Algorithm Complexity

Lionel Kitihoun

Introduction

We often need to evaluate a program cost, in terms of:

- processor usage (number of operations needed to solve the problem),
- memory usage (how much RAM will we need).

Why?

- We often have limited resources.
  - We have a lot of other things to do.
  - We need the result of our program as soon as possible.
  - Computation power and memory are costly.
- We have several programs that can solve our problem.
  - We need to pick the best one.

Algorithm complexity

Complexity of an algorithm is a measure of the amount of time or space required by an algorithm for an input of a given size (n)[1].

[1] http://www.dcs.gla.ac.uk/~pat/52233/complexity.html

Your task

You should be able to deduce the complexity of a given algorithm in order to decide if it fits your need.

Initiation examples

Three examples.

- Sum of the first N positive integers.
- Primality test.
- Fast exponentiation.

Sum of the first N positive integers (Naive)

```
function sum(n: int): int
var
  s: integer
  i: integer
begin
  s := 0
  for i := 1 to n
      s = s + i
  return s
end
```

Sum of the first N positive integers (Naive)

So, how many operations are needed?

Answer

This program needs $N + 1$ operations to get the answer.

Sum of the first N positive integers (Better)

```
function sum(n: int): int
var
  s: integer
begin
  s := n * (n + 1) / 2
  return s
end
```

Number of operations

This version only needs one instruction[2].

[2] Okay, three operations. But you get the idea.

Make your choice

I prefer the second version.

- It is faster, imagine if $N = 10^{10}$.
- It makes me look smart.

Second example: primality test

Given a number, check if it is prime or not.

Primality test: first take

Easy: just divide the number by all other integers below it.

Primality test pseudo-code

```
function is_prime(n: int): boolean
  var
    i: integer
  begin
    for i := 2 to n-1
      if n mod i = 0
        return false
    return true
end
```

Question

How many operations are needed?

Answer

- We loop $N - 2$ times.
- In each loop we do one test.
- $N - 2$ operations overall.

Primality test: better version

```
function is_prime(n: int): boolean
var
  i: integer
begin
  while i * i <= n
    if n mod i = 0
      return false
  return true
```

Cost

- We check every number up to $\sqrt{N}$.
- So, we do at most $\sqrt{N}$ operations.
- Better than the other version.

Last example: fast exponentiation

Given a real number x and an integer n, compute x^n.

Fast exponentiation insight

- No loop needed.
- We just need one key observation.

Fast exponentiation trick

$$x^n = \begin{cases} x^{n/2} * x^{n/2}, \ \ if\ n\ is\ even, \\ x * x^{n/2} * x^{n/2}, \ \ if\ n\ is\ odd \end{cases}$$

Pseudo-code

```
function exp(x: real, n: int): real
var
  h: integer
  p: real
begin
  if n = 0
    return 1
  h = n / 2
  p = exp(x, h)
  if n mod 2 = 0
    return h * h
  else
    return x * h * h
end
```

Cost

- You need a bit of analysis.
- The number of recursive call is roughly equal to $ln_2(n)$.
- We will say that we need $ln(n)$ recursive calls.

Next

This is just an introduction. Now, the real slides.