Design and Analysis of Algorithm

Complexity Analysis

Outline

Problem and Solution

Problem Description.

- A group of parameters: specifies the problem (set, variable, function, sequences, etc.), include descriptions of domain and relation among them
- Definition of solution: determined by optimization objective or constraints

Problem and Solution

Problem Description.

- A group of parameters: specifies the problem (set, variable, function, sequences, etc.), include descriptions of domain and relation among them
- Definition of solution: determined by optimization objective or constraints

Instance. An assignment of parameters $\rightarrow$ an instance of problem

Algorithm

Definition 1 (Algorithm)

An algorithm $\mathcal{A}$ is a finite sequence of well-defined, computer implementable instructions that solve a class of problems

- algorithms are always unambiguous
- specifications for performing calculations, data processing, automated reasoning, and other tasks.

Algorithm $\mathcal{A}$ for Problem P

- take any instance of P as $\mathcal{A}$'s input, computation of each step is deterministic
- $\mathcal{A}$ halts in finite steps
- always output the correct solution

Basic Computer Steps and Input Size

An insightful analysis is based on the right simplifications.

Basic computer steps. capture abstract atomic operation

- Example. compare, add, multiplication, swap, assign ...

$\boxed{\text{This is the first important simplification!}}$

Input size. capture the scale of instance: proportional to the length of instance encoding string

- Example. number of array, number of scheduling tasks, number of vertices and edges

Examples of Input Size and Basic Computer Steps (1/2)

Sorting. array $a[n]$

- n: the number of elements in the array
- element compares and movement

Searching. search x in array $a[n]$

- n: the number of elements in the array
- element compares between x and $a[i]$

Integer multiplication. $a \times b$

- the binary length of a and b, a.k.a. $m = \log_2 a$, $n = \log_2 b$
- bit-wise multiplication – $a \times b$ requires $\#mn$ bit-wise multiplication

Examples of Input Size and Basic Computer Steps (2/2)

Matrix multiplication. $\mathbf{A}_{n_1 \times n_2} \cdot \mathbf{B}_{n_2 \times n_3}$

- dimensions of $\mathbf{A}$ and $\mathbf{B}$, a.k.a. n_1, n_2, n_3
- point-wise multiplication – $\mathbf{A} \cdot \mathbf{B}$ requires $n_1 n_2 n_3$-times point-wise multiplication
- $n_1 = n_2 = n_3 = n \rightsquigarrow n^3$

Graph visit. $G = (V, E)$

- number of vertices and edges
- assignment of flag variable

Measurement of Algorithm's Efficiency

Express running time by counting the number of basic computer steps as a function of the size of the input.

uncluttered, machine-independent characterization

Measurement of Algorithm's Efficiency

Express running time by counting the number of basic computer steps as a function of the size of the input.

uncluttered, machine-independent characterization

For different inputs of the same instance size, the number of basic computer steps might vary $\Rightarrow$ functions could be different

Choose which one?

Three Types of Analyses

I am prepared for the worst, but hope for the best.

— Benjamin Disraeli

Three Types of Analyses

> *I am prepared for the worst, but hope for the best.*
>
> — Benjamin Disraeli

Worst-case. Maximum running time for any input of size n.

Example. Quicksort requires at most n^2 compares to sort n elements.

Three Types of Analyses

> *I am prepared for the worst, but hope for the best.*
>
> — Benjamin Disraeli

Worst-case. Maximum running time for any input of size n.

Example. Quicksort requires at most n^2 compares to sort n elements.

Best-case. Minimum running time for all inputs of size n

Example. Insertion sort only requires n compares when the input is sorted already.

Three Types of Analyses

> *I am prepared for the worst, but hope for the best.*
>
> — Benjamin Disraeli

Worst-case. Maximum running time for any input of size n.

Example. Quicksort requires at most n^2 compares to sort n elements.

Best-case. Minimum running time for all inputs of size n

Example. Insertion sort only requires n compares when the input is sorted already.

Average-case. Expected running time for a random input of size n

Example. expected number of element compares of Quicksort is $\sim n \log n$.

About Worst-Case

Algorithm. Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare.

- Linux `grep` command

About Worst-Case

Algorithm. Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare.

- Linux `grep` command

Cryptography. Require hard instance to be efficiently samplable — problems only have high worst-case complexity may not be suitable to be used as hardness assumption

About Worst-Case

Algorithm. Some exponential-time algorithms are used widely in practice because the worst-case instances seem to be rare.

- Linux `grep` command

Cryptography. Require hard instance to be efficiently samplable — problems only have high worst-case complexity may not be suitable to be used as hardness assumption

Good news to Algorithms = Bad news to Cryptography

Win-Win flavor

Formula of $A(n)$

$A(n)$: average-case complexity

- Let X be the set of all inputs of size n, $\Pr[x \in X] = p(x)$
- $t(x)$: the number of basic operations that $\mathcal{A}$ performs on input x

$$A(n) = \sum_{x \in X} p(x)t(x)$$

In many cases, we assume the input distribution is a uniform distribution.

Example of Search

Search Problem

Input. Array $a[n]$ with ascending order, search x

Output. $j \in [0, \ldots, n]$

- if $x \in a[n]$, then j is the first index such that $a[j] = x$
- else, $j = 0$

Basic operation. element compare between x and $a[i]$

Sequential Search Algorithm

Algorithm 1: Search($a[n], x$)

```
flag ← 0;
for j = 1 to n do
    if a[j] = x then
        flag = 1;
        break;
    end
end
if flag = 0 then  j = 0 ;
return j;
```

Example. $1, 2, 3, 4, 5$

- $x = 4$: 4 compares
- $x = 2.5$: 5 compares

Worst-case Complexity

There are $2n + 1$ types different inputs:

- Case inside: $x = a[1], x = a[2], \ldots, x = a[n]$
- Case outside: $x < a[1], a[1] < x < a[2], \ldots, a[n] < x$

Worse-case input. $x \notin A \vee x = A[n]$, requires n compares

Worse-case complexity. $T(n) = n$

Average-case complexity

Assume $\Pr[x \in A] = p$, and distributes on each position with equal probability.

$$\begin{aligned} T(n) &= \sum_{i=1}^{n} i \cdot \frac{p}{n} + (1-p)n \quad \text{//sum of arithmetic sequence} \\ &= \frac{p(n+1)}{2} + (1-p)n \end{aligned}$$

When $p = 1/2$

$$T(n) = \frac{n+1}{4} + \frac{n}{2} \approx \frac{3n}{4}$$

Outline

Pseudocode of Algorithm

Definition 2 (Pseudocode)

An informal high-level description of the operating principle of algorithms: uses the structural conventions of programming language, but is intended for human reading rather than machine reading.

Instruction	Symbol
Assignment	$\leftarrow$ or $:=$
Branch statement	if...then...[else...]
Loop structure	while, for, repeat until
Transfer statement	goto
Return statement	return
Function call	Func()
Comment	// or /* */

Example: Euclid Algorithm for Greatest Common Divisor

Algorithm 2: EuclidGCD(n, m)

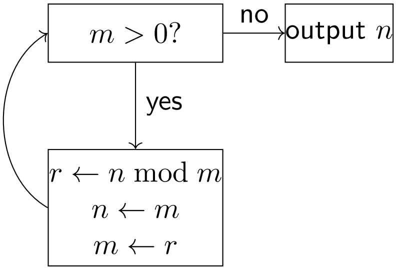
Input: $n, m \in \mathbb{Z}^+$, $n \geq m$
Output: GCD(n, m)

```
while m > 0 do
    r ← n mod m;
    n ← m;
    m ← r;
end
return n
```

Demo: $n = 36$, $m = 15$

while	n	m	r
1st loop	36	15	6
2nd loop	15	6	3
3rd loop	6	3	0
	3	0	0

output 3



Example of Insertion Sort

Algorithm 3: Algorithm InsertSort$(A[n])$

Input: array $A[n]$
Output: $A[n]$ in ascending order

for $j \leftarrow 2$ *to* n **do**
    $x \leftarrow A[j]$;
    $i \leftarrow j - 1$ //insert $A[j]$ to $A[1 ... j-1]$;
    while $i > 0$ *and* $x < A[i]$ **do**
        $A[i+1] \leftarrow A[i]$;
        $i \leftarrow i - 1$;
    end
    $A[i+1] \leftarrow x$;
end

i is the left neighbor index of the final insert position

Demo of Insertion Sort

2	4	1	5	3

$j = 3,\ x = A[3] = 1$

$i = 2,\ A[2] = 4$

$i > 0,\ x < A[2]$ ✓

2	4	4	5	3

$A[3] = 4,\ i = 1,\ x = 1$

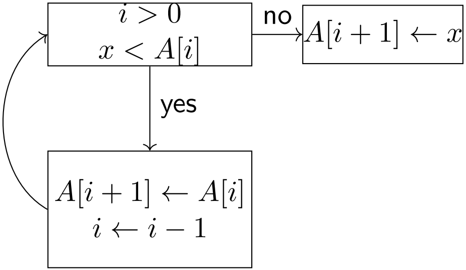
$i > 0,\ x < A[1]$ ✓

2	2	4	5	3

$A[2] = 2,\ i = 0,\ x = 1$

$i > 0$ ✗

1	2	4	5	3



Example of Binary Merge Sort

Algorithm 4: Algorithm MergeSort(A, l, r)

Input: array $A[l, r]$

Output: $A[l, r]$ in ascending order

if $l < r$ **then**
  $m \leftarrow \lfloor (l + r)/2 \rfloor$;
  MergeSort(A, l, m);
  MergeSort$(A, m + 1, r)$;
  Merge(A, l, m, r);
end

MergeSort is a recursive algorithm

- call itself from within its own code

Pseudocode of Algorithm $\mathcal{A}$

Algorithm 5: Algorithm $\mathcal{A}$

Input: Array $P[0,\ldots,n] \in \mathbb{R}^{n+1}$, $x \in \mathbb{R}$

Output: y

1: $y \leftarrow P[0]$; $power \leftarrow 1$;
2: **for** $i \leftarrow 1$ *to* n **do**
3: $\quad power \leftarrow power \times x$;
4: $\quad y \leftarrow y + P[i] \times power$;
5: **end**
6: **return** y;

What do 3-4 compute?

$$\text{for } i \in [n] \longrightarrow \boxed{\begin{array}{c} power \leftarrow power \times x \\ y \leftarrow y + P[i] \times power \end{array}}$$

loop	$power$	y
0	1	$P[0]$
1	x	$P[0] + P[1] \times x$
2	x^2	$P[0] + P[1] \times x + P[2] \times x^2$
3	x^3	$P[0] + P[1] \times x + P[2] \times x^2 + P[3] \times x^3$
		$\cdots$

Input $P[0, \ldots, n]$ is the coefficients of n-degree polynomial $P(x)$

- $\mathcal{A}$ compute $P(x) = \sum_{i=0}^{n} P[i]x^i$

Outline

Motivation

We use functions over $\mathbb{N}$ to capture how the running time or space requirements of algorithms grow as the input size increases.

Motivation

We use functions over $\mathbb{N}$ to capture how the running time or space requirements of algorithms grow as the input size increases.

How to compare them? How to classify them?

Motivation

We use functions over $\mathbb{N}$ to capture how the running time or space requirements of algorithms grow as the input size increases.

How to compare them? How to classify them?

The first simplification leads to another. Now, second simplification comes into play, consider the order of function rather than its concrete form.

Big-O Notations

Paul Bachmann and Edumund Landau invented a family of notations known as Big-O notation to describe the limiting behavior of a function when the input tends towrads infinity.


Figure: Paul Bachmann & Edumund Landau

- also known as Bachmann-Landau or asymptotic notation
- mathematical notation $\leadsto$ describe running times

Big-O Notation

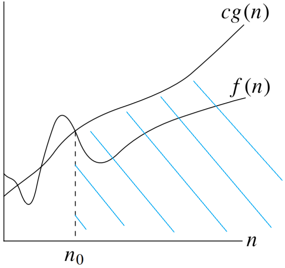
Definition 3 (Big-O)

$\exists c > 0$, $\exists n_0$, such that $\forall n \geq n_0$:

$$f(n) \leq cg(n)$$

f is bounded above by g (up to constant factor) asymptotically

$$f(n) = O(g(n))$$



Limit definition

$$\lim_{n \to \infty} \sup \frac{f(n)}{g(n)} < \infty$$

Some Remarks

Big-O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

- letter O is used because the growth rate of a function is also referred to as the order of the function.
- there are many (c, n_0), it suffices to find one tuple
- for finite values $n \leq n_0$, the inequality may not hold
- constant functions can be written as $O(1)$

More about Big-O

$f(n) = O(g(n))$: the order of $f(n)$ is less than that of $g(n)$

Typical usage: give upper bound

- Insertion sort makes $O(n^2)$ compares to sort n elements.

Example 1. $f(n) = n^2 + n$

- $f(n) = O(n^2) \leftarrow$ choose $c = 2$, $n_0 = 1$
- $f(n) = O(n^3) \leftarrow$ choose $c = 1$, $n_0 = 2$

Example 2. $f(n) = 32n^2 + 17n + 1$

- $f(n) = O(n^2) \leftarrow$ choose $c = 50$, $n_0 = 1$
- $f(n)$ is also $O(n^3)$
- $f(n)$ is neither $O(n)$ nor $O(n \log n)$

Limits of Big-O

Big-O notation only provides an upper bound on the growth rate of the function.

Associated with big-O notation are several related notations, using the symbols o, Ω, ω, and Θ, to describe other kinds of bounds on asymptotic growth rates.

Big-Ω Notation

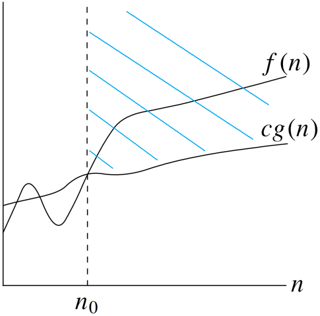
Definition 4 (Big-Ω)

$\exists c > 0,\ \exists n_0,\ \forall n \geq n_0$:

$$f(n) \geq cg(n)$$

f is bounded below by g asymptotically

$$f(n) = \Omega(g(n))$$



Limit definition

$$\lim_{n\to\infty} \inf \frac{f(n)}{g(n)} > 0$$

Example of Big-Ω

$f(n) = \Omega(g(n))$: the order of $f(n)$ is greater than $g(n)$.

Typical usage: give lower bound

- Any compare-based sorting algorithm requires $\Omega(n \log n)$ compares in the worst case.

Meaningless statement. Any compare-based sorting algorithm requires at least $O(n \log n)$ compares in the worst case.

- $O(\cdot)$ cannot give lower bound

Example. $f(n) = n^2 + n$

- $f(n) = \Omega(n^2) \leftarrow c = 1,\ n_0 = 1$
- $f(n) = \Omega(100n) \leftarrow c = 1/100,\ n_0 = 1$

Tight Bound

Big O and Ω notations are originally used as a tight upper-bound (resp. lower-bound) on the growth of an algorithm's effort

But, according to the definitions

$g(n)$ could be a loose upper-bound (resp. lower-bound).

Tight Bound

Big O and Ω notations are originally used as a tight upper-bound (resp. lower-bound) on the growth of an algorithm's effort

But, according to the definitions

$g(n)$ could be a loose upper-bound (resp. lower-bound).

To make the role as a tight upper-bound more clear, small o and ω notations are used to describe an upper-bound/lower-bound that cannot be tight.

Small o Notation

Definition 5 (Small-o)

$\forall c > 0$, $\exists n_0$, such that $\forall n \geq n_0$:

$$f(n) < cg(n)$$

f is dominated by g asymptotically:

$$f(n) = o(g(n))$$

Limit definition

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

More about Small-o

$f(n) = o(g(n))$: the order of $f(n)$ is strictly smaller than that of $g(n)$

Typical usage. $\log n = o(n)$

Example. $f(n) = n^2 + n$, $f(n) = o(n^3)$

- $c \geq 1$: obviously holds, choose $n_0 = 2 \Rightarrow n^2 + n < cn^3$

$$cn^3 \geq n^3 = n^2((n-1)+1) \geq n^2 + n, \text{when } n \geq n_0$$

- $0 < c < 1$: choose $n_0 > \lceil 2/c \rceil$, because

$$cn \geq cn_0 \geq 2$$
$$n^2 + n < 2n^2 \leq cn \cdot n^2 < cn^3$$

Small ω Notation

Definition 6

$\forall c > 0$, $\exists n_0$, $\forall n \geq n_0$:

$$f(n) > c \cdot g(n)$$

f dominates g asymptotically

$$f(n) = \omega(g(n))$$

Limit definition:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

Example of Small ω

$f(n) = \omega(g(n))$: the order of $f(n)$ is strictly larger than that of $g(n)$

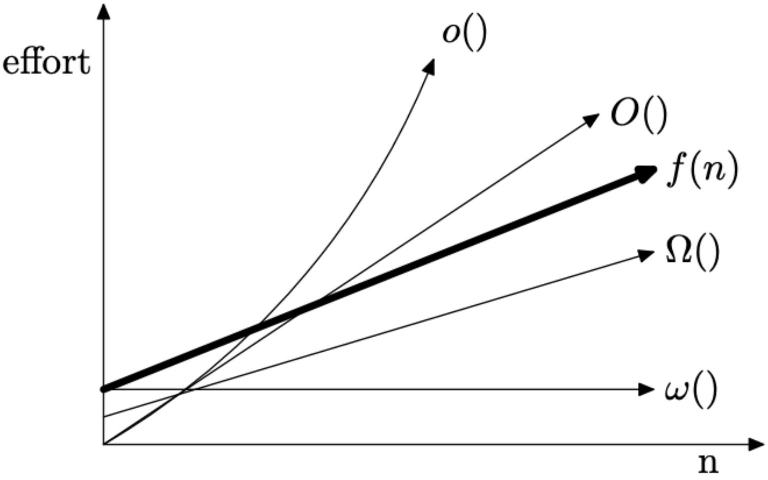
Typical usage. $n = \omega(\log n)$

Example. $f(n) = n^2 + n$, $f(n) = \omega(n)$

- $\lim_{n\to\infty} \frac{f(n)}{n} = \infty$
- $f(n) \neq \omega(n^2)$: choose $c = 2$, there does not exist n_0 such that $\forall n \geq n_0$

$$cn^2 = 2n^2 < n^2 + n$$

Visualization of the Relationships Between These Notations



Comparisons

Notation	$? \; c > 0$	$? \; n_0$	$f(n) \; ? \; c \cdot g(n)$	meaning
O	$\exists$	$\exists$	$\leq$	upper bound
o	$\forall$	$\exists$	$<$	non-tight upper bound
Ω	$\exists$	$\exists$	$\geq$	lower bound
ω	$\forall$	$\exists$	$>$	non-tight lower bound

While o and ω are not often used to described algorithms

- We define a combination of O and Ω: Θ, which means $g(n)$ is both a tight upper-bound and a tight lower-bound

Big-Θ Notation: Aims to a Tight Bound

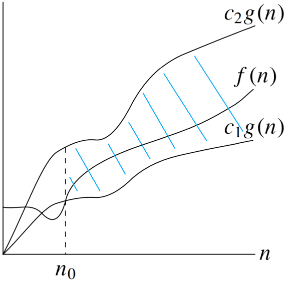
Definition 7 (Big-Θ)

$\exists c_1 > 0$, $\exists c_2 > 0$, $\exists n_0$, such that $\forall n > n_0$

$$c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$$

f is bounded both above and below by g asymptotically

$$f(n) = \Theta(g(n))$$



Limit definition

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

Proof of Equivalence

Proof. Definition of the limit $\Rightarrow \forall \varepsilon > 0$, $\exists n_0$, $\forall n \geq n_0$:

$$|f(n)/g(n) - c| < \varepsilon$$
$$c - \varepsilon < f(n)/g(n) < c + \varepsilon$$

Proof of Equivalence

Proof. Definition of the limit $\Rightarrow \forall \varepsilon > 0$, $\exists n_0$, $\forall n \geq n_0$:

$$|f(n)/g(n) - c| < \varepsilon$$
$$c - \varepsilon < f(n)/g(n) < c + \varepsilon$$

choose $\varepsilon = c/2 \Rightarrow c/2 < f(n)/g(n) < 3c/2$

- $\forall n \geq n_0$, $f(n) \leq (3c/2)g(n) \Rightarrow f(n) = O(g(n))$
- $\forall n \geq n_0$, $f(n) \geq (c/2)g(n) \Rightarrow f(n) = \Omega(g(n))$.

Proof of Equivalence

Proof. Definition of the limit $\Rightarrow \forall \varepsilon > 0$, $\exists n_0$, $\forall n \geq n_0$:

$$|f(n)/g(n) - c| < \varepsilon$$
$$c - \varepsilon < f(n)/g(n) < c + \varepsilon$$

choose $\varepsilon = c/2 \Rightarrow c/2 < f(n)/g(n) < 3c/2$

- $\forall n \geq n_0$, $f(n) \leq (3c/2)g(n) \Rightarrow f(n) = O(g(n))$
- $\forall n \geq n_0$, $f(n) \geq (c/2)g(n) \Rightarrow f(n) = \Omega(g(n))$.

This proves $f(n) = \Theta(g(n))$

More about Big-Θ

$f(n) = \Theta(g(n))$: $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$, $f(n)$ and $g(n)$ have the same order

Typical usage:

- Mergesort makes $\Theta(n \log n)$ compares to sort n elements.

Example 1. $f(n) = n^2 + n$, $g(n) = 100n^2$

$f(n) = \Theta(g(n))$

Example 2. $f(n) = 32n^2 + 17n + 1$

- $f(n)$ is $\Theta(n^2) \leftarrow$ choose $c_1 = 32$, $c_2 = 50$, $n_0 = 1$
- $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$

Example of Primality Test

Algorithm 6: PrimalityTest(n)

Input: odd integer $n > 2$

Output: true or false

```
s ← ⌊n^{1/2}⌋;
for j ← 2 to s do
    if j divides n then return false;
end
return true;
```

Example of Primality Test

Algorithm 7: PrimalityTest(n)

Input: odd integer $n > 2$

Output: true or false

$s \leftarrow \lfloor n^{1/2} \rfloor$;

for $j \leftarrow 2$ *to* s **do**

  if j *divides* n **then return** *false*;

end

return *true*;

If $n^{1/2}$ is computable in $O(1)$-time, the basic operation is divide

Example of Primality Test

Algorithm 8: PrimalityTest(n)

Input: odd integer $n > 2$
Output: true or false

$s \leftarrow \lfloor n^{1/2} \rfloor$;
for $j \leftarrow 2$ *to* s **do**
    if j *divides* n **then return** *false*;
end
return *true*;

If $n^{1/2}$ is computable in $O(1)$-time, the basic operation is divide

What is the worst-case complexity of naive primality test?

Input Size Matters: Case of Primality Test

Using n as input size of $W(\cdot)$

$$W(n) = O(n^{1/2}) ✓ \quad W(n) = \Omega(n^{1/2}) ✗$$

- Consider inputs of the form $3m$, then $n^{1/2}$ is not the lower bound

Using λ as input size (length of binary representation of n) of $W(\cdot)$.

$$W(n) = O(2^{\lambda/2}) ✓ \quad W(n) = \Omega(2^{\lambda/2}) ✓$$

a.k.a. $W(\lambda) = \Theta(2^{\lambda/2})$

Input size aims to capture the scale of a class of instances. This is what make this notion useful. For the first case, a class of instance degrades to a single instance, thus making worst-case complexity meaningless.

Big-O notation with multiple variables

Upper bounds. $f(m,n)$ is $O(g(m,n))$ if $\exists c > 0$, $m_0 \geq 0$ and $n_0 \geq 0$ such that $\forall n \geq n_0$ and $m \geq m_0$, $f(m,n) \leq c \cdot g(m,n)$

Example. $f(m,n) = 32mn^2 + 17mn + 32n^3$

- $f(m,n)$ is both $O(mn^2 + n^3)$ and $O(mn^3)$
- $f(m,n)$ is neither $O(n^3)$ nor $O(mn^2)$

Typical usage. Breadth-first search takes $O(m+n)$ time to find the shortest path from s to t in a digraph

Properties of Big-O Notations (1/2)

Transitivity. The order of functions are transitive.

- $f = O(g) \wedge g = O(h) \Rightarrow f = O(h)$
- $f = \Omega(g) \wedge g = \Omega(h) \Rightarrow f = \Omega(h)$
- $f = \Theta(g) \wedge g = \Theta(h) \Rightarrow f = \Theta(h)$
- $f = o(g) \wedge g = o(h) \Rightarrow f = o(h)$
- $f = \omega(g) \wedge g = \omega(h) \Rightarrow f = \omega(h)$

Properties of Big-O Notations (2/2)

Product

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$

Properties of Big-O Notations (2/2)

Product

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$

Sum

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(\max(g_1, g_2))$
- This implies $f_1 = O(g) \wedge f_2 = O(g) \Rightarrow f_1 + f_2 \in O(g)$, which means that $O(g)$ is a convex cone.

Properties of Big-O Notations (2/2)

Product

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$

Sum

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(\max(g_1, g_2))$
- This implies $f_1 = O(g) \wedge f_2 = O(g) \Rightarrow f_1 + f_2 \in O(g)$, which means that $O(g)$ is a convex cone.

This property extends to a finite composition of f_i

- Application. For an algorithm, if the running time of its each step is upper bounded by $h(n)$, and the algorithm only consists of constant steps, then the overall complexity is $O(h(n))$.

Properties of Big-O Notations (2/2)

Product

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 f_2 = O(g_1 g_2)$
- $f \cdot O(g) = O(fg)$

Sum

- $f_1 = O(g_1) \wedge f_2 = O(g_2) \Rightarrow f_1 + f_2 = O(\max(g_1, g_2))$
- This implies $f_1 = O(g) \wedge f_2 = O(g) \Rightarrow f_1 + f_2 \in O(g)$, which means that $O(g)$ is a convex cone.

This property extends to a finite composition of f_i

- Application. For an algorithm, if the running time of its each step is upper bounded by $h(n)$, and the algorithm only consists of constant steps, then the overall complexity is $O(h(n))$.

Multiplication by a constant. Let $k > 0$ be a constant. Then:

- $O(kg) = O(g)$, if $k \neq 0$.
- $f = O(g) \Rightarrow kf = O(g)$ (multiplicative constants can be omitted)

Outline

Important Function Classes (increasing order)

We list important function class in ascending order

- constant: $O(1)$
- double logarithmic: $\log \log n$
- logarithmic: $\log n$
- polylogarithmic: $(\log n)^c$, $c > 1$
- fractional power: n^c, $0 < c < 1$

- linear: $O(n)$
- loglinear or quasilinear: $n \log n$
- polynomial: n^c, $c > 1$ (quadratic: n^2, cubic n^3)

- exponential: c^n, $c > 1$
- factorial: $n!$

Asymptotic Bounds for some Common Functions (1/3)

Technical tool. Limit Definitions of $O, \Omega, \Theta, o, \omega$

Polynomials. Let $f(n) = a_0 + a_1 n + \cdots + a_d n^d$, then $f(n) = \Theta(n^d)$.

Proof.

$$\lim_{n \to \infty} \frac{a_0 + a_1 n + \cdots + a_d n^d}{n^d} = a_d > 0$$

Example. Let $f(n) = n^2/2 - 3n$, $f(n) = \Theta(n^2)$.

Asymptotic Bounds for some Common Functions (2/3)

Logarithms. $\Theta(\log_a n) \sim \Theta(\log_b n)$ for any constants $a, b > 0$

- no need to specify base (assuming it is a constant)

Asymptotic Bounds for some Common Functions (2/3)

Logarithms. $\Theta(\log_a n) \sim \Theta(\log_b n)$ for any constants $a, b > 0$

- no need to specify base (assuming it is a constant)

Logarithms vs. Polynomials. $\forall d > 1$, $\log n = o(n^d)$.

Asymptotic Bounds for some Common Functions (2/3)

Logarithms. $\Theta(\log_a n) \sim \Theta(\log_b n)$ for any constants $a, b > 0$

- no need to specify base (assuming it is a constant)

Logarithms vs. Polynomials. $\forall d > 1$, $\log n = o(n^d)$.

Proof.

- Both $\lim_{n\to\infty} \ln n = \infty$ and $\lim_{n\to\infty} n^d = \infty$ and are differentiable:
- Apply L'Hôpital (Bernoulli) rule once

$$\lim_{n\to\infty} \frac{\ln n}{n^d} = \lim_{n\to\infty} \frac{1/n}{dn^{d-1}} = \lim_{n\to\infty} \frac{1}{dn^d} = 0$$

Asymptotic Bounds for some Common Functions (3/3)

Exponentials vs. Polynomials. $\forall c > 1$ and $\forall d > 0$, $n^d = o(c^n)$.

Asymptotic Bounds for some Common Functions (3/3)

Exponentials vs. Polynomials. $\forall c > 1$ and $\forall d > 0$, $n^d = o(c^n)$.

Proof. W.L.O.G, choose d as a positive integer,

- Both $\lim_{n\to\infty} n^d = \infty$ and $\lim_{n\to\infty} c^n = \infty$ and are differentiable.
- Apply L'Hôpital (Bernoulli) rule repeatedly until the numerator is constant

$$\lim_{n\to\infty} \frac{n^d}{c^n} = \lim_{n\to\infty} \frac{dn^{d-1}}{c^n \ln c} = \lim_{n\to\infty} \frac{d(d-1)n^{d-2}}{c^n (\ln c)^2}$$
$$= \cdots = \lim_{n\to\infty} \frac{d!}{c^n (\ln c)^d} = 0$$

Factorial Function

Stirling Formula

(named after James Stirling, but it was first stated by Abraham de Moivre)

Factorial Function

Stirling Formula
(named after James Stirling, but it was first stated by Abraham de Moivre)

Precise form:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

Factorial Function

Stirling Formula
(named after James Stirling, but it was first stated by Abraham de Moivre)

Precise form:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Simple form:

$$\ln n! = n \ln n - n + O(\ln n)$$

Factorial Function

Stirling Formula
(named after James Stirling, but it was first stated by Abraham de Moivre)

Precise form:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

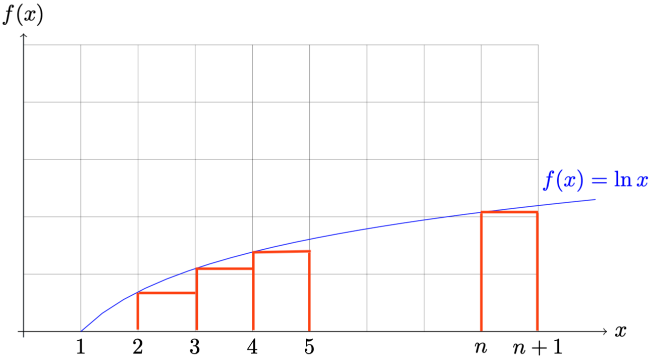
Simple form:

$$\ln n! = n \ln n - n + O(\ln n)$$

- $n! = o(n^n)$
- $n! = \omega(2^n)$
- $\underline{\ln n! = \Theta(n \ln n)}$ (proving via integral method as below)

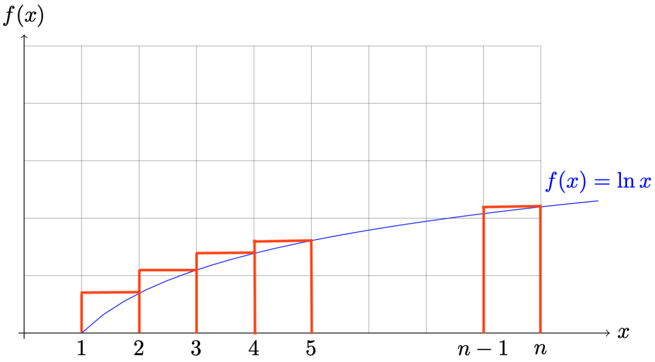
Proof of the Upper Bound

$$
\begin{aligned}
\ln n! = \sum_{k=1}^{n} \ln k &\le \int_{2}^{n+1} \ln x dx \\
&= (x \ln x - x)_2^{n+1} \\
&= O(n \ln n)
\end{aligned}
$$



Proof of the Lower Bound

$$
\begin{aligned}
\ln n! = \sum_{k=1}^{n} \ln k &\geq \int_1^n \ln x dx \\
&= (x \ln x - x)_1^n \\
&= n \ln n - n + 1 = \Omega(n \ln n)
\end{aligned}
$$



Application: Estimate the Size of Search Space

Recall the ROI optimization problem: the number of different investment schemes: m coins on n projects

$$
\begin{aligned}
C_{m+n-1}^{m} &= \frac{(m+n-1)!}{m!(n-1)!} \\
&= \frac{\sqrt{2\pi(m+n-1)}(m+n-1)^{m+n-1}\left(1+\Theta\left(\frac{1}{m+n-1}\right)\right)}{\sqrt{2\pi m}m^{m}\left(1+\Theta\left(\frac{1}{m}\right)\right)\sqrt{2\pi(n-1)}(n-1)^{n-1}\left(1+\Theta\left(\frac{1}{n-1}\right)\right)} \\
&= \Theta((1+\varepsilon)^{m+n-1})
\end{aligned}
$$

Rounding Function

Rounding a number means replacing it with a different number that is *approximately equal* to the original, but has a *shorter, simpler* representation

- round down (or take the floor)
  $y = \text{floor}(x) = \lfloor x \rfloor$: y is the largest integer that does not exceed x
- round up (or take the ceiling)
  $y = \text{ceil}(x) = \lceil x \rceil$: y is the smallest integer that is not less than x

Rounding Function

Rounding a number means replacing it with a different number that is *approximately equal* to the original, but has a *shorter, simpler* representation

- round down (or take the floor)
  $y = \text{floor}(x) = \lfloor x \rfloor$: y is the largest integer that does not exceed x
- round up (or take the ceiling)
  $y = \text{ceil}(x) = \lceil x \rceil$: y is the smallest integer that is not less than x

Example. $\lfloor 2.6 \rfloor = 2$, $\lceil 2.6 \rceil = 3$, $\lfloor 2 \rfloor = \lceil 2 \rceil = 2$

Rounding Function

Rounding a number means replacing it with a different number that is *approximately equal* to the original, but has a *shorter, simpler* representation

- round down (or take the floor)
  $y = \text{floor}(x) = \lfloor x \rfloor$: y is the largest integer that does not exceed x
- round up (or take the ceiling)
  $y = \text{ceil}(x) = \lceil x \rceil$: y is the smallest integer that is not less than x

Example. $\lfloor 2.6 \rfloor = 2$, $\lceil 2.6 \rceil = 3$, $\lfloor 2 \rfloor = \lceil 2 \rceil = 2$

Application. When performing binary search in $A[n]$, the index of median is $\lfloor n/2 \rfloor$, the subproblem is of size $\lfloor n/2 \rfloor$.

Properties of Rounding Function

Proposition 1. $x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$

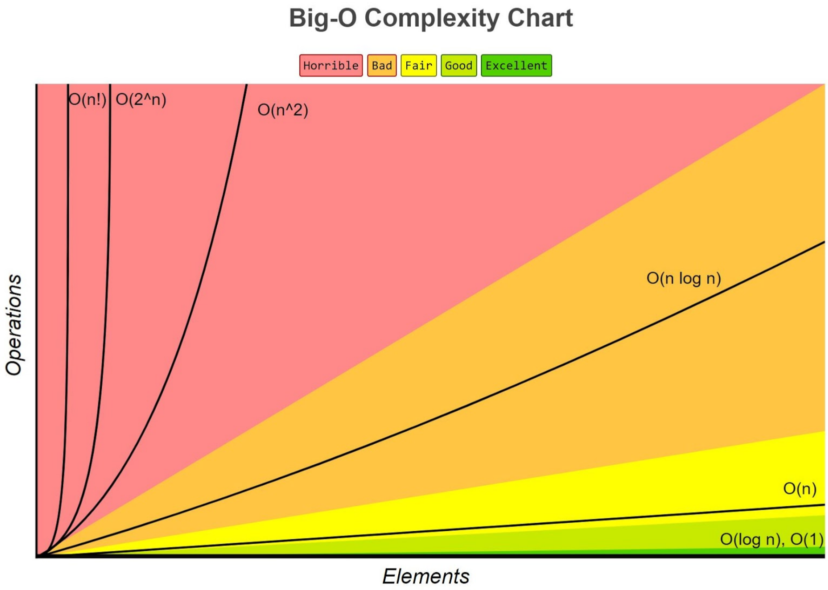
Proof. We proof this by considering two cases:

1. x is an integer: obvious
2. $\exists n \in \mathbb{Z}$ such that $n < x < n + 1$, definition of rounding function $\Rightarrow \lfloor x \rfloor = n$, $\lceil x \rceil = n + 1$

Proposition 2. Let $n, a, b \in \mathbb{Z}$, we have:

$$\lfloor x + n \rfloor = \lfloor x \rfloor + n, \lceil x + n \rceil = \lceil x \rceil + n$$
$$\left\lceil \frac{n}{2} \right\rceil + \left\lfloor \frac{n}{2} \right\rfloor = n$$
$$\left\lceil \frac{\lceil \frac{n}{a} \rceil}{b} \right\rceil = \left\lceil \frac{n}{ab} \right\rceil, \left\lfloor \frac{\lfloor \frac{n}{a} \rfloor}{b} \right\rfloor = \left\lfloor \frac{n}{ab} \right\rfloor$$

Running Ti



Big-O Complexity Chart
Horrible
Bad
Fair
Good
Excellent
O(n!)
O(2^n)
O(n^2)
O(n log n)
O(n)
O(log n), O(1)
Operations
Elements

Outline

Common Running Times (1/4)

Constant time. $T(n) = O(1)$

- Determine if a binary number is even or odd
- Random access of array $A[i]$ or hash map (key-value) access

Common Running Times (1/4)

Constant time. $T(n) = O(1)$

- Determine if a binary number is even or odd
- Random access of array $A[i]$ or hash map (key-value) access

Logarithmic time. $T(n) = O(\log n)$

- Search in a sorted array of size n: binary search

Common Running Times (1/4)

Constant time. $T(n) = O(1)$

- Determine if a binary number is even or odd
- Random access of array $A[i]$ or hash map (key-value) access

Logarithmic time. $T(n) = O(\log n)$

- Search in a sorted array of size n: binary search

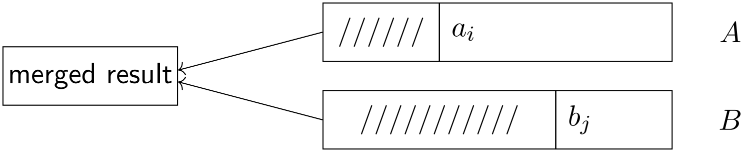
Fractional power. $T(n) = n^{1/2}$

- Primality test

Common Running Times (2/4)

Linear time. $T(n) = O(n)$: running time is proportional to input size

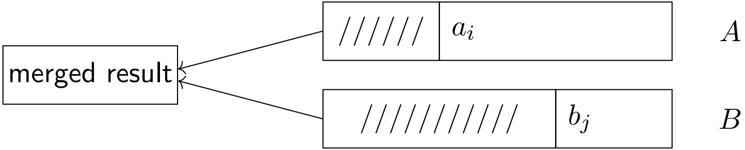
- Merge: combine two sorted lists $A = a_1, \ldots, a_n$ with $B = b_1, \ldots, b_n$ into sorted whole



Common Running Times (2/4)

Linear time. $T(n) = O(n)$: running time is proportional to input size

- Merge: combine two sorted lists $A = a_1, \ldots, a_n$ with $B = b_1, \ldots, b_n$ into sorted whole



After each compare, the length of output list increases by at least 1. When one list is empty, the rest part of another list is directly merged to the result list.

- Upper bound: $2n - 1$ vs. Lower bound: n

Common Running Times (3/4)

Loglinear time. $T(n) = O(n \log n)$ (arises in divide-and-conquer algorithms)

- Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ compares
- FFT

Common Running Times (3/4)

Loglinear time. $T(n) = O(n \log n)$ (arises in divide-and-conquer algorithms)

- Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ compares
- FFT

Quadratic time. $T(n) = O(n^2)$

- Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest. $O(n^2)$ solution: try all pairs of points

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion.

Common Running Times (3/4)

Loglinear time. $T(n) = O(n \log n)$ (arises in divide-and-conquer algorithms)

- Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ compares
- FFT

Quadratic time. $T(n) = O(n^2)$

- Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest. $O(n^2)$ solution: try all pairs of points

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion.

Cubic time. Enumerate all triples of elements

- Plain Matrix multiplication: $\mathbf{A}_{n\times n} \times \mathbf{B}_{n\times n}$: each $c_{i,j}$ requires $O(n)$ multiplications, totally n^2 elements in $\mathbf{C}_{n\times n}$

Common Running Times (4/4)

Polynomial time. $T(n) = O(n^k)$

- Independent set of size k: Given a graph of n nodes, are there k nodes such that no two are joined by an edge?

- enumerate all subsets of k nodes then check
  - check if S_k is an independent set takes $O(k^2)$ time
  - $\#(S_k) = C_n^k \leq n^k/k!$
- $O(k^2 n^k/k!) = O(n^k)$ (poly-time for $k = 17$, but not practical)

Common Running Times (4/4)

Polynomial time. $T(n) = O(n^k)$

- Independent set of size k: Given a graph of n nodes, are there k nodes such that no two are joined by an edge?

- enumerate all subsets of k nodes then check
  - check if S_k is an independent set takes $O(k^2)$ time
  - $\#(S_k) = C_n^k \leq n^k/k!$
- $O(k^2 n^k/k!) = O(n^k)$ (poly-time for $k = 17$, but not practical)

Exponential time. $T(n) = O(c^n)$

- Independent set: Given a graph, what is the maximum cardinality of an independent set?
- Enumerate all subsets and check: $O(n^2 2^n)$

About Polynomial Running Time

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor c.

About Polynomial Running Time

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor c.

Polynomial running time satisfies the above scaling property

- $T(n) = O(n^d) \leftarrow$ choose $c = 2^d$

About Polynomial Running Time

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor c.

Polynomial running time satisfies the above scaling property

- $T(n) = O(n^d) \leftarrow$ choose $c = 2^d$

We say that an algorithm is efficient if has a polynomial running time.

About Polynomial Running Time

Desirable scaling property. When the input size doubles, the algorithm should only slow down by some constant factor c.

Polynomial running time satisfies the above scaling property

- $T(n) = O(n^d) \leftarrow$ choose $c = 2^d$

We say that an algorithm is efficient if has a polynomial running time.

Exceptions. Some poly-time algorithms do have high constants and/or exponents, and/or $\rightsquigarrow$ useless in practice.

Question. Which would you prefer $20n^{100}$ vs. $n^{1+0.02 \ln n}$

Summary of This Lecture (1/2)

Introduce abstract definition of algorithm

How to capture algorithm's complexity?

- First simplification: functions that express number of basic computer steps of input size,

How to compare functions?

- Second simplification: Big-O notations (five standard asymptotic notations) capture order of functions. We study the definitions, typical usages, examples, properties

---

Big-O notations lets us focus on the big picture.

- Helpful analog: $O(\leq)$, $\Omega(\geq)$, $\Theta(=)$, $o(\ll)$, $\omega(\gg)$

Summary of This Lecture (1/2)

Study important running time functions and classical algorithm examples.

Notation abuses. $O(g(n))$ is a set of functions, but computer scientists often write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.

Bottom line. OK to abuse notation; not OK to misuse it.

Don't misunderstand this cavalier attitude towards constants. Programmers are *very* interested in constants and would gladly stay up nights in order to gain 5% efficiency improvement.

Constant Improvement Matters (勿以善小而不为)

Theoretical breakthrough is tooooooooo hard!