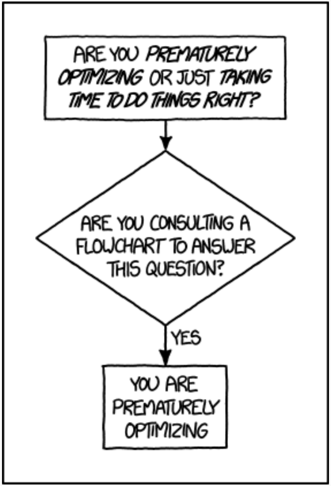
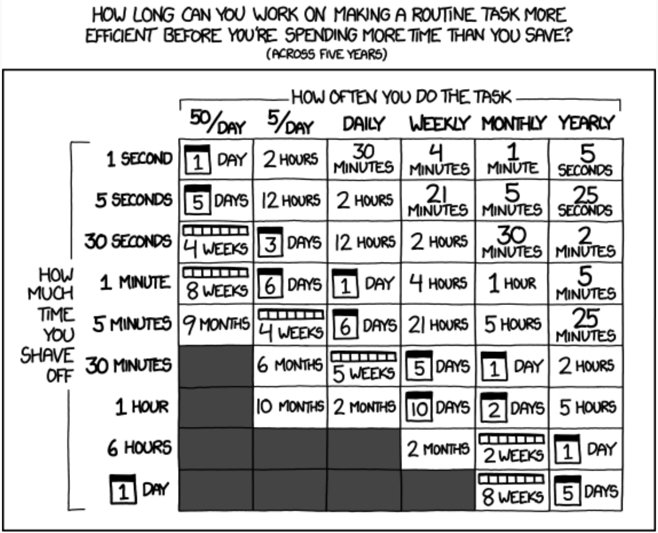
Refresher on algorithms

Louis Jachiet

Optimizing programs

Should you optimize a program?




HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)
HOW OFTEN YOU DO THE TASK
50/DAY
5/DAY
DAILY
WEEKLY
MONTHLY
YEARLY
HOW MUCH TIME YOU SHAVE OFF
1 SECOND
1 DAY
2 HOURS
30 MINUTES
4 MINUTES
1 MINUTE
5 SECONDS
5 SECONDS
5 DAYS
12 HOURS
2 HOURS
21 MINUTES
5 MINUTES
25 SECONDS
30 SECONDS
4 WEEKS
3 DAYS
12 HOURS
2 HOURS
30 MINUTES
2 MINUTES
1 MINUTE
8 WEEKS
6 DAYS
1 DAY
4 HOURS
1 HOUR
5 MINUTES
5 MINUTES
9 MONTHS
4 WEEKS
6 DAYS
21 HOURS
5 HOURS
25 MINUTES
30 MINUTES
6 MONTHS
5 WEEKS
5 DAYS
1 DAY
2 HOURS
1 HOUR
10 MONTHS
2 MONTHS
10 DAYS
2 DAYS
5 HOURS
6 HOURS
2 MONTHS
2 WEEKS
1 DAY
1 DAY
8 WEEKS
5 DAYS

Should you optimize a program?

If you have a solution that works and is fast enough. . .

Should you optimize a program?

If you have a solution that works and is fast enough... NO.

Also you should only optimize the time-consuming parts of your program

Should you optimize a program?

If you have a solution that works and is fast enough... NO.

Also you should only optimize the time-consuming parts of your program which means you should measure what takes time.

Why optimize?

If you have a program that is too slow you have three ways of optimizing it:

- Do some little tweaking

You can go for a $2\times$ *speed-up*

Why optimize?

If you have a program that is too slow you have three ways of optimizing it:

- Do some little tweaking

  You can go for a $2\times$ *speed-up*

- Change the language or use a well-optimized library

  You can have a $100\times$ *speed-up, in a very favorable case*

Why optimize?

If you have a program that is too slow you have three ways of optimizing it:

- Do some little tweaking

  You can go for a $2\times$ *speed-up*

- Change the language or use a well-optimized library

  You can have a $100\times$ *speed-up, in a very favorable case*

- Use multiple computer

  $n\times$ *speed-up with* n *computers*

Why optimize?

If you have a program that is too slow you have three ways of optimizing it:

- Do some little tweaking

  You can go for a $2\times$ *speed-up*

- Change the language or use a well-optimized library

  You can have a $100\times$ *speed-up, in a very favorable case*

- Use multiple computer

  $n\times$ *speed-up with* n *computers*

- Change the algorithm

  No limit on speed-up!

Numbers Everyone Should Know

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3 000 ns
Send 2K bytes over 1 Gbps network	20 000 ns
Read 1 MB sequentially from memory	250 000 ns
Round trip within same datacenter	500 000 ns
Disk seek (hard drive)	10 000 000 ns
Read 1 MB sequentially from disk (hard drive)	20 000 000 ns
Send packet CA $\rightarrow$ Netherlands $\rightarrow$ CA	150 000 000 ns

Defining algorithmic complexity

Defining algorithmic complexity

Turing machines

Church Turing thesis

Everything that can be computed, can be computed with a Turing Machine.

Strong Church Turing thesis

Everything that can be computed efficiently, can be efficiently computed with a deterministic Turing Machine.

Turing machines

In practice

Turing machines are **great** at modeling **large** complexity classes P, EXPTIME, L, etc. but **bad** for **fined-grained** complexity.

Example

Testing whether a string contains n times the letter a followed by n times the letter b cannot be recognized by a deterministic Turing Machine in linear time.

How to define computational complexity?

In practice

We use a ill-defined, vague but useful notion of RAM-model:

- the memory is divided in register of limited size (64 in actual computers)
- we have a memory indexed by addresses (this allows for arrays and pointers)
- we can do basic arithmetic operation $(+, -, \times, /, \%$, etc.)
- all basic operation takes $O(1)$

Defining algorithmic complexity

Notations

Bachmann-Landau notation

The parameter n

Usually the **length** of the problem. On TM this is the number of **bits**, on RAM machines this is usually the number of machine **words**.

- **Small** o**:** $g(n) = o(f(n))$ means $g(n)/f(n)$ tends to 0.
- **Big** $\mathcal{O}$**:** $g(n) = \mathcal{O}(f(n))$ means $g(n)/f(n)$ is bounded.
- **Big** Ω**:** $g(n) = \Omega(f(n))$ means $f(n)/g(n)$ is bounded, i.e. $f(n) = \mathcal{O}(g(n))$
- **Big** Θ**:** $g(n) = \theta(n)$ means $\exists c$ s.t. $c^{-1} < f(n)/g(n) < c$.

Bachmann-Landau notation

The parameter n

Usually the **length** of the problem. On TM this is the number of **bits**, on RAM machines this is usually the number of machine **words**.

- **Small** o**:** $g(n) = o(f(n))$ means $g(n)/f(n)$ tends to 0.
- **Big** $\mathcal{O}$**:** $g(n) = \mathcal{O}(f(n))$ means $g(n)/f(n)$ is bounded.
- **Big** Ω**:** $g(n) = \Omega(f(n))$ means $f(n)/g(n)$ is bounded, i.e. $f(n) = \mathcal{O}(g(n))$
- **Big** Θ**:** $g(n) = \theta(n)$ means $\exists c$ s.t. $c^{-1} < f(n)/g(n) < c$.
- **Big** $\tilde{\Omega}$**:** $g(n) = \tilde{\Omega}(f(n))$ means $\exists c$ s.t. $f(n)/g(n) < c$ for infinitely many n.
- **Big** $\tilde{\Theta}$**:** $g(n) = \tilde{\Theta}(f(n))$ means $\exists c$ s.t. $c^{-1} < f(n)/g(n) < c$ for infinitely many n.

Importance of the constant

The $\mathcal{O}$ notation “hides” the actual performance in the constant:

- it is very useful to develop algorithms
- it is generally gives the fastest algorithms
- but there are cases where the constant is huge

However, keep in mind that all computers have a finite memory. . .

Generic algorithmic approach

Know the basics of algorithms

- Divide and conquer

Know the basics of algorithms

- Divide and conquer
- Sliding windows

Know the basics of algorithms

- Divide and conquer
- Sliding windows
- Dynamic algorithms

Know the basics of algorithms

- Divide and conquer
- Sliding windows
- Dynamic algorithms
- Math-trick

Know the basics of algorithms

- Divide and conquer
- Sliding windows
- Dynamic algorithms
- Math-trick
- Reduction of complexity

Know the basics of algorithms

- Divide and conquer
- Sliding windows
- Dynamic algorithms
- Math-trick
- Reduction of complexity
- Data structure

Use the right datastructure

- Array

Use the right datastructure

- Array
- Linked Lists

Use the right datastructure

- Array
- Linked Lists
- Hash table

Use the right datastructure

- Array
- Linked Lists
- Hash table
- Balanced binary tree

Use the right datastructure

- Array
- Linked Lists
- Hash table
- Balanced binary tree
- Queues

Exercises

- Sort a list of integers
- Given two strings, are they anagrams?
- Given a list of pair (people,phone) and a list (people,mail), what are the people that have both a phone and a mail?
- We define $F_{n+2} = F_n + F_{n+1}$ with $F_0 = F_1 = 0$, how to compute F_n?
- Given a list l, compute $max_{i,j}(sum(l[i:j]))$