上海科技大学
ShanghaiTech University

CS240 Algorithm Design and Analysis

Lecture 0

Introduction and Overview

Quan Li

Fall 2025
2025.09.16

Organization

Instructor

- Quan Li (李权)
- Tel: (021) 20684425
- Office: Room 1-404B, SIST Building
- Email: liquan@shanghaitech.edu.cn
- Homepage:
- http://faculty.sist.shanghaitech.edu.cn/liquan/

TAs

- 邵煜恒 shaoyh2024@shanghaitech.edu.cn
- 史少寒 shishh2023@shanghaitech.edu.cn
- 熊俊杰 xiongjj2025@shanghaitech.edu.cn
- 姜浩然 jianghr2023@shanghaitech.edu.cn

Organization

上海科技大学 ShanghaiTech University

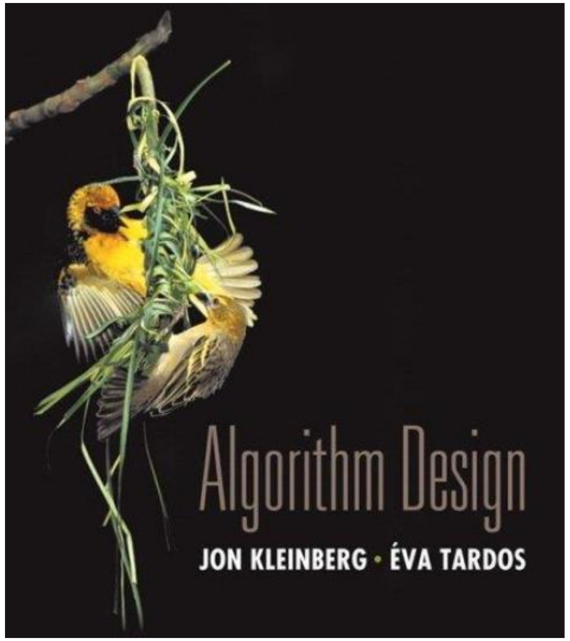
Prerequisites

- **Algorithm and Data Structure (Undergraduate course)**
  - Sorting and searching, divide & conquer, greedy, dynamic programming, graph algorithms
  - Analysis of algorithms
- **Basic discrete mathematics**
  - Recurrences, logic and proofs, basic graph theory
- **Basic probability theory**
  - Probability space, random variables, expectation, variance
- **Computer programming**
  - Doesn't matter which language(s) you know
  - But you should be capable of translating high-level algorithm descriptions into working programs in some programming language

立志成才报国裕民

Organization

- **Textbook**
  - [KT] Algorithm Design, by Jon Kleinberg and Eva Tardos.
  - [CLRS] Introduction to Algorithms (3rd edition), by T. Cormen, C. Leiserson, R. Rivest, and C. Stein.
  - [V] Approximation Algorithms, by Vijay V. Vazirani.
  - [MR] Randomized Algorithms, by Rajeev Motwani and Prabhakar Raghavan.
- **Piazza** (https://piazza.com/shanghaitech.edu.cn/fall2025/cs240) **(Please JOIN as students!)**
  - Lecture slides, announcements, homework assignment, QA and discussions, etc.
- **Gradescope**
  - Homework submission and grading
- **Academic Integrity**
  - Unless explicitly noted, work turned in should reflect your own/independent capabilities
  - No cheating (We will check carefully!)
    - Don't share your homework/code!
    - No fake solutions!
    - No plagiarism!
    - Serious consequences!



Organization

- Grading
  - Assignments (20%)
  - Midterm (35%)
  - Final (35%)
  - Course Project (10%)
  - Exams will be open-book with only one A4 cheating cheet

上海科技大学2025-2026学年校历

	八月		九月				十月					十一月				十二月				
星期一	18	25	1	8	15	22	29	6 中秋节	13	20	27	3	10	17	24	1	8	15	22	29
星期二	19	26	2	9	16	23	30	7	14	21	28	4	11	18	25	2	9	16	23	30
星期三	20	27	3	10	17	24	1 国庆节	8	15	22	29	5	12	19	26	3	10	17	24	31
星期四	21	28	4	11	18	25	2	9	16	23	30	6	13	20	27	4	11	18	25	1 元旦
星期五	22	29	5	12	19	26	3	10	17	24	31	7	14	21	28	5	12	19	26	2
星期六	23	30	6	13	20	27	4	11	18	25	1	8	15	22	29	6	13	20	27	3
星期日	24	31	7	14	21	28	5	12	19	26	2	9	16	23	30	7	14	21	28	4
周数	5	6	7	8	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
学期	暑假				秋学期															

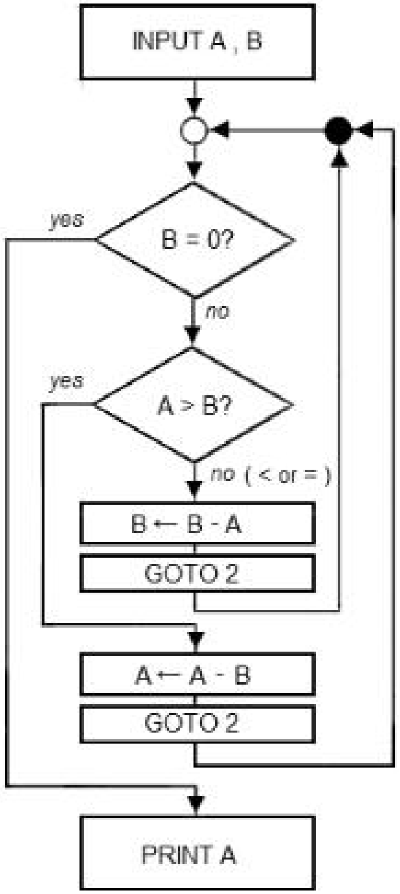
秋学期(2025.9.15-2026.1.18)

一. 9月14日老生注册,9月15日本科生、研究生上课
二. 17、18周本科生、研究生考试
三. 1月19日-2月28日放寒假

Dates for homework assignments, midterm, and course project will be announced in due course~

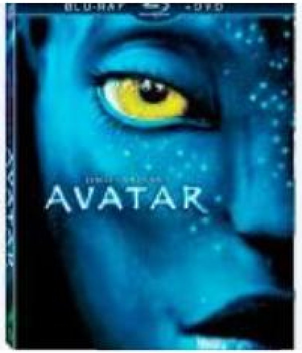
Algorithms

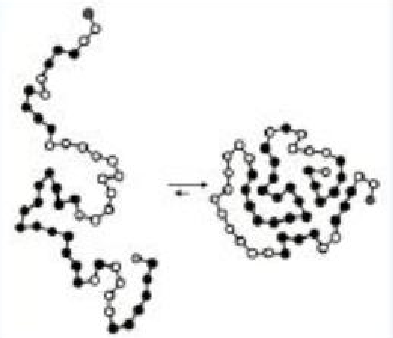
- [Knuth, TAOCP] An algorithm is a finite, definite, effective procedure, with some input and some output
- [Wikipedia] An algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation
- Important for all other branches of computer science
- Plays a key role in modern technological innovation
- Provides novel "lens" on processes outside of computer science and technology
    - Internet: Web search, packet routing, distributed file sharing, …
    - Biology: Human genome project, protein folding, …
    - Computers: Circuit layout, databases, caching, networking, compilers, …
    - Computer graphics: Movies, video games, virtual reality, …
    - Security: Cell phones, e-commerce, voting machines, federated learning, …
    - Multimedia: MP3, JPG, DivX, HDTV, face recognition, …
    - Social networks: Recommendations, news feeds, advertisements, …
    - Physics. N-body simulation, particle collision simulation, …



Why Study Algorithms?

- Wide range of applications
  - Internet. Web search, packet routing, distributed file sharing, …
  - Biology. Human genome project, protein folding, …
  - Computers. Circuit layout, databases, caching, networking, compilers, …
  - Computer graphics. Movies, video games, virtual reality, …
  - Security. Cell phones, e-commerce, voting machines, …
  - Multimedia. MP3, JPG, DivX, HDTV, face recognition, …
  - Social networks. Recommendations, news feeds, advertisements, …
  - Physics. N-body simulation, particle collision simulation, …



Typical Undergraduate Algorithm Course

Understanding and implementing classic algorithms

- Sorting
- Searching
- String algorithms
- Graph algorithms

Critical thinking, problem-solving, coding

This Course

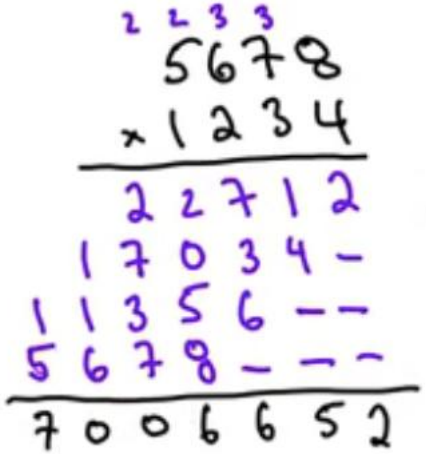
Design and analysis of computer algorithms

- Vocabulary for design and analysis of algorithms
- Greedy algorithms
- Divide-and-conquer
- Dynamic programming
- Network flow
- Intractability (complexity classes)
- Amortized analysis
- Approximation algorithms
- Randomized algorithms
- Local search

Critical thinking, problem-solving, rigorous analysis

Integer Multiplication

- Input: two n-digit numbers x and y
- Output: the product x · y
- "Primitive operation": add or multiply two single-digit numbers
- The grade-school algorithm: 5678 * 1234 = 7006652
- **2n** operations per row and there are **n** rows
- Upshot: #operations overall <= constant · n^2



- The Algorithm Designers' Mantra

"*Perhaps the most important principle for the good algorithm designers is to refuse to be content.*" – Aho, Hopcroft, and Ullman, The Design and Analysis of Computer Algorithms, 1974

- **CAN WE DO BETTER?**

Karatsuba Multiplication

- Example: $x = 5678$ $y = 1234$ to compute the product $x \cdot y$
- Assume $56 = a$, $78 = b$, $12 = c$, $34 = d$
- Step 1: Compute $a \cdot c = 672$
- Step 2: Compute $b \cdot d = 2652$
- Step 3: Compute $(a+b) \cdot (c+d) = 134 \cdot 46 = 6164$
- Step 4: Compute Step3 − step2 − step1 = 2840

$$\begin{array}{r} 6720000 \\ 2652 \\ 284000 \\ \hline 7006652 \end{array}$$

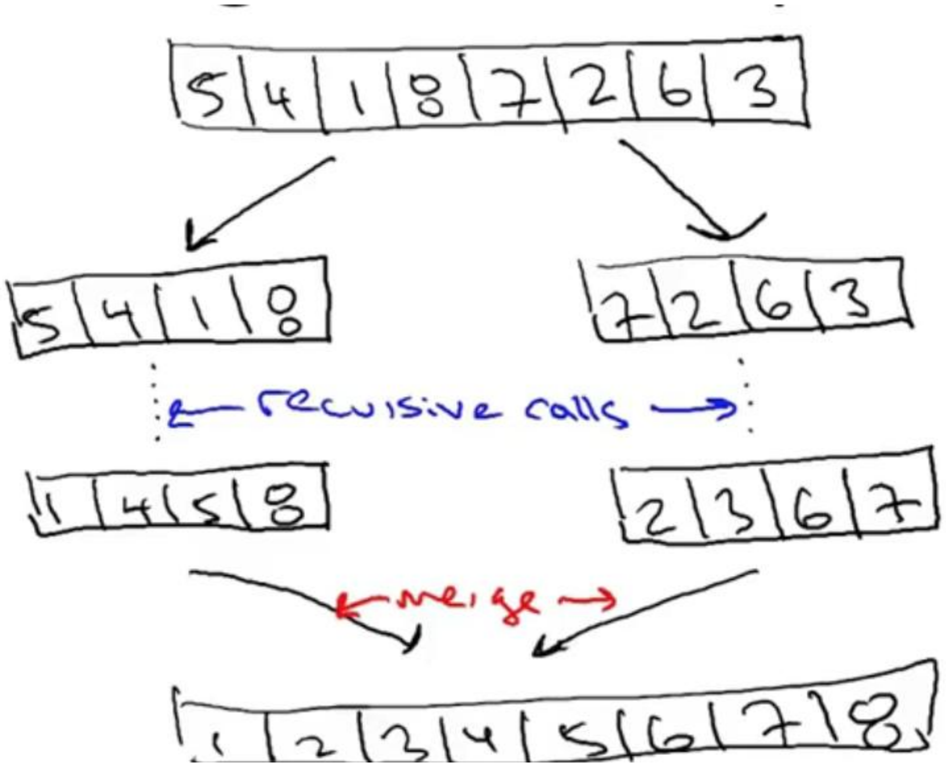
A Recursive Algorithm

- Write $x = 10^{n/2}a + b$ and $y = 10^{n/2}c + d$ where a, b, c, d are n/2-digit numbers
- [Example: a = 56, b = 78, c = 12, d = 34]
- Then: $x \cdot y = (10^{n/2}a + b) \cdot (10^{n/2}c + d) = 10^{n}ac + 10^{n/2}(ad + bc) + bd$
- Idea; recursively compute ac, ad, bc, bd, then compute the above equation in the straightforward way
- Simple base case omitted (if input is very small, get the result immediately)

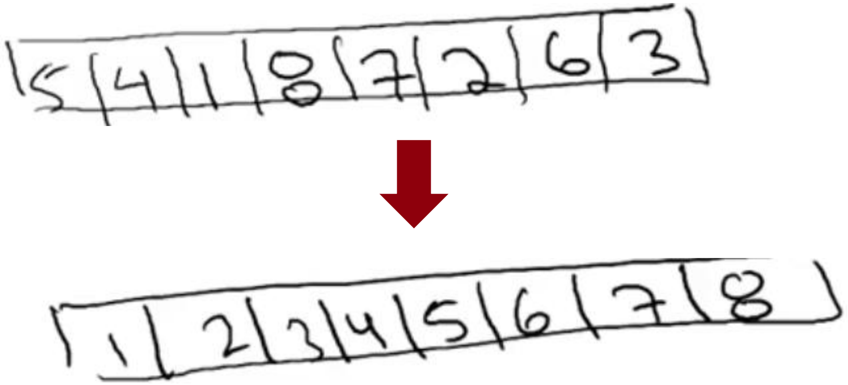
- **Karatsuba Multiplication**
  - Recall: $x \cdot y = 10^{n}ac + 10^{n/2}(ad + bc) + bd$ (seems having 4 recursive multiplications…)
  - Step 1: recursively compute ac
  - Step 2: recursively compute bd
  - Step 3: recursively compute (a+b)(c+d) = ac+ad+bc+bd
  - **Gaoss's trick: step3 – step1 – step2 = ad + bc**
  - Upshot: only need 3 recursive multiplications and some additions

Why study merge sort?

- Good introduction to divide & conquer
  - Improves over selection, insertion, bubble sorts
- Motivates guiding principles for algorithm analysis (worst-case and asymptotic analysis)
- Analysis generalizes to "Master Method"



- The Sorting Problem
- Input: array of n numbers, unsorted
- Output: Same numbers, sorted in increasing order



Merge Sort: Pseudocode

- Recursively sort 1st half of input array
- Recursively sort 2nd half of input array
- Merge two sorted sublists into one

- Pseudocode for Merge:

C = output array [length = n]

A = 1st sorted array [n/2]

B = 2nd sorted array [n/2]

i = 1

j = 1

```
for k  =  1 to n
    if A[i] < B[j]
        C[k] = A[i]
        i++
    else B[j] < A[i]
        C[k] = B[j]
        j++
End
(ignores end cases)
```

Merge Sort Running Time?

- **Key question:** running the MergesSort on array of n numbers?
- Running time ≈ # of lines of code executed

- Pseudocode for Merge:

C = output array [length = n]

A = 1st sorted array [n/2]

B = 2nd sorted array [n/2]

i = 1, j = 1

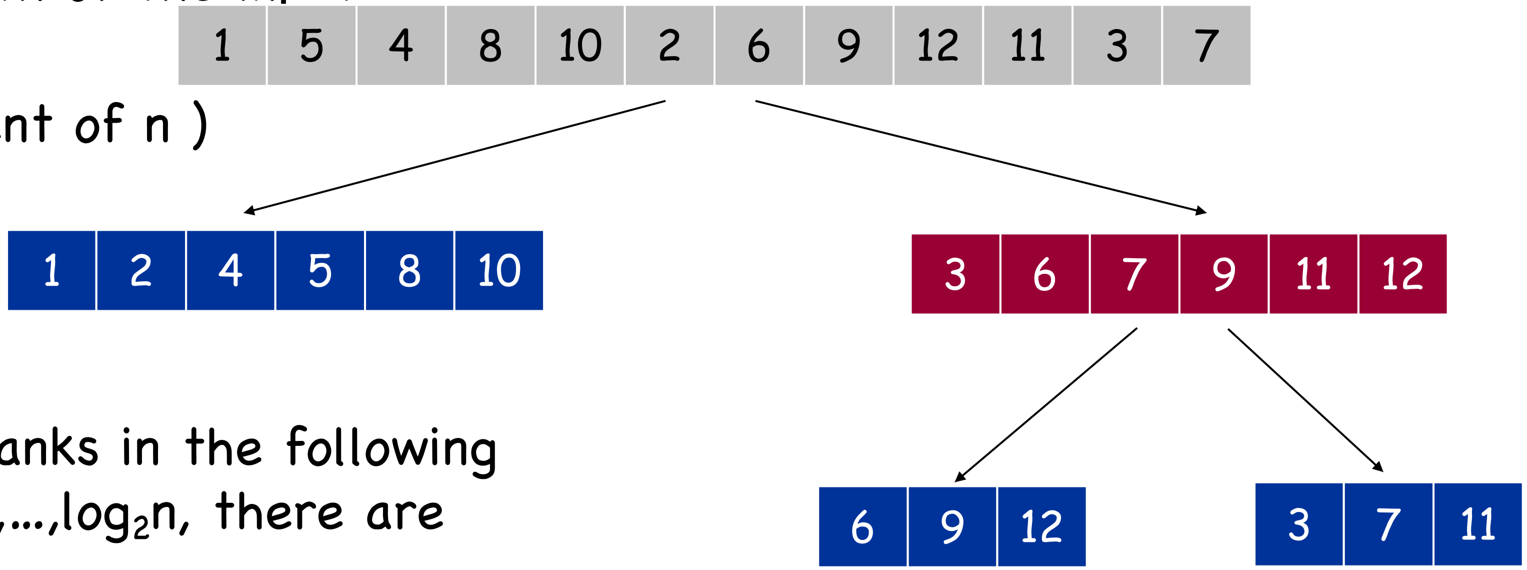
2 operations

```
for k  =  1 to n
      if A[i] < B[j]
            C[k] = A[i]
            i++
      else B[j] < A[i]
            C[k] = B[j]
            j++
End
(ignores end cases)
```

- **Upshot:** running time of merge on array of n numbers is <= 4n + 2 <= 6n (since n >= 1)
- **Claim:** MergeSort requires <= 6nlogn + 6n operations to sort n numbers
- **Recall:** logn = # d times you divide by 2 until you get down to 1

Proof of claim (assuming n = power of 2)

- Will use “recursion tree”
- Q: Roughly how many levels does this recursion tree have (as a function of n, the length of the input array)?
  - A constant number (independent of n)
  - $\sqrt{n}$
  - $\log_2 n$
  - n
- Q: What is the pattern? Fill in blanks in the following statement: at each level $j = 0,1,2,\ldots,\log_2 n$, there are ___ subproblems, each of size ____.
  - 2^j and 2^j, respectively
  - $n/2^j$ and $n/2^j$, respectively
  - 2^j and $n/2^j$, respectively
  - $n/2^j$ and 2^j, respectively



Proof of claim (assuming n = power of 2)

- At each level $j = 0,1,2,\ldots,\log_2 n$, there are 2^j subproblems, each of size $n/2^j$
- Total # of operations at level j:

[each $j=0,1,2,\ldots,\log_2 n$]

$<= 2^j * 6(n/2^j) = 6n$ ← independent of j

- Total $<= 6n(\log_2 n+1)$

- **Claim:** For every input array of n numbers, MergeSort produces a sorted output array and uses at most $6n\log_2 n+6n$ operations

Guiding Principles

- **Guiding Principle 1**
  - "worst-case analysis" : running time bound holds for every input of length n
  - Particularly appropriate for "general-purpose" routines
  - As opposed to "average-case" analysis and benchmarks (requires domain knowledge)
  - Worst case usually easier to analyze
- **Guiding Principle 2**
  - Won't pay much attention to constant factors, lower-order terms
  - Way easier
  - Constants depend on architecture/compiler/programmer anyways
  - Lose very little predictive power
- **Guiding Principle 3**
  - Asymptotic analysis: focus on running time for large input size n
  - E.g., $6n\log_2 n + 6n$ "better than" $\frac{1}{2} n^2$ (e.g., insertion sort)
  - Justification: any big problems are interesting

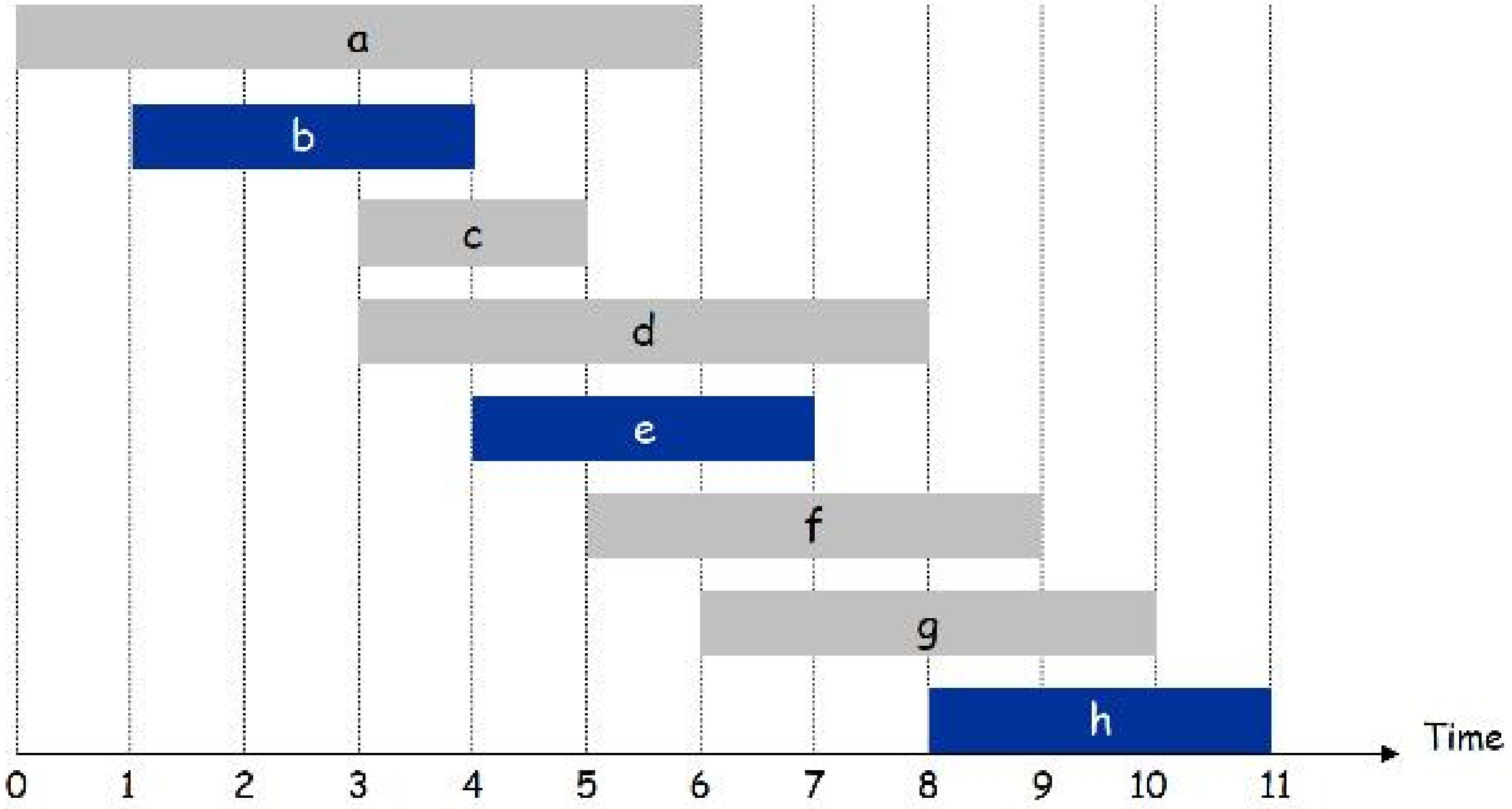
Fast algorithm: worst-case running time grows slowly with input size

立志成才报国裕民

Five Representative Problems

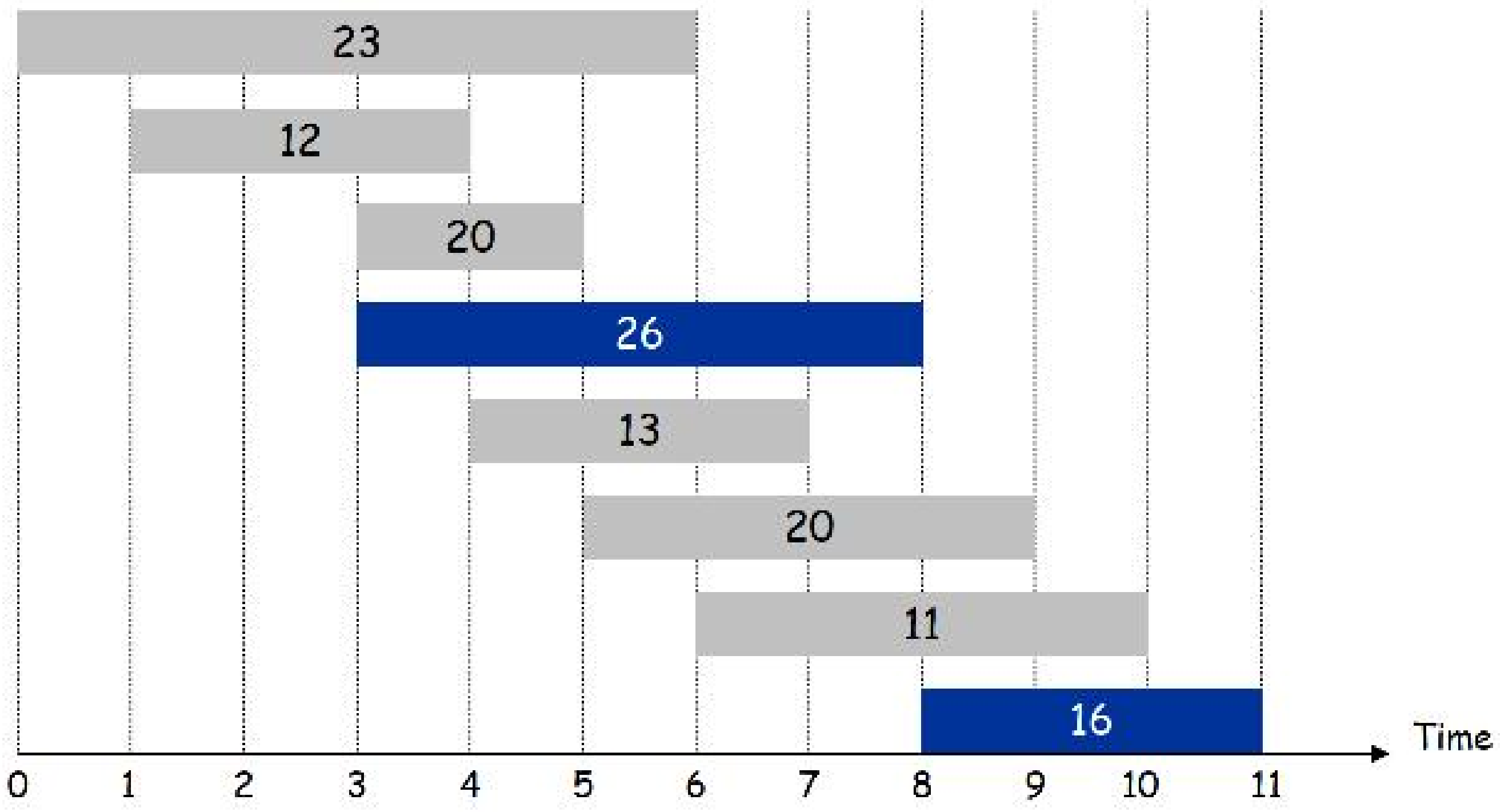
Interval Scheduling

- **Input:** Set of jobs with start times and finish times
- **Goal:** Find **maximum cardinality** subset of mutually compatible (i.e., jobs don't overlap) jobs



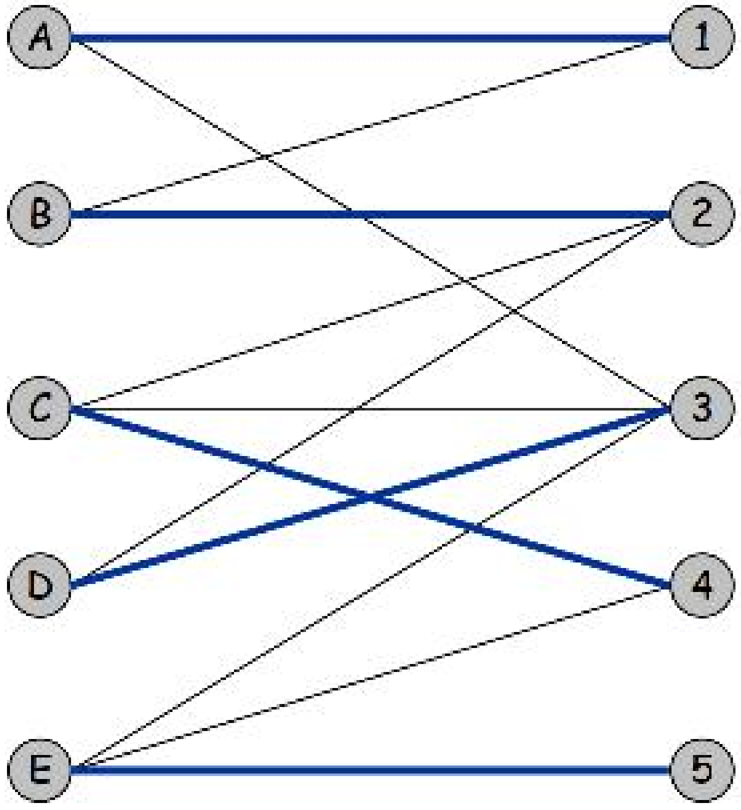
Weighted Interval Scheduling

- **Input:** Set of jobs with start times, finish times, and weights
- **Goal:** Find maximum weight subset of mutually compatible jobs



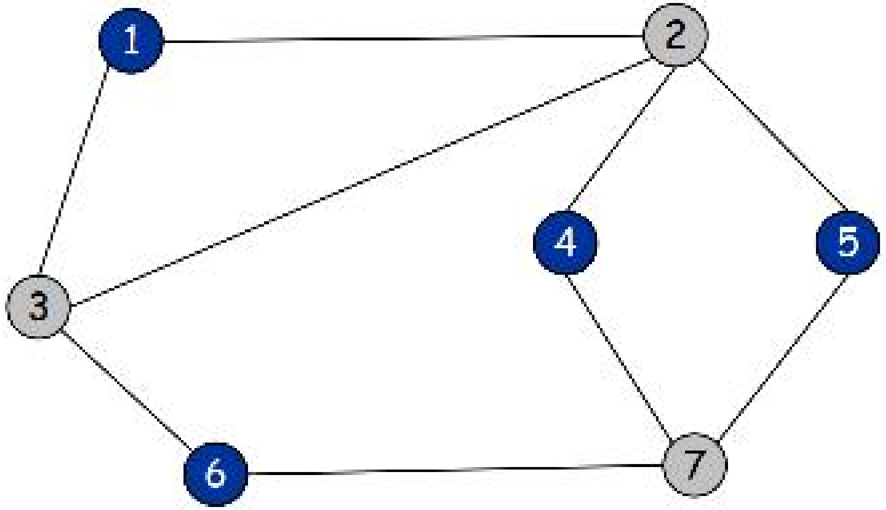
Bipartite Matching

- **Input:** Bipartite graph
- **Goal:** Find **maximum cardinality** matching



Independent Set

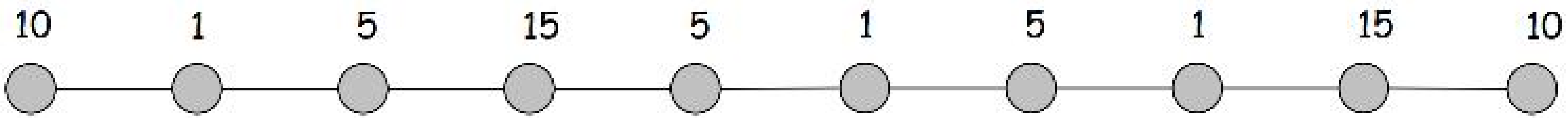
- **Input:** Graph
- **Goal:** Find **maximum cardinality** independent (i.e., subset of nodes such that no two joined by an edge) set



- **Extension:** Weighted independent set

Competitive Facility Location

- **Input:** Graph with weight on each node
- **Game:** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected
- **Goal:** Select a **maximum weight** subset of nodes



Second player can guarantee 20, but not 25

Five Representative Problems

- **Variations on a theme:** independent set

- **Interval scheduling:** nlogn greedy algorithm
- **Weighted interval scheduling:** nlogn dynamic programming algorithm
- **Bipartite matching:** n^2 max-flow-based algorithm
- **Independent set:** NP-complete
- **Competitive facility location:** PSPACE-complete

CS240 Algorithm Design and Analysis

Lecture 1

Computational Tractability

Quan Li

Fall 2025
2025.09.16

"For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing." – Francis Sullivan, Science, Vol. 287, No. 5454, p.799, February 2000

Polynomial-Time

- **Brute force.** For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution
  - Typically takes 2^N time or worse for inputs of size N
  - Unacceptable in practice

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

Poly-time: There exists constants $c > 0$ and $d > 0$ such that on every input of size N, its running time is bounded by cN^d steps

- **Thm.** An algorithm is poly-time iff the above scaling property holds (i.e., choose $C = 2^d$)

Polynomial-Time

- **Def.** An algorithm is **efficient** if its running time is polynomial

- **Exceptions**
  - Some poly-time algorithms do have high constants and/or exponents, and are useless in practice. Ex. $6.02 * 10^{23} * N^{20}$

- **Justification: It really works in practice!**
  - In practice, the poly-time algorithms that people develop almost always have low constants and low exponents
  - Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem

Why It Matters

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second.
In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time

Worst-Case Analysis

- **Worst case running time.** Obtain bound on largest possible running time of algorithm on input of a given size N
  - Generally captures efficiency in practice

- **Exceptions**
  - Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare (e.g., simplex method Unix grep)

Average-Case Analysis

- **Average case running time.** Obtain bound on running time of algorithm on **random** input as a function of input size N
  - Need to choose a distribution over input instances
  - Algorithm tuned for a certain distribution may perform poorly on other inputs
  - Average-case analysis may tell us more about the choice of distributions than about the algorithm itself

Asymptotic Order of Growth

Asymptotic Order of Growth

- **Importance:** vocabulary for the design and analysis of algorithms (e.g., bit-oh notation)
- Sweet spot for high-level reasoning about algorithms
- Coarse enough to suppress architecture/language/compiler-dependent details
- Sharp enough to make useful comparisons between different algorithms, especially on large inputs (e.g., sorting or integer multiplication)
- High-level idea: Suppress constant factors (too system-dependent) and lower-order terms → irrelevant for large inputs
- **Example:** equate $6n\log_2 n + 6n$ with just $n\log n$
- **Terminology:** running time is $O(n\log n)$ where n = input size (e.g., length of input array)

Examples

- Problem: does array A contain the integer t?
- Given A (array of length n) and t (an integer)

```
for i = 1 to n
    if A[i] == t return TRUE
return FALSE
```

- Problem: Given A, B (array of length n) and t (an integer), does A or B contain t?

```
for i = 1 to n
    if A[i] == t return TRUE
for i = 1
    if B[i] == t return TRUE
return FALSE
```

- Problem: Do arrays A and B have a number in common?

```
for i = 1 to n
    for j = 1 to n
        if A[i] == B[j] return
        TRUE
return FALSE
```

- Problem: Do arrays A have duplicated entities?

```
for i = 1 to n
    for j = i+1 to n
        if A[i] == A[j] return
        TRUE
return FALSE
```

Question: What is the running time?

$O(1)$
$O(n)$
$O(\log n)$
$O(n^2)$

Asymptotic Order of Growth

- **Upper bounds.** $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 >= 0$ such that for all $n >= n_0$ we have $T(n) <= c \cdot f(n)$
- Example #1
  - Claim: If $T(n) = a_k n^k + \ldots + a_1 n + a_0$ then $T(n) = O(n^k)$
  - Proof: Choose $n_0 = 1$ and $c = |a_k|+|a_{k-1}|+\ldots+|a_1|+|a_0|$
  - Need to show that exist $n >= 1$, $T(n) <= cn^k$
  - We have for every $n >= 1$, $T(n) <= |a_k|n^k + \ldots + |a_1|n + |a_0| <= |a_k|n^k + \ldots + |a_1|n^k + |a_0|n^k = cn^k$
- Example #2
  - Claim: for every $k >= 1$, n^k is not $O(n^{k-1})$
  - Proof: by contradiction. Suppose $n^k = O(n^{k-1})$
  - Then exist constants c, $n_0 > 0$ such that $n^k <= cn^{k-1}$ for every $n >= n_0$
  - But then cancelling n^{k-1} from both sides → $n <= c$ which is clearly false

Asymptotic Order of Growth

- **Upper bounds.** $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 >= 0$ such that for all $n >= n_0$ we have $T(n) <= c \cdot f(n)$
- **Lower bounds.** $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 >= 0$ such that for all $n >= n_0$ we have $T(n) >= c \cdot f(n)$
- **Tight bounds.** $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.
  - Exist constants c_1, c_2, n_0, such that $c_1 f(n) <= T(n) <= c_2 f(n)$ for all $n >= n_0$
- Let $T(n) = \frac{1}{2}n^2 + 3n$. Which of the following statements are true?
  - $T(n) = O(n)$
  - $T(n) = \Omega(n)$
  - $T(n) = \Theta(n)$
  - $T(n) = O(n^3)$

Ex: $T(n) = 32n^2 + 17n + 32$
$T(n)$ is $O(n^2)$, $O(n^3)$ ← choose $c = 50$, $n_0 = 1$
$T(n)$ is $\Omega(n^2)$, $\Omega(n)$ ← choose $c=32$, $n_0 = 1$
$T(n)$ is $\Theta(n^2)$
$T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$

More Examples

- **Claim:** $2^{n+10} = O(2^n)$
- **Proof:** need to pick constants c, n_0 such that $2^{n+10} <= c2^n$ for every $n >= n_0$
- **Note:** $2^{n+10} = 2^{10}2^n = (1024)2^n$
- **So:** if we choose $c = 1024$, $n_0 = 1$, then $2^{n+10} <= c2^n$ holds

- **Claim:** 2^{10n} is not $O(2^n)$
- **Proof:** By contradiction. If $2^{10}n = O(2^n)$, then exist constants c, $n_0 > 0$, such that
- $2^{10n} <= c2^n$ for every $n >= n_0$
- But then cancelling 2^n:
- $2^{9n} <= c$ for every $n >= n_0$ which is certainly false

More Examples

- **Claim:** for every pair of (positive) functions $f(n)$, $g(n)$, $\max\{f, g\} = \Theta(f(n) + g(n))$
- **Proof:** $\max\{f, g\} = \Theta(f(n) + g(n))$

For every n, we have $\max\{f(n), g(n)\} <= f(n) + g(n)$

and

$2\max\{f(n), g(n)\} >= f(n) + g(n) \rightarrow \max\{f(n), g(n)\} >= ½ (f(n) + g(n))$

Thus: $½ (f(n) + g(n)) <= \max\{f(n), g(n\} <= f(n) + g(n)$ for all $n >= 1$

$\rightarrow \max\{f, g\} = \Theta(f(n) + g(n))$, where $n_0 = 1$, $c_1 = ½$, $c_2 = 1$

Notation and Properties

- **Slight abuse of notation.** $T(n) = O(f(n))$
- **Asymmetric**
  - $f(n) = 5n^3$; $g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - But $f(n) \neq g(n)$
  - Better notation: $T(n) \in O(f(n))$

$O(f(n))$ is a set of functions, but we often write $T(n) = O(f(n))$ instead of $T(n) \in O(f(n))$

- **Transitivity**
  - If $f=O(g)$ and $g=O(h)$ then $f=O(h)$
  - If $f=\Omega(g)$ and $g=\Omega(h)$ then $f=\Omega(h)$
  - If $f=\Theta(g)$ and $g=\Theta(h)$ then $f=\Theta(h)$
- **Additivity**
  - If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$
  - If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$
  - If $f = \Theta(h)$ and $g = \Theta(h)$ then $f + g = \Theta(h)$

Asymptotic Bounds for Some Common Functions

- **Polynomials.** $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$
- **Polynomial time.** Running time is $O(n^d)$ for some constant d independent of the input size n
- **Logarithms.** $O(\log_a n) = O(\log_b n)$ for any constants a, b > 1
- **Logarithms.** For every x > 0, $\log n = O(n^x)$

log grows slower than every polynomial

- **Exponentials.** For every r > 1 and every d > 0, $n^d = O(r^n)$

every exponential grows faster than every polynomial

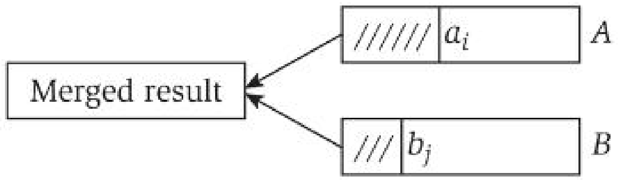
A Survey of Common Running Times

Linear Time: O(n)

- **Linear time.** Running time is at most a constant factor times the size of the input
- **Computing the maximum.** Compute the maximum of n numbers $a_1, \ldots, a_n$

```
max ← a1
for i = 2 to n {
    if (ai > max)
        max ← ai
}
```

- **Merge.** Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole




```
i = 1, j = 1
while (both lists are nonempty) {
   if (ai ≤ bj) append ai to output list and increment i
   else         append bj to output list and increment j
}
append remainder of nonempty list to output list
```

- **Claim.** Merging two lists of size n takes O(n) time
- **Pf.** After each comparison, the length of output list increases by 1

O(nlogn) Time

- **O(nlogn) time.** Arises in divide-and-conquer algorithms
- **Sorting.** Mergesort and heapsort are sorting algorithms that perform O(nlogn) comparisons
- **Largest empty interval.** Given n timestamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?
- **O(nlogn) solution.** Sort the timestamps. Scan the sorted list in order, identifying the maximum gap between successive timestamps

Quadratic Time: $O(n^2)$

- **Quadratic time.** Enumerate all pairs of elements
- **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

```
min ← (x1 - x2)^2 + (y1 - y2)^2
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)^2 + (yi - yj)^2
      if (d < min)
         min ← d
   }
}
```

Don't need to take square roots

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

Cubic Time: $O(n^3)$

上海科技大学 ShanghaiTech University

- **Cubic time.** Enumerate all triples of elements
- **Set disjointness.** Given n sets $S_1, \ldots, S_n$ each of which is a subset of 1, 2, …, n, is there some pair of these which are disjoint?
- **$O(n^3)$ solution.** For each pairs of sets, determine if they are disjoint

```
foreach set Si {
    foreach other set Sj {
        foreach element p of Si {
            determine whether p also belongs to Sj
        }
        if (no element of Si belongs to Sj)
            report that Si and Sj are disjoint
    }
}
```

立志成才 报国裕民

Polynomial Time: $O(n^k)$ Time

- **Independent set of size k.** Given a graph, are there k nodes such that no two are joined by an edge?
- **$O(n^k)$ solution.** Enumerate all subsets of k nodes

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set = $O(k^2)$
- Number of k element subsets = $$\binom{n}{k} = \frac{n\,(n-1)\,(n-2)\cdots(n-k+1)}{k\,(k-1)\,(k-2)\cdots(2)\,(1)} \leq \frac{n^k}{k!}$$
- $O(k^2 n^k / k!) = O(n^k)$

poly-time for k=17,
but not practical

Exponential Time

上海科技大学 ShanghaiTech University

- **Independent set.** Given a graph, what is maximum size of an independent set?
- **$O(n^2 2^n)$ solution.** Enumerate all subsets.

```
S* ← ϕ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

立志成才报国裕民