Algorithms Design I

Prologue

Guoqiang Li
School of Software, Shanghai Jiao Tong University

SHANGHAI JIAO TONG UNIVERSITY

Instructor

Instructor and Teaching Assistants

SHANGHAI JIAO TONG UNIVERSITY

Guoqiang LI

Instructor and Teaching Assistants

SHANGHAI JIAO TONG UNIVERSITY

Guoqiang LI

- Homepage: https://basics.sjtu.edu.cn/%7Eliguoqiang
- Course page: https://basics.sjtu.edu.cn/%7Eliguoqiang/teaching/SE3352/
- Canvas: https://oc.sjtu.edu.cn/courses/34409
- Email: li.g (AT) outlook (DOT) com
- Office: Rm. 1212, Building of Software
- Phone: 3420-4167

Instructor and Teaching Assistants

SHANGHAI JIAO TONG UNIVERSITY

Guoqiang LI

- Homepage: `https://basics.sjtu.edu.cn/%7Eliguoqiang`
- Course page: `https://basics.sjtu.edu.cn/%7Eliguoqiang/teaching/SE3352/`
- Canvas: https://oc.sjtu.edu.cn/courses/34409
- Email: li.g (AT) outlook (DOT) com
- Office: Rm. 1212, Building of Software
- Phone: 3420-4167

TA:

Instructor and Teaching Assistants

Guoqiang LI

- Homepage: https://basics.sjtu.edu.cn/%7Eliguoqiang
- Course page: https://basics.sjtu.edu.cn/%7Eliguoqiang/teaching/SE3352/
- Canvas: https://oc.sjtu.edu.cn/courses/34409
- Email: li.g (AT) outlook (DOT) com
- Office: Rm. 1212, Building of Software
- Phone: 3420-4167

TA:

- Jingyang LI: 394598772 (AT) qq (DOT) com
- Minyu CHEN: minkowchen (AT) qq (DOT) com

Instructor and Teaching Assistants

SHANGHAI JIAO TONG UNIVERSITY

Guoqiang LI

- Homepage: https://basics.sjtu.edu.cn/%7Eliguoqiang
- Course page: https://basics.sjtu.edu.cn/%7Eliguoqiang/teaching/SE3352/
- Canvas: https://oc.sjtu.edu.cn/courses/34409
- Email: li.g (AT) outlook (DOT) com
- Office: Rm. 1212, Building of Software
- Phone: 3420-4167

TA:

- Jingyang LI: 394598772 (AT) qq (DOT) com
- Minyu CHEN: minkowchen (AT) qq (DOT) com

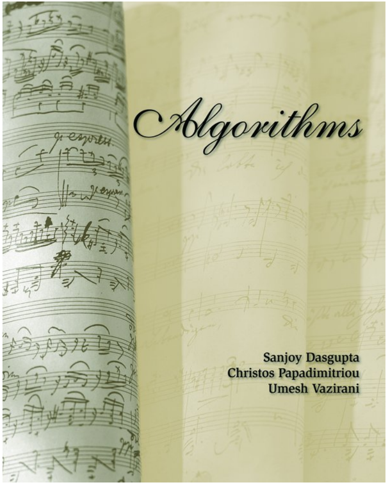
Office hour: Wed. 14:00-17:00 @ Software Building 3203

Reference Book

Textbook

Algorithms

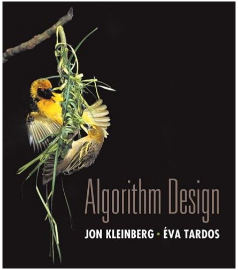
- Sanjoy Dasgupta
- San Diego Christos Papadimitriou
- Umesh Vazirani
- McGraw-Hill, 2007.

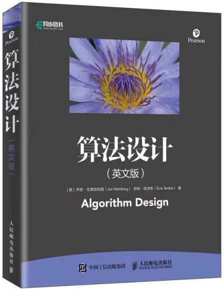


Reference book

Algorithm Design

- Jon Kleinberg, Éva Tardos
- Addison-Wesley, 2005.

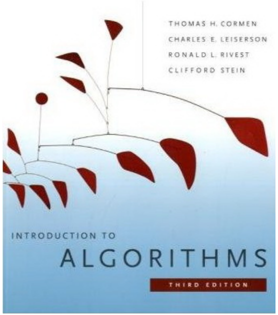




Reference Book

Introduction to Algorithms

- Thomas H. Cormen
- Charles E. Leiserson
- Ronald L. Rivest
- Clifford Stein
- The MIT Press (3rd edition), 2009.



Scoring Policy

30% Homework.

- Six assignments.
- Each one is 5pts.
- Work out individually.
- Each assignment will be evaluated by A, B, C, D, F (Excellent(5), Good(5), Fair(4), Delay(3), Fail(0))

10% Project.

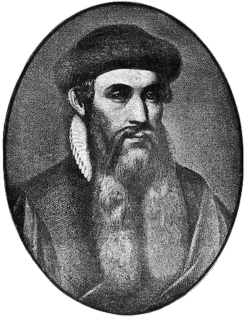
- A comprehensive report.
- To be announced shortly.

60% Final exam.

Any Questions?

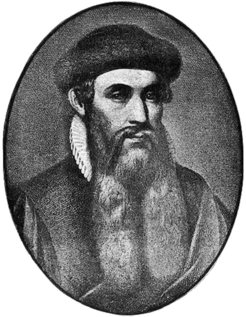
Two Things Change the World

Johann Gutenberg



Johann Gutenberg (1398 - 1468)

Johann Gutenberg



Johann Gutenberg (1398 - 1468)

In 1448 in the German city of Mainz a goldsmith named Johann Gutenberg discovered a way to print books by putting together movable metallic pieces.

Two Ideas Changed the World

Because of the typography, literacy spread, the Dark Ages ended, the human intellect was liberated, science and technology triumphed, the Industrial Revolution happened.

Many historians say we owe all this to typography.

Others insist that the key development was not typography, but algorithms.

Decimal System

Gutenberg would write the number 1448 as $MCDXLVIII$.

Decimal System

Gutenberg would write the number 1448 as $MCDXLVIII$.

How to add two Roman numerals? What is

$$MCDXLVIII + DCCCXII$$

Decimal System

Gutenberg would write the number 1448 as $MCDXLVIII$.

How to add two Roman numerals? What is

$$MCDXLVIII + DCCCXII$$

The decimal system was invented in India around AD 600. Using only 10 symbols, even very large numbers were written down compactly, and arithmetic is done efficiently by elementary steps.

Al Khwarizmi

Al Khwarizmi (780 - 850)

Al Khwarizmi

Al Khwarizmi (780 - 850)

In the 12th century, Latin translations of his work on the Indian numerals, introduced the decimal system to the Western world. (Source: Wikipedia)

Algorithms

Al Khwarizmi laid out the basic methods for

- adding,
- multiplying,
- dividing numbers,
- extracting square roots,
- calculating digits of π.

Algorithms

Al Khwarizmi laid out the basic methods for

- adding,
- multiplying,
- dividing numbers,
- extracting square roots,
- calculating digits of π.

These procedures were precise, unambiguous, mechanical, efficient, correct.

They were algorithms, a term coined to honor the wise man after the decimal system was finally adopted in Europe, many centuries later.

What Is An Algorithm

What Is An Algorithm

A step by step procedure for solving a problem or accomplishing some end.

What Is An Algorithm

A step by step procedure for solving a problem or accomplishing some end.

An abstract recipe, prescribing a process which may be carried out by a human, a computer or by other means.

What Is An Algorithm

A step by step procedure for solving a problem or accomplishing some end.

An abstract recipe, prescribing a process which may be carried out by a human, a computer or by other means.

Any well-defined computational procedure that makes some value, or set of values, as input and produces some value, of set of values, as output. An algorithm is thus a finite sequence of computational steps that transform the input into the output.

What Is An Algorithm

An algorithm is a procedure that consists of

What Is An Algorithm

An algorithm is a procedure that consists of

- a finite set of instructions which,

What Is An Algorithm

An algorithm is a procedure that consists of

- a finite set of instructions which,
- given an input from some set of possible inputs,

What Is An Algorithm

An algorithm is a procedure that consists of

- a finite set of instructions which,
- given an input from some set of possible inputs,
- enables us to obtain an output through a systematic execution of the instructions

What Is An Algorithm

An algorithm is a procedure that consists of

- a finite set of instructions which,
- given an input from some set of possible inputs,
- enables us to obtain an output through a systematic execution of the instructions
- that terminates in a finite number of steps.

What Is An Algorithm

An algorithm is a procedure that consists of

- a finite set of instructions which,
- given an input from some set of possible inputs,
- enables us to obtain an output through a systematic execution of the instructions
- that terminates in a finite number of steps.

A program is

What Is An Algorithm

An algorithm is a procedure that consists of

- a finite set of instructions which,
- given an input from some set of possible inputs,
- enables us to obtain an output through a systematic execution of the instructions
- that terminates in a finite number of steps.

A program is

- an implementation of an algorithm, or algorithms.

What Is An Algorithm

An algorithm is a procedure that consists of

- a finite set of instructions which,
- given an input from some set of possible inputs,
- enables us to obtain an output through a systematic execution of the instructions
- that terminates in a finite number of steps.

A program is

- an implementation of an algorithm, or algorithms.
- A program does not necessarily terminate.

program不一定终止，但是一个算法一定会终止。

Fibonacci Algorithm

Leonardo Fibonacci

Leonardo Fibonacci (1170 - 1250)

Leonardo Fibonacci

Leonardo Fibonacci (1170 - 1250)

Fibonacci helped the spread of the decimal system in Europe, primarily through the publication in the early 13th century of his Book of Calculation, the Liber Abaci. (Source: Wikipedia)

Fibonacci Sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Fibonacci Sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Formally,

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

Fibonacci Sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots$$

Formally,

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

Q: What is F_{100} or F_{200}?

An Exponential Algorithm

SHANGHAI JIAO TONG UNIVERSITY

```
FIBO1(n)
```

a nature number n;

if $n = 0$ **then** `return(`*0*`);`

if $n = 1$ **then** `return(`*1*`);`

`return(`$\texttt{FIBO1}(n-1)$ + $\texttt{FIBO1}(n-2)$`);`

Three Questions about An Algorithm

1. Is it correct?
2. How much time does it take, as a function of n?
3. Can we do better?

Three Questions about An Algorithm

1. Is it correct?
2. How much time does it take, as a function of n?
3. Can we do better?

The first question is trivial, as this algorithm is precisely Fibonacci's definition of F_n

How Much Time

Let $T(n)$ be the number of computer steps needed to compute `FIBO1`(n)

How Much Time

Let $T(n)$ be the number of computer steps needed to compute $\texttt{FIBO1}(n)$

For $n \le 1$,

$$T(n) \le 2$$

How Much Time

Let $T(n)$ be the number of computer steps needed to compute $\texttt{FIBO1}(n)$

For $n \leq 1$,

$$T(n) \leq 2$$

For $n \geq 1$,

$$T(n) = T(n-1) + T(n-2) + 3$$

How Much Time

Let $T(n)$ be the number of computer steps needed to compute $\texttt{FIBO1}(n)$

For $n \leq 1$,

$$T(n) \leq 2$$

For $n \geq 1$,

$$T(n) = T(n-1) + T(n-2) + 3$$

这里的"+3"是因为要先执行两个if(均失败),之后还要对n-1与n-2两种情况做一个额外的加法。这里还认为加法需要常数时间。

It is easy to shown, for all $n \in \mathbb{N}$,

$$T(n) \geq F_n$$

How Much Time

Let $T(n)$ be the number of computer steps needed to compute `FIBO1`(n)

For $n \leq 1$,

$$T(n) \leq 2$$

For $n \geq 1$,

$$T(n) = T(n-1) + T(n-2) + 3$$

It is easy to shown, for all $n \in \mathbb{N}$,

$$T(n) \geq F_n$$

It is exponential to n.

递归树，对于F(n)，其递归树的高度为n-1层，故为2^(n-1)。为啥是n-1层？因为每一次递归都有F(n-1)项，所以会从n一直减到1.

Why Exponential Is Bad?

$$T(200) \geq F_{200} \geq 2^{138} \approx 2.56 \times 10^{42}$$

Why Exponential Is Bad?

$$T(200) \geq F_{200} \geq 2^{138} \approx 2.56 \times 10^{42}$$

In 2010, the fastest computer in the world is the Tianhe-1A system at the National Supercomputer Center in Tianjin.

Why Exponential Is Bad?

$$T(200) \geq F_{200} \geq 2^{138} \approx 2.56 \times 10^{42}$$

In 2010, the fastest computer in the world is the Tianhe-1A system at the National Supercomputer Center in Tianjin.

Its speed is

$$2.57 \times 10^{15}$$

steps per second.

Why Exponential Is Bad?

$$T(200) \geq F_{200} \geq 2^{138} \approx 2.56 \times 10^{42}$$

In 2010, the fastest computer in the world is the Tianhe-1A system at the National Supercomputer Center in Tianjin.

Its speed is

$$2.57 \times 10^{15}$$

steps per second.

Thus to compute F_{200} Tianhe-1A needs roughly

$$10^{27} \text{ seconds} \geq 10^{22} \text{ years.}$$

Why Exponential Is Bad?

$$T(200) \geq F_{200} \geq 2^{138} \approx 2.56 \times 10^{42}$$

In 2010, the fastest computer in the world is the Tianhe-1A system at the National Supercomputer Center in Tianjin.

Its speed is

$$2.57 \times 10^{15}$$

steps per second.

Thus to compute F_{200} Tianhe-1A needs roughly

$$10^{27} \text{ seconds} \geq 10^{22} \text{ years.}$$

In 2022, the fastest is Frontier, 1.102×10^{18} per second.

Moore's Law

Moore's Law:

Computer speeds have been doubling roughly every 18 months.

Moore’s Law

Moore’s Law:

Computer speeds have been doubling roughly every 18 months.

The running time of `FIB01` is proportional to

$$2^{0.694n} \approx 1.6^n$$

Thus, it takes 1.6 times longer to compute F_{n+1} than F_n.

Moore's Law

Moore's Law:

Computer speeds have been doubling roughly every 18 months.

The running time of `FIB01` is proportional to

$$2^{0.694n} \approx 1.6^n$$

Thus, it takes 1.6 times longer to compute F_{n+1} than F_n.

So if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}, and so on . . .

Moore's Law

Moore's Law:

Computer speeds have been doubling roughly every 18 months.

The running time of `FIBO1` is proportional to

$$2^{0.694n} \approx 1.6^n$$

Thus, it takes 1.6 times longer to compute F_{n+1} than F_n.

So if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}, and so on . . .

Just one more number every year!

上述里说明了，一个算法不光需要正确性，还需要一个合适的时间开销！
注：一般而言，多项式时间的算法计算机都是可以接受的，只不过有很多多项式时间算法都是可以优化的。

Moore's Law

Moore's Law:

Computer speeds have been doubling roughly every 18 months.

The running time of `FIB01` is proportional to

$$2^{0.694n} \approx 1.6^n$$

Thus, it takes 1.6 times longer to compute F_{n+1} than F_n.

So if we can reasonably compute F_{100} with this year's technology, then next year we will manage F_{101}, and so on ...

Just one more number every year!

Such is the curse of exponential time.

Three Questions

1. Is it correct?
2. How much time does it take, as a function of n?
3. Can we do better?

Three Questions

1. Is it correct?
2. How much time does it take, as a function of n?
3. Can we do better?

Now we know $\texttt{FIB1}(n)$ is correct and inefficient, so can we do better?

An Polynomial Algorithm


```
FIBO2(n)
a nature number n;

if n = 0 then return(0);
create an array f[0...n];
f[0] = 0; f[1] = 1;
for i = 2 to n do
    f[i] = f[i-1] + f[i-2];
end
return(f[n]);
```

在认为加法花费常数项时间的情况下，该优化之后的算法时间复杂度为O(n)。

动态规划！

An Analysis

The correctness of `FIBO2` is trivial.

An Analysis

The correctness of `FIBO2` is trivial.

How long does it take?

An Analysis

SHANGHAI JIAO TONG UNIVERSITY

The correctness of FIBO2 is trivial.

How long does it take?

The inner loop consists of a single computer step and is executed $n - 1$ times. Therefore the number of computer steps used by FIBO2 is linear in n.

A More Careful Analysis

We count the number of basic computer steps executed by each algorithm and regard these basic steps as taking a constant amount of time.

A More Careful Analysis

We count the number of basic computer steps executed by each algorithm and regard these basic steps as taking a constant amount of time.

It is reasonable to treat addition as a single computer step if small numbers are being added, e.g., 32-bit numbers.

A More Careful Analysis

We count the number of basic computer steps executed by each algorithm and regard these basic steps as taking a constant amount of time.

It is reasonable to treat addition as a single computer step if small numbers are being added, e.g., 32-bit numbers.

The n-th Fibonacci number is about $0.694n$ bits long, and this can far exceed 32 as n grows.

A More Careful Analysis

We count the number of basic computer steps executed by each algorithm and regard these basic steps as taking a constant amount of time.

It is reasonable to treat addition as a single computer step if small numbers are being added, e.g., 32-bit numbers.

The n-th Fibonacci number is about $0.694n$ bits long, and this can far exceed 32 as n grows.

Arithmetic operations on arbitrarily large numbers cannot possibly be performed in a single, constant-time step.

A More Careful Analysis

Shanghai Jiao Tong University

The addition of two n-bit numbers takes time roughly proportional to n (next lecture).

与n成正比

A More Careful Analysis

The addition of two n-bit numbers takes time roughly proportional to n (next lecture).

`FIB01`, which performs about F_n additions, uses a number of basic step roughly proportional to nF_n.

A More Careful Analysis

The addition of two n-bit numbers takes time roughly proportional to n (next lecture).

`FIBO1`, which performs about F_n additions, uses a number of basic step roughly proportional to nF_n.

The number of steps taken by `FIBO2` is proportional to n^2, and still polynomial in n.

多项式的

A More Careful Analysis

The addition of two n-bit numbers takes time roughly proportional to n (next lecture).

`FIB01`, which performs about F_n additions, uses a number of basic step roughly proportional to nF_n.

The number of steps taken by `FIB02` is proportional to n^2, and still polynomial in n.

Q: Can we do better?

A More Careful Analysis

The addition of two n-bit numbers takes time roughly proportional to n (next lecture).

`FIB01`, which performs about F_n additions, uses a number of basic step roughly proportional to nF_n.

The number of steps taken by `FIB02` is proportional to n^2, and still polynomial in n.

Q: Can we do better?

- Exercise 0.4

Big-O Notation

Counting the Number of Steps

We see how sloppiness in the analysis of running times can lead to unacceptable inaccuracy.

Counting the Number of Steps

We see how sloppiness in the analysis of running times can lead to unacceptable inaccuracy.

It is also possible to be too precise to be useful.

Counting the Number of Steps

We see how sloppiness in the analysis of running times can lead to unacceptable inaccuracy.

It is also possible to be too precise to be useful.

Expressing running time in terms of basic computer steps is already a simplification. The time taken by one such step depends crucially on the particular processor, etc.

Counting the Number of Steps

We see how sloppiness in the analysis of running times can lead to unacceptable inaccuracy.

It is also possible to be too precise to be useful.

Expressing running time in terms of basic computer steps is already a simplification. The time taken by one such step depends crucially on the particular processor, etc.

Accounting for these architecture-specific details is too complicated and yields a result that does not generalize from one computer to the next.

Counting the Number of Steps

It makes more sense to seek a machine independent characterization of an algorithm's efficiency.

Counting the Number of Steps

It makes more sense to seek a machine independent characterization of an algorithm's efficiency.

We always express running time by counting the number of basic computer steps, as a function of the size of the input.

Counting the Number of Steps

It makes more sense to seek a machine independent characterization of an algorithm's efficiency.

We always express running time by counting the number of basic computer steps, as a function of the size of the input.

Instead of reporting that an algorithm takes, say, $7n^3 + 4n + 1$ steps on an input of size n, it is much simpler to leave out lower-order terms such as $4n$ and 1.

Counting the Number of Steps

It makes more sense to seek a machine independent characterization of an algorithm's efficiency.

We always express running time by counting the number of basic computer steps, as a function of the size of the input.

Instead of reporting that an algorithm takes, say, $7n^3 + 4n + 1$ steps on an input of size n, it is much simpler to leave out lower-order terms such as $4n$ and 1.

The Coefficient 7 in the leading term is also left out, and just say that the algorithm takes time $O(n^3)$ (pronounced big oh of n^3).

Big-O Notation

$f(n)$ and $g(n)$ are the running times of two algorithms on inputs of size n.

Big-O Notation

$f(n)$ and $g(n)$ are the running times of two algorithms on inputs of size n.

- Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals.

Big-O Notation

$f(n)$ and $g(n)$ are the running times of two algorithms on inputs of size n.

- Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals.
- $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.

Big-O Notation

大0表示法

$f(n)$ and $g(n)$ are the running times of two algorithms on inputs of size n.

- Let $f(n)$ and $g(n)$ be functions from positive integers to positive reals.
- $f = O(g)$ if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.

$f = O(g)$ is very loose analog of "$f \leq g$". It differs from the usual notion of $\leq$ because of the constant c, so that for instance $10n = O(n)$.

Why Disregard the Constant?

SHANGHAI JIAO TONG UNIVERSITY

We are choosing between two algorithms: One takes $f_1(n) = n^2$ steps, while the other takes $f_2(n) = 2n + 20$ steps.

Why Disregard the Constant?

We are choosing between two algorithms: One takes $f_1(n) = n^2$ steps, while the other takes $f_2(n) = 2n + 20$ steps.

Which is better?

Why Disregard the Constant?

We are choosing between two algorithms: One takes $f_1(n) = n^2$ steps, while the other takes $f_2(n) = 2n + 20$ steps.

Which is better?

The answer depends on n:

Why Disregard the Constant?

We are choosing between two algorithms: One takes $f_1(n) = n^2$ steps, while the other takes $f_2(n) = 2n + 20$ steps.

Which is better?

The answer depends on n:

- If $n \leq 5$, then $f_1(n) \leq f_2(n)$.
- If $n > 5$, then $f_1(n) > f_2(n)$.

Why Disregard the Constant?

SHANGHAI JIAO TONG UNIVERSITY

We are choosing between two algorithms: One takes $f_1(n) = n^2$ steps, while the other takes $f_2(n) = 2n + 20$ steps.

Which is better?

The answer depends on n:

- If $n \leq 5$, then $f_1(n) \leq f_2(n)$.
- If $n > 5$, then $f_1(n) > f_2(n)$.

f_2 scales much better as n grows, and therefore it is superior.

Why Disregard the Constant?

This superiority is captured by the big-O notion: $f_2 = O(f_1)$.

$$\frac{f_2(n)}{f_1(n)} = \frac{2n+20}{n^2} \leq 22$$

for all $n \in \mathbb{N}$.

Why Disregard the Constant?

This superiority is captured by the big-O notion: $f_2 = O(f_1)$.

$$\frac{f_2(n)}{f_1(n)} = \frac{2n+20}{n^2} \leq 22$$

for all $n \in \mathbb{N}$.

On the other hand, $f_1 \neq O(f_2)$, since the ratio

$$\frac{f_1(n)}{f_2(n)} = \frac{n^2}{2n+20}$$

can get arbitrarily large.

Why Disregard the Constant?

Recall $f_1(n) = n^2$ and $f_2(n) = 2n + 20$, suppose we have a third algorithm which uses $f_3(n) = n + 1$ steps.

Why Disregard the Constant?

Recall $f_1(n) = n^2$ and $f_2(n) = 2n + 20$, suppose we have a third algorithm which uses $f_3(n) = n + 1$ steps.

Is this better than f_2?

Why Disregard the Constant?

Recall $f_1(n) = n^2$ and $f_2(n) = 2n + 20$, suppose we have a third algorithm which uses $f_3(n) = n + 1$ steps.

Is this better than f_2?

Certainly, but only by a constant factor.

Why Disregard the Constant?

Recall $f_1(n) = n^2$ and $f_2(n) = 2n + 20$, suppose we have a third algorithm which uses $f_3(n) = n + 1$ steps.

Is this better than f_2?

Certainly, but only by a constant factor.

The discrepancy between f_2 and f_3 is tiny compared to the huge gap between f_1 and f_2.

Why Disregard the Constant?

Recall $f_1(n) = n^2$ and $f_2(n) = 2n + 20$, suppose we have a third algorithm which uses $f_3(n) = n + 1$ steps.

Is this better than f_2?

Certainly, but only by a constant factor.

The discrepancy between f_2 and f_3 is tiny compared to the huge gap between f_1 and f_2.

In order to stay focused on the big picture, we treat functions as equivalent if they differ only by multiplicative constants.

Why Disregard the Constant?

Recall $f_1(n) = n^2$ and $f_2(n) = 2n + 20$, suppose we have a third algorithm which uses $f_3(n) = n + 1$ steps.

Is this better than f_2?

Certainly, but only by a constant factor.

The discrepancy between f_2 and f_3 is tiny compared to the huge gap between f_1 and f_2.

In order to stay focused on the big picture, we treat functions as equivalent if they differ only by multiplicative constants.

$f_2 = O(f_3)$ and $f_3 = O(f_2)$.

Other Similar Notations

Just as $O(\cdot)$ is an analog of $\leq$, we also define analogs of $\geq$ and $=$ as follows,

- $f = \Omega(g)$ means $g = O(f)$.
- $f = \Theta(g)$ means $f = \Omega(g)$ and $f = O(g)$.

Other Similar Notations

Just as $O(\cdot)$ is an analog of $\leq$, we also define analogs of $\geq$ and $=$ as follows,

- $f = \Omega(g)$ means $g = O(f)$.
- $f = \Theta(g)$ means $f = \Omega(g)$ and $f = O(g)$.

Recall $f_1(n) = n^2$, $f_2(n) = 2n + 20$, and $f_3(n) = n + 1$, then

- $f_2 = \Theta(f_3)$ and $f_1 = \Omega(f_2)$

老师上课的思考题：为什么归并排序的时间复杂度一定要是O(NlogN),而不能写为其O的O(N^2)之类的呢？因为NlogN的O实际上应该是θ，但是一般我们只关注平均性能，所以一般使用O就行了。因为对于一个算法而言，其是可以存在一个最好情况和最坏情况的。但是当N足够大时，平均性能更有意义。

Some Simple Rules

Multiplicative constants can be omitted: $14n^2$ becomes n^2.

n^a dominates n^b if $a > b$, for instance, n^2 dominates n

a^n dominates b^n if $a > b$, for instance, 3^n dominates 2^n

Any exponential dominates any polynomial: 3^n dominates n^5

Any polynomial dominates any logarithm: n dominates $(\log n)^3$.

Some Simple Rules

Shanghai Jiao Tong University

Multiplicative constants can be omitted: $14n^2$ becomes n^2.

n^a dominates n^b if $a > b$, for instance, n^2 dominates n

a^n dominates b^n if $a > b$, for instance, 3^n dominates 2^n

Any exponential dominates any polynomial: 3^n dominates n^5

Any polynomial dominates any logarithm: n dominates $(\log n)^3$. This also means, for example, n^2 dominates $n \log n$.