Prerequisites

- Algorithm complexity: **Big Oh** notation
- Java/c++

PROGRAM EXECUTION

Today's general-purpose computers use a set of instructions called a program to process data

A computer executes the program to create output data from input data

Machine cycle

The CPU uses repeating machine cycles to execute instructions in the program, one by one, from beginning to end. A simplified cycle can consist of three phases: fetch, decode and execute

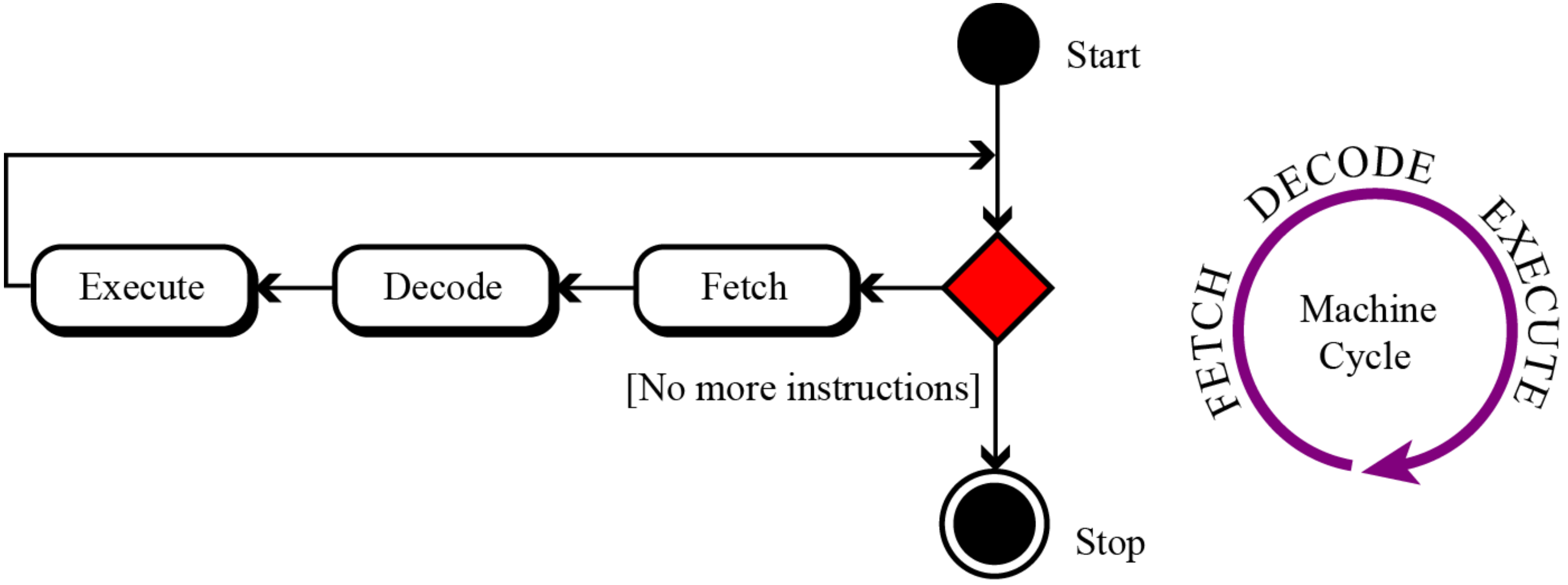

Figure: The steps of a cycle

Algorithm Complexity

Depends on # instructions (steps or cycles) –
depends on # inputs

In general, we are not so much interested in the time and space complexity for small inputs

For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with $n = 10$,

but it is significant for $n = 2^{30}$

Algorithm Complexity

For example, let us assume two algorithms $\mathfrak{A}$ and $\mathfrak{B}$ that solve the same class of problems.

The time complexity of $\mathfrak{A}$ is **$5{,}000n$**, the one for $\mathfrak{B}$ is **1.1^n** for an input with n elements.

<u>For n = 10</u>, $\mathfrak{A}$ requires 50,000 steps, but $\mathfrak{B}$ only 3, so $\mathfrak{B}$ <u>seems to be superior to</u> $\mathfrak{A}$.

<u>For n = 1000</u>, however, $\mathfrak{A}$ requires 5,000,000 steps, while $\mathfrak{B}$ requires $2.5 \cdot 10^{41}$ steps.

Algorithm Complexity

- Comparison: time complexity of algorithms 𝔄 and 𝔅

Input Size	Algorithm 𝔄	Algorithm 𝔅
n	5,000n	1.1^n
10	50,000	3
100	500,000	13,781
1,000	5,000,000	2.5×10^{41}
1,000,000	5×10^9	4.8×10^{41392}

Algorithm Complexity

This means that algorithm $\mathbb{B}$ cannot be used for large inputs, while running algorithm $\mathbb{A}$ is still feasible.

So what is important is the **growth** of the complexity functions.

The growth of time and space complexity with increasing input size n is a suitable measure for the comparison of algorithms.

The Growth of Functions

The growth of functions is usually described using the **big-O notation**

Definition: Let f and g be functions from the integers or the real numbers to the real numbers.
We say that <u>f(n) is O(g(n))</u> if
there are +ve constants C and n_0 such that

$$|f(n)| \leq C|g(n)|$$

whenever $n > n_0$

The Growth of Functions

When we analyze the growth of **complexity functions**, f(n) and g(n) are always positive

Therefore, we can simplify the big-O requirement to

$$0 \leq f(n) \leq C \cdot g(n) \quad \text{whenever } n > n_0$$

If we want to show that f(n) is O(g(n)), we only need to find **one** pair (C, n_0) (which is never unique)

The Growth of Functions

The idea behind the big-O notation is to establish an **upper boundary** for the growth of a function f(n) for large n

This boundary is specified by a function g(n) that is usually much **simpler** than f(n)

We accept the constant C in the requirement

$$0 \leq f(n) \leq C \cdot g(n) \quad \text{whenever } n > n_0,$$

because **C does not grow with n**

We are only interested in large n, so it is OK if

$$f(n) > C \cdot g(n) \quad \text{for } n \leq n_0$$

The Growth of Functions

Example:

Show that $f(n) = n^2 + 2n + 1$ is $O(n^2)$.

For $n > 1$ we have:

$$n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2$$
$$\Rightarrow \qquad \leq 4n^2$$

Therefore, for $C = 4$ and $n_0 = 1$:

$f(n) \leq Cn^2$ whenever $n > n_0$

$\Rightarrow$ $f(n)$ is $O(n^2)$

The Growth of Functions

Question: If $f(n)$ is $O(n^2)$, is it also $O(n^3)$?

Yes. n^3 grows faster than n^2, so n^3 also grows faster than $f(n)$.

$$f(n) \le C_1 n^2 \le C_2 n^3$$

Therefore, we always have to find the **smallest** simple function $g(n)$ for which $f(n)$ is $O(g(n))$.

The Growth of Functions

"Popular" functions $g(n)$ are
$n \log n$, 1, 2^n, n^2, $n!$, n, n^3, $\log n$

Listed from slowest to fastest growth:

1
$\log n$
n
$n \log n$
n^2
n^3
2^n
$n!$

Complexity Examples

What does the following algorithm compute?

procedure MAX_Diff(a_1, a_2, …, a_n: integers)
m := 0
for i := 1 to n-1
    for j := i + 1 to n
        if $|a_i - a_j| > m$ **then** m := $|a_i - a_j|$
{m is the maximum difference between any two numbers in the input sequence}

Comparisons: n-1 + n-2 + n-3 + … + 1

$$= (n-1)n/2$$

$$= 0.5n^2 - 0.5n$$

Time complexity is $O(n^2)$.

Complexity Examples

Another algorithm solving the same problem:

```
procedure MAX_Diff(a_1, a_2, ..., a_n: integers)
min := a1
max := a1
for i := 2 to n
        if a_i < min then min := a_i
        else if a_i > max then max := a_i
m := max - min
```

Comparisons: **$2(n - 1) = 2n-2$**

Time complexity is **$O(n)$**

Complexity of Algorithms

Is $\mathbf{O(n^2)}$ too much time?

Is the algorithm practical?

Practical Complexities

10^9 instructions/second

n	n	$n\log n$	n^2	n^3
1000	1mic	10mic	1milli	1sec
10000	10mic	130mic	100milli	17min
10^6	1milli	20milli	17min	32years

Impractical Complexities

10^9 instructions/second

n	n^4	n^{10}	2^n
1000	17min	3.2×10^{13} years	3.2×10^{283} years
10000	116 days	???	???
10^6	3×10^7 years	??????	??????

Prerequisites

- Algorithm complexity: **Big Oh** notation
- Java/c++

Procedural oriented Programming

- Top-down approach
- Procedure (Function) is building block

Ex: Calculator → scientific

→ non-scientific → float

→ int → add.

→ subs.

→ mult.

→ div.

Small Fun's

Ex: C - language

Procedural oriented Programming

- **Adv:**
  - Re-usability
  - Easy to debug
- **Disadv:**
  - Concentrate on what we want to do, not on who will use it
  - Data does not have a owner (sharing)
  - All functions are global
  - No data security
    - Ex: let there are three functions a(),b() and c(). Data d is used by a() and b() but how to restrict from c() [data is either global or local]
  - No data integrity [Stack:{add, delete},Queue:{add, delete}]
    - How to distinguish which fun associated with which data structure