## Algorithms and Algorithm Analysis

Algorithm: A clearly specified set of instructions the computer will follow to solve a problem.

Given an algorithm, we want to determine the amount of memory it uses, and how much time it requires to solve a problem.

## Etymology of “Algorithm”

Algorism = process of doing arithmetic using Arabic numerals.

A misperception: algiros [painful] + arithmos [number].

True origin: Abū ’Abdallāh Muhammad ibn Mūsā al-Khwārizmī was a 9th-century Persian mathematician, astronomer, and geographer, who wrote Kitab al-jabr wa’l-muqabala (Rules of restoring and equating), which evolved into today’s high school mathematics text.

## Maximum contiguous subsequence sum

Given an array with integers $a_1, a_2, \ldots, a_n$, find the maximum value of $\sum_{k=i}^{j} a_k$.

4, -3, 5, -2, -1, 2, 6, -2

How many possible subsequences are there?

## The naive algorithm

```
var maxSum = 0
for (i <- 0 until A.length) {
  for (j <- i until A.length) {
    var sum = 0
    for (k <- i to j)
      sum += A(k)
    if (sum > maxSum)
      maxSum   = sum
  }
}
```

Number of additions: $\sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 1)$

## A faster algorithm

```
var maxSum = 0
for (i <- 0 until A.length) {
  var sum = 0
  for (j <- i until A.length) {
    sum += A(j)
      if (sum > maxSum)
        maxSum = sum
  }
}
```

Number of additions: $\sum_{i=0}^{n-1}(n-i)$

## A recursive algorithm

How can we apply recursion to this problem?

4, -3, 5, -2, | -1, 2, 6, -2

Split the array in the middle.

(1) The maximal subsequence is in the left half.
(2) The maximal subsequence is in the right half.
(3) The maximal subsequence begins in the left half and ends in the right half.

How many additions?

## Divide and Conquer (Divide et impera)

- Split the problem into subproblems.
- Solve the subproblems recursively.
- Combine the solutions to the subproblems.

## Experimental analysis of algorithms

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like `scala.compat.Platform.currentTime` to get an accurate measure of the actual running time

| $n$ | Naive | Faster | Recursive |
|---:|---:|---:|---:|
| 10 | 2 | 1 | 1 |
| 100 | 760 | 31 | 19 |
| 1,000 | 652,285 | 2,411 | 236 |
| 10,000 | – | 218,210 | 2,378 |
| 100,000 | – | 23,033,000 | 25,037 |
| 1,000,000 | – | – | 260,375 |

## Experimental analysis of algorithms

Limitations:

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used. Programming language, programming style, fine tuning should not be measured.

## Theoretical analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size $n$
- Takes into account all possible inputs, and looks at worst-case
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Theoretical analysis counts primitive operations.

Primitive operations are:

- Assigning a value to a variable
- Calling a method
- Arithmetic operations (e.g. adding two numbers)
- Indexing into an array
- Following a reference
- Returning from a method

## Counting primitive operations

While we do not know the exact cost of a primitive operation (it depends on the processor speed, processor architecture, programming language, compiler, etc.), we know that all primitive operations take constant time.

There is a fixed, finite number of primitive operations.
Let $a$ be the time taken by the fastest primitive operation, let $b$ be the time taken by the slowest primitive operation.
If our algorithm uses $k$ primitive operations, then its running time $T(n)$ is bounded by

$$ak \leq T(n) \leq bk$$

## Simplifying the analysis

We are more interested in the growth rate of the running time than in the exact formula. A quadratic algorithm will always be faster than a cubic algorithm if the input size is sufficiently large.

The growth rate determines the scaling behavior of an algorithm: If we increase the problem size by a factor $10$, how much does the running time increase?

Or, put differently: If we buy a computer that is ten times faster, how much larger problems can we solve?

| Time complexity | Problem size after speedup |
|---|---|
| $n$ | $10s$ |
| $n^2$ | $3.16s$ |
| $n^3$ | $2.15s$ |
| $2^n$ | $s + 3.3$ |

## Big-Oh notation

Since we only want to know the growth rate of an algorithm, we can simplify the analysis using Big-Oh notation.

Definition of Big-Oh:
Let $f(n)$, $g(n)$ be functions from $\{1, 2, 3, 4, \ldots\}$ to $\mathbb{R}$.
We say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer $n_0 \geq 1$ such that

$$f(n) \leq cg(n) \qquad \text{for } n \geq n_0.$$

$4n + 1$ is $O(n)$

$2n^2 + 3n + 5$ is not $O(n)$

$2n^2 + 3n + 5$ is $O(n^2)$

## Simplest terms

We want to express the running time in the simplest possible Big-Oh notation.

$4n \log n + 3n - 12$ is $O(n \log n + 3n)$ is correct,
but we should say that it is $O(n \log n)$.

Any polynomial

$$f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \cdots + a_d n^d$$

with $a_d > 0$ is just $O(n^d)$.

$5n^2 + 3n \log n + 2n + 5$ is $O(n^2)$
$20n^3 + 10n \log n + 5$ is $O(n^3)$
$3 \log n + 2$ is $O(\log n)$
$2^{n+2}$ is $O(2^n)$
$2n + 100 \log n$ is $O(n)$

## Asymptotic analysis

The asymptotic analysis of an algorithm determines the running time in big- Oh notation.
To perform the asymptotic analysis
- We find the worst-case number of primitive operations executed as a function of the input size
- We express this function with big-Oh notation

Since constant factors and lower- order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

A word of caution:
What is better, $10^{100} n$, $n^{100}$, or $2^n$?

## Keeping your perspective

- Throughout the course of an analysis, keep in mind that you are interested only in significant differences in efficiency
- When choosing an ADT implementation, consider how frequently particular ADT operations occur in a given application
- Some seldom-used but critical operations must be efficient
- If the problem size is always small, you can probably ignore an algorithm's efficiency
- Weigh the trade-offs between an algorithm's time requirements and its memory requirements