ANALYSIS OF ALGORITHMS

Analyzing an algorithm means **predicting** the resources that the algorithm requires. There are mainly two types of resources that we want to estimate, i.e. time and space**.**

Before we can analyze an algorithm, we must have a model of the **implementation technology** that we will use, including a model for the resources of that technology and their costs. We will do all our design and analysis for the one-processor **RAM** (random-access machine**) model** of computation, which is based on following assumptions:

- In the RAM model, instructions are executed one after another, with no concurrent operations.
- Each "**simple**" operation (+, -, =, if, call, ...) takes exactly 1 step (a constant amount of time).
- Each memory access takes exactly 1 step.
- In the RAM model, we do not consider effects of memory hierarchy (caches, virtual memory, ...) that is common in contemporary computers.
- The RAM model excludes an instruction like “sort”, because sorting is not a single step operation and real computers do not have such instructions. Similarly, Loops and subroutine calls are not simple operations, but depend upon the size of the data and the contents of a subroutine.

The step count method for Estimating Running Time

1. Each executable statement of the algorithm (or the corresponding program) is assigned some number of ‘steps’. The number of steps any statement is assigned depends on the kind of statement. For example,
    - Comments are assigned zero steps because comments are not executable statements, and so they take no time**.**
    - Assignment statement is counted as one step.
    - Each execution of the control part (loop header) of a simple ‘for’ or ‘while’ loop is counted as one step.
2. Find the **total** number of steps contributed by each statement, which is given by the number of steps per execution (**s/e**) of the statement multiplied by the number of times (i.e. frequency) that statement is executed.
3. Add the running times (total steps) contributed by each statement to obtain step count of the entire algorithm.

Note: The cost/time needed for executing each statement is expressed in terms of steps. Here, a step may be thought as a constant amount of time. One statement may take a different amount of time than another statement. Consequently, one statement may be assigned different number of steps than another. So the cost of a statement is the number of steps assigned to that statement.

Thus, the **running time** of an algorithm on a particular input is the number of primitive operations or “steps” executed. In general, the time taken by an algorithm grows with the size of input, so it is traditional to describe the running time of a program as a function of the size of its input.

The **input size** depends on the problem being studied. For example, for a sorting problem, input size is the number of items in the input (or array size, n). For many other problems, such as multiplying two integers, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation.

EXAMPLE 1: Consider the following **insertion sort** algorithm on array A containing 'n' items. Dry run this algorithm to get an understanding of how it works. In each iteration of for loop, we pick up an element $A[j]$ and insert it into the sorted subarray $A[1\,..\,j-1]$.

```
INSERTION-SORT(A)
for j = 2 to A.length
{
  Key = A[j] ;
  //insert A[j] into the sorted sequence A[1..j-1]
  i = j - 1 ;
  While ( i > 0 && A[i] > key )
    {
     A[i + 1] = A[i];
     i = i - 1;
    }
  A[i + 1] = key;
 }
```

The step table for analyzing insertion sort algorithm follows:

	Statement	s/e	frequency	Total steps
1	INSERTION-SORT(A)	0	-	0
2	$for\ j = 2\ to\ A.length$	1	n	n
3	{	0	-	0
4	$Key = A[j]$;	1	$n-1$	$n-1$
5	$i = j - 1$;	1	$n-1$	$n-1$
6	$While\ (\ i > 0\ \&\&\ A[i] > key\ )$	1	$\sum_{j=2}^{n} t_j$	$\sum_{j=2}^{n} t_j$
7	{	0	-	0
8	$A[i + 1] = A[i];$	1	$\sum_{j=2}^{n}(t_j - 1)$	$\sum_{j=2}^{n}(t_j - 1)$
9	$i = i - 1;$	1	$\sum_{j=2}^{n}(t_j - 1)$	$\sum_{j=2}^{n}(t_j - 1)$
10	}	0	-	0
11	$A[i + 1] = key;$	1	$n-1$	$n-1$
12	}	0	-	0

For each $j = 2,3, \ldots, n$, where $n = A.length$, let t_j denote the number of times the while loop test in line 6 is executed for that value of j. Further, note that when a **for** or **while** loop exits in the usual way (i.e., due to the test in the loop header), the test is executed one time more than the loop body.

Thus, the total running time, T(n), of insertion sort algorithm is given by:

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

Best, worst, and average Case complexities

Best case: Insertion sort takes minimum number of 'steps' when the input array is already sorted.

By carrying out the dry run of the above algorithm when input array is already sorted in increasing order (best case), you can easily note the following:

$$\begin{aligned} when\ j = 2 \quad & t_2 = 1 \\ j = 3 \quad & t_3 = 1 \\ \cdots \quad & \cdots \\ j = n \quad & t_n = 1 \end{aligned}$$

Thus, $\sum_{j=2}^{n} t_j = t_2 + t_3 + \cdots + t_n = 1 + 1 + \cdots + 1 \quad [(n-1)times] = n - 1$

$\sum_{j=2}^{n}(t_j - 1) = (t_2 - 1) + (t_3 - 1) + \cdots + (t_n - 1) = (1 - 1) + (1 - 1) + \cdots + (1 - 1) = 0$

Therefore, $T(n) = n + (n-1) + (n-1) + (n-1) + (n-1)$

$= 5n - 4 \ \ (a\ linear\ function)$

$= \theta(n)$

Worst case: Insertion sort takes maximum number of steps when the input array is reverse sorted (decreasing order). In worst case, you can easily notice that

$$\begin{aligned} when\ j = 2 \quad & t_2 = 2 \\ j = 3 \quad & t_3 = 3 \\ \cdots \quad & \cdots \\ j = n \quad & t_n = n \end{aligned}$$

Thus, $\sum_{j=2}^{n} t_j = t_2 + t_3 + \cdots + t_n = 2 + 3 + \cdots + n = \frac{n(n+1)}{2} - 1$

$\sum_{j=2}^{n}(t_j - 1) = (t_2 - 1) + (t_3 - 1) + \cdots + (t_n - 1)$

$= (2 - 1) + (3 - 1) + \cdots + (n - 1)$

$= 1 + 2 + 3 + \cdots + n - 1$

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

$= \frac{n(n-1)}{2}$

Therefore, the worst case running time of insertion sort is given by

$T(n) = n + (n-1) + (n-1) + \frac{n(n+1)}{2} - 1 + \frac{n(n-1)}{2} + \frac{n(n-1)}{2} + (n-1)$

$= \frac{3}{2}n^2 + \frac{7}{2}n - 4 \ \ (a\ quadratic\ function)$

$= \theta(n^2)$

We usually concentrate on finding only the worst-case running time, that is, the longest running time for any input of size n. We give three reasons for this orientation.

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer.
- For some algorithms, the worst case occurs fairly often. For example, in searching a database for a particular piece of information, the searching algorithm's worst case will often occur when the information is not present in the database. In some applications, searches for absent information may be frequent.
- The "**average case**" is often roughly as bad as the worst case. Suppose that we randomly choose n numbers and apply insertion sort. How long does it take to determine where in subarray $A[1\,..\,j-1]$ to insert element $A[j]$? On average, half the elements in $A[1\,..\,j-1]$ are less than $A[j]$, and half the elements are greater. On average, therefore, we check half of the subarray $A[1\,..\,j-1]$, and so t_j is about $\frac{j}{2}$. The resulting average-case running time turns out to be a quadratic function of the input size, just like the worst-case running time.

EXAMPLE 2: The Fibonacci sequence of numbers is given as:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Each next term is obtained by taking the sum of the two previous terms. Thus,

$$f_n = \begin{cases} 0\,, & if\ n = 0 \\ 1, & if\ n = 1 \\ f_{n-1} + f_{n-2}\,, & if\ n \geq 2 \end{cases}$$

The algorithm to compute the n[th] Fibonacci number along with step table is given below:

Statement	s/e	Frequency		Total steps	
		n=0 or 1	n>1	n=0 or 1	n>1
1. FIBONACCI(n)	0	-	-	0	0
2. $If(n \leq 1)$	1	1	1	1	1
3. $Write(n);$	1	1	0	1	0
4. $else$	0	-	-	0	0
5. {	0	-	-	0	0
6. $f0 = 0;$	1	0	1	0	1
7. $f1 = 1;$	1	0	1	0	1
8. $for\ i = 2\ to\ n$	1	0	n	0	n
9. {	0	-	-	0	0
10. $fn = f0 + f1;$	1	0	n-1	0	n-1
11. $f0 = f1;$	1	0	n-1	0	n-1
12. $f1 = fn;$	1	0	n-1	0	n-1
13. }	0	-	-	0	0
14. $Write(fn);$	1	0	1	0	1
15. }	0	-	-	0	0
Total				2	$\boldsymbol{4n+1}$

EXAMPLE 3: Analysis of Matrix multiplication.

If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then in the product $C = A.B$, we define the entry c_{ij} , for $i, j = 1,2,3, \dots, n$, by

$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj}$

The following procedure takes $n \times n$ matrices A and B and multiplies them, returning their $n \times n$ product C.

SQUARE-MATRIX-MULTIPLY(A,B)

```
1.  n = A.rows
2.  Let C be a new n × n matrix
3.  for i = 1 to n
4.      for j = 1 to n
5.      {
6.          c_ij = 0;
7.          for k = 1 to n
8.              c_ij = c_ij + a_ik × b_kj
9.      }
10. return C
```

The step table for only significant statements is given below. Note that the 'k loop' is executed $n + 1$ times for each iteration of the outer two loops. Thus, for n^2 iterations of outer two loops, the 'k loop' will be executed $n^2(n + 1)\ times$. Also any statement inside the 'k loop' will be executed n^3 times.

Statement	Total steps
1. $for\ i = 1\ to\ n$	$n + 1$
2. $for\ j = 1\ to\ n$	$n(n + 1)$
3. {	0
4. $c_{ij} = 0$;	n^2
5. $for\ k = 1\ to\ n$	$(n + 1)\ n^2$
6. $c_{ij} = c_{ij} + a_{ik} \times b_{kj}$	$n \times n^2$
7. }	0
Total	$2n^3 + 3n^2 + 2n + 1 = \theta(n^3)$

Note: there is an efficient recursive algorithm for matrix multiplication called Strassen's matrix multiplication that runs in $\theta\left(n^{\log 7}\right) = \theta(n^{2.8})\ time$.

RECURSIVE ALGORITHMS AND THEIR TIME COMPLEXITY

The running time of recursive algorithm is naturally described by a recurrence relation. A recurrence relation is an equation or inequality that describes a function in terms of its value on smaller inputs.

EXAMPLE 1: **Towers of Hanoi**.

According to legend, there is a temple in Hanoi with three posts and 64 gold disks of different sizes. Each disk has a hole through the center so that it fits on a post. In the misty past, all the disks were on the first post, with the largest on the bottom and the smallest on top, as shown in Figure 1.

Monks in the temple have labored through the years since to move all the disks to one of the other two posts according to the following rules:

- The only permitted action is removing the top disk from one post and dropping it onto another post.
- A larger disk can never lie above a smaller disk on any post.

So, for example, picking up the whole stack of disks at once and dropping them on another post is illegal. That's good, because the legend says that when the monks complete the puzzle, the world will end!

The goal of the puzzle is to have all the disks on one of the two other posts in order of size, with the largest on the bottom.

Solution: Recursion provides an elegant solution for the Towers of Hanoi puzzle. To clarify, suppose there are n=3 disks, then the puzzle can be solved in 7 steps as shown in **figure 1** below.

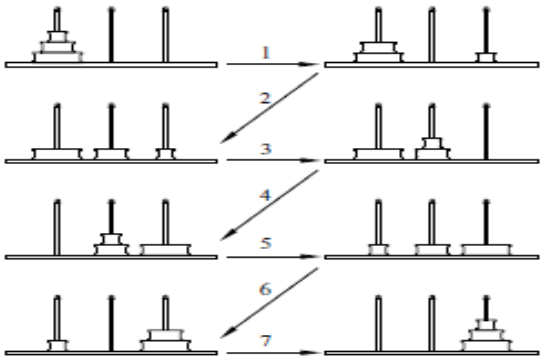

Figure 1. the 7-step solution to the towers of Hanoi problem when there are n=3 disks

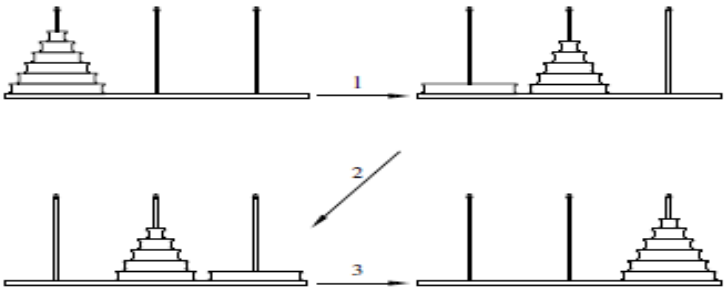

Figure 2. a recursive solution to the towers of Hanoi problem.

In general, the recursive solution has three stages that are described below and illustrated in **figure 2.** Let H_n denote the minimum number of moves (steps) needed to solve the n-disk Tower of Hanoi problem.

1. Move the top n-1 disks from the first post to the second using the solution for n-1 disks. This can be done in H_{n-1} steps.
2. Move the largest disk from the first post to the third post. This takes just 1 step.
3. Move the $n - 1$ disks from the second post to the third post, again using the solution for $n - 1$ disks. This can also be done in H_{n-1} steps.

This algorithm shows that H_n, the minimum number of steps required to move n disks to a different post, is at most $H_{n-1} + 1 + H_{n-1} = 2H_{n-1} + 1$.

Thus, $H_n = 2H_{n-1} + 1$.

The following recursive procedure solves Towers of Hanoi puzzle:

```
1. TOWERS − OF − HANOI(n, x, y, z)
2. //move the top n disks from tower x to tower y
3. If(n > 1)
4. {
5. TOWERS − OF − HANOI(n − 1, x, z, y);
6. Write(move top disk from tower 'x' to top of tower 'y');
7. TOWERS − OF − HANOI(n − 1, z, y, x);
8. }
```

The step table for significant statements of the algorithm is given below:

Statement	s/e	Frequency	Total steps
1. $\text{TOWERS} - \text{OF} - \text{HANOI}(n-1, x, z, y);$	H_{n-1}	1	H_{n-1}
2. $Write(move\ top\ disk\ from\ tower\ 'x'\ to\ top\ of\ tow$	1	1	1
3. $\text{TOWERS} - \text{OF} - \text{HANOI}(n-1, z, y, x);$	H_{n-1}	1	H_{n-1}
Total			$2H_{n-1} + 1$

You can easily note that $H_1 = 1\ and\ H_2 = 3$.

The total number of steps obtained is a recurrence relation, which can be solved using the iteration method (repeated substitutions) as follows:

Given, $H_n = 2H_{n-1} + 1$

$= 2(2H_{n-2} + 1) + 1$

$= 2^2 H_{n-2} + 2 + 1$

$= 2^2(2H_{n-3} + 1) + 1$

$= 2^3 H_{n-3} + 2^2 + 1$

$= 2^4 H_{n-4} + 2^3 + 2^2 + 1$

…

$= 2^{n-1} H_1 + 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2 + 1$

$= 2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2 + 1$

$= 2^n - 1$

we know that, sum of geometric progressions is given by

$a + ar + ar^2 + ar^3 + ar^4 + \cdots + ar^n = \frac{ar^{n+1}-1}{r-1}$

Substituting a=1and r=2 in this formula, we get

$2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2^2 + 2 + 1 = 2^n - 1$

Summary

A step is a computation unit that is independent of the instance characteristics like size of each input, number of inputs etc. A step is chosen so that it is as machine independent as possible. For example, 10 additions can be one step, 50 multiplications can also be one step but n additions cannot be regarded as a step. Nor can be $\frac{n}{2}$ $additions, m \times n$ $subtractions$, $p +$ q multiplications because these depend on characteristics of problem instance.

Obtaining the exact step count may be challenging and exceedingly difficult for complex problems. Further, the notion of step itself is inexact. Therefore, in order to compare several candidate algorithms for a given problem we usually abstract out only the important details. For example, in case of sorting or searching algorithms we usually count only the number of comparisons (because the number of comparisons made by any sorting/ searching algorithm depend on the input array size 'n') and ignore the other insignificant statements in the process. The algorithm that makes lower number of comparisons is then rated as efficient as compared to others.

Note that to represent the cost of each statement we assign some number of steps to it but ignore the actual cost. We then again ignore the cost of insignificant statements and add up the costs of only those statements whose cost depends on the characteristics of problem instance (e.g. those statements whose cost involves the parameter $n, i.e. input\ size$). We can go a step further and discard all the lower-order terms in the final expression for the running time of an algorithm. For example, in $an^2 + bn + c$, we can discard the terms cn and c and express the running time of algorithm as an^2. The reason is that the lower-order terms become relatively insignificant for sufficiently large values of n. Furthermore, we can also discard the coefficient of the leading term. In other words, we concentrate only on the **order of growth**/ **rate of growth** of the running time.

The concept of rate of growth gives rise to asymptotic notations like θ, O,Ω.

We usually consider one algorithm to be more efficient than another if its worst case running time has a lower order of growth. Due to constant factors and lower-order terms, an algorithm whose running time has a higher order of growth might take less time for small inputs than an algorithm whose running time has a lower order of growth. But for large enough inputs, a $\theta(n^2)$ algorithm, for example, will run more quickly in the worst case than a $\theta(n^3)$ algorithm.

Compiled by: Gulnawaz Gani
Department of Computer Sciences,
University of Kashmir,
North campus, Delina Baramulla.
03 November 2016.