Introduction

How efficient is an algorithm or piece of code? Efficiency covers lots of resources, including:

- CPU (time) usage
- memory usage
- disk usage
- network usage

All are important but we will mostly talk about time complexity (CPU usage).

Be careful to differentiate between:

1. Performance: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
2. Complexity: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger?

Complexity affects performance but not the other way around.

The time required by a function/procedure is proportional to the number of "basic operations" that it performs. Here are some examples of basic operations:

- one arithmetic operation (e.g., +, *).
- one assignment (e.g. x := 0)
- one test (e.g., x = 0)
- one read (of a primitive type: integer, float, character, boolean)
- one write (of a primitive type: integer, float, character, boolean)

Some functions/procedures perform the same number of operations every time they are called. For example, StackSize in the Stack implementation always returns the number of elements currently in the stack or states that the stack is empty, then we say that StackSize takes ***constant time***.

Other functions/ procedures may perform different numbers of operations, depending on the value of a parameter. For example, in the BubbleSort algorithm, the number of elements in the array, determines the number of operations performed by the algorithm. This parameter (number of elements) is called the ***problem size***/ ***input size***.

When we are trying to find the complexity of the function/ procedure/ algorithm/ program, we are ***not*** interested in the ***exact*** number of operations that are being performed. Instead, we are interested in the relation of the ***number of operations*** to the ***problem size***.

Typically, we are usually interested in the ***worst case***: what is the ***maximum*** number of operations that might be performed for a given problem size. For example, inserting an element into an array, we have to move the current element and all of the elements that come after it one place to the right in the array. In the worst case, inserting at the beginning of the array, ***all*** of the elements in the array must be moved. Therefore, in the worst case, the time for insertion is proportional to the number of elements in the array, and we say that the worst-case time for the insertion operation is ***linear*** in the number of elements in the array. For a linear-time algorithm, if the problem size doubles, the number of operations also doubles.

Big-O notation

We express complexity using **big-O notation**.

For a problem of size N:

- a constant-time algorithm is "order 1": O(1)
- a linear-time algorithm is "order N": O(N)
- a quadratic-time algorithm is "order N squared": $O(N^2)$

Note that the big-O expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time algorithm will be faster than a linear-time algorithm, which will be faster than a quadratic-time algorithm).

Formal definition:

A function T(N) is O(F(N)) if for some constant c and for values of N greater than some value n_0:

$$T(N) <= c * F(N)$$

The idea is that T(N) is the ***exact*** complexity of a procedure/function/algorithm as a function of the problem size N, and that F(N) is an upper-bound on that complexity (i.e., the actual time/space or whatever for a problem of size N will be no worse than F(N)).

In practice, we want the smallest F(N) -- the ***least*** upper bound on the actual complexity. For example, consider:

$$T(N) = 3 * N^2 + 5.$$

We can show that T(N) is $O(N^2)$ by choosing c = 4 and $n_0 = 2$.

This is because for all values of N greater than 2:

$$3 * N^2 + 5 <= 4 * N^2$$

T(N) is **not** O(N), because whatever constant c and value n_0 you choose, There is always a value of $N > n_0$ such that $(3 * N^2 + 5) > (c * N)$

Determining Complexity

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

Sequence of statements

```
statement 1;
statement 2;
  ...
statement k;
```

The total time is found by adding the times for all statements:

total time = time(statement 1) + time(statement 2) + ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1).

If-Then-Else

```
if (cond) then
   block 1 (sequence of statements)
else
   block 2 (sequence of statements)
end if;
```

Here, either block 1 will execute, or block 2 will execute. Therefore, the worst-case time is the slower of the two possibilities:

max(time(block 1), time(block 2))

If block 1 takes O(1) and block 2 takes O(N), the if-then-else statement would be O(N).

Loops

```
for I in 1 .. N loop
   sequence of statements
end loop;
```

The loop executes N times, so the sequence of statements also executes N times. If we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

Nested loops

```
for I in 1 .. N loop
   for J in 1 .. M loop
      sequence of statements
   end loop;
end loop;
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M).

In a common special case where the stopping condition of the inner loop is `J < N` instead of `J < M` (i.e., the inner loop also executes N times), the total complexity for the two loops is O(N^2).

Statements with function/ procedure calls

When a statement involves a function/ procedure call, the complexity of the statement includes the complexity of the function/ procedure. Assume that you know that function/ procedure *f* takes constant time, and that function/procedure *g* takes time proportional to (linear in) the value of its parameter *k*. Then the statements below have the time complexities indicated.

f(k) has O(1)
g(k) has O(k)

When a loop is involved, the same rule applies. For example:

```
for J in 1 .. N loop
    g(J);
end loop;
```

has complexity (N^2). The loop executes N times and each function/procedure call `g(N)` is complexity O(N).

Recurrence Equations

A recurrence equation is just a recursive function definition. It defines a function on an input in terms of its values on smaller inputs.

Recurrence equations are used to characterize the running time of algorithms. T(n) typically stands for the running time (usually worst case) of a given algorithm on an input of size n.

We refer to algorithms that define a function in terms of itself but with a smaller input space as *divide and conquer* algorithms.

Recurrence equations for divide and conquer algorithms typically have the form:

General case ($n \geq 1$):
T(n) = [number of partitions] T (size of each partition) + cost of dividing and gluing

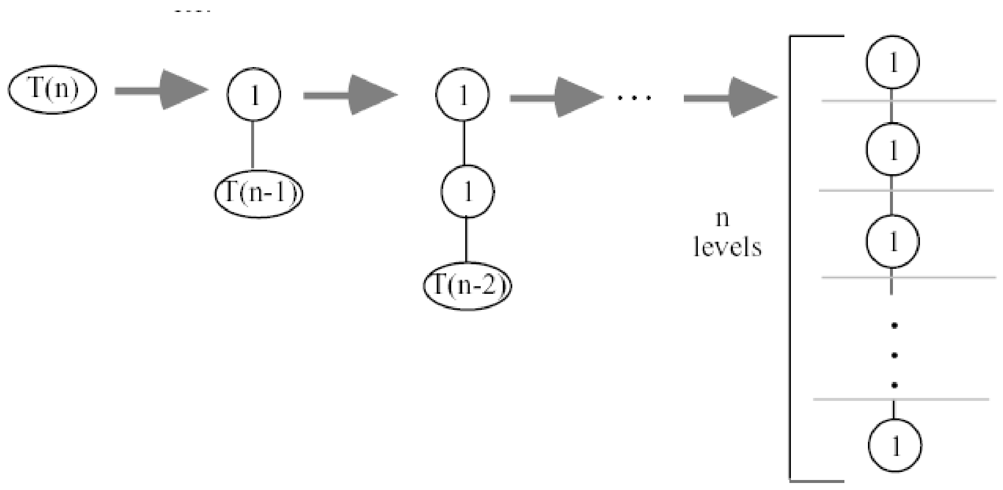
Base case ($n \leq 0$)

Techniques for solving Recurrences:

1. Recursion-Tree method
    a. Construct the tree by unwinding the recurrence equation.
    b. Determine the entire cost of the tree by summing the cost of the nodes.
2. Iteration
3. Master Theorem

Example 1: T(n) = T(n - 1) + 1
Substitute T(n - 1) = T(n - 2) + 1, T(n - 2) = T(n - 3) + 1 …..

What does the recursion tree look like?

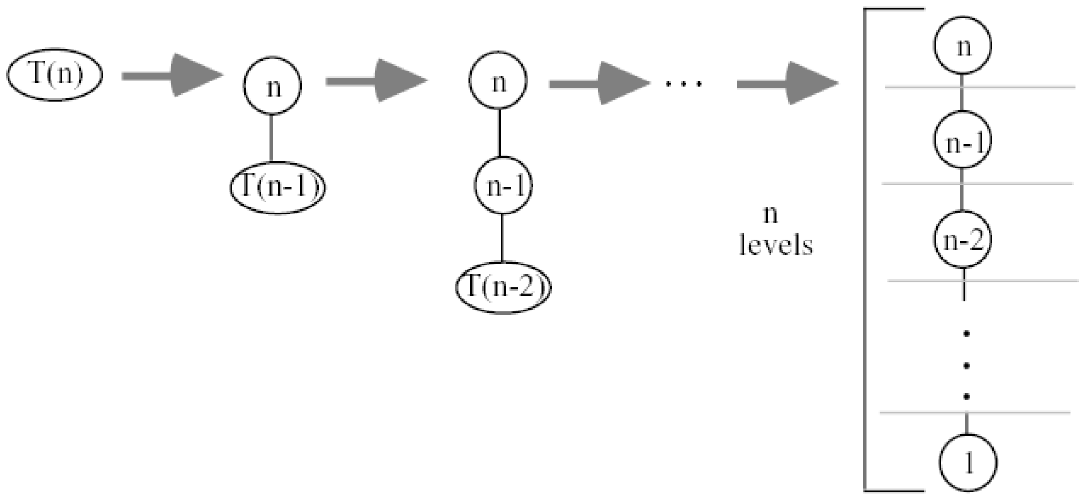


What is the total cost?

Example 2: T(n) = T(n - 1) + n

Substitute: T(n - 1) = T(n - 2) + (n - 1), T(n - 2) = T(n - 3) + (n - 2), ….

The Recursion Tree:



Cost = 1 + 2+ … + n which is an arithmetic series.

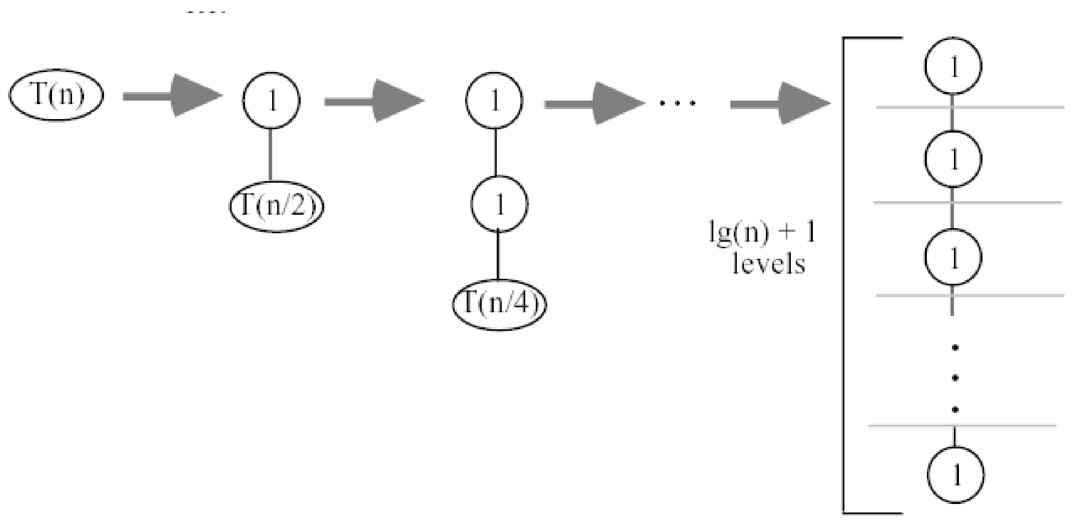
Note: A series is arithmetic if ak = c + ak-1, it can be computed using a formula

$$\sum_{k=1}^{n} a_k = \frac{n}{2}(a_1 + a_n)$$

Example 3: T(n) = T(n/2) + 1

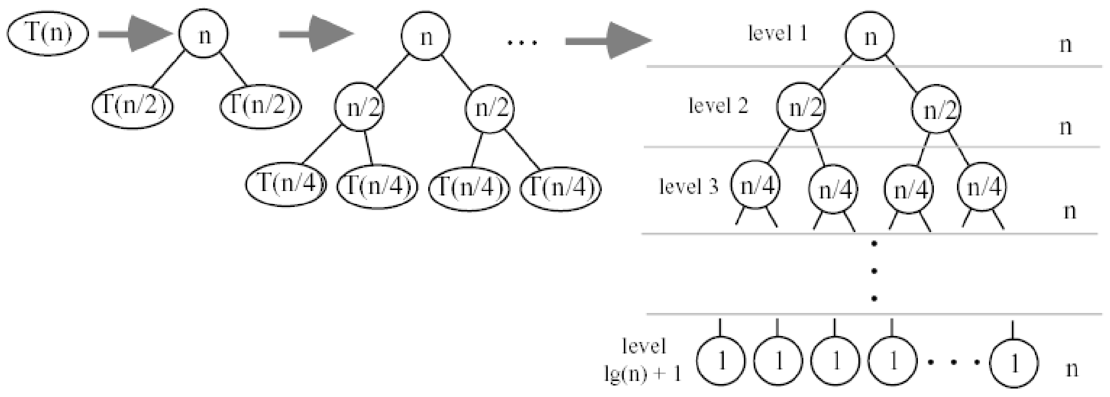
Substitute: T(n/2) = T(n/4) + 1, T(n /4) = T(n/8) + 1, ….

The recursion tree is



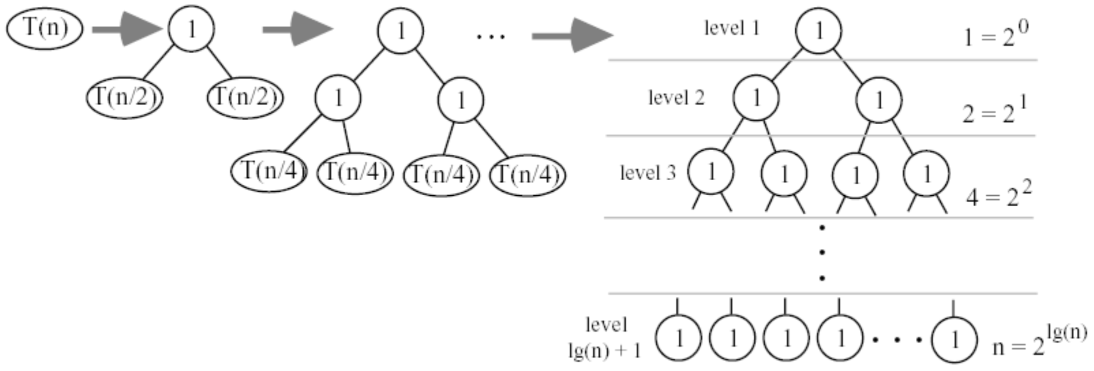
Example 4: T(n) = 2T(n/2) + n

Substitute: T(n/2) = 2T(n/4) + n/2, T(n/4) = 2T(n/8) + n/4, ….



Example 5: $T(n) = 2T(n/2) + 1$

Substitute: $T(n/2) = 2T(n/4) + 1$, $T(n/4) = 2T(n/8) + 1$, ….



Total cost = number of nodes

= sum of nodes at each level

= $1 + 2 + 4 + 8 + \ldots + 2^{\lg(n)}$ {Geometric series!}

$$= \sum_{k=0}^{\lg(n)} 2^k$$

Note: A series is geometric if it has the form $a_k = c.\ a_{k-1}$

Let $S(n) = \sum_{k=0}^{n} a_0 c^k = a_0 + a_0 c + a_0 c^2 + a_0 c^3 + \ldots + a_0 c^n$

$$\begin{aligned} cS(n) &= a_0 c + a_0 c^2 + a_0 c^3 + \ldots + a_0 c^n + a_0 c^{n+1} \\ -\quad S(n) &= a_0 + a_0 c + a_0 c^2 + a_0 c^3 + \ldots + a_0 c^n \end{aligned}$$

$(c - 1)\, S(n) = a_0 c^{n+1} - a_0 = a_0(c^{n+1} - 1)$

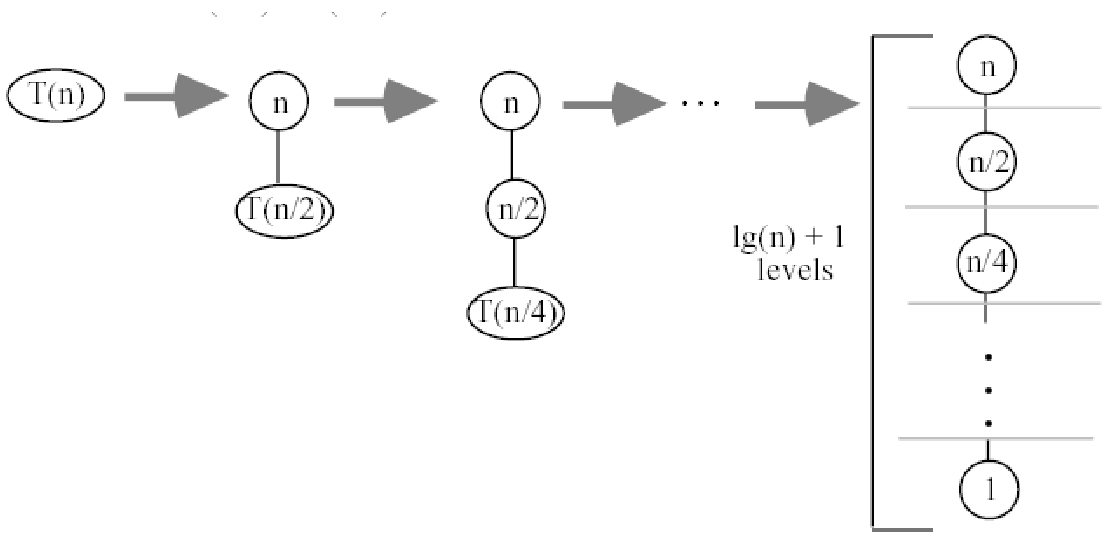
$$\Rightarrow \quad S(n) = \frac{a_0(c^{n+1} - 1)}{c - 1}$$

If $0 < c < 1$ and $n \to \infty$, the above formula can be rewritten as:

$$\lim_{n \to \infty} S(n) = \frac{a_0}{1 - c} \quad 0 < c < 1$$

Example 6: T(n) = T(n/2) + n

Substitute: T(n/2) = T(n/4) + n/2, T(n/4) = T(n/8) + n/4, ….



What is the formula?

Consider the recurrence equation

$$T(n) = \begin{cases} c & n = 1 \\ 2T\left(\frac{n}{2}\right) + c & n > 1 \end{cases}$$

$$\begin{aligned}
T(n) &= 2T(n/2) + c \\
&= 2(2T(n/2/2) + c) + c \\
&= 2^2T(n/2^2) + 2c + c \\
&= 2^2(2T(n/2^2/2) + c) + 3c \\
&= 2^3T(n/2^3) + 4c + 3c \\
&= 2^3T(n/2^3) + 7c \\
&= 2^3(2T(n/2^3/2) + c) + 7c \\
&= 2^4T(n/2^4) + 15c \\
&\dots \\
&= 2^kT(n/2^k) + (2^k - 1)c
\end{aligned}$$

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

Consider the recurrence equation

$$T(n) = \begin{cases} c & n = 1 \\ aT\left(\frac{n}{b}\right) + cn & n > 1 \end{cases}$$

$$\begin{aligned}
T(n) \quad &= \quad aT(n/b) + cn \\
&= \quad a(aT(n/b/b) + cn/b) + cn \\
&= \quad a^2T(n/b^2) + cna/b + cn \\
&= \quad a^2T(n/b^2) + cn(a/b + 1) \\
&= \quad a^2(aT(n/b^2/b) + cn/b^2) + cn(a/b + 1) \\
&= \quad a^3T(n/b^3) + cn(a^2/b^2) + cn(a/b + 1) \\
&= \quad a^3T(n/b^3) + cn(a^2/b^2 + a/b + 1) \\
&\dots \\
&= \quad a^kT(n/b^k) + cn(a^{k-1}/b^{k-1} + a^{k-2}/b^{k-2} + \dots + a^2/b^2 + a/b + 1)
\end{aligned}$$

if $k = \log_b n$, then $n = b^k$

Why?

$T(n) = cn(a^k/b^k + ... + a^2/b^2 + a/b + 1)$

If a = b, then the equation becomes cn (k+1) = cnlog(n) + cn = O(n log n)

Note again : $S(x^k + x^{k-1} + \dots + x + 1) = (x^{k+1} -1)/(x-1)$

If a< b then

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \frac{1 - (a/b)^{k+1}}{1 - (a/b)} \quad < \quad \frac{1}{1 - a/b}$$

T(n) = cn ·Q(1) = Q(n)

If a > b, then

$$\frac{a^k}{b^k} + \frac{a^{k-1}}{b^{k-1}} + \cdots + \frac{a}{b} + 1 \quad = \quad \frac{(a/b)^{k+1} - 1}{(a/b) - 1} \quad = \quad \Theta\left((a/b)^k\right)$$

Note: $\mathbf{a^{\log n} = n^{\log a}}$

T(n) $= cn \cdot \Theta(a^k / b^k)$
$= cn \cdot \Theta(a^{\log n} / b^{\log n})$
$= cn \cdot \Theta(a^{\log n} / n)$

Putting all three results together

$$T(n) = \begin{cases} \Theta(n) & a < b \\ \Theta(n \log_b n) & a = b \\ \Theta\left(n^{\log_b a}\right) & a > b \end{cases}$$

This is a specific instance of the master theorem. The simple but more generic version of the master theorem was presented in class.

Jayakanth Srinivasan
I. Kristina Lundqvist