algorithm to sort a sequence of n numbers

```
Input: An array $A$ of $n$ elements $\langle a_1, ..., a_n \rangle$
Output: A permutation (reordering) $\langle a_1^{'}, ..., a_n^{'} \rangle$ such that $a_1^{'} \leq a_2^{'} \leq ... \leq a_n^{'}$.
procedure INSERTION_SORT($A$)
    for $j = 2$ to $len(A)$ do
        $key = A[j]$
        $i = j - 1$            ▷ Insert $A[.]$ into the sorted sequence $A[1, ..., j-1]$
        while $i > 0$ & $A[i] > key$ do
            $A[i+1] = A[i]$
            $i = i - 1$
        end while
        $A[i+1] = key$
    end for
end procedure
```

Source: CLRS

Proposition

Insertion sort algorithm is correct.

Outline

Analyzing algorithms (Complexity analysis)

- Provides a framework for analyzing the performance of an algorithm in terms of elementary operations (assignment, arithmetic, logical and control) it performs.
- One measure of efficiency of an algorithm is running time.
  - Wall clock time can be "problematic" since it may depend the programming language, hardware, proficiency of the programmer, compiler, etc.
- The running time for solving a problem may vary by the instance, algorithm, and hardware used.
- Three approaches for measuring the performance of an algorithm
  1. Empirical analysis: Estimates how the algorithm performs in real-life on specific instances.
  2. Average-case analysis: Estimates the expected running time of the algorithm based on sampling from a probability distribution on the problem instances.
  3. Worst-case analysis: Provides an upper bound on the number of steps the algorithm can take on any instance.

Analyzing algorithms

- Worst-case analysis has became quite popular in the theoretical sense as it does not depend on the programming language, compiler, probability distribution on the problem instances, etc.
- However, it is influenced by pathological instances. For example, the performance of simplex method versus ellipsoid method for solving linear programs.

Example(s). Assume that worst-case time-complexity (we'll define it formally shortly) of solving the shortest path problem with non-negative length costs is $2n^2$, meaning that number of computations grows no more than 2 times the square of number of nodes in the graph.

Remark. We say that an algorithm is good if its computations are bounded by a polynomial in the problem input size. On the other hand, we say that an algorithm is bad if its computations grow exponentially when applied to specific instances.

Remark. An algorithm typically takes varying amount of time/space on different instances.

Analyzing algorithms

Definition (Time complexity/running time).: A time complexity function for an algorithm is a function describing the time taken by the algorithm in terms of its input size.

Definition (Space complexity). A space complexity function for an algorithm is a function describing the memory required by the algorithm in terms of its input size.

Outline

Analyzing insertion sort algorithm

- $T(n)$: running time of the algorithm for input array of size n.
- t_j: # of times the "while" loop is executed.
- Assume constant time is needed for executing each line of our pseudo-code.

```
procedure INSERTION_SORT($A$)                  ▷ Cost    Times
    for $j = 2$ to $len(A)$ do                  ▷ $c_1$   $n$
        $key = A[j]$                           ▷ $c_2$   $n-1$
        $i = j - 1$                            ▷ $c_3$   $n-1$
        while $i > 0$ & $A[i] > key$ do        ▷ $c_4$   $\sum_{j=2}^n t_j$
            $A[i+1] = A[i]$                    ▷ $c_5$   $\sum_{j=2}^n (t_j - 1)$
            $i = i - 1$                        ▷ $c_6$   $\sum_{j=2}^n (t_j - 1)$
        end while
        $A[i+1] = key$                         ▷ $c_7$   $n-1$
    end for
end procedure
```

Source: CLRS

- "for" loop will come back to check $n+1$, therefore runs n times
- $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\sum_{j=2}^n t_j) + c_5(\sum_{j=2}^n (t_j - 1)) + c_6(\sum_{j=2}^n (t_j - 1)) + c_7(n-1)$

Analyzing insertion sort

$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\sum_{j=2}^{n} t_j) + c_5(\sum_{j=2}^{n}(t_j - 1)) + c_6(\sum_{j=2}^{n}(t_j - 1)) + c_7(n-1)$

1. Best-case scenario: The array is already sorted, i.e., $A[i] < key$ always, implying that $t_j = 1$ for $j = 2, ..., n$.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \end{aligned}$$

$T(n)$ is a linear function of n.

1. Worst-case scenario: The array is reverse-sorted (Note: $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ and $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$).

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

$T(n)$ is a quadratic function of n. Worst-case scenario gives you an upper bound on the running time for any input.

Outline

Order of growth/Asymptotic efficiency

Definition (Asymptotic notations).: Let $g : \mathbb{N} \mapsto \mathbb{R}_+$. We define

$$\boxed{O(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}_+, n_0 \in \mathbb{N} \text{ s.t. } 0 \leq f(n) \leq cg(n), \forall n \geq n_0\}}$$

$$\boxed{\Omega(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}_+, n_0 \in \mathbb{N} \text{ s.t. } 0 \leq cg(n) \leq f(n), \forall n \geq n_0\}}$$

$$\boxed{\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2 \in \mathbb{R}_+, n_0 \in \mathbb{N} \text{ s.t. } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0\}}.$$

Remark. We write $f(n) = O(g(n))$ rather than $f(n) \in O(g(n))$.

Remark. An algorithm is efficient if its worst case running time is polynomial in its input.

Order of growth/Asymptotic efficiency

O-notation

- characterizes an UB on the asymptotic behavior of a function, i.e., a function grows no faster than the certain rate
- $f(n) = O(g(n))$ pronounced as $f(n)$ is "big-O" of $g(n)$, which means that g grows at least as fast as f.

Proposition

$4n^2 + 100n + 500 = O(n^2)$

Proof.

To show this, we need to find $c > 0$ and $n_0 \in \mathbb{N}$ such that $4n^2 + 100n + 500 \leq cn^2, \forall n \geq n_0 \implies 4 + \frac{100}{n} + \frac{500}{n^2} \leq c$. This is satisfied for many c and n_0. For example, take $n_0 = 1, c = 604$. $\square$

Order of growth/Asymptotic efficiency

Ω-notation

- characterizes a LB on the asymptotic behavior of a function, i.e., a function grows at least as fast as a certain rate, based - as in the O-notation -on the highest term.
- $f(n) = \Omega(g(n))$ pronounced as f is "Omega" of g, which means that f grows at least as fast as g.

Proposition

$4n^2 + 100n + 500 = \Omega(n^2)$

Proof.

To show this, we need to find $c > 0$ and $n_0 \in \mathbb{N}$ such that $4n^2 + 100n + 500 \geq cn^2, \forall n \geq n_0 \implies 4 + \frac{100}{n} + \frac{500}{n^2} \geq c$. This holds for any n_0 and $c = 4$. $\square$

Order of growth/Asymptotic efficiency

Θ-notation

- characterizes a "tight bound" on the asymptotic behavior of a function, i.e., a function grows "precisely" at a certain rate, based-on the highest order term.

Theorem

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Corollary

$4n^2 + 100n + 500 = \Theta(n^2)$

Definition (Other asymptotic notations).: Let $g : \mathbb{N} \mapsto \mathbb{R}_+$. We define

$$\boxed{o(g(n)) = \{f(n) \mid \text{ for any } c > 0, \exists n_0 > 0 \text{ s.t. } 0 \leq f(n) < cg(n), \forall n \geq n_0\}}$$

$$\boxed{\omega(g(n)) = \{f(n) \mid \text{ for any } c > 0, \exists n_0 > 0 \text{ s.t. } 0 < cg(n) < f(n), \forall n \geq n_0\}}$$

Analyzing insertion sort algorithm

```
 1: procedure INSERTION_SORT($A$)
 2:     for $j = 2$ to $len(A)$ do
 3:         $key = A[j]$
 4:         $i = j - 1$
 5:         while $i > 0$ & $A[i] > key$ do
 6:             $A[i+1] = A[i]$
 7:             $i = i - 1$
 8:         end while
 9:         $A[i+1] = key$
10:     end for
11: end procedure
```

Proposition

The worst-case complexity of insertion sort algorithm is $\Theta(n^2)$.

Proof.

Let us first show that it is $O(n^2)$. The outer "for" loop runs $n-1$ times regardless of values being sorted. Lines 3-4 run in constant time. The inner "while" loop might run $j-1$ times. Further, Lines 6-7 take constant time per iteration. Therefore, $\#$ of iterations $\leq (n-1)(j-1) \leq (n-1)(n-1) \leq n^2$ (since $j \leq n$). The running time for any case of insertion sort algorithm is $O(n^2)$. $\square$

Proof (contd.)

Next, let us show that worst-case complexity of insertion sort algorithm is $\Omega(n^2)$. For a value to end-up k positions to the right of where it started, Line 6 must have been executed k times. Let us assume that n is a multiple of 3, so we divide the array A into three groups of $n/3$ positions.

A$[1 : \frac{n}{3}]$	A$[\frac{n}{3} + 1 : \frac{2n}{3}]$	A$[\frac{2n}{3} + 1 : n]$

Suppose that in the input to the algorithm, the $n/3$ largest values occupy the first $n/3$ positions in the array $A[1 : n/3]$. Once the array has been sorted, each of these $n/3$ values will endup somewhere in the last $n/3$ positions, i.e., $A[2n/3 + 1 : n]$. For that to happen, each of these $n/3$ values must pass through each of the middle $n/3$ positions, i.e., $A[n/3 + 1 : 2n/3]$. Each of these values passes through $n/3$ position one at a time, by at least $n/3$ executions of Line 6. Because at least $n/3$ values have to pass through at least $n/3$ positions, the time taken by the insertion sort algorithm in worst-case is at least proportional to $(n/3)(n/3) = n^2/9$, which is $\Omega(n^2)$.
Since worst-case complexity of insertion sort algorithm is both $O(n^2)$ and $\Omega(n^2)$, it is also $\Theta(n^2)$.

$\square$

Outline

Polynomial- versus exponential-time algorithms

- Generally, algorithms with worst-case complexity bounded by a polynomial function of problem parameters are considered to be "good".
- Problems for which there is an algorithm with polynomial running time (or better) are called polynomially solvable.
- Let $n, m, C,$ and U represent no. of nodes, no. of links, largest link cost, and largest link capacity respectively. Examples of polynomial-time bounds are $O(n^2)$, $O(nm)$, $O(m + nlogC)$, $O(nm + n^2logU)$, etc.
- An algorithm is called exponential-time algorithm if its worst-case running time grows as a function that cannot be polynomially bounded by the input length. Examples - $O(nC)$, $O(2^n)$, $O(n!)$, $O(n^{logn})$, etc.
- Generally, algorithms with exponential worst-case complexity are considered to be "bad".

Polynomial- versus exponential-time algorithms

There are many interesting problems for which it is not known if there is a polynomial-time algorithm. Such problems are considered "difficult". Examples: TSP, VRP, SAT, etc.

One of the open questions in mathematics is whether these "difficult" problems really are difficult or if we just haven't discovered the right algorithm yet. By answering this question, one can win a million dollars!

https://www.claymath.org/millennium/p-vs-np/(Click for more details).

I'm avoiding providing more information right now for obvious reasons.

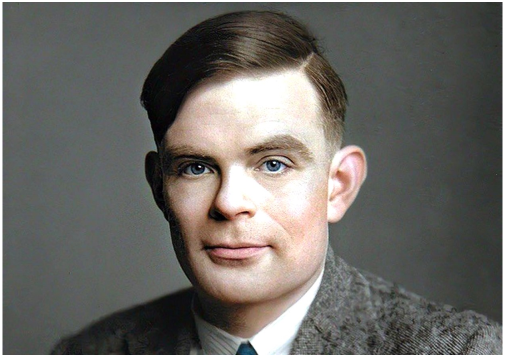

Figure: Alan Turing (Source:facts.net)

Alan Turing, considered to be the father of theoretical computer science, played a key role in formalizing the concepts of algorithm and computation with the Turing machine, which can be considered a model of general-purpose computer (Source: Wiki).

Suggested reading

1. CLRS Chapter 1 and 2

Thank you!