Programmeren (Ectrie)

Lecture 2: Computing

Tommi Tervonen

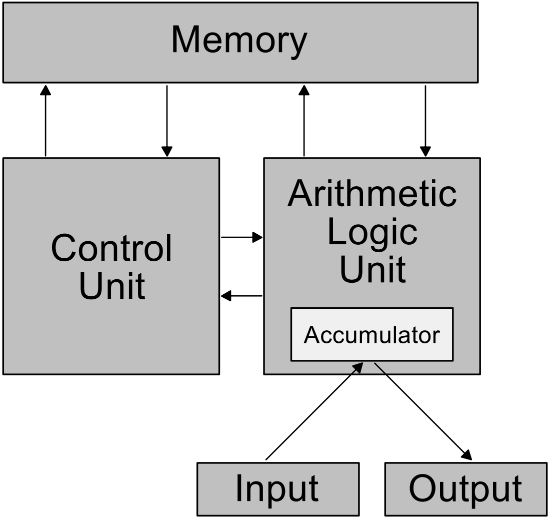
Econometric Institute, Erasmus University Rotterdam

What's the difference?

Stored-program computers

- Enable to write, compile, and run code on the same machine
- Implement *von Neumann* architecture

von Neumann architecture



Numerical representation

- Computers have instruction sets (e.g. MOV, MUL, ADD)
- Each instruction has a binary *opcode*
- Numbers (integers and reals) are also just sequences of bits
- Standard computers operate with a certain number of bits (32/64)
- We give *semantics* to the sequences of bits to represent integers, reals, characters, opcodes, ...

Floating point numbers

- Integers are within a certain range (e.g. standard 32-bit: $[-2^{31}, 2^{31} - 1]$) that contains all values
- Reals are represented as *floating point numbers* with a fixed base b, signed fraction f, and exponent e as:

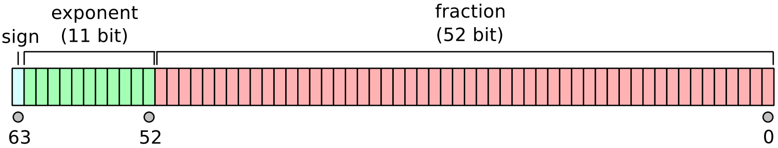
$$(e, f) = f \times b^e$$

- E.g. floating decimal ($b = 10$) with 8 digits can represent Plank's constant (6.6261×10^{-27}) as

$$(-26, +.66261000)$$

IEEE 754

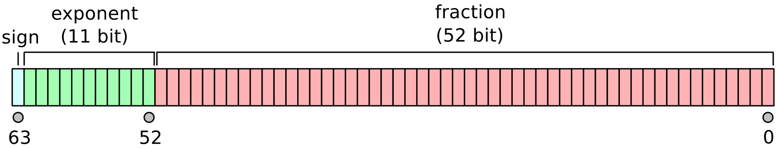
- We still need to choose b and bit sizes for e and f
- Most processors support IEEE 754 double precision (64 bit) floating point standard with $b = 2$:



$$value = (-1)^{sign}(1 + \sum_{i=1}^{52} b_{-i}2^{-i}) \times 2^{(e-1023)}$$

- Java's `double` and Matlab's numbers are 64 bit floats

IEEE 754



Implications:

- The decimal point is floating, and precision of the fraction is $53 \log_{10} 2 \approx 15.955$
- 23000000000000000000 ok
- 23000000000000000001 not

Problems with floating point numbers (1)

Operations on floating point numbers performed on computers are neither associative nor distributive, that is,

$$a + (b + c) \neq (a + b) + c, \text{ for many } a, b, c$$
$$a * (b + c) \neq (a * b) + (a * c), \text{ for many } a, b, c$$

when a, b, and c are floating point numbers. For example, consider

$$a = 0.42$$
$$b = -0.5$$
$$c = 0.08$$

now, with IEEE 754 double-precision binary floats, we get

$$(a + b) + c = -1.3878 \times 10^{-17}$$
$$a + (b + c) = 0$$

Problems with floating point numbers (1)

- So not all numbers can be represented exactly with IEEE 754 floating point numbers
- The interval between numbers that can be represented depends on the magnitude of the number
- Floating point numbers can be thought of representing an interval around the given value (e.g. 0.42 is actually $[0.42 - \epsilon_1, 0.42 + \epsilon_2]$)

Problems with floating point numbers (2)

The limited amount of bits used to store the numbers can cause under- and overflows:

$$1.2345678 + 1.7654321 = 3$$

due to the inherent imprecision of the floating point representation.

$\Rightarrow$ never compare results against an exact value, rather check whether they are within a threshold ϵ:

$$1.2345678 + 1.7654321 - 3 \leq \epsilon$$

Problems with floating point numbers (3)

Accuracy of floating point numbers highly depends on the operations performed:

- multiplication is less precise than addition
- repeated application of addition/substraction can result in arbitrarily large errors if incorrect rounding scheme is used (though in practice it isn't)

If you need very high precision floats, don't use Matlab

Computational complexity

- The only two resources for algorithms are computation time and memory
- Computational complexity refers to their use - how complex is the algorithm given an input of size n
- Complexity theory forms the basis for all computational sciences
- Complexity can be analyzed by counting the amount of resources used
- Example: adding together two integers 12345 and 53766

Insertion sort

2	3	1	5	4

- Sort in the way card game players sort their hands

Insertion sort

2	3	1	5	4

- Sort in the way card game players sort their hands

```
function [a] = insertionSort(a)
  for j=2:length(a)
    key = a(j);
    i = j-1;
    while i > 0 && a(i) > key
      a(i+1) = a(i);
      i = i-1;
    end
  a(i+1) = key
  end
end
```

Insertion sort: example

With romanian folk dance

Insertion sort: analysis

Assumptions:

- We are computing with a single-processor random access machine
- No parallel processing
- Instructions are processsed sequentially
- The machine has unlimited memory

Insertion sort: analysis

- **Memory**: a constant amount of additional memory (i.e. for the temporary variables) is used - insertion sort does the sorting *in place*

- **Running time**: count the amount of primitive operations performed
  - Primitive operations = arithmetic operations, comparisons, assignments, etc

  - Exact number of CPU cycles / operation depends on compiler and hardware

  - Analyze on more abstract level by counting the amount *computation steps*

```
1  function [a] = insertionSort(a)
2     for j=2:length(a)
3        key = a(j);
4        i = j-1;
5        while i > 0 && a(i) > key
6           a(i+1) = a(i);
7           i = i-1;
8        end
9     a(i+1) = key
10    end
11 end
```

- Amount of times each line is executed
- c_i: the cost of executing line i
- t_j the amount of times the while loop test on line 5 is executed

```
1   function [a] = insertionSort(a)
2     for j=2:length(a)
3       key = a(j);
4       i = j-1;
5       while i > 0 && a(i) > key
6         a(i+1) = a(i);
7         i = i-1;
8       end
9     a(i+1) = key
10    end
11  end
```

Line	2	3	4	5	6	7	9
Cost	c_2	c_3	c_4	c_5	c_6	c_7	c_9
Times	n	$n-1$	$n-1$	$\sum_{j=2}^{n} t_j$	$\sum_{j=2}^{n}(t_j-1)$	$\sum_{j=2}^{n}(t_j-1)$	$n-1$

Line	2	3	4	5	6	7	9
Cost	c_2	c_3	c_4	c_5	c_6	c_7	c_9
Times	n	$n-1$	$n-1$	$\sum_{j=2}^{n} t_j$	$\sum_{j=2}^{n}(t_j-1)$	$\sum_{j=2}^{n}(t_j-1)$	$n-1$

$$
\begin{aligned}
T(n) = & c_2 n + c_3(n-1) + c_4(n-1) \\
& + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j-1) \\
& + c_7 \sum_{j=2}^{n}(t_j-1) + c_9(n-1)
\end{aligned}
$$

```
function [a] = insertionSort(a)
  for j=2:length(a)
    key = a(j);
    i = j-1;
    while i > 0 && a(i) > key
      a(i+1) = a(i);
      i = i-1;
    end
  a(i+1) = key
  end
end
```

- The running time depends on size of the input n *and* times the inner loop is executed t_j

- $a(i) \leq a(j)\ \forall i < j,\ i, j \in \{1, \ldots, n\} \Rightarrow t_j = 1\ \forall j \in \{1, \ldots, n\}$

Best-case running time

$$
\begin{aligned}
T(n) = & c_2 n + c_3(n-1) + c_4(n-1) \\
& + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
& + c_7 \sum_{j=2}^{n} (t_j - 1) + c_9(n-1)
\end{aligned}
$$

if $a(i) \leq a(j)\ \forall i < j,\ i, j \in \{1, \dots, n\} \Rightarrow t_j = 1\ \forall j \in \{1, \dots, n\}$

Best-case running time

$$
\begin{aligned}
T(n) = {} & c_2 n + c_3(n-1) + c_4(n-1) \\
& + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\
& + c_7 \sum_{j=2}^{n} (t_j - 1) + c_9(n-1)
\end{aligned}
$$

if $a(i) \leq a(j)\ \forall i < j,\ \ i, j \in \{1, \ldots, n\} \Rightarrow t_j = 1\ \forall j \in \{1, \ldots, n\}$

$$
\begin{aligned}
\Rightarrow T(n) &= c_2 n + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) \\
&= (c_2 + c_3 + c_4 + c_5 + c_9)n - (c_2 + c_4 + c_5 + c_9)
\end{aligned}
$$

Best-case running time

$$\begin{aligned} T(n) = & c_2 n + c_3(n-1) + c_4(n-1) \\ & + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ & + c_7 \sum_{j=2}^{n} (t_j - 1) + c_9(n-1) \end{aligned}$$

if $a(i) \leq a(j) \; \forall i < j, \; i, j \in \{1, \ldots, n\} \Rightarrow t_j = 1 \; \forall j \in \{1, \ldots, n\}$

$$\begin{aligned} \Rightarrow T(n) &= c_2 n + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) \\ &= (c_2 + c_3 + c_4 + c_5 + c_9)n - (c_2 + c_4 + c_5 + c_9) \end{aligned}$$

replace $c_2 + c_3 + c_4 + c_5 + c_9 = a$ and $c_2 + c_4 + c_5 + c_9 = b$

Best-case running time

$$\begin{aligned} T(n) = & c_2 n + c_3(n-1) + c_4(n-1) \\ & + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) \\ & + c_7 \sum_{j=2}^{n} (t_j - 1) + c_9(n-1) \end{aligned}$$

if $a(i) \leq a(j) \; \forall i < j, \; i, j \in \{1, \dots, n\} \Rightarrow t_j = 1 \; \forall j \in \{1, \dots, n\}$

$$\begin{aligned} \Rightarrow T(n) &= c_2 n + c_3(n-1) + c_4(n-1) + c_5(n-1) + c_9(n-1) \\ &= (c_2 + c_3 + c_4 + c_5 + c_9)n - (c_2 + c_4 + c_5 + c_9) \end{aligned}$$

replace $c_2 + c_3 + c_4 + c_5 + c_9 = a$ and $c_2 + c_4 + c_5 + c_9 = b$

$$\Rightarrow T(n) = an + b$$

```
function [a] = insertionSort(a)
  for j=2:length(a)
    key = a(j);
    i = j-1;
    while i > 0 && a(i) > key
      a(i+1) = a(i);
      i = i-1;
    end
  a(i+1) = key
  end
end
```

If $a(i) > a(j)\ \forall i < j,\ i,j \in \{1, \ldots, n\}$

$\Rightarrow$ in every iteration of the while loop the current element a(i) must be compared with each of the elements in the already sorted subarray $a(1), \ldots, a(i-1)$, so $t_j = j\ \forall j \in \{2, \ldots, n\}$

$$
\begin{aligned}
T(n) =& c_2 n + c_3(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} j \\
&+ c_6 \sum_{j=2}^{n}(j-1) + c_7 \sum_{j=2}^{n}(j-1) + c_0(n-1)
\end{aligned}
$$

note that

$$
\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}
$$

$$
\begin{aligned}
T(n) = & c_2 n + c_3(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} j \\
& + c_6 \sum_{j=2}^{n} (j-1) + c_7 \sum_{j=2}^{n} (j-1) + c_0(n-1)
\end{aligned}
$$

note that

$$
\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \text{ and } \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}
$$

$$
\begin{aligned}
\Rightarrow T(n) = & c_2 n + c_3(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
& + (c_6 + c_7)\left(\frac{n(n-1)}{2}\right) + c_9(n-1) \\
= & \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 \\
& + \left(c_2 + c_3 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9\right) n \\
& - (c_3 + c_4 + c_5 + c_9)
\end{aligned}
$$

Worst-case running time

$$
\begin{aligned}
T(n) =& \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 \\
& + \left(c_2 + c_3 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9\right) n \\
& - \left(c_3 + c_4 + c_5 + c_9\right)
\end{aligned}
$$

replace sets of c_i's with constants a, b, and c

Worst-case running time

$$
\begin{aligned}
T(n) =& \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 \\
&+ \left(c_2 + c_3 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9\right)n \\
&- (c_3 + c_4 + c_5 + c_9)
\end{aligned}
$$

replace sets of c_i's with constants a, b, and c

$$\Rightarrow T(n) = an^2 + bn + c$$

Analysis: conclusions

Insertion sort

- Sorts in place - requires constant amount of memory not dependent on the input size
- Has linear best-case complexity
- Has quadratic worst-case complexity

Worst-case analyses

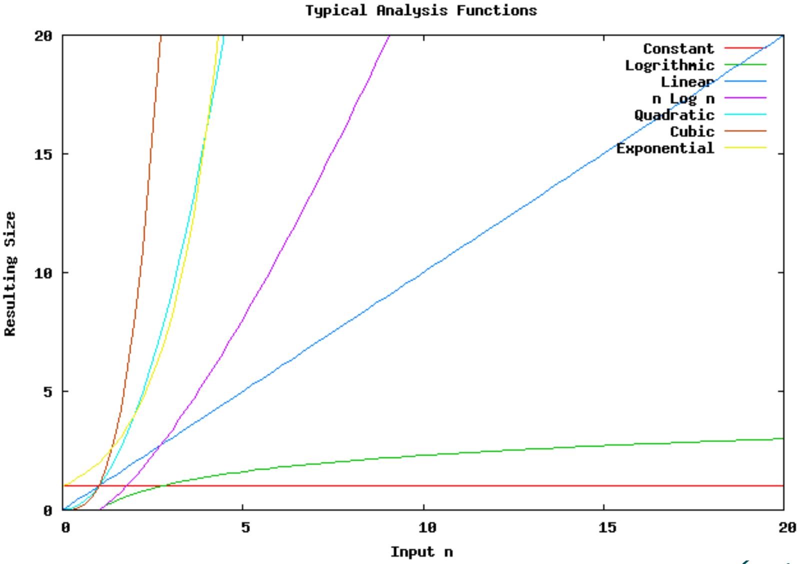
Usually we are interested *only* in the worst-case complexity, as

- It gives us an upper-bound on how bad the algorithm can perform
- Worst-case occurs fairly often with some algorithms
- Worst-case can occur with extremely high probability when input is from real-life processes (e.g. sorting customers)
- Algorithms are executed often, so worst case happens almost surely sometime

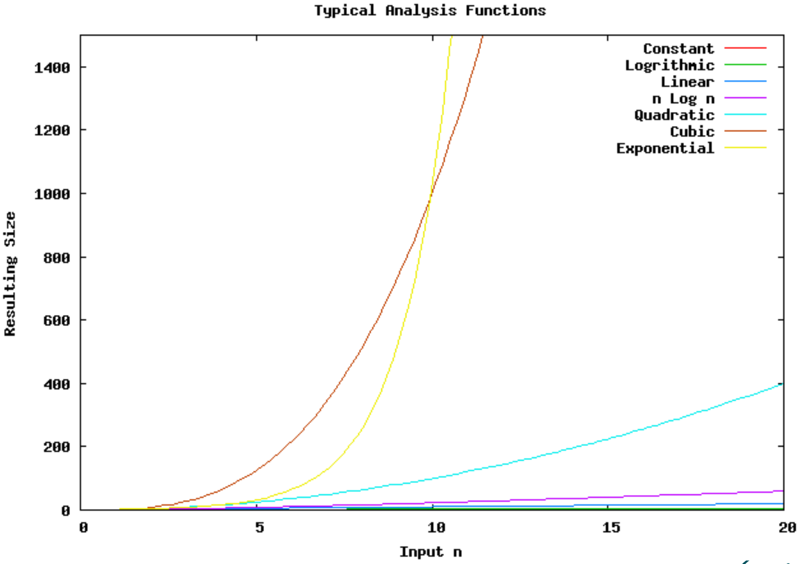
Running times with a computer processing 10^9 ops/s

f(n)	10	100	1000	10^4	10^5	10^6
n	10^{-8}s	10^{-7}s	10^{-6}s	10^{-5}s	10^{-4}s	10^{-3}s
$n \log n$	10^{-8}s	2.4×10^{-8}s	2.0×10^{-6}s	3.5×10^{-4}s	0.1s	56s
n^2	10^{-7}s	10^{-5}s	10^{-3}s	0.1s	10s	17min
n^3	10^{-6}s	10^{-3}s	1s	17min	12d	32y
2^n	10^{-6}s	4.0×10^{13}y	3.3×10^{284}y			
$n!$	3.6×10^{-3}s	3.0×10^{141}y				

Growth of functions

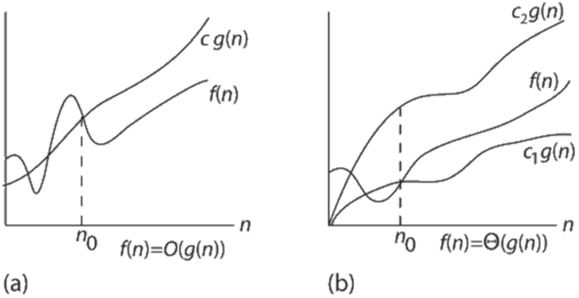


Growth of functions



Asymptotic complexity

- Exact analysis as we did before (with c_i's) is not meaningful - only *asymptotic* complexity matters
- Given an input size $n > n_0$, where n_0 is some constant value, how fast does the running time grow?
- The asymptotic behaviour of a function depends only on the highest order term, and not at all of the constants (the c's)

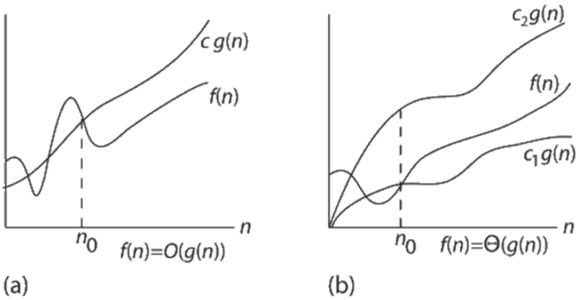


big-O notation

- For asymptotic worst-case complexity, we use the big-O notation. Given a function $g(n)$, the set of functions

$$O(g(n)) = \{f(n) : \exists c > 0, n_0 > 0 : \ 0 \leq f(n) \leq cg(n) \quad \forall n \geq n_0\}$$

are asymptotically O-equivalent.



O(g(n)) is the asymptotical upper bound, that is not necessary tight

big-O

- For example, our previous quadratic complexity
  $an^2 + bn + c \in O(n^2)$, also
  $3n^2 \in O(n^2)$ and
  m$n^2 + nlogn \in O(n^2)$

- Common complexity classes
  - O(1)
  - O(n)
  - O(n log n)
  - O(n^2)
  - O(n^3)
  - O(2^n)

About hardness

- Any problem with a known algorithm for solving it in polynomial time (O(g(n)) where g(n) is a polynomial) is called *tractable*
- Unfortunately many practical problems are intractable
- The most significant problem in mathematics: P=NP?