Introduction to Algorithms: Lecture 1

Xue Chen
xuechen1989@ustc.edu.cn
2024 spring

Outline

Introduction

This course focuses on *algorithms* — a classical math concept.

Algorithm Euclid's algorithm for GCD

```
function EUCLID(a, b)
    if b=0 then
        return a
    else
        return Euclid(b, a mod b)
    end if
end function
```

Introduction

This course focuses on *algorithms* — a classical math concept.

Algorithm Euclid's algorithm for GCD

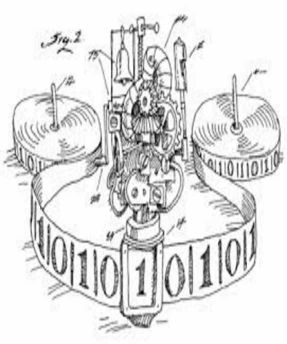
```
function EUCLID(a, b)
    if b=0 then
        return a
    else
        return Euclid(b, a mod b)
    end if
end function
```

More examples: Approximations of π, finding roots of quadratic/cubic polynomials, Newton method, ...

Introduction (II)

Modern concepts of algorithms:

1. Efficiency: **Running time**
2. Correctness: **It does what we want**

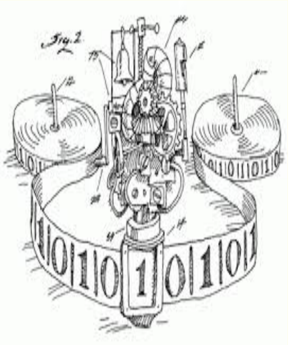


Turing Machine 1936

Introduction (II)

Modern concepts of algorithms:

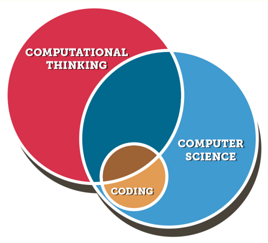
1. Efficiency: **Running time**
2. Correctness: **It does what we want**
3. Other computational resources: Space, randomness, communication, ...
4. Reliability: Easy to implement and maintain
5. Scalability: Parallel computing and distributed computing
6. Functionality
7. Robustness
8. ...



Turing Machine 1936

Why study algorithms?

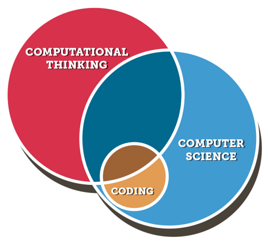
1. Write better codes — Knuth: Computer Programming is an art form
2. Solve problems



My answer: Algorithms are the core of computer science and give a computational thinking (undecidable, efficient algorithms, . . .).

Why study algorithms?

1. Write better codes — Knuth: Computer Programming is an art form
2. Solve problems

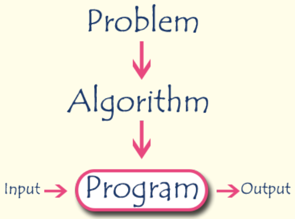


My answer: Algorithms are the core of computer science and give a computational thinking (undecidable, efficient algorithms, . . .).

Two goals

- Algorithm design & analysis — 2/3 load
- Implementation: Understand real programs' performance & solve practical problems — 1/3 load

Programs = Algorithms + Data Structures



Algorithmic Techniques

1. divide and conquer
2. dynamic program
3. greedy method
4. linear program
5. max flow algorithms
6. ...

Advanced Data Structures

1. heaps and priority queues
2. hash tables
3. binary search trees and red-black trees
4. disjoint-set operations
5. ...

Example: The Experts/Multiplicative Weights Alg.

Description

- n experts (models) and m events on day 1,...,day m.
- Each expert predicates event i on the night of day $i-1$ and know the actual result σ_i at day i.
- Task: Generate a prediction every night and minimize the number of mistakes — compared to the best expert!

Example: The Experts/Multiplicative Weights Alg.

Description

- n experts (models) and m events on day 1,. . .,day m.
- Each expert predicates event i on the night of day $i-1$ and know the actual result σ_i at day i.
- Task: Generate a prediction every night and minimize the number of mistakes — compared to the best expert!

1. Implemented in the top of DeekSeek (called Mixture of Experts)
2. Lots of applications: learning, solve LP, Nash equilibrium, . . .
3. Lots of interesting ideas: gradient descent (mirror descent), multi-bandit problems, . . .

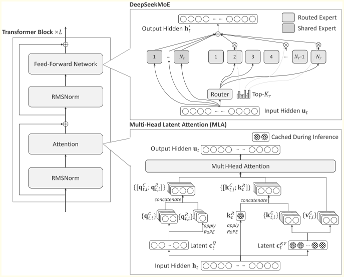
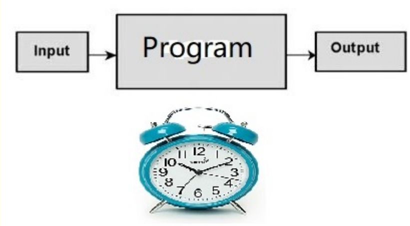


Figure 2 | Illustration of the basic architecture of DeepSeek-V3. Following DeepSeek-V2, we adopt MLA and DeepSeekMoE for efficient inference and economical training.

Outline

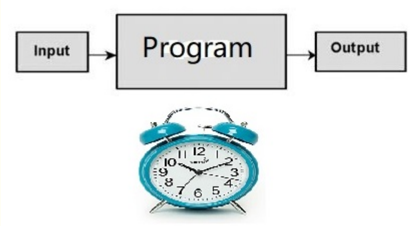
Running Time (I)



However, the actual running time of the program depends on lots of factors:

1. Input & data size

Running Time (I)



However, the actual running time of the program depends on lots of factors:

1. Input & data size
2. Programming languages
3. Hardware: Memory, cache, CPU& GPU (instruction set, # cores, . . .)

Many issues affect the time by fixed constant factors except input/data size

Running Time (II)

Consider *running time* $\approx$ number of steps/instructions

For simplicity

Our algorithms in each step can

1. $+,\ -,\ \times,\ /,\ \mod$ for all integers $< 2^{64}$
2. load, store, copy an integer
3. control operations: If-Else, subroutine call, ...

Be careful for large integers and real numbers!

Notations

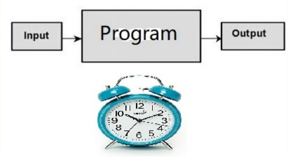
Asymptotic Analysis

This course will ignore those constant factors and focus on the relation (asymptotically) between running time and input size (or input length).

Notations

Asymptotic Analysis

This course will ignore those constant factors and focus on the relation (asymptotically) between running time and input size (or input length).



Notation:

1. Let $n :=$ parameter about the input (like string length or # vertices)
2. Let $T(n)$ be the maximum # steps (instructions) on inputs with parameter n

Example: Fibonacci number

$n :=$ parameter about the input
$T(n) :=$ maximum # steps (or instructions) on inputs of parameter n

---

Compute the Fibonacci number (I)

---

function $\textsc{Fib}(n)$
    if $n \leqslant 1$ **then**
        return 1
    end if
    return $\textsc{Fib}(n-2)$+$\textsc{Fib}(n-1)$
end function

Question

Assume $+$ operation is always in 1 step, what is $T(n)$ for $\textsc{Fib}(n)$?

Example (II)

$n :=$ parameter about the input
$T(n) :=$ maximum # steps (or instructions) on inputs of parameter n

---

Compute the Fibonacci number (II)

---

function FIBONACCI(n)
  $f[0] \leftarrow 1; f[1] \leftarrow 1$
  for $i = 2, \ldots, n$ **do**
    $f[i] \leftarrow f[i-1] + f[i-2]$
  end for
  return $f(n)$
end function

Example (II)

$n :=$ parameter about the input
$T(n) :=$ maximum # steps (or instructions) on inputs of parameter n

---

Compute the Fibonacci number (II)

---

function FIBONACCI(n)
  $f[0] \leftarrow 1; f[1] \leftarrow 1$
  for $i = 2, \ldots, n$ **do**
    $f[i] \leftarrow f[i-1] + f[i-2]$
  end for
  return $f(n)$
end function

Question

What is $T(n)$ for FIBONACCI(n)?

Asymptotic Analysis

The language to analyze running times, like $\int$, ∂ and d in calculus

— basically, ignore those annoying constants

Asymptotic Analysis

The language to analyze running times, like $\int$, ∂ and d in calculus
— basically, ignore those annoying constants

Compute the Fibonacci number

---

function FIBONACCI(n)
  $f[0] \leftarrow 1$; $f[1] \leftarrow 1$
  for $i = 2, \ldots, n$ **do**
    $f[i] \leftarrow f[i-1] + f[i-2]$
  end for
  return $f(n)$
end function

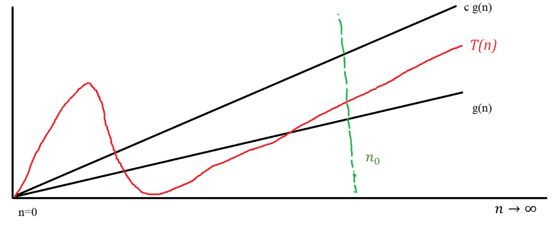
$T(n) = 2n + 1$ is in the same order of n asymptotically if we ignore the constant 2.

Asymptotic Notation

O-notation

$T(n) = O\big(g(n)\big)$ if there exist c and n_0 such that $T(n) \leqslant c \cdot g(n)$ for all $n > n_0$.

Call $T(n)$ is in the order of $g(n)$ or $T(n)$ is $O(g(n))$

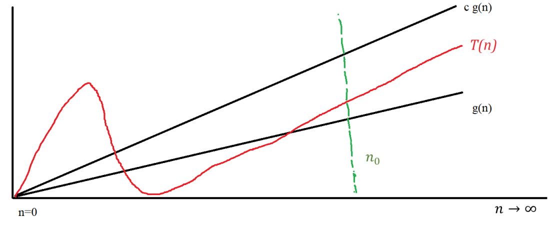


Asymptotic Notation

O-notation

$T(n) = O(g(n))$ if there exist c and n_0 such that $T(n) \leqslant c \cdot g(n)$ for all $n > n_0$.

Call $T(n)$ is in the order of $g(n)$ or $T(n)$ is $O(g(n))$



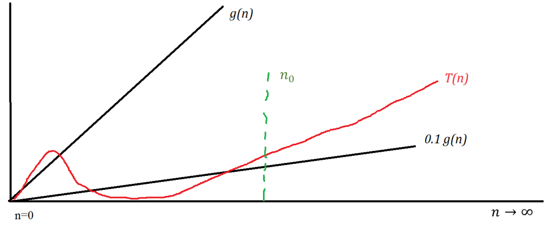
Example: $T(n) := 0.2n^3 + 100\frac{n^3}{\log\log n} + 5n^2 \log n + 2^{\sqrt{\log n}}$ is $O(n^4)$ and $O(n^3)$

— O is an upper bound of $T(n)$ like $\leqslant$

Ω-notation

Definition

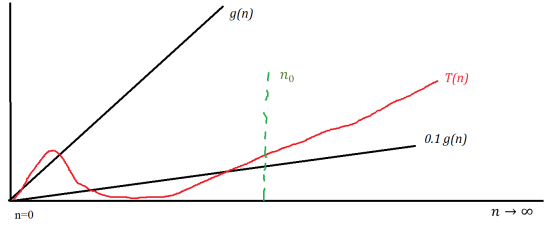
$T(n) = \Omega(g(n))$ if there exist c and n_0 such that $T(n) \geqslant c \cdot g(n)$ for all $n > n_0$.



Ω-notation

Definition

$T(n) = \Omega\big(g(n)\big)$ if there exist c and n_0 such that $T(n) \geqslant c \cdot g(n)$ for all $n > n_0$.



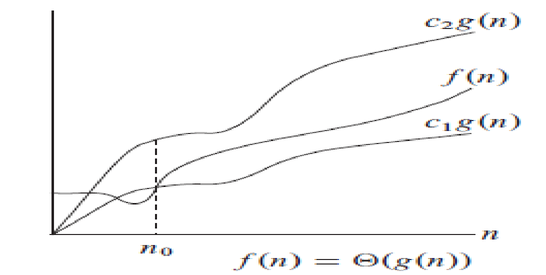
Example: $T(n) := 0.2n^3 + 100\frac{n^3}{\log\log n} + 5n^2 \log n + 2^{\sqrt{\log n}}$ is $\Omega(n^2)$, $\Omega(n^3)$ and so on

— Ω is a lower bound of $T(n)$ like $\geqslant$

Θ-notation

Definition

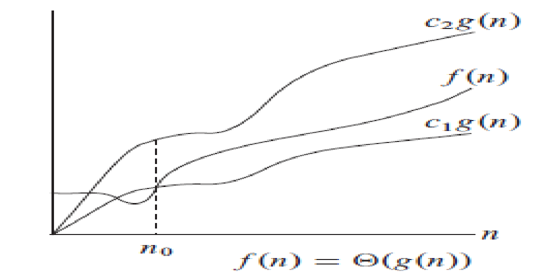
$T(n) = \Theta\big(g(n)\big)$ iff $T(n) = O\big(g(n)\big)$ and $T(n) = \Omega\big(g(n)\big)$. Equivalently, there exist c_1, c_2 and n_0 such that $T(n) \in \big[c_1 \cdot g(n), c_2 \cdot g(n)\big] \forall n > n_0$.



Θ-notation

Definition

$T(n) = \Theta\big(g(n)\big)$ iff $T(n) = O\big(g(n)\big)$ and $T(n) = \Omega\big(g(n)\big)$. Equivalently, there exist c_1, c_2 and n_0 such that $T(n) \in \big[c_1 \cdot g(n), c_2 \cdot g(n)\big] \forall n > n_0$.



Questions

1. Is $0.1n^3 + 10n^{2.99} = \Theta(n^3)$?
2. Is $n^5 = 2^{\Theta(\log n)}$?
3. Is $n \log n = \Theta(n)$?

More Notations

O can be thought as $\leqslant$, let us define o for strictly $<$.

o-notation

$T(n) = o\big(g(n)\big)$ if $\forall c > 0$, $\exists n_0$ such that $T(n) < c \cdot g(n)$ for all $n > n_0$.
Example: $n = o(n \log n)$ but $\frac{n}{100} \neq o(n)$.

More Notations

O can be thought as $\leqslant$, let us define o for strictly $<$.

o-notation

$T(n) = o\big(g(n)\big)$ if $\forall c > 0$, $\exists n_0$ such that $T(n) < c \cdot g(n)$ for all $n > n_0$.
Example: $n = o(n \log n)$ but $\frac{n}{100} \neq o(n)$.

Let us define ω for strictly $>$.

ω-notation

$T(n) = \omega\big(g(n)\big)$ if $\forall c > 0$, $\exists n_0$ such that $T(n) > c \cdot g(n)$ for all $n > n_0$.
Example: $n^2 = \omega(n \log n)$ but $100n \neq \omega(n)$.

Discussions about Asymptotic Analysis

Disadvantages

(1) Cannot tell you whether algorithm is practical on given inputs (like $100n^{2.73}$ vs n^3 for $n \leqslant 10^4$).
(2) Ignores constant factor improvements which are important in practice.

Discussions about Asymptotic Analysis

Disadvantages

(1) Cannot tell you whether algorithm is practical on given inputs (like $100n^{2.73}$ vs n^3 for $n \leqslant 10^4$).
(2) Ignores constant factor improvements which are important in practice.

Advantages:
(1) Independent of hardware and implementation.
(2) Compare behavior on sufficiently large inputs.
(3) Usually an algorithm with better asymptotic behavior will do better in practice (though there are notable exceptions).

Because of its advantages, this class will almost exclusively use big-O analysis — running time := asymptotic order

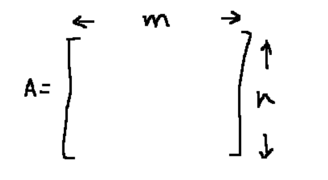
Outline

Input Size

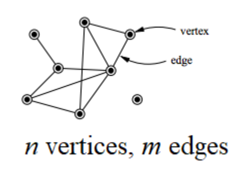
We focus on the relation between asymptotic time and input size.

Recall n is a parameter of input size

1. For an array $A = [\underbrace{0, 2, 3, \ldots, 99}_{n}]$, the input size = array-length n.
2. For a matrix of dimension $n \times m$, the input size = nm.



3. For a graph with n vertices and m edges, the input size = $n + m$



Input Size of Numbers

Say an algorithm is in linear time only if its running time is $O(\text{input size})$.

Compute the Fibonacci number

---

function FIBONACCI(n)
    $f[0] \leftarrow 1$; $f[1] \leftarrow 1$
    for $i = 2, \ldots, n$ **do**
        $f[i] \leftarrow f[i-1] + f[i-2]$
    end for
    return $f(n)$
end function

Question: (1) Is Algorithm FIBONACCI in linear time or not?
(2) What is the input size?

Formal Definition: Input size in binary encoding

Given an instance Φ as the input problem, there are many ways to encode it as the input.

Example

For FIBONACCI, the instance is an integer n — But the input could be either $\underbrace{1 \cdots 1}_{n}$ or the binary presentation of n.

Formal Definition: Input size in binary encoding

Given an instance Φ as the input problem, there are many ways to encode it as the input.

Example

For FIBONACCI, the instance is an integer n — But the input could be either $\underbrace{1 \cdots 1}_{n}$ or the binary presentation of n.

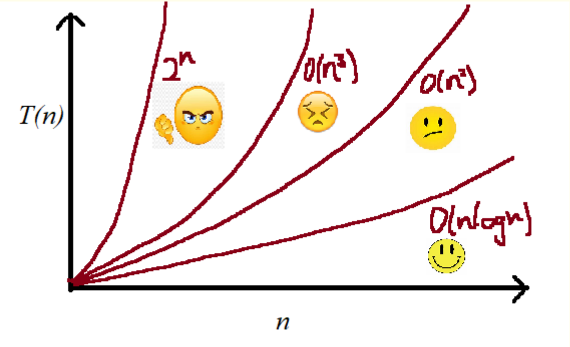
Always define the input size as the shortest length to encode it as binary numbers.

Question

Formally, input size of an integer n is $O(\log_2 n)$. What is the running time of Algorithm FIBONACCI in terms of the input size?

More

- Similarly, we say an algorithm is in almost-linear time only if its running time is $O(\textit{input size})^{1+o(1)}$.
- Almost quadratic time means $O(\text{input size})^{2+o(1)}$.
- Cubic time means $O(\text{input size})^{3}$.
- Exponential time means $2^{O(\text{input size})}$.



Running time of Euclid's algorithm?

Algorithm Euclid's algorithm for GCD

```
function EUCLID(a, b)
    if b=0 then
        return a
    else
        return Euclid(b, a mod b)
    end if
end function
```

Fact: Euclid's algorithm is in linear time $O(\log a + \log b)$!

Running time of Euclid's algorithm?

Algorithm Euclid's algorithm for GCD

```
function EUCLID(a, b)
    if b=0 then
        return a
    else
        return Euclid(b, a mod b)
    end if
end function
```

Fact: Euclid's algorithm is in linear time $O(\log a + \log b)$!

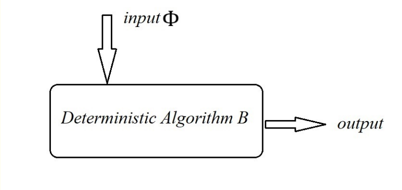
Remark

In fact, it is $(\log a + \log b)^{O(1)}$ because division and module takes $(\log a + \log b)^{O(1)}$ instructions for large integers a and b say $> 2^{64}$.

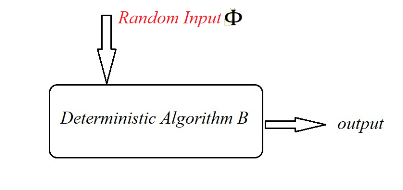
Outline

Two concepts

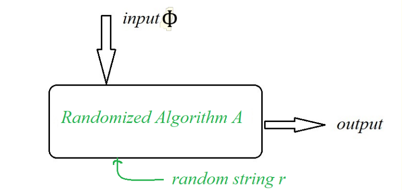
Running time $T(n)$ denotes the longest time of an deterministic algorithm B on all inputs Φ of size parameter n.



Many extensions:



Algorithm on a random input

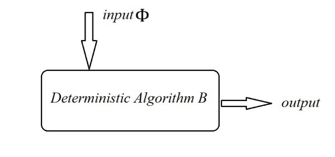


A randomized algorithm

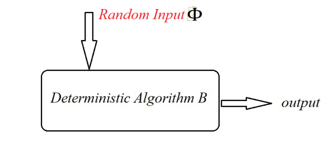
Random Data

Definition: Worst case running time

Given a deterministic ALG B and n, $T(n) :=$ the longest running time of B among all inputs of parameter $n \Leftrightarrow T(n) = \max_{\Phi:|\Phi|=n} \mathrm{Time}\Big(B(\Phi)\Big)$.



(a) Worst case

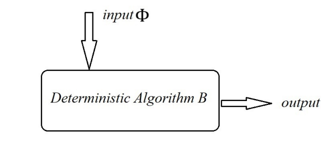


(b) Average case

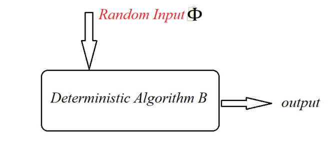
Random Data

Definition: Worst case running time

Given a deterministic ALG B and n, $T(n) :=$ the longest running time of B among all inputs of parameter $n \Leftrightarrow T(n) = \max_{\Phi:|\Phi|=n} \text{Time}\Big(B(\Phi)\Big)$.



(c) Worst case



(d) Average case

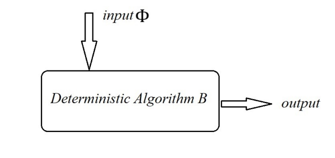
Definition: Average-case running time

The average-case time denotes the average time of B on all input of parameter n, i.e., $\mathbb{E}_{\Phi:|\Phi|=n} \text{Time}\Big(B(\Phi)\Big)$.

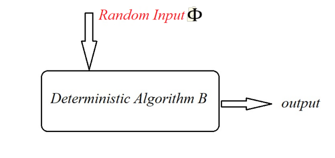
Random Data

Definition: Worst case running time

Given a deterministic ALG B and n, $T(n) :=$ the longest running time of B among all inputs of parameter $n \Leftrightarrow T(n) = \max_{\Phi:|\Phi|=n} \mathrm{Time}\Big(B(\Phi)\Big)$.



(e) Worst case



(f) Average case

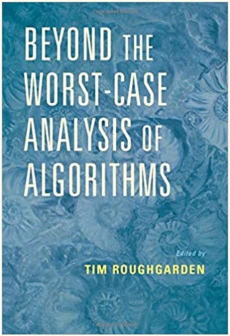
Definition: Average-case running time

The average-case time denotes the average time of B on all input of parameter n, i.e., $\mathbb{E}_{\Phi:|\Phi|=n} \mathrm{Time}\Big(B(\Phi)\Big)$.

Example: Define the average-time on random graphs with n vertices?

Discussion

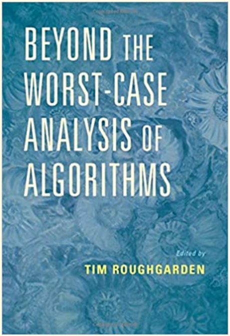
Why we care about the average-case running time?



1. Usually, it is faster than the worst-case time — if not, this problem may be used for cryptography like lattice-based problems.

Discussion

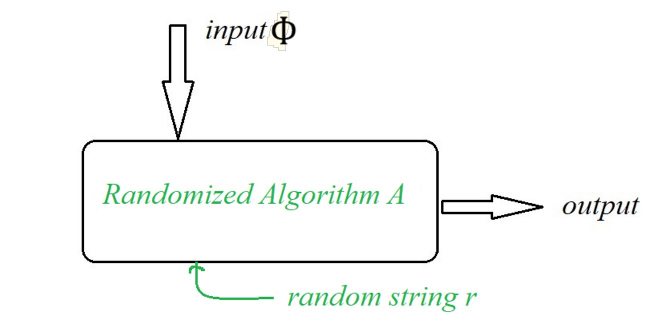
Why we care about the average-case running time?



1. Usually, it is faster than the worst-case time — if not, this problem may be used for cryptography like lattice-based problems.
2. It provides provable guarantees for practical applications.
3. Many problems like sorting can reduce the worst-case to the average case.

Randomized Algorithms

A randomized Algorithm with input Φ and random string r



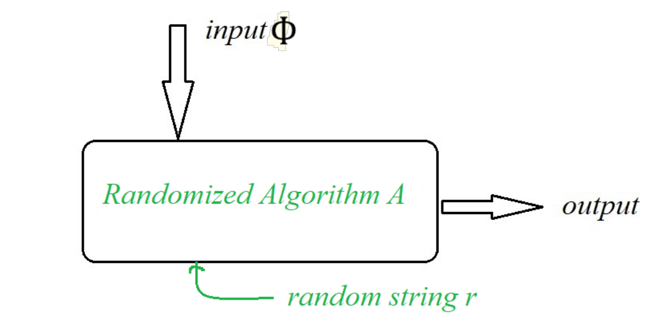
- The expected running time of A is

$$T(n) = \max_{\Phi:|\Phi|=n} \left\{ \mathop{\mathbb{E}}_{r} \left[\mathrm{Time}\big(A(\Phi, r)\big) \right] \right\}$$

— also called running time in the worst case.

Randomized Algorithms

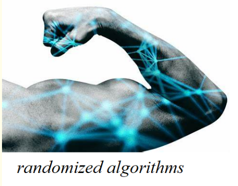
A randomized Algorithm with input Φ and random string *r*



- The expected running time of *A* is
$$T(n) = \max_{\Phi:|\Phi|=n} \left\{ \mathop{\mathbb{E}}_{r}\left[\mathrm{Time}\big(A(\Phi, r)\big) \right] \right\}$$
— also called running time in the worst case.
- The average-case running time of a random algorithm is
$$T(n) = \mathop{\mathbb{E}} \left\{ \mathbb{E}\left[\mathrm{Time}\big(A(\Phi, r)\big) \right] \right\}.$$

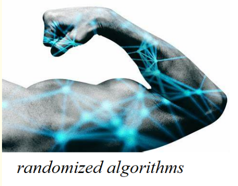
More about randomized algorithms

Randomized algorithms are faster, simpler, and more powerful than their deterministic counterparts.



More about randomized algorithms

Randomized algorithms are faster, simpler, and more powerful than their deterministic counterparts.



1. Disadvantages: Trickier to analyze and hard to control.
2. Fail with non-zero probability — but could be tiny $< 2^{-100}$ (in theory).

More about randomized algorithms

Randomized algorithms are faster, simpler, and more powerful than their deterministic counterparts.

randomized algorithms

1. Disadvantages: Trickier to analyze and hard to control.
2. Fail with non-zero probability — but could be tiny $< 2^{-100}$ (in theory).
3. One central question in CS: How much stronger are randomized algorithms than their deterministic counterparts?

Questions?