# Algorithms

algorithm: a finite set of instructions that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems

An algorithm must possess the following properties:

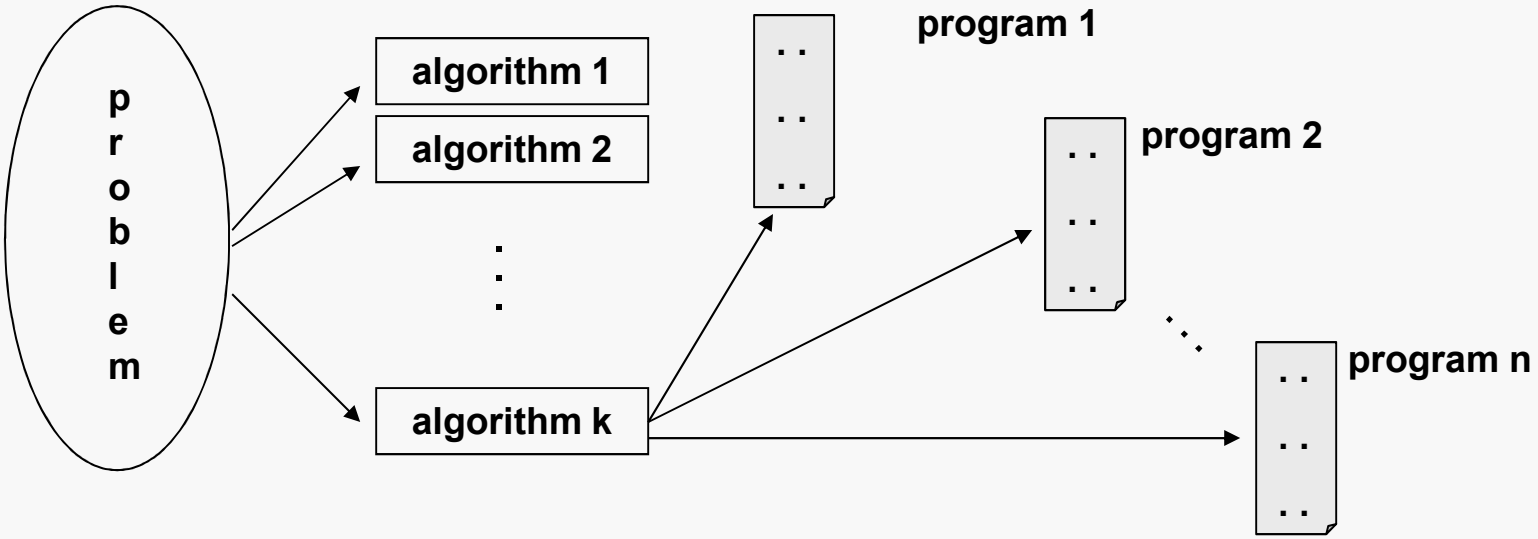
| | |
|---|---|
| finiteness: | Algorithm must complete after a finite number of instructions have been executed. |
| absence of ambiguity: | Each step must be clearly defined, having only one interpretation. |
| definition of sequence: | Each step must have a unique defined preceding & succeeding step. The first step (start step) & last step (halt step) must be clearly noted. |
| input/output: | Number and types of required inputs and results must be specified. |
| feasibility: | It must be possible to perform each instruction. |

# Algorithms vs Programs

program: the concrete expression of an algorithm in a particular programming language

Given a problem to solve, the design phase produces an algorithm.

The implementation phase then produces a program that expresses the designed algorithm.



# Design Considerations

Given a particular problem, there are typically a number of different algorithms that will solve that problem. A designer must make a rational choice among those algorithms.

Design considerations:

- to design an algorithm that is easy to understand, implement, and debug (software engineering)
- to design an algorithm that makes efficient use of the available computational resources (data structures and algorithm analysis)

We will be primarily concerned with the second area.

But, how do we measure the efficiency of an algorithm?

Note that the number of operations to be performed and the space required will depend on the number of input values that must be processed.

# Benchmarking

It is tempting to measure the efficiency of an algorithm by producing an implementation and then performing benchmarking analyses by running the program on input data of varying sizes and measuring the "wall clock" time for execution.

However:

- the program may be a poor representation of the algorithm's possibilities.
- the results will depend upon the particular characteristics of the hardware used for the benchmarking, perhaps in subtle ways.
- the choice of test data may not provide a representative sampling of the various factors that influence the algorithm's behavior

# Complexity Analysis

Complexity analysis is the systematic study of the cost of a computation, measured either in time units or in operations performed, or in the amount of storage space required.

The goal is to have a meaningful measure that permits comparison of algorithms and/or implementations independent of operating platform.

Complexity analysis involves two distinct phases:

- algorithm analysis: analysis of the algorithm or data structure to produce a function T(n) measuring the complexity
- order of magnitude (asymptotic) analysis: analysis of the function T(n) to determine the general complexity category to which it belongs.

# Algorithm Analysis

Algorithm analysis requires a set of rules to determine how operations are to be counted.

There is no generally accepted set of rules for algorithm analysis.

In some cases, an exact count of operations is desired; in other cases, a general approximation is sufficient.

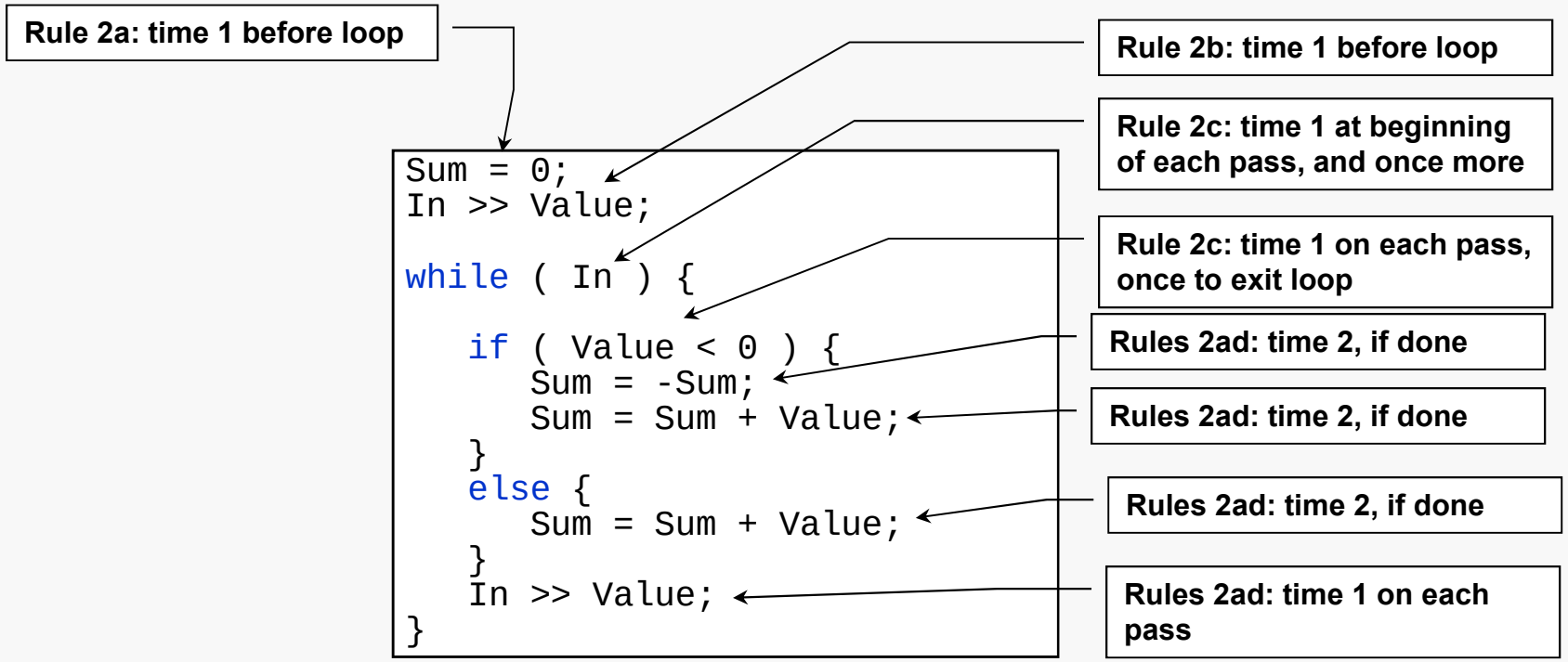
The rules presented that follow are typical of those intended to produce an exact count of operations.

# Exact Analysis Rules

1. We assume an arbitrary time unit.
2. Execution of one of the following operations takes time 1:
   a) assignment operation
   b) single I/O operations
   c) single Boolean operations, numeric comparisons
   d) single arithmetic operations
   e) function return
   f) array index operations, pointer dereferences
3. Running time of a selection statement (if, switch) is the time for the condition evaluation + the maximum of the running times for the individual clauses in the selection.
4. Loop execution time is the sum, over the number of times the loop is executed, of the body time + time for the loop check and update operations, + time for the loop setup.

   Always assume that the loop executes the maximum number of iterations possible
5. Running time of a function call is 1 for setup + the time for any parameter calculations + the time required for the execution of the function body.

# Analysis Example 1




So, assuming n input values are received, the total time T(n) is given by:

$$T(n) = 2 + \sum_{k=1}^{n} \big(3 + \max(4,2)\big) + 1 = 7n + 3$$

# Analysis Example 2

**Given:**

```
for (i = 0; i < n-1; i++) {
    for (j = 0; j <= i; j++) {
        aray[i][j] = 0;
    }
}
```

**Rules 4 and 2a: time 1 before loop**

**Rules 4, 2c and 2d: time 3 on each iteration of outer loop, and one more test to exit**

**Rules 4 and 2a: time 1 on each iteration of outer loop**

**Rules 4, 2c and 2d: time 2 (on each iteration of inner loop) and one more test to exit**

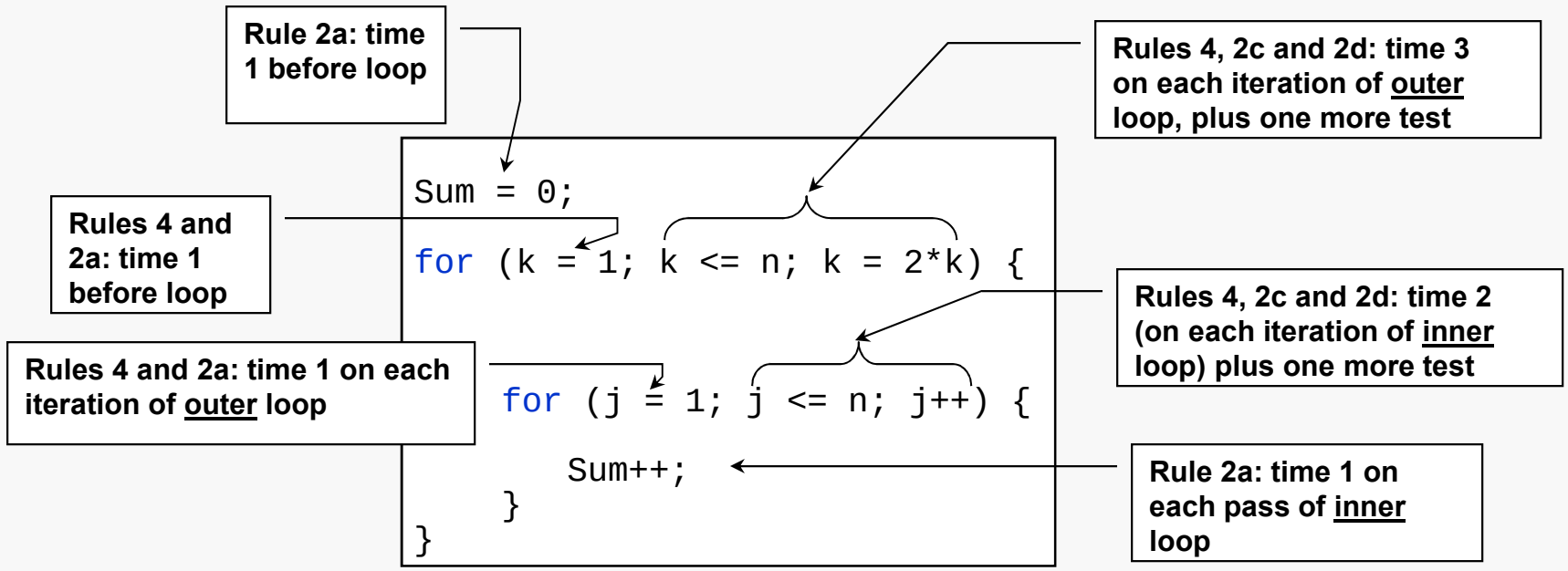
**Rule 2a and 2f: time 3 on each pass of inner loop**

So, the total time T(n) is given by:

$$T(n) = 1 + \sum_{i=1}^{n-1} \left( 4 + \sum_{j=1}^{i} 5 + 1 \right) + 2 = \frac{5}{2} n^2 - \frac{5}{2} n - 2$$

# Analysis Example 3






The tricky part is that the outer loop will be executed about log(n) times. Precisely, since n equals $2^{\log(n)}$, we can argue that if p is the number of the current pass (numbering starting at 1) then:

$$k \le n \leftrightarrow 2^{p-1} \le 2^{\log n} \leftrightarrow p-1 \le \log n \leftrightarrow p-1 \le \lfloor \log n \rfloor \leftrightarrow p \le 1 + \lfloor \log n \rfloor$$

Here, $\lfloor x \rfloor$ is the largest integer that's less than or equal to x, commonly called the *floor*.

# Analysis Example 3

So, the total time T(n) for the previous algorithm is given by:

$$T(n) = 2 + \sum_{p=1}^{1+\lfloor \log n \rfloor} \left( 4 + \sum_{j=1}^{n} 3 + 1 \right) + 1 = 3n\lfloor \log n \rfloor + 3n + 5\lfloor \log n \rfloor + 8$$

If we assume that n is a power of 2, the floor notation may be dropped. It is common to do so when expressing complexity functions, since the difference is minor.

# Analysis Example 4

Now let's consider a simple linear search function:

```
int linearSearch(int List[], int Target, int Sz) {


   for (int i = 0; i < Sz; i++) {          // 1 before, 2 each pass, 1 exit
      if ( Target == List[i] )             // 2
         return i;                         // 1, if done
   }

   return Sz;                              // 1, if done
}
```

The worst-case cost would be incurred if `Target` does not occur in `List`. In that case:

$$T(N) = 1 + \sum_{i=0}^{N-1} 4 + 1 + 1 = 4N + 3$$

The best-case cost would be incurred if `Target` occurs at index 0 in `List`. In that case:

$$T(N) = 5$$

# Analysis Example 4

What about the average cost? If `Target` occurs at index `K` in `List`, the cost of the search would be:

$$T(N,K) = 1 + \sum_{i=0}^{K-1} 4 + 1 + 2 + 1 = 4K + 5$$

The average-case cost, assuming `Target` is in `List` would be:

$$T(N) = \frac{1}{N}\sum_{K=0}^{N-1}(4K+5) = \frac{1}{N}\left(4\frac{(N-1)N}{2} + 5N\right) = 2N + 3$$

The true average-case cost would depend on the probability that `Target` does occur in the list. Obviously, the cost of the search when Target is not in the list would be worst case cost found earlier. But the true average cost would depend on how many searches did achieve the worst case performance.

As a side note, if we only count comparisons of data objects, the average-case cost, assuming `Target` is in `List` would be:

$$\overline{C}(N) = \frac{N+1}{2}$$

# Analysis Example 5

Now let's consider a simple binary search function:

```
int binSearch(int List[], int Target, int Sz) {

   unsigned int Mid,
                Lo = 0,        // 1
                Hi = Sz - 1;   // 2

   while ( Lo <= Hi ) {                // 1 per pass + 1 for exit, if done

      Mid = (Lo + Hi) / 2;             // 3

      if ( List[Mid] == Target )       // 2
         return Mid;                   // 1, if done

      else if ( Target < List[Mid] ) // 2, if done
         Hi = Mid - 1;                 // 2, if done

      else
         Lo = Mid + 1;                 // 2, if done
   }

   return Sz;                          // 1, if done
}
```

The worst-case cost of one pass through the loop is easily seen to be 10...

# Analysis Example 5

… but how many passes will be required, in the worst case?

Consider the loop condition in this form: `Hi - Lo >= 0`

Each pass through the loop (worst case) either raises `Lo` or lowers `Hi`. No matter which is done, simple algebra reveals that the successive loop tests are just:

$$\frac{Hi-Lo}{2}-1 \qquad \frac{Hi-Lo}{4}-\frac{3}{2} \qquad \frac{Hi-Lo}{8}-\frac{7}{4} \qquad \frac{Hi-Lo}{16}-\frac{15}{8}$$

The denominator is $2^p$ where p is the number of the pass about to be performed (starting with p = 0). The constant term is bounded by 2.

Now, from the initializations of `Hi` and `Lo`, the value of `Hi - Lo` is just `Sz - 1`. So, the question is essentially, when will we achieve:

$$\frac{Sz-1}{2^p} < 2-\frac{1}{2^{p-1}}$$

That's messy, but it's easy enough to show we need $p > \log_2(Sz+1)-1$

So, the binary search function is about: $T(N) = 10\log(N+1)+5$

# Why Does Complexity Matter?

Consider the following chart of some simple complexity functions:

| n | log n | n | n log n | n^2 | n^3 | 2^n |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 | 2.E+00 |
| 10 | 3 | 10 | 33 | 100 | 1000 | 1.E+03 |
| 20 | 4 | 20 | 86 | 400 | 8000 | 1.E+06 |
| 30 | 5 | 30 | 147 | 900 | 27000 | 1.E+09 |
| 40 | 5 | 40 | 213 | 1600 | 64000 | 1.E+12 |
| 50 | 6 | 50 | 282 | 2500 | 125000 | 1.E+15 |
| 60 | 6 | 60 | 354 | 3600 | 216000 | 1.E+18 |
| 70 | 6 | 70 | 429 | 4900 | 343000 | 1.E+21 |
| 80 | 6 | 80 | 506 | 6400 | 512000 | 1.E+24 |
| 90 | 6 | 90 | 584 | 8100 | 729000 | 1.E+27 |
| 100 | 7 | 100 | 664 | 10000 | 1000000 | 1.E+30 |

# Running Time Estimation

Suppose we have hardware capable of executing $10^6$ instructions per second.

How long would it take to execute an algorithm whose complexity function was:

$$T(N) = N^2$$

on an input of size N = $10^8$?

The total number of operations to be performed would be T($10^8$):

$$T(10^8) = (10^8)^2 = 10^{16}$$

The total number of seconds required would be given by T($10^8$)/$10^6$ so:

$$\text{Running Time} = 10^{16} / 10^6 = 10^{10}$$

The number of seconds/day is 86,400 so this is about 115,740 days (317 years).

# Running Time Estimation

What if we used an algorithm whose complexity function was:

$$T(N) = N \log N$$

on an input of size N = $10^8$?

The total number of operations to be performed would be T($10^8$):

$$T(10^8) = (10^8)\log(10^8) \approx 2.66 \times 10^9$$

The total number of seconds required would be given by T($10^8$)/$10^6$ so:

$$\text{Running Time} \approx 2.66 \times 10^9 / 10^6 = 2.66 \times 10^3$$

This is about 44.33 minutes.

# Maximum Problem Size

Another way of looking at this is to ask, what is the largest problem size that can be handled in a given amount of time, given a complexity function for an algorithm and the hardware speed?

Assuming the same hardware speed as before, what's the largest input size that could be handled in one hour, if the complexity function is once again:

$$T(N) = N^2$$

The total number of seconds required would again be given by T(N)/$10^6$ so we're asking what is the maximum value of N for which:

$$\frac{T(N)}{10^6} \leq 3600 \quad \text{or} \quad N^2 \leq 3600 \times 10^6$$

This yields $N \leq 60{,}000$

# Maximum Problem Size

Applying the same logic, with the complexity function: $T(N) = N \log N$

The total number of seconds required would be T(N)/$10^6$ so we're asking what is the maximum value of N for which:

$$N \log N \leq 3600 \times 10^6$$

Solving for equality (Newton's Method) yields about $N \leq 133{,}000{,}000$

The first moral is that for large N, the complexity function matters.

The minor first moral is that for large N, Nlog(N) is a LOT faster than $N^2$.

The second moral involves applying this logic to the question of hardware speedup…

# Faster Hardware?

If we apply the same analysis, assuming that we can now find hardware that is, say, 100 times faster than the previous hardware (so $10^8$ operations per second), the results are revealing:

| T(N) | time for N = $10^8$ | max N in 1 hour |
|---|---|---|
| N log(N) | .4433 minutes | ~10 billion |
| $N^2$ | 3.17 years | ~600,000 |

Comparing to the earlier results, speeding up the hardware by a factor of 100:

- reduces time for same sized problem by a factor of 100 in both cases, so the relative advantage of the N log(N) algorithm is retained
- increases the max problem size by a factor of 10 for the $N^2$ case, versus a factor of almost 75 for the N log(N) case