CSU22011: ALGORITHMS AND DATA STRUCTURES

Lecture 3: Mathematical Approach to the Analysis of Algorithms

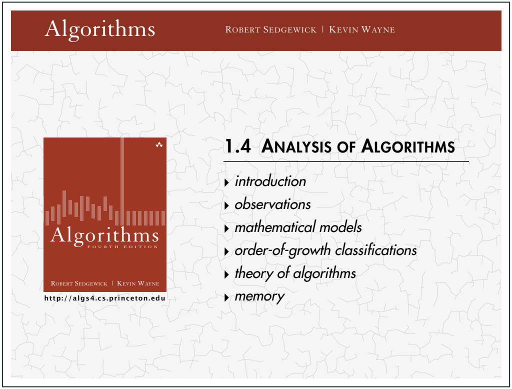
Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

Analysis of Algorithms: evaluates the efficiency of our programs

- How the **running time/memory footprint** of the algorithm scales when the input size increases.
  - Express running time as a function T(N), where N is the size of the input.
- For any given input size N, we will be focusing on the worst-case inputs (the worst case value of T(N))
- **Scientific method**: measure running times through experiments and discover T(N)

TODAY



Mathematical methods – consider a model of computation and count the number of program steps for worst-case inputs of size N.

- → Parts from S&W 1.4
- → Evaluate the performance of algorithms by Mathematical Calculations

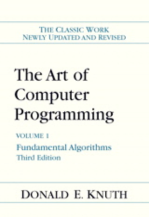
Mathematical Approach:

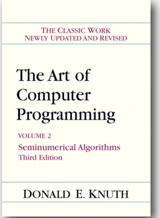
Evaluating Performance by **Calculations**

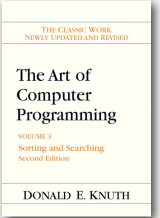
Mathematical models for running time

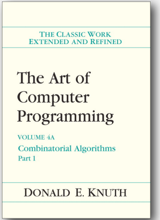
Total running time: sum of cost × frequency for all operations.

- Need to analyze program to determine set of operations.
- Cost depends on machine, compiler.
- Frequency depends on algorithm, input data.









Donald Knuth
1974 Turing Award

In principle, accurate mathematical models are available.

Cost of basic operations

Challenge. How to estimate constants.

operation	example	nanoseconds †
integer add	`a + b`	2.1
integer multiply	`a * b`	2.4
integer divide	`a / b`	5.4
floating-point add	`a + b`	4.6
floating-point multiply	`a * b`	4.2
floating-point divide	`a / b`	13.5
sine	`Math.sin(theta)`	91.3
arctangent	`Math.atan2(y, x)`	129.0
...	...	...

† Running OS X on Macbook Pro 2.2GHz with 2GB RAM

Cost of basic operations

Observation. Most primitive operations take constant time.

operation	example	nanoseconds †
variable declaration	`int a`	c_1
assignment statement	`a = b`	c_2
integer compare	`a < b`	c_3
array element access	`a[i]`	c_4
array length	`a.length`	c_5
1D array allocation	`new int[N]`	$c_6\ N$
2D array allocation	`new int[N][N]`	$c_7\ N^2$

Caveat. Non-primitive operations often take more than constant time.

novice mistake: abusive string concatenation

Example: 1-SUM

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   if (a[i] == 0)
      count++;
```

N array accesses

operation	frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	N to $2N$

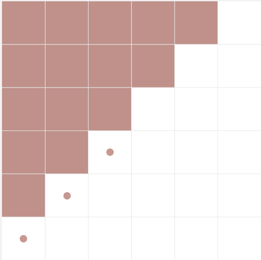
Example: 2-SUM

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0 + 1 + 2 + \ldots + (N-1) \;=\; \frac{1}{2}N(N-1) \;=\; \binom{N}{2}$$

Pf. [n even]



$$0 + 1 + 2 + \ldots + (N-1) \;=\; \underbrace{\frac{1}{2}N^2}_{\text{half of square}} - \underbrace{\frac{1}{2}N}_{\text{half of diagonal}}$$

String theory infinite sum

$$1 + 2 + 3 + 4 + \ \ldots = \ -\frac{1}{12}$$

http://www.nytimes.com/2014/02/04/science/in-the-end-it-all-adds-up-to.html

Example: 2-SUM

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N(N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$½ (N+1)(N+2)$
equal to compare	$½ N(N-1)$
array access	$N(N-1)$
increment	$½ N(N-1)$ to $N(N-1)$

tedious to count exactly

Example: 2-SUM

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0 + 1 + 2 + \ldots + (N-1) \;=\; \frac{1}{2} N(N-1) \;=\; \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2}(N+1)(N+2)$
equal to compare	$\frac{1}{2} N(N-1)$
array access	$N(N-1)$
increment	$\frac{1}{2} N(N-1)$ to $N(N-1)$

tedious to count exactly

"best case" vs "worst case" input of size N. We are

CALCULATING PRECISE RUNNING TIME

$$T(N) = c_1 A(N) + c_2 B(N) + c_3 C(N) + c_4 D(N) + c_5 E(N)$$

Where

c_1	:cost of array access	$A(N)$	:number of array accesses
c_2	:cost of integer addition	$B(N)$	:number of integer additions
c_3	:cost of integer comparison	$C(N)$	:number of integer comparisons
c_4	:cost of increment	$D(N)$	:number of increments
c_5	:cost of assignment	$E(N)$	:number of assignments
			(functions of the input size N)

CALCULATING PRECISE RUNNING TIME

$$T(N) = c_1 A(N) + c_2 B(N) + c_3 C(N) + c_4 D(N) + c_5 E(N)$$

Where

c_1 :cost of array access	$A(N)$:number of array accesses
c_2 :cost of integer addition	$B(N)$:number of integer additions
c_3 :cost of integer comparison	$C(N)$:number of integer comparisons
c_4 :cost of increment	$D(N)$:number of increments
c_5 :cost of assignment	$E(N)$:number of assignments
	(functions of the input size N)

Q. Advantages / Disadvantages over scientific method?

Mathematical models for running time

In principle, accurate mathematical models are available.

In practice,

- Formulas can be complicated.
- Advanced mathematics might be required.
- Exact models best left for experts.

costs (depend on machine, compiler)

$$T_N = c_1 A + c_2 B + c_3 C + c_4 D + c_5 E$$

$A =$ array access

$B =$ integer add

$C =$ integer compare

$D =$ increment

$E =$ variable assignment

frequencies (depend on algorithm, input)

Bottom line. We use approximate models in this course: $T(N) \sim c N^3$.

We will simplify calculations using approximations based on Cost Models

Cost Model 1: all constant costs = 1

COST MODEL 1: ALL CONSTANT COSTS = 1

→ New generation computers have smaller constants than previous generation

$$\boxed{c_i = 1}$$

$$T_N = A + B + C + D + E$$

Where

A :number of array accesses

B :number of integer additions

C :number of integer comparisons

D :number of increments

E :number of assignments

Careful!

There are operations that **do not** have a constant cost:

- → Naive string concatenation: `s = str + "ABCDEFG";`
- → Method calls: `max = Collections.max(myList);`

Careful!

There are operations that **do not** have a constant cost:

- Naive string concatenation: `s = str + "ABCDEFG";`
  - the cost of this operation is linear to the size of `str`

- Method calls: `max = Collections.max(myList);`

Careful!

There are operations that **do not** have a constant cost:

- Naive string concatenation: `s = str + ”ABCDEFG”;`
  - the cost of this operation is linear to the size of `str`
  - when efficiency is important use `StringBuilder`
- Method calls: `max = Collections.max(myList);`

Careful!

There are operations that **do not** have a constant cost:

- Naive string concatenation: `s = str + "ABCDEFG";`
  - the cost of this operation is linear to the size of `str`
  - when efficiency is important use `StringBuilder`
- Method calls: `max = Collections.max(myList);`
  - the cost of this operation is the cost of running the algorithm in `Collections.max` with an input of size `myList.size()`

Example: 2-SUM

Q. How many instructions as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$½ (N + 1) (N + 2)$
equal to compare	$½ N (N - 1)$
array access	$N (N - 1)$
increment	$½ N (N - 1)$ to $N (N - 1)$

tedious to count exactly

Estimate performance by adding up frequencies

Cost model 2: only highest order terms count

Simplification 2: tilde notation

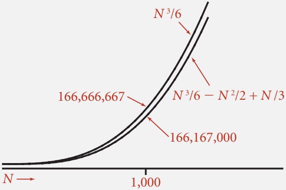
- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care

Ex 1. $\frac{1}{6} N^3 + 20 N + 16 \quad \sim \frac{1}{6} N^3$

Ex 2. $\frac{1}{6} N^3 + 100 N^{4/3} + 56 \quad \sim \frac{1}{6} N^3$

Ex 3. $\frac{1}{6} N^3 - \underbrace{\frac{1}{2} N^2 + \frac{1}{3} N} \quad \sim \frac{1}{6} N^3$

discard lower-order terms
(e.g., N = 1000: 166.67 million vs. 166.17 million)



Leading-term approximation

Technical definition. $f(N) \sim g(N)$ means $\displaystyle\lim_{N \to \infty} \frac{f(N)}{g(N)} = 1$

Simplification 2: tilde notation

- Estimate running time (or memory) as a function of input size N.
- Ignore lower order terms.
  - when N is large, terms are negligible
  - when N is small, we don't care

operation	frequency	tilde notation
variable declaration	$N + 2$	$\sim N$
assignment statement	$N + 2$	$\sim N$
less than compare	$\frac{1}{2}(N+1)(N+2)$	$\sim \frac{1}{2} N^2$
equal to compare	$\frac{1}{2} N(N-1)$	$\sim \frac{1}{2} N^2$
array access	$N(N-1)$	$\sim N^2$
increment	$\frac{1}{2} N(N-1)$ to $N(N-1)$	$\sim \frac{1}{2} N^2$ to $\sim N^2$

Estimate performance by adding up **simplified** frequencies

Cost model 3: count only SOME operations

Simplifying the calculations

" It is convenient to have a measure of the amount of work involved in a computing process, even though it be a very crude one. We may count up the number of times that various elementary operations are applied in the whole process and then given them various weights. We might, for instance, count the number of additions, subtractions, multiplications, divisions, recording of numbers, and extractions of figures from tables. In the case of computing with matrices most of the work consists of multiplications and writing down numbers, and we shall therefore only attempt to count the number of multiplications and recordings. " — *Alan Turing*

ROUNDING-OFF ERRORS IN MATRIX PROCESSES

By A. M. TURING

(*National Physical Laboratory, Teddington, Middlesex*)

[Received 4 November 1947]

SUMMARY

A number of methods of solving sets of linear equations and inverting matrices are discussed. The theory of the rounding-off errors involved is investigated for some of the methods. In all cases examined, including the well-known 'Gauss elimination process', it is found that the errors are normally quite moderate: no exponential build-up need occur.

Simplification 1: cost model

Cost model. Use some basic operation as a proxy for running time.

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N(N-1) = \binom{N}{2}$$

operation	frequency
variable declaration	$N + 2$
assignment statement	$N + 2$
less than compare	$\frac{1}{2}(N+1)(N+2)$
equal to compare	$\frac{1}{2} N(N-1)$
array access	$\boxed{N(N-1)}$
increment	$\frac{1}{2} N(N-1)$ to $N(N-1)$

cost model = array accesses

(we assume compiler/JVM do not optimize any array accesses away!)

Performance estimate = (array accesses) $\times c_{\text{arraccess}}$

DON'T OVER-SIMPLIFY!

Careful!

Make sure that the operations you are not counting add up to a factor **lower** than the operations you do count.

Combinations of Cost Models

Each cost model makes a **simplification** in the calculation of running time.

$\Longrightarrow$ approximation of running time.

We can even **combine** the assumptions of different cost models.

Example: 2-SUM

Q. Approximately how many array accesses as a function of input size N?

```
int count = 0;
for (int i = 0; i < N; i++)
   for (int j = i+1; j < N; j++)
      if (a[i] + a[j] == 0)
         count++;
```

"inner loop"

$$0 + 1 + 2 + \ldots + (N-1) = \frac{1}{2} N (N-1) = \binom{N}{2}$$

A. $\sim N^2$ array accesses.

Performance estimate = simplified number of array accesses

Bottom line. Use cost model and tilde notation to simplify counts.