







## What do we Want from a Program?

We want that a program...

- Does what it is supposed to do (**correctness**, computability, decidability),
- Does it fast (**efficiency**, tractability),
- Works on challenging inputs (scalability),
- Consumes few resources (memory, storage, communication),
- Be easy to use (user-friendliness, Human-Computer Interface),
- Be written fast and/or at low cost (development),
- Be easy to maintain and modify if needed (maintenance),
- And many other things (throughput, high availability, energy consumption, etc.).









Where could the Bug Be?

## Where could the Bug Be?

Errors in hardware, operating systems, compilers, interpreters and commercial application software occur. They are well publicized when discovered and fixed in the following versions (in particular when they can be security threats: see “Top 25 Most Dangerous Software Errors” <http://cwe.mitre.org/top25>).

Hardware errors are the rarest but occur: The Pentium FDIV bug discovered by Professor Thomas R. Nicely in October 1994 - “An error in a lookup table created the infamous bug in Intel’s latest processor”, by Tom R. Halfhill, BYTE (March 1995).

Errors can be due to interactions between components, for instance in operating systems, the interaction between drivers and applications.



## Typing Errors

Discrepancy between the types of variables and the expected types of functions and operations.

```
1 var i = "hello";  
2 i = Math.abs(i);  
3 document.write("The number is: " + i + "<BR>")  
   ;\
```

JavaScript is dynamically and weakly typed. Flexible but does not help preventing typing errors.

## Semantic Errors

A program with no error may not do what was intended.

```
1 {for (i = 0; i = i+1; i < 5;)
2   {document.write("The number is: " + i + "<
   BR>");}}
```

“ $i = i+1$ ” returns the value assigned to  $i$  also as the value of a condition. This value is interpreted as “true” if it is positive and as “false” if it is 0. So “ $i = i+1$ ” is a syntactical legal condition. Furthermore, there are no constraints how the third condition updates  $i$ , anything is permitted in Java Script. Different programming languages are more or less permissive.



## Infinite Loops

Program execution may not terminate! Sometimes by design.

```
1 {for (i = 0; i >= 0; i = i+1)
2   {document.write("Wait for me!");}}
```

```
1 {var something;
2 while (true)
3   {something = wait_for();
4   take_action(something);}}
```

Note: See JavaScript event handling.



## Example: Fibonacci Numbers

The first two Fibonacci numbers are 0 and 1. Each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

```
1 function fibonacci(z)
2   {var r;
3     r = fibonacci(z-1) * fibonacci(z-2);
4     return r;}
5 for (i = 0; i < 10; i++)
6 {document.write(fibonacci(i) + "<BR>");}
```



Most likely, **the bug is in your code...**

## How to Deal with the Bug?

- **Prevent the bug:** follow disciplined, rigorous **software engineering** (design and development) approach (software analysis, software verification, correctness proofs);
- **Track the potential bug:** test the software;
- **Find the bug:** instrument the code (put print statements to **trace the execution**; you may have used **defensive programming**, **design by contract** or other software engineering approaches; use **debugging tools**, **analysis tools** and other software engineering tools);
- **Fix the bug:** remove the bugs;





## Example of Specification

Euclid's algorithm computes the greatest common divisor of two natural numbers.

- Input: two natural numbers  $x$  and  $y$ ;
- Output a natural number  $z$  such that:
  - $z$  divides  $x$  and  $z$  divides  $y$  (i.e.  $z$  is a common divisor of  $x$  and  $y$ );
  - Any other natural number  $u$  greater than  $z$  does not divide  $x$  or  $y$  ( $z$  is the greatest common divisor of  $x$  and  $y$ ).

## Correctness and Termination

- **Total correctness**: an algorithm or a program is totally correct if it always terminates and returns a correct answer;
- **Termination**: the program terminates;
- **Partial correctness**: an algorithm or a program is partially correct if it returns the correct answer when it terminates.

There is no solution to the “**Halting Problem**”.







```
2 var v = x; var w = y;
```

## Correctness

- The three assertions are true by definition of  $z$  since we have copied  $x$  and  $y$  into  $v$  and  $w$ .

































## Example

Find an element  $e$  in a sorted array of size  $n$ . The following algorithm is called “Binary Search”.

```
1 function binSearch(A,e)
2 {var lower=0; var upper = A.length;
3 while (lower+2<=upper)
4     {var middle = Math.floor((lower+upper)/2)
5       ;
6       if (A[middle]<=target)
7           {lower=middle;}
8       else {upper=middle;}}
9 if (list[lower]==e)
   {return(true);} else {return(false);}}
```

## Logarithmic Time

At each iteration, the size of the part of the array searched is halved. The algorithm takes logarithmic time:  $O(\log(n))$ .

## Merge Sort

- Split the list in two halves;
- Sort the two halves by recursively applying the algorithm;
- Merge the two lists using at most one comparison per element.

## Recurrence Relation

- The number of comparisons is  $C(n)$  defined by the following recurrence relation (equation).

$$\begin{aligned} \text{If } n \leq 2 \quad \text{then} \quad C(n) &= 1 \\ \text{else} \quad C(n) &= n + 2 \times C\left(\frac{n}{2}\right) \end{aligned}$$

- How to solve a recurrence equation?

## Recurrence Equation

We consider the case where  $n$  is a power of 2 (i.e.  $n = 2^m$ ).

$$\begin{aligned}
 C(n) &= C(2^m) \\
 &= 2^m + 2 \times C(2^{m-1}) \\
 &= 2^m + 2 \times (2^{m-1} + 2 \times C(2^{m-2})) \\
 &= 2^m + 2^m + 2^2 \times C(2^{m-2}) \\
 &= k \times 2^m + 2^k \times C(2^{m-k}) \\
 &= (m-1) \times 2^m + 2^{m-1} \times C(2^1) \\
 &\leq m \times 2^m = \log(n) \times n
 \end{aligned}$$

## Pseudo-linear

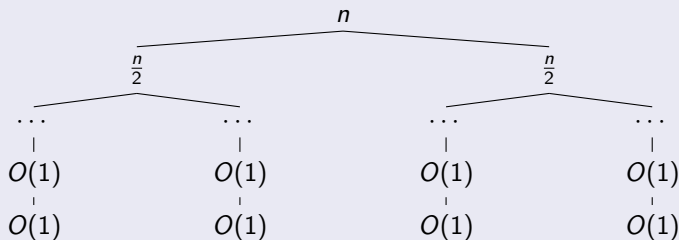
- Merge Sort is  $O(n \times \log(n))$ .

## Recurrence Relation

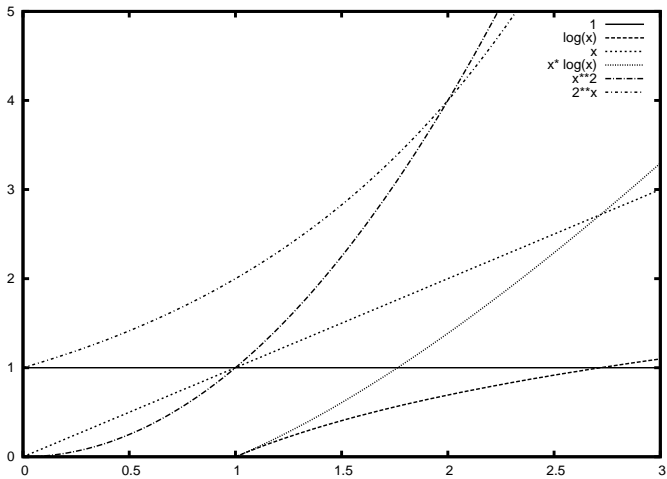
If  $n \leq 2$  then  $C(n) = 1$

else  $C(n) = n + 2 \times C\left(\frac{n}{2}\right)$

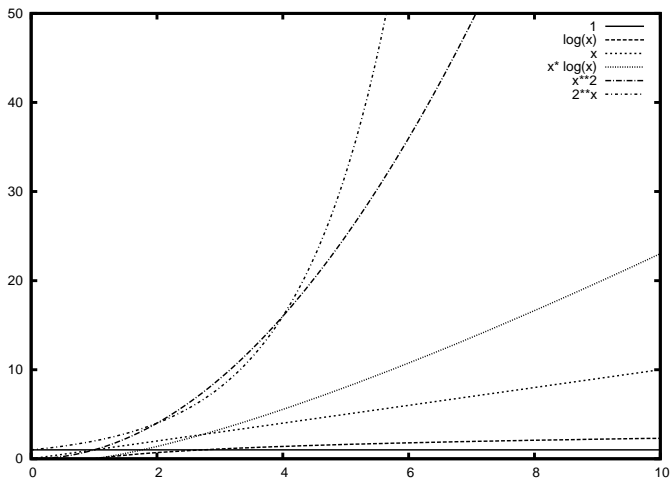
## Recursion Tree



## Theory



## Theory



## Order of Functions: Big O

Let  $g$  be a function on Real Numbers ( $g : \mathbb{R} \rightarrow \mathbb{R}$ ).

The set  $O(g)$  is the set of functions on Real Numbers defined as follows.

$$O(g) = \{f : \mathbb{R} \rightarrow \mathbb{R} \mid \exists c \in \mathbb{R} \forall x \in \mathbb{R} \\ ((c < x) \Rightarrow (f(x) \leq c \times g(x)))\}$$

- $f \in O(g)$  is often written  $f = O(g)$ ;
- $f \in O(g)$  reads “ $f$  is of the order of  $g$ ”;
- $f \in O(g)$  reads “ $f$  is big O of  $g$ ”;
- $f \in O(g) \Leftrightarrow O(f) \subseteq O(g)$ ;
- $(f \in O(g) \wedge g \in O(f)) \Leftrightarrow O(f) = O(g)$ .

## Examples

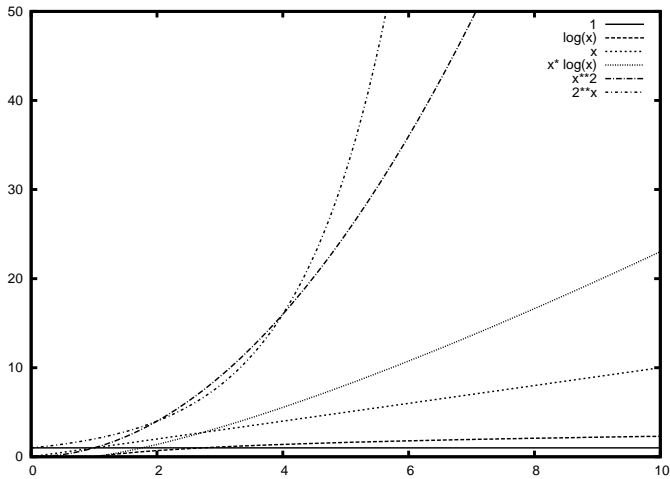
- $n^2 + n + 5 = O(n^2)$ ;
- $n^2 + n + 5 = O(n^3)$ ;
- $O(n^2) \subset O(n^3)$ .

## Important Sets

- Constant:  $O(1)$ ;
- Logarithmic:  $O(\log(n))$ ;
- Linear  $O(n)$ ;
- Pseudo-linear:  $O(n \times \log(n))$ ;
- Quadratic:  $O(n^2)$ ;
- Polynomial:  $O(n^c)$ , for any given  $c$ ;
- Exponential:  $O(2^{n^c})$ , for any given  $c$ ;

## Important Relationships

- $O(1) \subset O(\log(n))$ ;
- $O(\log(n)) \subset O(n)$ ;
- $O(n) \subset O(n \times \log(n))$ ;
- $O(n \times \log(n)) \subset O(n^2)$ ;
- $O(n^2) \subset O(n^3)$ ;
- $O(n^3) \subset O(2^n)$ .



## Sorting

Sorting: Given list  $(a_1, \dots, a_n)$  ( $n$  elements) output a list  $(a'_1, \dots, a'_n)$  that is a permutation such that  $a'_i \leq a'_j$  whenever  $i \leq j$ .

## Many Algorithms!

- Bubble sort
- Heap sort
- Insertion sort
- Quick sort
- Merge sort
- Selection sort
- etc.

## What can we Expect?

- A sorting algorithm can compare and copy and move elements. No other information is available.
- The running time of a sorting algorithm depends on the number of comparisons.

## Sorting Two Elements

There are two possible cases. One comparison allows sorting two elements.

- If  $e_1 < e_2$  then  $(e_1, e_2)$  is sorted.
- If  $e_1 > e_2$  then  $(e_2, e_1)$  is sorted.

## Sorting Three Elements

There are six possible cases. Three comparisons allow sorting three elements.

- If  $e_1 < e_2$  and  $e_2 < e_3$  and  $e_1 < e_3$  then  $(e_1, e_2, e_3)$  is sorted.
- If  $e_1 < e_2$  and  $e_2 < e_3$  and  $e_3 < e_1$  is impossible.
- If  $e_1 < e_2$  and  $e_3 < e_2$  and  $e_1 < e_3$  then  $(e_1, e_3, e_2)$  is sorted.
- If  $e_1 < e_2$  and  $e_3 < e_2$  and  $e_3 < e_1$  then  $(e_3, e_1, e_2)$  is sorted.
- If  $e_2 < e_1$  and  $e_2 < e_3$  and  $e_1 < e_3$  then  $(e_2, e_1, e_3)$  is sorted.
- If  $e_2 < e_1$  and  $e_2 < e_3$  and  $e_3 < e_1$  then  $(e_2, e_3, e_1)$  is sorted.
- If  $e_2 < e_1$  and  $e_3 < e_2$  and  $e_1 < e_3$  is impossible
- If  $e_2 < e_1$  and  $e_3 < e_2$  and  $e_3 < e_1$  then  $(e_3, e_2, e_1)$  is sorted.

## Sorting $n$ Elements

- There are  $n!$  possible cases (permutation of  $n$  elements).
- With  $m$  comparisons we can cater for  $2^m$  cases (some impossible).
- We need  $m$  comparisons such that  $2^m \geq n!$ .
- What do we know about  $\log(n!)$ ?
- Therefore  $m > \log\left(\frac{n!}{2}\right) = n \times \frac{(\log(n)-1)}{2}$ .
- We cannot expect better than an  $O(n \log(n))$  algorithm.

## Count Sort

The list contains all numbers between 0 and some known maximum, possibly with some duplicates.

- Count the number of occurrences of each number.
- Insert each number as many times as it occurs in the list in increasing order.

## Linear

- It takes  $n$  steps to count the occurrences (with an ancillary array of size  $n$ ).
- It takes  $n$  insertions to create the sorted list.
- Count Sort is  $O(n)$ .
- How is it possible? We were expecting  $O(n \log(n))$  at best!

# Count Sort

```
1 function count_sort(A)
2 { var m; var n=A.length; var i; var j;
3 var count = new Array(100);
4 for (i=0;i<100;i=i+1)
5     {count[i]=0;}
6 for (m=0;m<n;m=m+1)
7     {count[A[m]] = count[A[m]]+1;}
8 m=0;
9 for (i=0;i<100;i=i+1)
10    {for (j=0;j<count[i];j=j+1)
11        {A[m]=i; m=m+1;}}
12 return A;}
```

## Lower Bound of Tower of Hanoi

- Having  $T(n)$ , one can analyse the fastest algorithm for  $n + 1$ .
  - move  $n$  smaller rings;
  - move the larger ring
  - move  $n$  smaller rings;
- Let  $T(n)$  be the minimal number of moves needed to move  $n$  rings.
  - $T(n)$ ;
  - $T(1) = 1$ .
  - $T(n)$ ;
- $T(n + 1) = 2 \times T(n) + 1$ ;
- $2^n - 1$  is lower bound which matches known algorithm.

- Four instead of three pegs, 8 rings: 33 instead of 255 moves, improvement might be possible.
- Five instead of three pegs, 10 rings: 31 instead of 1023 moves.





## Summary

- Correctness
  - Sources and types of errors
  - Correctness and termination
- Efficiency
  - Efficiency of programs
  - Complexity of algorithms

## Next Lecture

- Growth of functions;
- Computing versus verifying solutions;
- NP-complete problems;
- Is P equal NP?
- Even worse problems.

## Attribution

The images and media files used in this presentation are either in the public domain or are licensed under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation, the Creative Commons Attribution-Share Alike 3.0 Unported license or the Creative Commons Attribution-Share Alike 2.5 Generic, 2.0 Generic and 1.0 Generic license.

The pocket watch photograph is by Isabelle Grosjean ZA (19 March, 1994).