Content:

- Asymptotics & Logarithms 1-3
- Proof Methods (Induction, Contradiction, & Combinatorial Arguments) 4-5
- Recurrences and the Master Theorem 6-7
- Heaps 8-9 & Heapsort 9-11
- Quicksort 12
- Sorting and Searching in Linear time (Counting 13-15, Radix 15-18)
- BSTs 20-22 & RBTs 23
- Hashing 24 appendix after 69
- Dynamic Programming 25-27
- Greedy Algorithms 28-37 first process: 32
- Amortized Analysis 38-40
- Graph Algorithm 41-46
    - BFS 41, DFS 43, Topological 46
- MST 47-48
- Shortest Path 49-50
- Max Flow 51-62
    - Bipartite Matching 59
- NP completeness 63-68

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Given an array of integers, the worst-case time for each algorithm:

Insertion sort: $\mathcal{O}(n^2)$
Merge sort: $\mathcal{O}(n \log n)$
Heap sort: $\mathcal{O}(n \log n)$
Quick sort: $\mathcal{O}(n^2)$
Counting sort: $\mathcal{O}(n+k)$

1. Note 1: k can be larger than n, and it will change the complexity
2. Note 2: Won't work with real numbers or any sets S that are not countable (*i.e.* no surjective $f : \mathbb{N} \to S$) Only work on finite sets.

Radix sort: $\mathcal{O}(d(n+k))$ (Uses counting sort, so restrictions of counting sort applies)
Bucket sort: $\mathcal{O}(n^2)$ (Worst case when all numbers in the same bucket)

Visualization: `https://www.toptal.com/developers/sorting-algorithms`

BFS&DFS: O(V+E)
Prims: O(Elog(V)) binary heap; O(E+Vlog(V)) Fibonacci Heap
Dijkstra: O(Elog(V)); Bellman-ford: O(VE)
Ford-Fulkerson: O(E |f*|) f*是最大流的数值
Edmonds-Karp: O(VE^2)

Asymoptotics

def

- $f(n) = \mathcal{O}(g(n)) \Leftrightarrow \exists c, n_0 > 0$ s.t. $0 \leq f(n) \leq cg(n), \forall n \geq n_0$
- $f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_0 > 0$ s.t. $0 \leq cg(n) \leq f(n), \forall n \geq n_0$
- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n) \Leftrightarrow \exists c_1, c_2, n_0 > 0$ s.t. $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0$

e.g. (2022 final): What does it mean by $n! = n^n e^{-n}\sqrt{2\pi n}\left(1 + \mathcal{O}\left(\frac{1}{n}\right)\right)$ (Stirling formula[1])?
Note here $f(n) = \frac{n!}{n^n e^{-n}\sqrt{2\pi n}} - 1$, $g(n) = \frac{1}{n}$,
So $\frac{n!}{n^n e^{-n}\sqrt{2\pi n}} - 1 \leq \frac{c}{n}$ for some $c, n_0 > 0$ and all $n \geq n_0$

properties

Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
Proof: $\exists c_1, c_2, n_1 > 0$ s.t. $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_1$.
$\exists c_3, c_4, n_2 > 0$ s.t. $0 \leq c_3 h(n) \leq g(n) \leq c_4 h(n), \forall n \geq n_2$.
$\therefore 0 \leq c_1 c_3 h(n) \leq c_1 g(n)$ and $c_2 g(n) \leq c_2 c_4 h(n), \forall n \geq n_2$.
$\therefore 0 \leq c_1 c_3 h(n) \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \leq c_2 c_4 h(n), \forall n \geq \max(n_1, n_2)$.

Transpose: $f(n) = \mathcal{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$
Proof: $\exists c, n_0 > 0$ s.t. $0 \leq f(n) \leq cg(n), \forall n \geq n_0 \Leftrightarrow \exists c, n_0 > 0$ s.t. $0 \leq \frac{1}{c} f(n) \leq g(n), \forall n \geq n_0$

Symmetry: $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
Proof: $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$ and $g(n) = \mathcal{O}(f(n))$

Limit Method

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$[2] (The 2 and 3 here are refering to the footnote numbers.)
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \Rightarrow f(n) = \mathcal{O}(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$[3]
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c,\ c \in (0,\infty) \Rightarrow f(n) = \Theta(g(n))$

L'Hopital's rule: $\lim_{x\to a} \frac{f(x)}{g(x)} = \frac{0}{0}$ or $\frac{\infty}{\infty} \Rightarrow \lim_{x\to a} \frac{f(x)}{g(x)} = \lim_{x\to a} \frac{f'(x)}{g'(x)}$

[2] $f(n) = o(g(n))$ if and only if $\forall c > 0$, $\exists n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.
[3] $f(n) = \omega(g(n))$ if and only if $\forall c > 0$, $\exists n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$.

UNIVERSITY OF TORONTO

Limit Method (More precisely)

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = o(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in [0,\infty) \Rightarrow f(n) = \mathcal{O}(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in (0,\infty) \Rightarrow f(n) = \Theta(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in (0,\infty] \Rightarrow f(n) = \Omega(g(n))$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \omega(g(n))$

- log*n remark

Remark

Notation (in this course): $(\log n)^2 = (\log n)(\log n)$ and $\log^{(2)} n = \log(\log n)$
$\log^* n = \min\{i \geq 0 : \log^{(i)} n \leq 1\}$

- Bounded Functions

Polylogarithmically bounded: $\exists k > 0,\ f(n) = \mathcal{O}((\log n)^k)$
Polynomially bounded: $\exists k > 0,\ f(n) = \mathcal{O}(n^k)$
Exponentially bounded: $\exists k > 0,\ f(n) = \mathcal{O}(k^n)$

Polynomially-Bounded Functions

Theorem

$f(n) = \mathcal{O}(n^k) \Leftrightarrow \log(f(n)) = \mathcal{O}(\log n)$

Theorem

All Logarithmically bounded functions are polynomically bounded. i.e. $f(n) = \mathcal{O}((\log n)^a) \Rightarrow f(n) = \mathcal{O}(n^b),\ \forall a, b \geq 0$

Theorem

All polynomially bounded functions are exponentially bounded. i.e. $f(n) = \mathcal{O}(n^a) \Rightarrow f(n) = \mathcal{O}(b^n),\ \forall a > 0, b > 1$

- Logrithm method

Limit of logs: $\lim_{x \to a}(\log_b f(x)) = \log_b \left(\lim_{x \to a} f(x)\right)$ ($\log_b(\cdot)$ is continuous)

Suppose we want to compute $\lim_{n \to \infty} \frac{f(n)}{g(n)} = L.$

$$\log\left(\lim_{n \to \infty} \frac{f(n)}{g(n)}\right) = \log L$$

$$\lim_{n \to \infty}\left(\log \frac{f(n)}{g(n)}\right) = \log L$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = L = 2^{\lim_{n \to \infty}\left(\log \frac{f(n)}{g(n)}\right)}$$

▼ Comparing Functions

Short hand notation: $f(n) << g(n) \Leftrightarrow f(n) = \mathcal{O}(g(n))$
Assume f and h are eventually positive, i.e. $\lim_{n\to\infty} f(n) > 0$ and $\lim_{n\to\infty} h(n) > 0$

$1 << \log^*(n) << \log^{(i)} n << (\log n)^a << \sqrt{n} << n << n\log n << n^{1+b} << c^n << n!$, for all positive i, a, b, c

$f(n) << g(n) \Rightarrow h(n)f(n) << h(n)g(n)$
Proof: $\lim_{n\to\infty} \frac{h(n)f(n)}{h(n)g(n)} = \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

$f(n) << g(n) \Rightarrow f(n)^{h(n)} << g(n)^{h(n)}$
Proof: $\log\left(\lim_{n\to\infty} \frac{f(n)^{h(n)}}{g(n)^{h(n)}}\right) = \lim_{n\to\infty} h(n)\log\frac{f(n)}{g(n)} = \lim_{n\to\infty} h\log\lim_{n\to\infty}\frac{f}{g} = -\infty$, $\lim_{n\to\infty}\frac{f^h}{g^h} = 0$

$f(n) << g(n)$ and $\lim_{n\to\infty} h(n) > 1 \Rightarrow h(n)^{f(n)} << h(n)^{g(n)}$
Proof: $\log\left(\lim_{n\to\infty} \frac{h(n)^{f(n)}}{h(n)^{g(n)}}\right) = \lim_{n\to\infty}(f-g)\log h = -\infty$, $\lim_{n\to\infty}\frac{h^f}{h^g} = 0$

UNIVERSITY OF TORONTO

▼ Simple Examples (Direct Solve)

Prove that $2^{n+1} = \mathcal{O}(2^n)$.

Solution: $0 \le 2^{n+1} = 2\cdot 2^n \le c\cdot 2^n$, $\forall n \ge 0$, if $c \ge 2$.
$\therefore$ we can choose $c = 2$, $n_0 = 1$, $0 \le 2^{n+1} \le c2^n$, $\forall n \ge n_0$.
$\therefore 2^{n+1} = \mathcal{O}(2^n)$.

Prove that $2^{n+1} = \Omega(2^n)$.

Solution: $0 \le c\cdot 2^n \le 2\cdot 2^n = 2^{n+1}$, $\forall n \ge 0$, if $c \le 2$.
$\therefore$ we can choose $c = 1$, $n_0 = 1$, $0 \le c2^n \le 2^{n+1}$, $\forall n \ge n_0$.
$\therefore 2^{n+1} = \Omega(2^n)$.

Proof Methods (Induction, Contradiction, & Combinatorial Arguments)

- **Weak Induction**
  - def

    Weak induction:
    1. Basis: show $P(n_0)$ is true
    2. Hypothesis: Assume $P(n)$ is true (**!!!Note:** You should not assume it is true for all n. This is what you need to prove. Assume $P(n)$ is true for all n will cost you 2-3 marks in exams.)
    3. Induction: Show $P(n) \Rightarrow P(n+1)$
  - simplest example

    Prove $n! \leq n^n$, $\forall n \geq 1$
    Proof: **Base Step**: $n = 1$, $1! = 1 = 1^1$
    (i) Using n only:
    Induction Hypothesis: Assume $n! \leq n^n$
    Induction Step: For $n+1$, $(n+1)! = (n+1)n! \leq (n+1)n^n$ by IH
    $\leq (n+1)(n+1)^n = (n+1)^{n+1}$
  - simplest example: use k (same thing as previous)

    (ii)Introducing a new variable k:
    Induction Hypothesis: Assume when $n = k \geq 1$, $k! \leq k^k$
    Induction Step: When $n = k+1$, $(k+1)! = (k+1)k! \leq (k+1)k^k$ by IH
    $\leq (k+1)(k+1)^k = (k+1)^{k+1}$

Weak induction examples

Prove $11^n - 6$ is divisible by 5, $\forall n \geq 1$
Let $P(n) = 5|(11^n - 6)$ (5 divides $11^n - 6$)
Base Step: $n = 1$, $11^1 - 6 = 5$ $5|5$
Induction Hypothesis: Assume $P(n)$ is true
Induction Step: Check $P(n+1)$, $11^{n+1} - 6 = 11 \cdot 11^n - 6 = 11 \cdot (5m + 6 - 6)$
$= 55m + 66 - 6 = 55m + 60 = 5(11m + 12)$ for some m
So $5|11^{n+1} - 6$, $P(n+1)$ is true.

Strong Induction

1. Basis: show $P(n_0), P(n_1), \ldots$ are true
2. Hypothesis: Assume $P(k)$ is true, $\forall k \leq n$
3. Induction: Show $P(n_0) \wedge \cdots \wedge P(k) \wedge \cdots \wedge P(n) \Rightarrow P(n+1)$

e.g. The Fundamental Theorem of Arithmetic: all integers $n \geq 2$ can be expressed as the product of one or more prime numbers
Proof: **Base Step**: $n = 2$, 2 is a prime
Induction Hypothesis: assume all $k \in [2, n]$ can be written as the product of one or more primes
Induction Step:
$n+1$ is prime. Then it can be expressed as the product of itself.
$n+1$ is not prime. Then $n+1 = k_1 k_2$ for some integers $k_1, k_2 < n+1$. By IH, k_1, k_2 can be written as product of primes. Thus $n+1$ can be written as product of primes

Prove that using \$2 and \$5, we can make any amount $\geq$ \$4
Proof: **Base Step**: $n = 4$ can be made using \$2+\$2, $n = 5$ can be made using \$5
Induction Hypothesis: assume $\forall k \in [4, n]$, \$$k$ can be made using \$2 and \$5
Induction Step: $(n+1) = (n-1) + 2$ and $n-1$ can be made using \$2 and \$5 by IH

Prove by Contradiction

1. Assume toward a contradiction $\neg P$/ $\bar{P}$(not P)
2. Make some argument
3. arrive at a contradiction
4. $\therefore P$ must be true

e.g. Prove that there are infinitely many prime number
Proof: Assume that there are a finite number of primes
Let S be the complete set of primes
let $P = \prod_{x \in S} x + 1$
$P \notin S$ because $P > x$, $\forall x \in S$
P is a prime because P is not divisible by any prime, $1 \equiv P \mod x$, $\forall x \in S$
P is a prime but not in S, contradiction.

Recurrence:

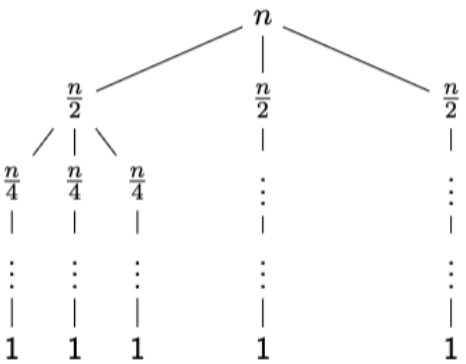
Recurrence:

Find asymptotic expressions for the following $T(n)$:
e.g. $T(n) = T(n-1) + n$, $T(1) = 1$
Proof: $T(n) = T(n-1) + n = T(n-2) + (n-1) + n = \cdots = T(1) + 2 + 3 + 4 + \cdots + (n-1) + n$
$T(n) = \sum_{i=1}^{n} = \frac{n(n+1)}{2}$, $T(n) = \Theta(n^2)$

e.g. $T(n) = T(n/2) + n$, $T(1) = 1$
Proof: $T(n) = T(n/2) + n = T(n/4) + n/2 + n = T(n/8) + n/4 + n/2 + n = \cdots =$
$T(1) + 2 + 4 + 8 + \cdots + (n/4) + (n/2) + n$
$T(n) = \sum_{i=0}^{\log n} \frac{n}{2^i} = n\frac{1 - (1/2)^{\log n + 1}}{1 - 1/2} = n(2 - 1/n) = 2n - 1$, $T(n) = \Theta(n)$

Basic examples

Find asymptotic expressions for the following $T(n)$:
e.g. $T(n) = 3T(n/2) + n$, $T(1) = 1$
Proof: $T(n) = 3T(n/2) + n = 3(3T(n/4) + n/2) + n = 3^2T(n/4) + 3(n/2) + n = \cdots = 3^kT(1) + 3^{k-1}(n/2^{k-1}) + \cdots + 3^2(n/2^2) + 3(n/2) + n = 3^kT(1) + \sum_{i=0}^{k-1} \left(\frac{3}{2}\right)^i n$
Assume $n = 2^k$, $k = \log n$ $T(n) = 3^{\log n} + \sum_{i=0}^{\log n - 1} \left(\frac{3}{2}\right)^i n = n^{\log 3} + n\frac{(3/2)^{\log n} - 1}{(3/2) - 1} =$
$n^{\log 3} + 2n(n^{\log 3}/n - 1) = 3n^{\log 3} - 2n$,
$T(n) = \Theta(n^{\log 3})$



Master Theorem:

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$, where $a \geq 1, b \geq 1$, $f(n)$ is asymptotically positive

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$, $\epsilon > 0$. Then $T(n) = \Theta(n^{\log_b a})$ (cost of solving the sub-problems at each level increases by a certain factor, the last level dominates)
- Case 2: $f(n) = \Theta(n^{\log_b a})$. Then $T(n) = \Theta(n^{\log_b a} \log n)$ (cost to solve subproblem at each level is nearly equal)
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$, $\epsilon > 0$ and $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ and $n > n_0$ (regularity condition, always holds for polynomials). Then $T(n) = \Theta(f(n))$ (cost of solving the subproblems at each level decreases by a certain factor)

Method:

1. identify a and b and compute $\log_b a$
2. compare $n^{\log_b a}$ to $f(n)$ and decide which case applies
3. don't forget to check the regularity condition for case 3

UNIVERSITY OF TORONTO

Examples

$T(n) = 7T\left(\frac{n}{2}\right) + n^2$ (Works for $n^2 \log n$)
Solution: $a = 7, b = 2, \log_2 7 \approx 2.8, f(n) = n^2 = \mathcal{O}(n^{\log_2 7 - \epsilon})$ for $\epsilon \in (0, \log_2 7 - 2)$
Case 1, $T(n) = \Theta(n^{\log_2 7})$

$T(n) = 4T\left(\frac{n}{2}\right) + n^2\sqrt{n}$
Solution: $a = 4, b = 2, \log_2 4 = 2, f(n) = n^{2.5} = \Omega(n^{2+\epsilon})$ for $\epsilon \in (0, 0.5]$
Case 3, Check regularity: $af\left(\frac{n}{b}\right) = 4\left(\frac{n}{2}\right)^{2.5} \leq cn^{2.5}$. Choose $c \in \left[\frac{1}{\sqrt{2}}, 1\right)$
$T(n) = \Theta(f(n)) = \Theta(n^{2.5})$

$T(n) = T(\sqrt{n}) + 1$
Solution: let $n = 2^m$, $S(m) = T(2^m) = T(n)$
Then $T(2^m) = T(2^{\frac{m}{2}}) + 1 \Leftrightarrow S(m) = S\left(\frac{m}{2}\right) + 1$
$a = 1, b = 2, \log_2 1 = 0, f(m) = 1 = m^0 = \Theta(m^0)$
Case 2, $T(2^m) = S(m) = \Theta(\log m)$, $T(n) = \Theta(\log \log n)$

Heaps & Heapsort:

Heap Definition:

Definition: A heap is an array $A = [a_1, a_2, ..., a_n]$ of elements such that:

- Heap shape property: the heap is an almost complete binary tree (all except the last row is full)
- Heap order property:
  - Max-heap: $\forall i,\ A[\text{parent}(i)] \geq A[i]$
  - Min-heap: $\forall i,\ A[\text{parent}(i)] \leq A[i]$

Indexing parents and children (Assume index of array starts at 1)

- $\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$
- $\text{left}(i) = 2i$
- $\text{right}(i) = 2i + 1$

Heap Properties:

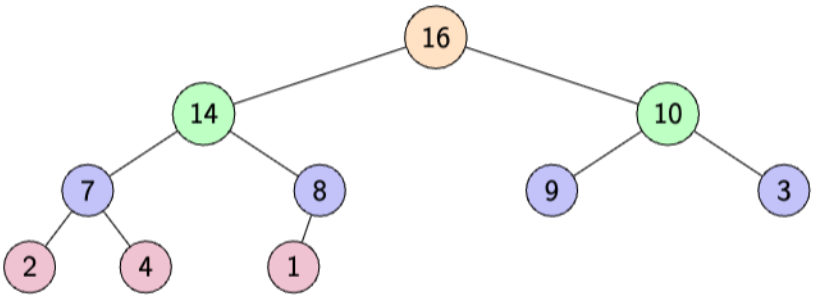
Key Take-Aways

- $2^h \leq n \leq 2^{h+1} - 1 \Leftrightarrow h = \lfloor \log n \rfloor$
- For a max-heap, the maximum value will always occur at the root
- For a min-heap, the minimum value will always occur at the root
- Heapsort is $\Theta(n \log n)$ in time and $\Theta(1)$ in space (space complexity only considers extra memory needed excluding input)
- Heapsort is an in-place algorithm
- Heapsort is not stable

Note: A stable sorting algorithm is a sorting algo that preserves the relative order of the same value in the previous step

Example

$A = [16, 14, 10, 7, 8, 9, 3, 2, 4, 1]$



Basic proofs

Let h be the height and n the number of nodes in a heap, $2^h \leq n \leq 2^{h+1} - 1$
Proof: the number of elements in a full binary tree of height h is $2^{h+1} - 1$. Min occurs if only 1 element in the last level.

Let h be the height and n the number of nodes in a heap, $h = \lfloor \log n \rfloor$
Proof: $2^h \leq n \leq 2^{h+1} - 1 \Rightarrow h \leq \log n < h + 1 \Rightarrow h = \lfloor \log n \rfloor$ since h is integer

In a max-heap, the root contains the largest value
Proof by induction:
Base Step: $h = 0$, root is the only element
Induction Hypothesis: Assume the propsition is true for heaps of size $0 < k < n$
Induction Step: Consider the left and right subtree of the root, by IH, their roots are the maximum of the subheap, then by max-heap property, the root is larger than all its children.

Heap Sort:

操作	作用	单次耗时	总耗时
Max-Heapify(A, i)	让以 `i` 为根的子树恢复最大堆性质	`O(h)`，h = 子树高度 ≤ `O(log n)`	—
Build-Max-Heap(A)	从无序数组建立最大堆	`O(n)`	—
Extract-Max(A)	取出堆顶最大元素并重新调整堆	`O(log n)`	`O(n log n)`（在排序中执行 n 次）
HeapSort(A)	先建堆，再不断取出最大元素	`O(n log n)`	`O(n log n)`
Insert(A, key)	插入一个新元素到堆中	`O(log n)`	—
Increase-Key(A, i, key)	增大节点值并向上调整	`O(log n)`	—
Find-Max(A)	取堆顶元素，不改动	`O(1)`	—

Max-Heapify:

- **Max-Heapify** 只“直接操作”以 **i** 为根的当前层和它的下一层（父节点与两个孩子），
- 但通过递归调用，它间接修复整个以 **i** 为根的整棵子树。

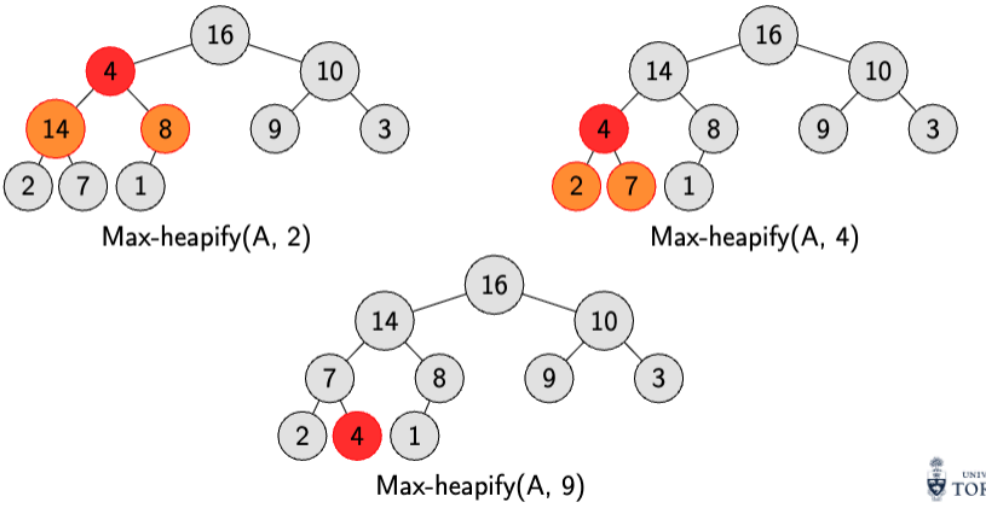
Max-Heapify

Enforces the heap order property if it is violated

1: **function** Max-Heapify(A, i)
2: Compare $A[i]$ with $A[2i]$ and $A[2i+1]$
3: **if** $A[i]$ is smaller **then**
4: swap $A[i] \leftrightarrow \max(A[2i], A[2i+1])$
5: Recurse downwards, until property is not violated or hit a leaf node

Runtime: $\mathcal{O}(h) = \mathcal{O}(\log n)$

Example



Built Max Heap:

把一个“乱序数组”一步步改造成合法的最大堆（Max-Heap）

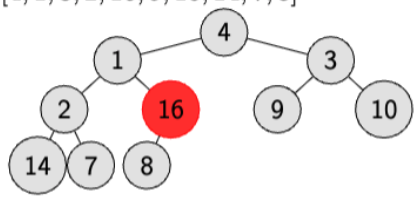
1: **function** Build-Max-Heap(A, n)
2: **for each** $i = \lfloor \frac{n}{2} \rfloor : 1$ **do** ▷ the rest are leaf nodes
3: Max-Heapify(A, i)

Runtime:

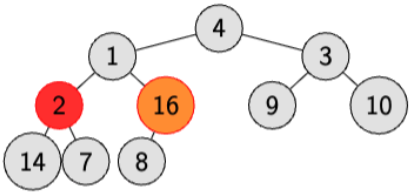
- Simple: $\mathcal{O}(n \log n)$ (for loop $\times$ cost for Heapify)
- Actual: Time to run Max-Heapify is linear in the height of the node it is run on and most node have smaller height
  - At height h, there are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes.
  - height of heap is $\lfloor \log n \rfloor$
  - Runtime: $\sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \mathcal{O}(h) \leq cn \sum_{h=0}^{\infty} \frac{h}{2^h} = \cdots = \mathcal{O}(n)$

Example

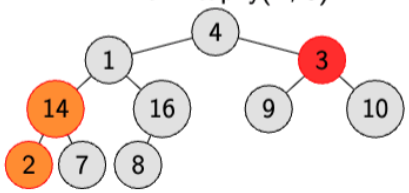
$A = [4, 1, 3, 2, 16, 9, 10, 14, 7, 8]$



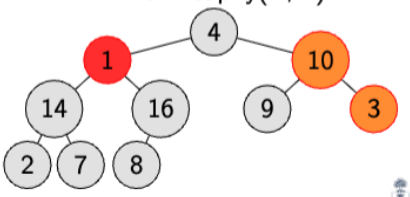
Max-heapify(A, 5)



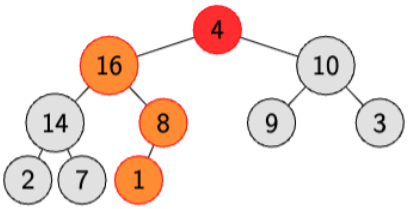
Max-heapify(A, 4)



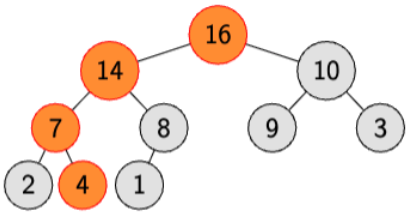
Max-heapify(A, 3)



Max-heapify(A, 2)



Max-heapify(A, 1)



Final result: $A = [16, 14, 10, 7, 8, 9, 3, 2, 4, 1]$

Heapsort:

Build-Max-Heap 只是 Heapsort 的前半步，它的目标是“构造堆”；
而 Heapsort 是一个完整的排序算法，在建堆之后还会“反复取最大值、放到末尾”——那才是排序。

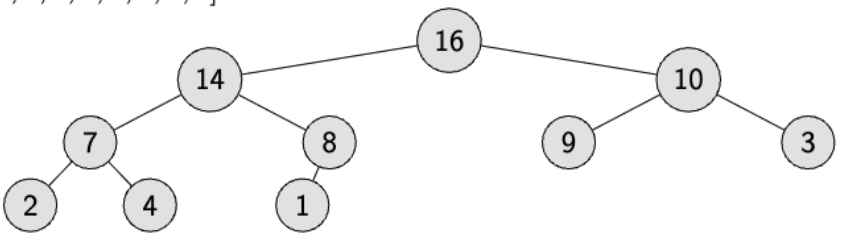
```
1: function Extract-Max(A)
2:     Swap A[1] ↔ A[A.size()]
3:     A.size = A.size-1
4:     Max-Heapify(A, 1)
```

```
1: function Heapsort(A, n)
2:     Build-Max-Heap(A, n)
3:     for each i = n : 2 do
4:         Extract-Max(A)
```

Runtime: $\mathcal{O}(n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$

Example

$A = [16, 14, 10, 7, 8, 9, 3, 2, 4, 1]$



Iteration 1: $A = [14, 8, 10, 7, 1, 9, 3, 2, 4|16]$
Iteration 2: $A = [10, 8, 9, 7, 1, 4, 3, 2|14, 16]$
Iteration 3: $A = [9, 8, 4, 7, 1, 2, 3|10, 14, 16]$
Iteration 4: $A = [8, 7, 4, 3, 1, 2|9, 10, 14, 16]$
Iteration 5: $A = [7, 3, 4, 1, 2|8, 9, 10, 14, 16]$
Iteration 6: $A = [4, 3, 1, 2|7, 8, 9, 10, 14, 16]$
Iteration 7: $A = [3, 2, 1|4, 7, 8, 9, 10, 14, 16]$
Iteration 8: $A = [2, 1|3, 4, 7, 8, 9, 10, 14, 16]$
Iteration 9: $A = [1|2, 3, 4, 7, 8, 9, 10, 14, 16]$
Iteration 10: $A = [1, 2, 3, 4, 7, 8, 9, 10, 14, 16]$

Quick Sort:

https://www.bilibili.com/video/BV1dmzHYZEBp/?spm_id_from=333.337.search-card.all.click&vd_source=3717cb523f98d837907369c5ba2ca663

Quicksort

Key Take-Aways

- Worst case run time: $\Theta(n^2)$ (the list is already sorted)
  - pivot always the largest/smallest. Everytime we get one empty array and one array of size $n-1$. $T(n) = T(n-1) + \mathcal{O}(n)$
- Best case run time: $\Theta(n \log n)$
  - pivot always median, $T(n) = 2T(n/2) + \mathcal{O}(n)$
- Average/Expected run time: $\Theta(n \log n)$
  - $T(n) = T(an) + T(bn) + \mathcal{O}(n)$, where $a + b = 1$ as we have to run through the full array.
- Quicksort tends to have the smallest constant in front of its runtime
- Quicksort is not stable (although can modify to be stable)
- Quicksort is in-place (although recursion stores stuff on stack, based on exact definition of in-place)
- The idea of a randomized algorithm. Why randomization helps and how to analyze a random algorithm

UNIVERSITY OF TORONTO

Quicksort

```
1: function Partition(A, p, r)
2:     x = A[r]
3:     i = p − 1
4:     for each j = p : r − 1 do
5:         if A[j] < x then
6:             i = i + 1
7:             Swap A[i] ↔ A[j]
8:     Swap A[i + 1] ↔ A[r]
9:     Return i + 1
10: function Randomized-Partition(A, p, r)
11:     i = RAND(p, r)
12:     Swap A[i] ↔ A[r]
13:     Return Partition(A, p, r)
```

```
1: function Quicksort(A, l, r)        ▷ Checking A[l, ..., r]
2:     if l < r then                  ▷ Otherwise no subarray
3:         p =Partition(A, l, r)      ▷ Split the array by p
4:         Quicksort(A, l, p − 1)     ▷ Recurse on ≤ p
5:         Quicksort(A, p − 1, r)     ▷ Recurse on > p
```

If we randomly shuffle input or choose pivot, we reduce the chance of getting the worst case scenario.
The worst case scenario is still $\mathcal{O}(n^2)$, but the chance is lower.

Partition

The array is separated into 4 parts [pivot | $\leq$ pivot | $>$ pivot | unanalyzed]
At each iteration, examine an element in the unanalyzed part
Compare to pivot and put in the appropriate section
$[\boxed{26}|||33, 35, 29, 19, 12, 22]$
$[\boxed{26}||33, 35, 29|19, 12, 22]$ $(19 \leq 26)$
$[\boxed{26}|19|33, 35, 29|12, 22]$ $(12 \leq 26)$
$[\boxed{26}|19, 12|33, 35, 29|22]$ $(22 \leq 26)$
$[\boxed{26}|19, 12, 22|33, 35, 29|]$
Swap
$[22, 19, 12, \boxed{26}, 33, 35, 29]$

Sorting and Searching in Linear time

Counting Sort:

Counting Sort

Key Take-Aways

- Counting sort assumes elements are integers ranging from 0 to k
- Runtime: $\Theta(n + k)$ (if $k = \mathcal{O}(n)$, then runtime is $\Theta(n)$)
- Counting sort uses no comparisons (it uses values of elements to determine the position)
- Counting sort is not in-place
- Counting sort is stable

Note: A stable sorting algorithm is a sorting algo that preserves the relative order of the same value in the previous step

Algorithm

Input: $A[1, ..., n]$, $A[j] \in \{1, ..., k\}$
Output: $B[1, ..., n]$ sorted
Create an auxiliary array $C[1, ...k]$, recording the number of elements in A with value $\leq x$

```
1: function Counting-Sort(A[1, .., n], B[1, ..., n], n, k)
2:     for each i = 0 : k do
3:         C[i] = 0
4:     for each j = 1 : n do
5:         C[A[j]] = C[A[j]] + 1          ▷ # elements in A with value x
6:     for each i = 1 : k do
7:         C[i] = C[i] + C[i − 1]         ▷ # elements in A with value ≤ x
8:     for each j = n : 1 do              ▷ iterate A backwards
9:         B[C[A[j]]] = A[j]              ▷ C[A[j]] implies the sorted position of A[j]
10:        C[A[j]] = C[A[j]] − 1
```

假设要排序数组：

```ini
A = [4, 2, 2, 8, 3, 3, 1]
```

Step 1️⃣ 找出最大最小值

最大值 `max = 8`
最小值 `min = 1`

范围长度：`k = 8 - 1 + 1 = 8`

Step 2️⃣ 建立计数数组 `C[0..k-1]`

初始化全为 0：

```makefile
C = [0, 0, 0, 0, 0, 0, 0, 0]
       ↓
       对应原数组中的值 1~8
```

Step 3️⃣ 统计每个元素出现次数

扫描 `A`：

元素	操作
4	C[3]++ → [0,0,0,1,0,0,0,0]
2	C[1]++ → [0,1,0,1,0,0,0,0]
2	C[1]++ → [0,2,0,1,0,0,0,0]
8	C[7]++ → [0,2,0,1,0,0,0,1]
3	C[2]++ → [0,2,1,1,0,0,0,1]
3	C[2]++ → [0,2,2,1,0,0,0,1]
1	C[0]++ → [1,2,2,1,0,0,0,1]

Step 4️⃣ 累加计数数组（Prefix Sum）

让 `C[i]` 表示"小于等于 i 的元素个数"。

```ini
C = [1, 3, 5, 6, 6, 6, 6, 7]
```

解释：

- 有 1 个 ≤1 的数
- 有 3 个 ≤2 的数
- 有 5 个 ≤3 的数
- 有 6 个 ≤4 的数
- ...
- 有 7 个 ≤8 的数（全部）

Step 5 填充输出数组 B

从原数组的 **最后一个元素往前** 遍历（保证稳定性）：

元素	查C[]	放入B[]	更新C[]
1	C[0]=1	B[1]=1	C[0]=0
3	C[2]=5	B[5]=3	C[2]=4
3	C[2]=4	B[4]=3	C[2]=3
8	C[7]=7	B[7]=8	C[7]=6
2	C[1]=3	B[3]=2	C[1]=2
2	C[1]=2	B[2]=2	C[1]=1
4	C[3]=6	B[6]=4	C[3]=5

得到：

```ini
B = [1, 2, 2, 3, 3, 4, 8]
```

✅ 排序完成！

Radix Sort:

Radix Sort

Key Take-Aways

- Radix sort assumes all elements have $\leq d$-digits
- Runtime $\Theta(d(n+k))$ for d-digit numbers and each digit $\in [0, k]$
- Runtime $\Theta\left(\frac{b}{r}(n+2^r)\right)$ for b-bit numbers and $r = \min(b, \lceil \log n \rceil)$
- Overall, it sorts in $\Theta(n)$
- Radix sort uses no comparisons
- Radix sort is not in-place
- Radix sort is stable

Algorithm

```
1: function Radix-Sort(A, d)
2:     for each i = 1 : d do                    ▷ Sort least sig digit first
3:         Stable sort A on digit i    ▷ Relative order in previous step is preserved
```

Example

3	2	9
4	5	7
6	5	7
8	3	9
4	3	6
7	2	0
3	5	5

7	2	0
3	5	3
4	3	6
4	5	7
6	5	7
3	2	9
8	3	9

7	2	0
3	2	9
4	3	6
8	3	9
3	5	5
4	5	7
6	5	7

3	2	9
3	5	5
4	3	6
4	5	7
6	5	7
7	2	0
8	3	9

Runtime

Lemma

Given n-bit numbers, and any positive integer $r \leq b$. Radix sort sorts in $\Theta\left(\frac{b}{r}\left(n+2^r\right)\right)$, if stable sort is $\Theta(n+k)$.

Let $b = \#$ bits per element, $r = \#$ bits per word, $d = \#$ digits per word, $k = \#$ possible values per digit.
If $d = \left\lceil \frac{b}{r} \right\rceil$ and $k = 2^r$, then $(b, r) \Leftrightarrow (d, k)$

Choose the base $d = \left\lceil \frac{b}{r} \right\rceil$ to use for radix sort
e.g. if $b = 16$, and $r = 2$, we can make $k = r^2 = 2^2 = 4$ possible values and $d = \lceil \frac{16}{2} \rceil = 8$

Choose r to minimize the run time
Intuition: We need to balance the terms $\frac{b}{r}$ and 2^r
If r large, $\frac{b}{r}$ is small and 2^r is large
If r small, $\frac{b}{r}$ is large and 2^r is small

For $\Theta\left(\frac{b}{r}\left(n+2^r\right)\right)$ to be $\Theta(n)$, we need $\frac{b}{r} = \mathcal{O}(1)$ and $2^r = \mathcal{O}(n)$
$r = \mathcal{O}(b)$ and $r = \mathcal{O}(\log n)$, so $r = \mathcal{O}(\min(b, \log n))$

UNIVERSITY OF TORONTO

📘 二、符号说明

符号	含义
`b`	每个元素的总位数（bit 数）
`r`	每次处理的位数（即 radix 的大小）
`d = ⌈b / r⌉`	总共需要多少趟排序（多少"位组"）
`k = 2^r`	每组位的取值范围（即计数排序的桶数）

例如：

- 如果 `b = 16`，`r = 2`，
  那么 `d = 8`，每次 2 位，一共分 8 组；
  每组可能的取值是 `2^2 = 4` 种（00, 01, 10, 11）。

⚙️ 三、Radix Sort 的时间复杂度推导

Radix Sort 的每一趟（对一组 r 位的排序）要花：

> Θ(n + k) 时间
> （因为内部使用 Counting Sort）

而总共有 `d = b/r` 趟。
所以总时间为：

$$T(n) = \Theta\left(\frac{b}{r}(n+2^r)\right)$$

🧱 五、找到最佳 r

我们希望：
1️⃣ $\frac{b}{r} = O(1)$，也就是 r 不太小；
2️⃣ $2^r = O(n)$，也就是 r 不太大。

第二个式子 $2^r = O(n)$ → 得出

$$r = O(\log n)$$

结合起来：

$$r = O(\min(b, \log n))$$

✅ 六、最终结论

> 当 `r = θ(log n)` 时，Radix Sort 的时间复杂度为：

$$T(n) = \Theta(n)$$

换句话说：

- 对"字长 b 固定"的机器（例如 b = 32 位整数）
  → Radix Sort = **线性时间 O(n)**。
- 对超大数（b 很大）
  → 时间 = Θ((b/log n) * n)。

这一页解释了 **Radix Sort 的运行时间在不同输入规模下（b 与 log n 的关系）** 的两种情况，说明了当数字位数多或少时，最优的 `r`（每次处理的位数）不同，
最终得出结论：

$$T(n) = \begin{cases} \Theta(n), & \text{if } b < \log n, \\ \Theta\left(\frac{bn}{\log n}\right), & \text{if } b \geq \log n. \end{cases}$$

BSTs

Definition

BST properties:

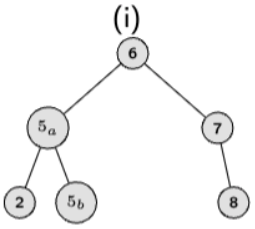
- If y is in the left subtree of x, then key$[y] \leq$ key$[x]$
- If y is in the right subtree of x, then key$[y] \geq$ key$[x]$

Key Take-Aways

- In general, if a BST has n nodes, $h = \mathcal{O}(n)$. Only if the BST is balanced, $h = \mathcal{O}(\log n)$
- Minimum/Maximum and searching for any arbitrary key $\mathcal{O}(h)$
- Successor/Predecessor $\mathcal{O}(h)$
- Insertion/Deletion $\mathcal{O}(h)$
- Build-BST: $\mathcal{O}(n^2)$ worst case (chain)

Basic operations

(i)



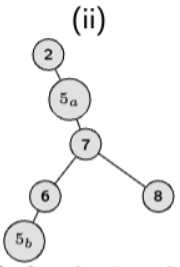

```
1: function In-Order(x)
2:     if x ≠NIL then
3:         In-Order(x.left)
4:         Process x
5:         In-Order(x.right)
```

(i)2, 5_a, 5_b, 6, 7, 8
(ii)2, 5_a, 5_b, 6, 7, 8
Used for sorting

```
1: function Pre-Order(x)
2:     if x ≠NIL then
3:         Process x
4:         Pre-Order(x.left)
5:         Pre-Order(x.right)
```

(i)6, 5_a, 2, 5_b, 7, 8
(ii)2, 5_a, 7, 6, 5_b, 8
Used for rotation

(ii)




```
1: function Post-Order(x)
2:     if x ≠NIL then
3:         Post-Order(x.left)
4:         Post-Order(x.right)
5:         Process x
```

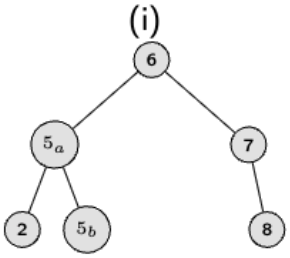
(i)2, 5_b, 5_a, 8, 7, 6
(ii)5_b, 6, 8, 7, 5_a, 2
Used for deleting

$\mathcal{O}(n)$, since we process each node exactly once

UNIVERSITY OF TORONTO

Basic operations

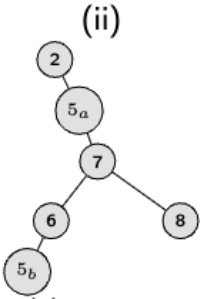
(i)




```
1: function Minimum(x)
2:     while x.left ≠ NIL do
3:         x = x.left
4:     Return x
```

(i)$6 \to 5_a \to 2$
(ii)2

(ii)




```
1: function Maximum(x)
2:     while x.right ≠ NIL do
3:         x = x.right
4:     Return x
```

(i)$6 \to 7 \to 8$
(ii)$2 \to 5 \to 7 \to 8$

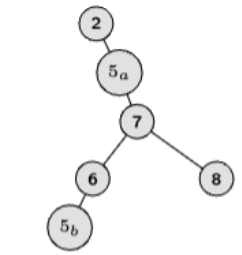
$\mathcal{O}(h)$, since in the worst case we have to traverse the full height

Predecessor(x) 是指 在二叉搜索树（**BST**）中，键值比 **x** 小的节点里最大的一个。

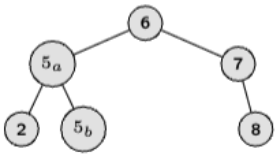
Basic operations

```
1: function Predecessor(x)
2:     if x.left ≠ NIL then
3:         Return Maximum(x.left)
       y =parent(x)
4:     while y ≠ NIL and x == y.left do
5:         x = y
6:         y =parent(y)
7:     Return y
```

$\mathcal{O}(h)$, since in the worst case we have to traverse the full height



Predecessor$(5_b) = 5_a$



Predecessor$(6) = 5_b$

Predecessor(x) 是 BST 中比 x 小的最大节点。
如果有左子树 → 左子树中最大节点；
如果没有左子树 → 向上找第一个让你在“右边”的祖先。

Insert

```
1: function Insert(T, z)
2:     y=NIL, x = T.root
3:     while x ≠NIL do
4:         y = x                    ▷ Find where z should connect
5:         if z.key < x.key then
6:             x = x.left
7:         else
8:             x = x.right
9:     parent(z)=y
10:    if y==NIL then
11:        T.root = z
12:    else if z.key < y.key then
13:        y.left=z
14:    else
15:        y.right=z
```

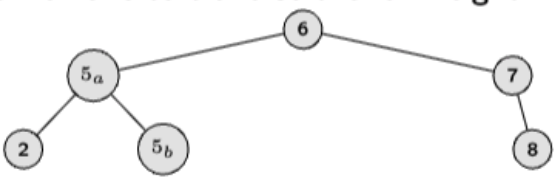
$\mathcal{O}(h)$, since in the worst case we have to traverse the full height

Build a BST from $[6, 5, 7, 5, 8, 2]$

Search

```
1: function Search(x, k)
2:     if x==NIL or x.key == k then
3:         Return x
4:     if x.key > k then
5:         Return Search(x.left, k)
6:     if x.key < k then
7:         Return Search(x.right, k)
```

$\mathcal{O}(h)$, since in the worst case we have to traverse the full height



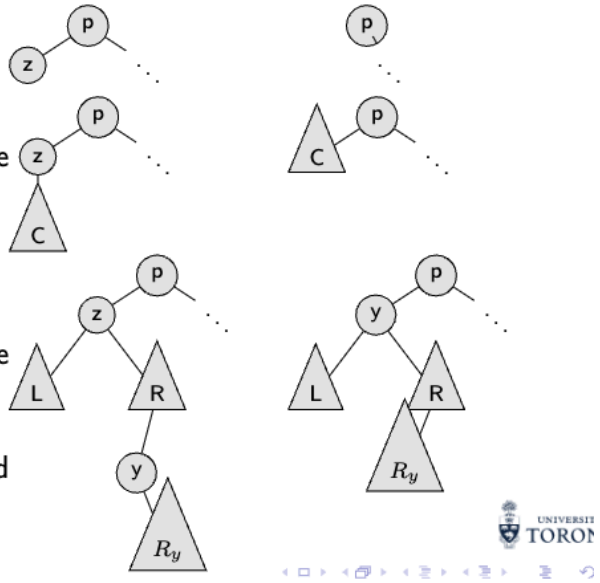
Delete

Successor(x) 是指 在二叉搜索树（BST）中，键值比 x 大的节点里最小的一个。

Case 1: z is a leaf, delete it.

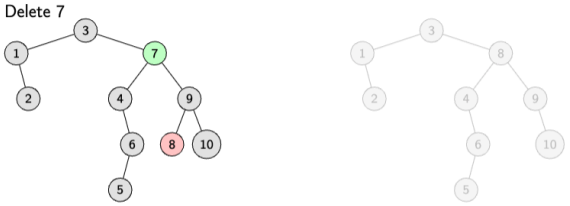
Case 2: z has one child, delete z and replace with child.

Case 3: z has 2 children.
Let y be z's successor, y never has a left subtree by defintion.
If y has no children, replace z with y
If y has a right subtree, replace z with y and replace y with right child



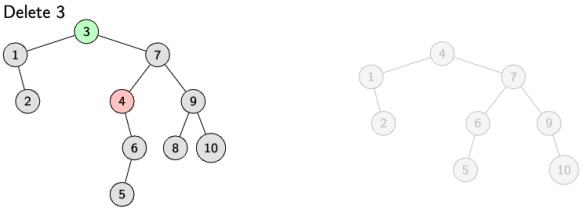
Example

Delete 7



Example

Delete 3



Example

Show if a BST node has 2 children, then the successor has no left child and the predecessor has no right child

Proof: Assume that node x has 2 children, and its successor s (minimum element of the BST rooted at x.right) has left child l, s.key $>$ l.key $>$ x.key, then l is the successor of x. Contradiction
Similarly, if its predecessor p (maximum element of the BST rooted at x.left) has right child r, we have x.key $>$ r.key $>$ p.key, r is the predecessor of x. Contradiction.

BST-sort works by constructing a BST out of the array and then calling In-Order traversal. What is the best and worst case?

Solution: In-Order always take $\Theta(n)$

Worst case: array is already sorted, BST is a linked list

Each BST at position i insert takes $\mathcal{O}(i)$, summing up gives $\sum_{i=1}^{n} \mathcal{O}(i) = \mathcal{O}(\sum_{i=1}^{n} i) = \mathcal{O}(n^2)$

Best case: BST is always perfectly balanced, height is always $\mathcal{O}(\log i)$

$$\sum_{i=1}^{n} \mathcal{O}(\log i) = \mathcal{O}(\sum_{i=1}^{n} \log i) = \mathcal{O}(\sum_{i=1}^{n} \log n) = \mathcal{O}(n \log n)$$

Red-Black Trees (RBTs)

左 < 根 < 右；根&叶 黑色；红色不连续；各路径黑色数量相同。

Defintion

Red Black Tree is a Binary Search Tree with the following additional properties:

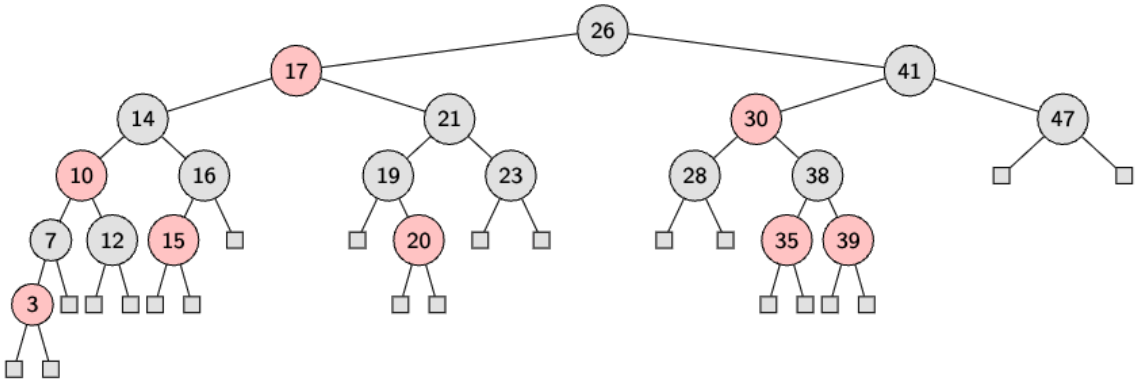
- Every node is either red or black
- The root is black
- All leaves (NIL) are black
- If a node is red, both its children are black
- For all nodes, all paths to all leaves have the same black-height (number of black nodes on path to a leaf, not including itself, but including NIL)

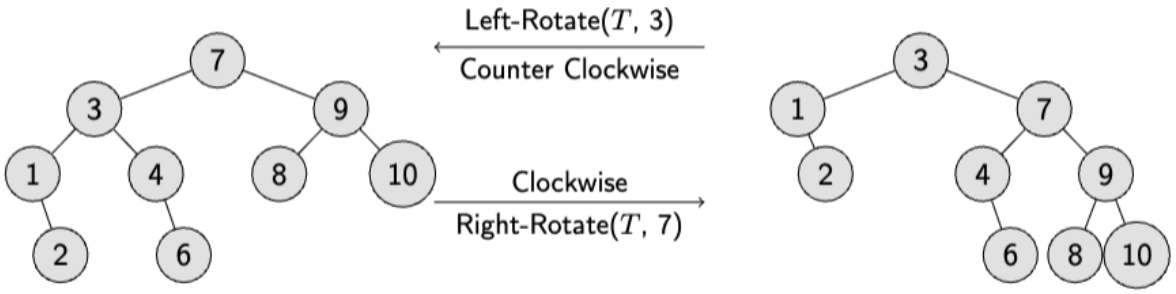
Key Take-Aways

- RB Treees are balanced $h = \mathcal{O}(\log n)$
- All read-only operations (e.g. traversals) are all the same as BST
- Rotation

UNIVERSITY OF TORONTO

Example





Hashing

(a) Demonstrate the insertion of keys 8, 15, 42, 17, 30, 27, 26, 33, 14, 24 into a doubly-hashed table where collisions are resolved with open addressing. Let the table have 11 slots, let the *primary* hash function be $h_p(k) = k \mod 11$ and let the *secondary* hash function be $h_s(k) = 3k \mod 4$.

2(a)

∵ 11 slots ∴ By the requirement, address is from 0 to 10

Put numbers by using double hashing and open addressing as defined

8: $hp(8) = 8 \mod 11 = 8$ → no collision → put 8

15: $hp(15) = 15 \mod 11 = 4$ → no collision → put 4

42: $hp(42) = 42 \mod 11 = 9$ → no collision → put 9

17: $hp(17) = 17 \mod 11 = 6$ → no collision → put 6

30: $hp(30) = 30 \mod 11 = 8$ → collision with $hp(8)$

→ $hs(30) = 3 \times 30 \mod 4 = 2$

→ 1st explore: $hp(8) + 1 \cdot hs(8) = 8 + 2 = 10$ → no collision → put 10

27: $hp(27) = 27 \mod 11 = 5$ → no collision → put 5

26: $hp(26) = 26 \mod 11 = 4$ → collision with $hp(15)$

→ $hs(26) = 3 \times 26 \mod 4 = 2$

$hp(26) + i \cdot hs(26)$ → 1st explore: $(4 + 1 \cdot 2) = 6$ → collision with $hp(17)$

→ 2nd explore: $4 + 2 \times 2 = 8$ → collision with $hp(8)$

→ 3rd explore: $4 + 3 \times 2 = 10$ → collision with $hp(30)$

→ 4th explore: $4 + 4 \times 2 = 12$ → $12 \mod 11 = 1$

→ no collision → put 1

33: $hp(33) = 33 \mod 11 = 0$ → no collision → put 0

14: $hp(14) = 14 \mod 11 = 3$ → no collision → put 3

24: $hp(24) = 2 \mod 11 = 2$ → no no collision → put 2

∴ Therefore, by this process, the final hash table is:

index	0	1	2	3	4	5	6	7	8	9	10
key	33	26	24	14	15	27	17	idle	8	42	30

FOR HASHING SUMMARY, SEE P71

Dynamic Programming (DP)

当问题同时具备“最优子结构”和“重叠子问题”特性时，DP 是最优解法。

Overview

Simple defintion: Efficient recursion for solving well-behaved optimization problems

Optimization problems: trying to find an optimal solution given some constraints (often discrete problems)

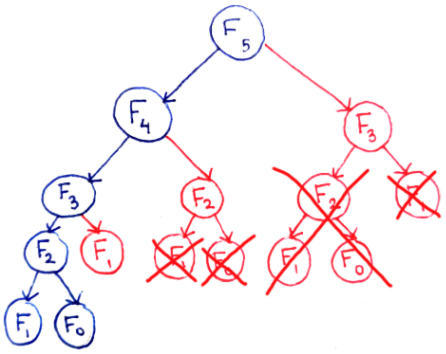
Used for problems with the following properties

- Optimal substructure: optimal solutions incorporate optimal solutions to related subproblems
- Overlapping subproblems: solving the same subproblems over and over again (Memorization exploits this redundancy)

In general, DP good for tasks with small but repetitive search spaces

Examples

Fibonacci series: $F_n = F_{n-1} + F_{n-2}$ with $F_1 = F_2 = 1$. Each value F_i needs to be solved as subproblem for F_{i+1} and F_{i+2}. Similarly, Pascal's triangle.



String/Array algorithms: longest common subsequence, longest increasing subsequence, longest common substring, edit distance, interleaving strings, balanced array, etc.

Graph algorithms: Bellman-Ford (to be discussed later in the course).

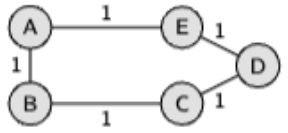
Backpropagation: compute the gradients for one layer at a time starting from the last layer. When calculating gradients for layer i, use the result from layer $i + 1$.

Viterbi algorithm: with a hidden markov model defined by $(S,\ O,\ P(s_{t=0} = s_i),\ P(s_{t+1} = s_j|s_t = s_i),\ P(O_t|s_t = s_i))$, find the most probable state path given the output path. Used for part-of-speech tagging, speech recognition etc. in NLP.

Anti-Examples

Sorting an array of n elements: not an optimization problem.

Longest path problem: find longest path between nodes in a graph without repeating an edge.
This problem does not exhibit optimal substructure.



LongestPath(A,D)=$A \to B \to C \to D$
LongestPath(A,B)=$A \to E \to D \to C \to B$
LongestPath(B,D)=$B \to A \to E \to D$

Compute $n!$ can be done with recursion:

$$\text{fact}(n) = \begin{cases} 1, \text{ if } n = 0 \\ n \cdot \text{fact}(n-1), \text{ if } n > 0 \end{cases}$$

There are no overlapping subproblems, once we compute fact(i), we don't need to compute it again.

UNIVERSITY OF TORONTO

Rod Cutting Problem:

Given: a rod of length n and a table of prices p_i for each rod length $i = 1, ..., n$
Goal: Determine maximum revenue you can obtain from the rod by cutting it into various size pieces and selling them

Visualization:
Suppose we have a rod of length 4 with prices:

length	1	2	3	4
price ($)	1	5	8	9

Possible cuts:
$[1,1,1,1]$: \$4; $[2,2]$: \$10 (optimal);
$[1,3]$: \$9; $[1,1,2]$: \$7; $[4]$: \$9

Naive solution: Try all possible cuts.
This will be $\mathcal{O}(2^n)$, since you either cut or not (2 possibilities for $n-1$ spots)

Rod Cutting Problem with DP

Optimal Substructure:
Suppose in the optimal solution, we make a cut at position k and we have 2 rods
The optimal solution must now contain the optimal way to cut the 2 sub-rods. If not, we could replace them with the optimal sub-solution to obtain a better optimal solution.
$\therefore$ this problem exhibits optimal substructure

Recursive relationship among subproblems:
Let r_n be the maximum revenue you can get from a rod of length n
$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1) = \max_{k \in [1,n]} (r_k + r_{n-k})$
We can simplify things. Instead of considering 2 subproblems, we fix the length of the rod on one side. $r_n = \max_{k \in [1,n]} (p_k + r_{n-k})$

Rod Cutting Problem Implementation

Naive implementation:

function Rod-Cut(p, n)
  if $n == 0$ **then**
    Return 0
  $q = -\infty$
  for each $i = 1 : n$ **do**
    $q = \max(q, p[i] + \text{Rod-Cut}(p, n - i))$

However, we can make it more efficient by adding a memory. This gives a top-down approach.

DP with bottom up approach ($\mathcal{O}(n^2)$):

function Rod-Cut(p, n)
  $r[0, ..., n] = 0$
  for each $j = 1 : n$ **do**
    $q = -\infty$
    for each $i = 1 : j$ **do**
      $q = \max(q, p[i] + r[j - i])$
    $r[i] = q$

Example run of Bottom-up approach

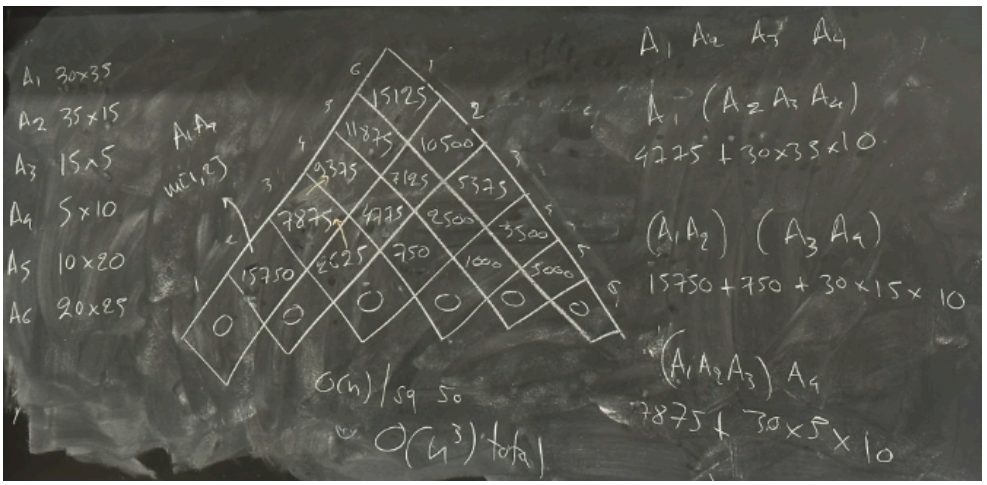
length	1	2	3	4
price ($)	1	5	8	9

p_i=price of rod with length i
r_i=optimal revenue of rod with length i

$j = 1$: $p_1 = 1$, $r_1 = \max(1) = 1$
$j = 2$: $p_2 = 5$, $p_1 + r_1 = 1 + 1 = 2$, $r_2 = \max(5, 2) = 5$
$j = 3$: $p_3 = 8$, $p_1 + r_2 = 1 + 5 = 6$, $p_2 + r_1 = 5 + 1 = 6$, $r_3 = \max(8, 6, 6) = 8$
$j = 4$: $p_4 = 9$, $p_1 + r_3 = 1 + 8 = 9$, $p_2 + r_2 = 5 + 5 = 10$, $p_3 + r_1 = 8 + 1 = 9$,
$r_4 = \max(9, 9, 10, 9) = 10$

Matrix Multiply Problem:

matrix multiplication in lecture:



Greedy Algorithm

2. 什么时候能用？ (Properties) 不是所有问题都能用贪心。能用的问题通常有两个特征：

- **Greedy Choice Property:** 你做出的第一个贪心选择，一定包含在某个最优解里（也就是第一步走对了，后面就不怕了）。
- **Optimal Substructure:** 你做完第一步选择后，剩下的问题依然是一个同类型的、规模更小的子问题。

Overview

Simplified Definition: an algorithm where locally optimal decisions lead to a globally optimal solution
Idea: when making a choice, take the one that looks the best right now. Local optimality leads to global optimality.
Note: Greedy is not always optimal, but good as approximation algorithms

Properties:

- Greedy choice property: the optimal solution agrees with the first greedy choice
- Smaller subproblems: after making the greedy choice, the resulting subproblem reduces in size
- Optimal substructure: an optimal solution contains optimal solutions to subproblems

A simple example where Greedy is not optimal

Suppose we want to make up x using coins from a given set S of denominations.

Case 1: $S = [1, 2, 4]$. This set is valid, since we can make either even (combinations of 2 and 4) or odd numbers (any even + 1). We can use greedy here, sort the coins from largest to smallest, then always use the largest coin first if not overshooting the target.

Case 2: $S = [1, 4, 5]$. Still valid, since we can make up anything just using \$1. However, greedy won't work here. e.g. if we want \$8, the most optimal way is \$4+\$4. The greedy gives us \$5+\$1+\$1+\$1.

There are only two types of greedy problems you need to consider in this course:

1. Car building: rearrange the full set of tasks to minimize the total penalty
2. Scheduling: find a feasible subset of tasks to maximize the profit

"重排顺序最小惩罚"（Car Building）和 "选取子集最大收益"（Scheduling）。

You can use these two examples as templates for the exams and HWs. The algorithms and proofs should be very similar.

Car building:

> "任务 A 每拖一小时罚 3 块，任务 B 每拖一小时罚 5 块。"

惩罚不是固定金额，而是**按完成时间乘以惩罚系数算的。**

🧱 二、惩罚计算公式（重点）

若我们执行顺序为：

$$J_1, J_2, ..., J_n$$

则每个任务的惩罚 = 它的权重 × 它完成时刻。

$$\text{Total Penalty} = \sum_{i=1}^{n} w_i \times C_i$$

其中：

$$C_i = t_1 + t_2 + ... + t_i$$

就是前面任务的执行时间总和（当前任务结束时刻）。

每个任务的"单位惩罚时间代价"不同，
直觉上：

> 如果某任务需要很长时间（t大），但惩罚又不高（w小），那它性价比低，应该靠后。

所以要最小化：

$$\sum w_i C_i$$

的最优排序是：

$$\boxed{\text{按 } \frac{t_i}{w_i} \text{ 升序排列执行。}}$$

Scheduling (**No Value Version**):

你有一堆任务（或会议、活动），每个都有：

- 开始时间：s_i
- 结束时间：f_i
- 可能还有价值（profit）：v_i

要求从中选出一个**互不重叠**的子集，使得：

- 若任务价值相同 → 选最多的；
- 若任务价值不同 → 总收益最大。

```
def Greedy_Schedule(jobs):
    sort jobs by finish time f_i ascending
    S = []
    last_finish = -∞
    for job in jobs:
        if job.start >= last_finish:
            S.append(job)
            last_finish = job.finish
    return S
```

Scheduling 型问题（调度类贪心）最优解的核心是：

> **"按结束时间早的任务优先"。**
> 也就是说：

$$\boxed{\text{按 } f_i \text{（任务结束时间）升序排序，依次选择不冲突的任务。}}$$

这个策略能保证你选出的任务集合具有**最大数量（或最大收益）**，并且是**全局最优。**

💡 二、贪心思路（为什么按结束时间排序）

假设你当前想排一整天的会议。

直觉上：

> 如果你选了一个很晚才结束的会议，它会挡住后面更多的可能；
> 如果你选一个早结束的，你有更大空间安排后续的会议。

⚙️ 举例

活动	开始 s	结束 f
A	1	4
B	3	5
C	0	6
D	5	7
E	8	9

🔢 步骤 1：按结束时间排序

A(4), B(5), C(6), D(7), E(9)

🔁 步骤 2：扫描选择

初始：空集

- A ✅ (结束于 4)
- B ❌ (3 < 4，有冲突)
- C ❌ (0 < 4，有冲突)
- D ✅ (5 ≥ 4)
- E ✅ (8 ≥ 7)

结果：**{A, D, E}**

🧠 思考：为什么这样选最优?

假设存在一个最优集合 S，它没选最早结束的活动 A，而是选了别的活动 X。
那么由于 A 结束得更早，把 X 替换成 A，不会导致冲突，还可能留出更多空间安排别的活动。
→ 所以，存在另一个不劣的最优解包含 A。

Proof of Correctness: 任务集合 T, 最优解是 J, 贪心算法输出是 G

Notation:
Optimal solution: $J = j_1, j_2, ..., j_n$
Greedy solution: $G = g_1, g_2, ..., g_n$
j_i and g_i are indices into some object set. If the task set $T = \{t_1, t_2, t_3\}$ and $J = j_1, j_2, j_3$, the optimal ordering is $t_{j_1}, t_{j_2}, t_{j_3}$
We treat J and G as ordered sets and use set operations $J \backslash \{j_1\} = j_2, j_3, ..., j_n$, $\{j_0\} \cup J = j_0, j_1, ..., j_n$

Proof of correctness:

1. let $G = g_1, g_2, ..., g_n$
2. First greedy choice: $\exists J = j_1, ..., j_n$ s.t. $g_1 = j_1$
3. Smaller subproblems: After making choice $j_1 = g_1$, we are left with a smaller subproblem
4. Optimal substructure: we must solve the subproblem optimally for the original solution to be optimal. $J \backslash \{j_1\} = j_2, j_3, ..., j_n$ must be an optimal solution to the subproblem
5. Recurse the argument to conclude $j_2 = g_2$, $j_3 = g_3$

$\therefore \exists J$ optimal solution s.t. $J = G$

UNIVERSITY OF TORONTO

🧩 Step 1：定义两个解

- $G = [g_1, g_2, g_3, \ldots]$：贪心算法选的活动集合；
- $J = [j_1, j_2, j_3, \ldots]$：最优解（最多活动数的集合）。

🧩 Step 2：证明贪心的第一个选择是"安全的"

> 贪心算法选的第一个活动 g_1（即最早结束的活动）
> 一定可以出现在某个最优解 J 里。

证明思路：

- 假设最优解 J 的第一个活动是 j_1，
  但它结束得比 g_1 晚（因为贪心选的是最早结束的）。
- 那么你可以把 J 里的 j_1 换成 g_1，
  不会造成任何冲突，还能留出更多时间。
- 所以存在另一个最优解 J'，其中 $j'_1 = g_1$。

✅ **结论：**
g_1 一定在某个最优解中，也就是说它是"安全的第一步"。

🧩 Step 3：形成子问题

既然 g_1 一定没问题，
那我们可以"移除所有与 g_1 冲突的活动"，
得到一个新的子问题：

> "在剩下不冲突的活动里，再选最多个活动。"

这就对应课件那句：

> "After making choice $j_1 = g_1$, we are left with a smaller subproblem."

🧩 Step 4：最优子结构

假设原问题的最优解是 J，

那它包含 g_1 后，剩下部分 $J \setminus \{g_1\}$
也必须是"子问题"的最优解。

换句话说：

> 如果整体最优解包含贪心第一步，
> 那它剩下的部分也必须最优。

否则你可以在剩下部分换成更好的组合，得到更优整体解——矛盾。

🧩 Step 5：递归证明

现在：

- 你知道第 1 步没错；
- 假设前 k 步都选对；
- 第 k+1 步同样可以用同样的逻辑推出；
  → 所以最终每一步都是对的。

结论：

$$\exists J \text{ such that } J = G$$

也就是说——贪心解和最优解完全一致。

🧠 直观理解

- 贪心第一步（选最早结束的活动）不会错；
- 选完它之后，剩下问题还是同样结构；
- 一直递归选下去；
- 每一步都不亏，整体一定最优。

✅ **一句话总结：**

> "贪心算法正确性"的标准证明流程：
> 1. **"第一步选择（greedy choice）不会破坏最优性；"**
> 2. **"剩下问题仍满足最优子结构；"**
> 3. **"递归地重复 → 贪心解 = 最优解。"**

就像"活动选择问题"这样，按结束时间选，
是因为你能用这套五步逻辑**严格证明它不会比最优解差。**

- ✅ **Greedy choice property（贪心选择性质）**
  → "第一步贪心选择可以出现在某个最优解中"。

现在这页是在证明第二个条件：

- ✅ **Optimal substructure（最优子结构）**
  → "拿掉第一步后，剩下的问题仍然必须是最优的"。

只有这两个都成立，贪心算法才整体正确。

Process - easy:

Given: $V = \{v_1, v_2, ..., v_n\}$ set of vehicles, $Q = \{p_1, p_2, ..., p_n\}$ set of corresponding penalties per day vehicle v_i is late.
Currently, all vehicles are late and can only produce one vehicle per day.
Goal: Compute optimal schedule for building vehicles to minimize your total penalty.
Suppose your schedule is $S = s_1, s_2, ..., s_n$, total penalty is $P(S) = \sum_{i=1}^{n} ip_{s_i}$

Devise a greedy algorithm:
Sort Q in descending order, build vehicles from highest penalty to lower. $\mathcal{O}(n \log n)$

```
def BuildCars(penalties):
    sort penalties in descending order
    total_penalty = 0
    for i in range(1, n+1):
        total_penalty += i * penalties[i]
    return total_penalty
```

Prove the greedy choice property:
let $J = j_1, j_2, ..., j_n$ be an optimal solution
Let $G = g_1, g_2, ..., g_n$ be the greedy solution
If $j_1 = g_1$, then done
Suppose $j_1 \neq g_1$
Since we must build all vehicles in the end, g_1 must appear somewhere in J
i.e. $J = j_1, j_2, ..., j_{m-1}, j_m = g_1, ..., j_n$
By definition of greedy choice, $p_{j_1} \leq p_{g_1} = p_{j_m}$ for some $m > 1$
Let J' be a solution where we swap j_1 and j_m, $J' = j_m, j_2, ..., j_1, ..., j_n$

$$P(J) = \sum_{i=1}^{n} i p_{j_i} = 1p_{j_1} + 2p_{j_2} + \cdots + mp_{j_m} + \cdots + np_{j_n}$$

$$P(J') = \sum_{i=1}^{n} i p_{j_m} = 1p_{j_m} + 2p_{j_2} + \cdots + mp_{j_1} + \cdots + np_{j_n}$$

$$P(J') = P(J) - 1p_{j_1} - mp_{j_m} + 1p_{j_m} + mp_{j_1} = P(J) + (m-1)(p_{j_1} - p_{j_m}) \leq P(J)$$

J' is an optimal solution that agrees with the first greedy choice

Prove the problem is reduced to a smaller subproblem after making the first greedy choice:
After scheduling a vehicle on day 1, we have $n-1$ vehicles to schedule in $n-1$ days. Same problem but smaller

How to prove the two props:

证明 Greedy Choice + 证明 Optimal Substructure 的根本意义，就是为了证明——这个贪心算法一定能得到最优解。

🧩 一、Greedy Choice Property（贪心选择性质）

🎯 目标：

证明"贪心算法的第一步选择"一定可以出现在某个最优解中。
👉 也就是"这步是安全的（safe choice）"。

🧠 思路：

用 **交换论证（Exchange Argument）**。

✍️ 通用证明模板：

> 假设存在一个最优解 $J = [j_1, j_2, ..., j_n]$，
> 但贪心算法的第一个选择 $g_1 \neq j_1$。
> 由于 g_1 是贪心算法选出的"最好"的元素，
> 所以有 $\text{value}(g_1) \geq \text{value}(j_1)$。
>
> 于是我们交换 g_1 和 j_1，构造一个新解 J'：
>
> $$J' = [g_1, j_2, ..., j_n]$$
>
> 计算总代价（或总收益）后可得：
>
> $$\text{Cost}(J') \leq \text{Cost}(J)$$
>
> 或
>
> $$\text{Profit}(J') \geq \text{Profit}(J)$$
>
> 所以 J' 仍然是最优的。
> 因此，贪心算法的第一步选择 g_1 一定可以出现在最优解中。

🧩 二、Optimal Substructure（最优子结构性质）

🎯 目标：

证明"做出贪心选择后，剩下的子问题仍是最优的"。

🧠 思路：

用 **反证法（Proof by Contradiction）**。

✍️ 通用证明模板：

> 假设我们已经做出第一步贪心选择 g_1，
> 并得到剩余子问题的最优解 J'。
>
> 假设 J' 不是最优的，
> 那么存在一个更好的子问题解 S，使得：
>
> $$\text{Cost}(S) < \text{Cost}(J')$$
>
> 我们把 g_1 和 S 合并成一个新解：
>
> $$J'' = \{g_1\} \cup S$$
>
> 得到：
>
> $$\text{Cost}(J'') < \text{Cost}(J)$$
>
> （或 Profit 更大）
>
> 这与"J 是最优解"矛盾。
> 因此，J' 必然是子问题的最优解。

Process - Scheduling Deadlines:

Scheduling Deadlines

Given $T = \{t_1, t_2, ..., t_n\}$ a set of tasks, $D = \{d_1, ..., d_n\}$ a set of corresponding deadlines in units of days, $Q = \{p_1, ..., p_n\}$ a set of corresponding profits if we complete the task by the deadline.
Each task takes exactly 1 day to complete. A feasible set is a set of tasks s.t. it is possible to complete all tasks before their deadline
Goal: Find a feasible set of tasks s.t. their profits $\sum_{j \in J} p_j$ are maximized.

e.g.

i	1	2	3	4
t_i	a	b	c	d
d_i	2	1	2	1
p_i	50	10	15	30

Invalid solution: $\{b, d\}$, both needs to completed on day 1
Solution 1: $\{d\}$, profit 30
Solution 2: $\{d, a\}$, profit 30 + 50

Devise a greedy algorithm assuming you have a function Feasible(J) that runs in constant time and check if J is a feasible schedule
Sort tasks in order of decreasing profit. $J = \{\}$, iterate through the tasks, if Feasible($J \cup \{j\}$), then $J = J \cup \{j\}$.

Prove the greedy choice property:
Let $J = j_1, j_2, ..., j_m$ be an optimal solution.
Let $G = g_1, g_2, ..., g_m$ be the greedy solution.
Note that the sequence of selection doesn't matter here. We need to finish all tasks in J and all tasks in G before the corresponding deadlines. If $g_1 \in J$, then done
Assume $g_1 \notin J$, then $p_{g_1} \geq p_j$, $\forall j \in J$ by construction of g_1
Construct $J' = J \backslash \{j\} \cup \{g_1\}$ for any $j \in J$ s.t. J' is feasible.
$\therefore P(J') = P(J) - P(\{j\}) + P(\{g_1\}) = P(J) - p_j + p_{g_1} \geq P(J)$
Thus J' is an optimal solution that agrees with our first greedy choice.

UNIVERSITY OF TORONTO

Prove the problem is reduced to a smaller subproblem after making the first greedy choice:
Start with $T = \{t_1, ..., t_n\}$, make first greedy choice g_1 with profit p_{g_1} where $p_{g_1} \geq p_k, \forall k \in \{1, ..., n\}$
Now, we have $T \backslash \{t_{g_1}\}$ possible tasks remaining, but with less days to complete them
So we have a subproblem T'.
Also, there are possibly some tasks that is now impossible to complete because we don't have enough days and we can remove them. $T' \subset T \backslash \{t_{g_1}\}$
$|T'| \leq |T| - 1 < |T|$, T' is a strictly smaller problem

Prove optimal substructure:
Let $J = j_1, j_2, ..., j_m$ s.t. $g_1 \in J$ be an optimal solution
Let $J' = J \backslash \{g_1\}$ be a solution to the subproblem
Assume that J is optimal and J' is not
Let S be an optimal solution to the subproblem $P(S) > P(J')$ and construct $J'' = \{g_1\} \cup S$. J'' is feasible because we can just do g_1 first and S is feasible for the subproblem.
$P(J'') = P(S) + P(\{g_1\}) > P(J') + P(\{g_1\}) = P(J)$. Contradiction, since J'' is now more optimal.

```
# J = 已选择的任务集合
J = ø
# Step 1: 按利润从大到小排序任务
Sort tasks by decreasing profit

# Step 2: 遍历任务
for each task j in sorted order:
    if Feasible(J ∪ {j}):    # 加进去后不违反截止时间
        J = J ∪ {j}          # 保留任务
```

Prove problem is reduced to a smaller subproblem after making the first greedy choice 短答

✍️ 三、考试怎么写最简版

一句话就够：

> "After making the first greedy choice g_1, the remaining tasks form a smaller instance of the same problem (same constraints, same objective), thus we can recursively apply the greedy rule."

或者更口语一点：

> "Once we fix the first greedy choice, what's left is the same scheduling problem but smaller."

在证明 **Greedy choice property** 时，
你只需要证明 ——**g_1**（第一次贪心选择）可以出现在某个最优解中，
而 不要求它一定在最优解的第一位。

2. **Algorithm Design, 15+10 points**

(a) **Algorithm**: To find a and b, we first insert the largest element of each array into a max-heap. While inserting keep track of the smallest element placed in the max-heap (the minimum of the maximum elements).

Obtain the maximum element in the heap (at the root) and the minimum element in the heap (found in the previous step). Calculate and store the difference between these two elements in the variable `range`.

Extract the maximum from the heap, and replace it with the next largest element in the list it came from (then re-heapify). If necessary, update the stored value for the smallest element in the heap and the `range`.

If the new `range` is smaller than any previous range, store that value in `minimum_range` along with the value of the corresponding elements. (i.e. `minimum_range` = min(`range`, `minimum_range`))

Repeat the above two steps until one of the arrays is empty. Return `minimum_range` and the corresponding elements that produces this range.

Runtime: This algorithm requires building a k-element max-heap, as well as removing/inserting $n \cdot k$ elements from/into that heap. Therefore the overall runtime is $O(k + n \cdot k \cdot \lg k) = O(n \cdot k \cdot \lg k)$.

Correctness: This algorithm can be incorrect in two possible ways. *(i)*, a, and b do not satisfy that the range between them contains at least one element from every list, or *(ii)*, $b - a$ is not minimized.

(i): Any possible selection for a and b are at some point the minimum and maximum elements in a max-heap containing one element from each array. Therefore, it must be the case that there is at least one element, c, in each array such that $a \leq c \leq b$, for any possible a, b under consideration. Therefore the algorithm will not be incorrect in this way.

(ii): If $b-a$ is not minimal, there must be some b^*, a^* such that $b^* - a^*$ is minimal. *ATaC* that $b^* - a^*$ is minimal, but our algorithm did not choose them. Consider that because we only remove one element at a time, and because b^* is greater than at least one element in each of the other lists, b^* must have been the root of our max heap at some point over the course of the algorithm. Given that we did not select a^*, b^*, it must be that a^* was never the minimum element in the heap when b^* was the root. However we know that at least one element from each list falls between a^* and b^*, and those elements would not have been removed from the heap before b^*, therefore if a^* was present in the heap with b^*, it must have been the minimum element. Therefore a^* must have never been in the heap with b^*. However, since we know that every other list contains some element between a^* and b^*, and whatever element from the list of a^* that was on the heap with b^* is greater than a^*, then whatever was the minimum element on the heap at this time would be greater than a^* while still, together with b^*, forming a range containing at least one element from each list. However, this contradicts our assumption that $b^* - a^*$ is minimized.

(b) The most efficient way to perform this algorithm is the same as above, however the lists must first be sorted. This takes $O(k \cdot n \lg n)$ time using heapsort. Therefore, in total, the algorithm will take $O(n \cdot k \cdot \lg k + k \cdot n \lg n)$. We did not specify that $k < n$, so this should not be simplified further.

3. **Dynamic Programming, 10+15 points**

(a) On the string `AAAABA`, the greedy algorithm would produce `AAAA|B|A`, but the optimal solution is `AAA|ABA`.

(b) The key idea is to isolate the last palindrome in the string, check all possible last palindromes, and then use the optimal substructure to obtain the number of cuts required for the rest of the string.

Let $C[j]$ be the minimum number of cuts required to partition the substring $s_{1,j}$ into palindromes.

$$C[j] = \begin{cases} 0 & \text{if } s_{0,j} \text{ is a palindrome} \\ \min\limits_{\substack{0 \le k < j \\ s_{k+1,j} \text{ is a palindrome}}} \Big(C[j],\ 1 + C[k] \Big) & \text{if } s_{0,j} \text{ is not palindrome} \end{cases}$$

Bottom-Up Solution:

```
1:  procedure MINIMUM-PALINDROMIC-CUTS-BOTTOM-UP(s)
2:      n ← LENGTH(s)
3:      initialize C[1..n]
4:      for j = 1 to n do                          ▷ initialization for comparison later
5:          C[j] ← n                               ▷ n upper bounds the total number of possible cuts
6:      for j = 1 to n − 1 do
7:          if IS-PALINDROME(s_{1,j}) then         ▷ s_{1,j} is a palindrome, therefore no cuts
8:              C[j] ← 0
9:          else                                   ▷ s_{1,j} is not a palindrome
10:             for k = 1 to j − 1 do              ▷ we need to make some cut
11:                 if IS-PALINDROME(s_{k+1,j}) then   ▷ s_{i+1,j} is a palindrome
12:                     C[j] ← min(C[j], 1 + C[k])     ▷ make cut s_{1,k}|s_{k+1,j}
13:     return C[n]
```

This algorithm has a total running time of $\mathcal{O}(n^2)$, since it requires 2 nested for loops.

The space complexity is $\mathcal{O}(n)$, since we need to keep track of a string of length n and an array $C[1..n]$.

Top-Down with Memoization Solution:

Note: Assume when the function is called, it is initially called with $C[1..n]$ with all elements equal to n.

```
1:  procedure MINIMUM-PALINDROMIC-CUTS-TOP-DOWN(s, n, C)
2:      if C[n] < n then return C[n]      ▷ This means we have already calculated this sub-problem
3:      if IS-PALINDROME(s_{1,n}) then    ▷ s_{1,n} is a palindrome, therefore no cuts
4:          C[n] = 0
5:          return C[n]
6:      for k = 1 to n − 1 do             ▷ we need to make some cut
7:          if IS-PALINDROME(s_{k+1,n}) then   ▷ s_{i+1,n} is a palindrome
8:              C_k = MINIMUM-PALINDROMIC-CUTS-TOP-DOWN(s, k, C)
9:              C[n] ← min(C[n], 1 + C_k)      ▷ make cut s_{1,k}|s_{k+1,n}
10:     return C[n]
```

This algorithm's running time is described by the recurrence $T(n) = \left(\sum_{k=1}^{n-1} T(k)\right) + \mathcal{O}(1)$. However, since Memoization is used, the value of $T(k)$ may have already been computed, making that call $\mathcal{O}(1)$. We notice there are n distinct inputs to the recurrence (all of the digits $1, \ldots, n$). Therefore, in exactly n of the calls to the recurrence we will have to compute to answer, and in the rest the answer is stored. Suppose all sub-problems have been pre-computed. Therefore, $T(n) = \left(\sum_{k=1}^{n-1} \mathcal{O}(1)\right) + \mathcal{O}(1) = \mathcal{O}(n)$. Now, we multiply by n to account for the n times where we do not have the answer stored (Since in the worst case each $T(k) = \mathcal{O}(n)$). Therefore, the $\boxed{\text{total running time is } \mathcal{O}(n^2)}$.

The $\boxed{\text{space complexity is } \mathcal{O}(n)}$ since we need to keep track of a string of length n and an array $C[1..n]$.

Amortized Analysis‘

1. Three Types of Costs (三种成本对比)

- **Average Cost (平均成本)**
  - **定义**: 所有可能输入成本的算术平均值。
  - **前提**: 假设输入服从**均匀分布 (Uniform Distribution)**，即每个输入出现的概率相等。
  - **公式**: $\text{Avg} = \int c(x)dx$ （连续） 或 $\frac{1}{N}\sum c_i$ （离散）。
- **Expected Cost (期望成本)**
  - **定义**: 考虑了输入**概率分布 (Probability Distribution)** 后的加权平均。
  - **前提**: 某些输入出现的概率 $p(x)$ 更高。
  - **公式**: $\text{Exp} = \int p(x)c(x)dx$。
  - **注**: 若分布均匀，则 Expected Cost = Average Cost。
- **Amortized Cost (平摊成本)**
  - **定义**: 在**最坏情况 (Worst Case)** 下，执行一个**操作序列 (Sequence)** 时，平均每个操作的成本。
  - **前提**: 不涉及概率，强调“长期平均”。昂贵操作很少发生，便宜操作经常发生。
  - **核心**: $\text{Amortized Cost} \approx \frac{\text{Total Cost of Sequence}}{\text{Number of Operations}}$

---

2. Amortized Analysis Methods (平摊分析方法)

- **Method 1: Aggregate Analysis (聚合/总量分析)**
  - **思路**: “算总账，平均分”。
  - **步骤**:
    1. 计算 n 个操作序列的总耗时 $T(n)$。
    2. 每个操作的平摊成本直接等于 $\frac{T(n)}{n}$。
  - **特点**: 赋予每种操作相同的平摊成本。
- **Method 2: Accounting Method (记账/核算法)**
  - **思路**: “多退少补，存钱罐”。
  - **定义**:
    - $\hat{c}_i$ **(Amortized Cost)**: 我们设定的平摊费用（通常 > 实际费用）。
    - c_i **(Actual Cost)**: 操作的实际开销。
  - **机制**:
    - **存钱 (Credit)**: 当 $\hat{c}_i > c_i$ 时，多余的钱存起来。
    - **消费**: 当 $\hat{c}_i < c_i$ 时（遇到昂贵操作），用之前的存款支付。
  - **Credit Invariant (信用约束)**:
    - 必须证明任意时刻 $\sum_{i=1}^{k} \hat{c}_i \geq \sum_{i=1}^{k} c_i$。
    - 即：**银行存款永远不能为负数。**

1. Aggregate Analysis :

→ Sequence of n operations takes $T(n)$ worst-case time

→ Amortized cost : Avg cost in worst case → $T(n)/n$

ex i) Stack Operation

→ Push : $O(1)$

→ Pop : $O(1)$

→ Multipop (s, k) : $O(\min\{n, k\})$.

In a serie of n operations, total # of pop & multipop ≤ total # of pushes

$\therefore$ { # of pop + multipop = $\min\{n - \#push, \#push\}$
# of push = #push

$\therefore$ in n operations : $T(n) = T(\#push) + \min\{T(n - \#push), T(\#push)\}$
$= O(n)$ (worst case).

$\therefore$ Amortized Cost = $O(n)/n = O(1)$ for each operation

For Accounting Mthd, { push : \$3 (1 for push, 2 to deposit).
pop : \$0
Multi-pop : \$0.

2. Accounting Method

→ Charge different amount of "cost" to different operations.

→ "cost" refered to as amortized cost

→ if actual cost { < amortized, save left-over for other operations
> amortized, use saved credit to pay for operation

Define ① Total Amortized cost = $\sum_1^n \hat{c}_i$ ② Total actual cost = $\sum_1^n c_i$

③ Total cost stored inside DS = $\sum_1^n \hat{c}_i - \sum_1^n c_i$

Similar to Aggregate analysis, we want the total amortized cost (fixed) to form an upper bound for the total actual cost (varys based on series of operations). Hence,

Rule ① → $\sum_1^n \hat{c}_i - \sum_1^n c_i \geq 0$

Apart from that, we also need to make sure at any time along the n operations, we cannot allow total amortized cost to fall below total actual, in which case the DS would be functioning in deficit.

Rule ② → $\sum_1^m \hat{c}_i - \sum_1^m c_i \geq 0 \;\forall\; m \leq n, m \geq 1$.

ex.). Amortized cost of pop is \$0., Actual cost of pop is \$1.

if first stack operation is a pop, cost stored in DS = $0 - 1 = -1 < 0$.

This is illegal since we're popping from an empty stack.

Example - Aggregate&Accounting 2^m

Ex 17.1-3

Suppose a data structure performs operations with cost $f(x) = \begin{cases} x, \text{ if } x = 2^m \\ 1, \text{ otherwise} \end{cases}$ (useful for array doubling). Determine the amortized cost on sequence $(1, 2, ..., n)$

Aggregate analysis: $\sum_{i=1}^{n} f(i) = \sum_{i \neq 2^m} 1 + \sum_{i=2^m} i \leq n + \sum_{m=0}^{\log n} 2^m = n + 2n - 1 < 3n = \mathcal{O}(n)$

$T(n) = \mathcal{O}(n)$, then $f(x)$ is $\frac{\mathcal{O}(n)}{n} = \mathcal{O}(1)$ amortized

Accounting method:
Charge \$3 for each operation i
If $i \neq 2^m$, use \$1 to pay for operation, and store \$2. If $i = 2^m$, use stored \$$i$ to pay for operation
Credit invariant: when $i = 2^m$, all elements in range $\left(2^{m-1}, 2^m\right]$ have \$2 stored
Since there are 2^{m-1} elements in $\left(2^{m-1}, 2^m\right]$, when $i = 2^m$ we have $\$2 \cdot 2^{m-1} = \$2^m$ total credit stored. So credit never goes negative when we use \$$i = \$2^m$
Amortized cost is $\mathcal{O}(1)$ since we need \$3 per operation

Example - stack operation

Stack with max elements

Do a sequence of stack operations where stack size never exceeds k. After k operations, copy (a single memory write for each element) the stack to back it up

Operations	Actual Cost	Amortized Cost	
PUSH(S, x)	$\mathcal{O}(1)$	\$2 ($\mathcal{O}(1)$)	Charge \$2 for PUSH, \$1 pays to push, \$1 stored on the stack
POP(S)	$\mathcal{O}(1)$	\$2 ($\mathcal{O}(1)$)	Charge \$2 for POP, \$1 pays to pop, \$1 stored on the stack
COPY(S)	$\mathcal{O}(n)$	\$0 ($\mathcal{O}(1)$)	

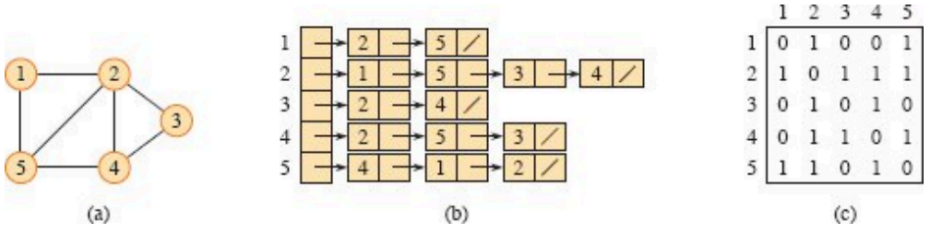
Credit invariant: The amount of credit on the stack equals the number of operations since the last copy, because after each push/pop, we store \$1 on the stack
The credit never goes negative because we copy after k operations, and the stack never contains more than k elements, so we always have \$$k$ stored on the stack to pay for the copy
Amortized cost of all operations is $\mathcal{O}(1)$.

Read exercise 17-2 on the published tutorial notes page 3. Similar, but harder.
It has been on exams for several times.

Graph Algorithm

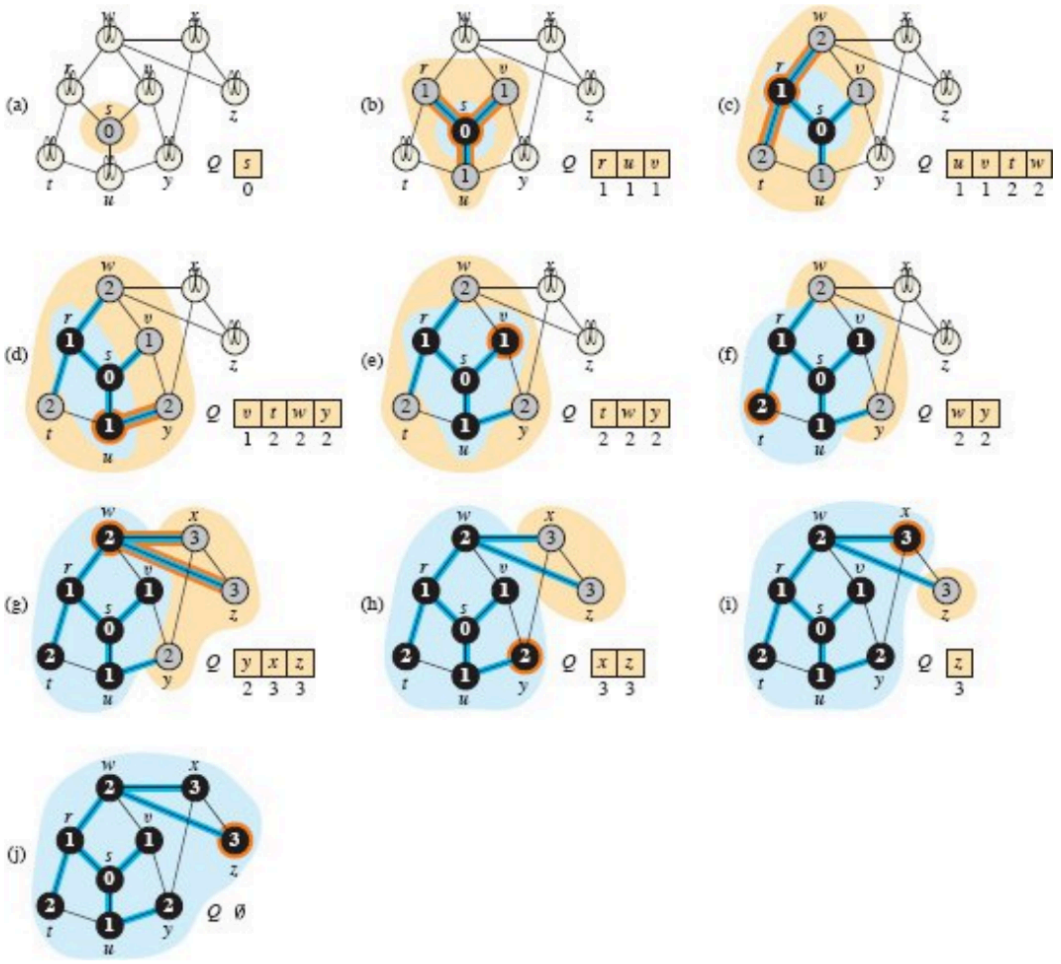
Graph Representation:

sparse graphs—those for which $|E|$ is much less than $|V|^2$—it is usually the method of choice. Most of the graph algorithms presented in this book assume that an input graph is represented in adjacency-list form. You might prefer an adjacency-matrix representation, however, when the graph is ***dense***—$|E|$ is close to $|V|^2$—or when you need to be able to tell quickly whether there is an edge connecting two given vertices. For example, two of the all-pairs shortest-paths algorithms presented in Chapter 23 assume that their input graphs are represented by adjacency matrices.



BFS:

```
BFS(G, s)
 1 for each vertex u ∈ G.V – {s}
 2     u.color = WHITE
 3     u.d = ∞
 4     u.π NIL
 5 s.color = GRAY
 6 s.d = 0
 7 s.π NIL
 8 Q = Ø
 9 ENQUEUE(Q, s)
10 while Q ≠ Ø
11     u = DEQUEUE(Q)
12     for each vertex v in G.Adj[u]      // search the neighbors of u
13         if v.color == WHITE            // is v being discovered now?
14             v.color = GRAY
15             v.d = u.d + 1
16             v.π = u
17             ENQUEUEd(Q, v)             // v is now on the frontier
18     u.color = BLACK                    // u is now behind the frontier
```



Key Take-Aways

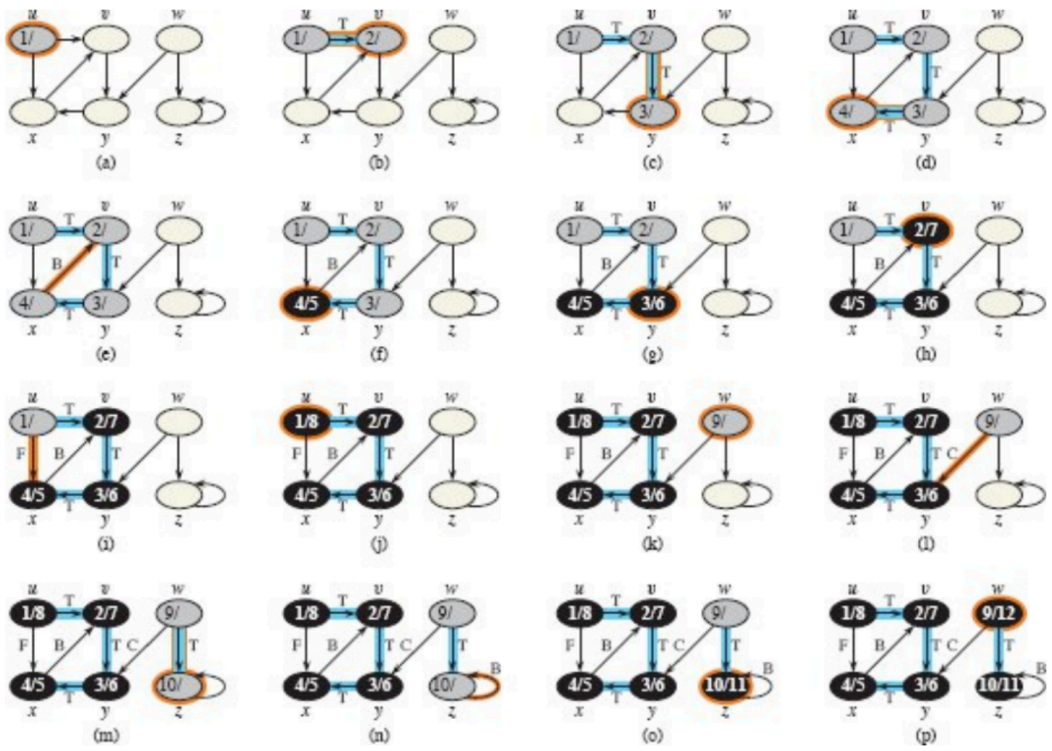
- Discover all nodes of depth k before discovering any node of depth $k+1$
- we only discover nodes **reachable** from s (i.e. $\exists\ s \xrightarrow{p} t$)
- Runtime: $\mathcal{O}(V+E)$
- Node types: WHITE (undiscovered), GREY (processing), BLACK (finished)
- maintain $v.d/d[v]$ and $v.\pi/\pi[v]$, $\forall v \in V$
  - $\forall v \in V$, $d[v] = \min\{\#\text{edges in any path from } s \text{ to } v\}$
  - if $\nexists\ s \xrightarrow{p} v$, $d[v] = \infty$
  - $\pi[v]$ is the predecessor of v on the shortest path $s \to v$

BFS can be thought of as a SSSP algorithm for a graph with all edge weights equal to 1. $d[v] = \delta(s, v)$.

DFS:

```
DFS(G)
 1 for each vertex u ∈ G.V
 2     u.color = WHITE
 3     u.π = NIL
 4 time = 0
 5 for each vertex u ∈ G.V
 6     if u.color == WHITE
 7         DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1 time = time + 1                    // white vertex u has just been discovered
 2 u.d = time
 3 u.color = GRAY
 4 for each vertex v in G.Adj[u]// explore each edge (u, v)
 5     if v.color == WHITE
 6         v.π = u
```

```
 7         DFS-VISIT(G, v)
 8 time = time + 1
 9 u.f = time
10 u.color = BLACK                    // blacken u; it is finished
```



IMPORTANT: Different kinds of Edges in DFS Graph

🔷 2️⃣ Tree edge

A **tree edge** is any edge that **leads to a vertex that was not yet discovered** when we first encounter it.

Formally:

$$(u, v) \text{ is a tree edge if } v.\text{color} = \text{WHITE}$$

when explored from u.

🔷 3️⃣ Forward edge

A **forward edge** connects a vertex u to a **descendant** v in the DFS tree,
but v has **already been discovered and finished**.

Formally:

$$(u, v) \text{ is a forward edge if } v \text{ is a descendant of } u \text{ in the DFS tree.}$$

That means:

- DFS had already fully processed v (so it's **BLACK**),
- and v's discovery/finish times satisfy $d[u] < d[v] < f[v] < f[u]$.

1. Tree Edge (树边) —— "发现新大陆"

- **定义：** 这是最普通的边。当我们从 u 出发，第一次发现 v 时，这条边就是树边。
- **判断标准：** 当你遇到 v 时，v 的颜色是 **WHITE** (未被发现)。
- **直观理解：** 这些边构成了 DFS 树的骨架，是你遍历时的"主干道"。

2. Back Edge (后向边) —— "回头路"

- **定义：** 从某个点 u 指回它的**祖先** v 的边。
- **判断标准：** 当你遇到 v 时，v 的颜色是 **GRAY** (正在处理中/在递归栈里)。
- **重要意义：** 如果图中存在 Back Edge，说明图里**一定有环 (Cycle)**。因为你撞见了一个"还没结束递归的长辈"，说明你绕了一圈回来了。

3. Forward Edge (前向边) —— "抄近道"

- **定义：** 从某个点 u 指向它的**后代** v 的边（但不是直接的树边）。
- **判断标准：**
  1. 当你遇到 v 时，v 的颜色是 **BLACK** (已完成)。
  2. 时间戳检查：u 的发现时间早于 v ($d[u] < d[v]$)。
- **直观理解：** u 是爷爷，v 是孙子。树边是"爷爷->爸爸->孙子"，Forward Edge 是"爷爷直接指向孙子"。虽然 v 已经处理完了，但 u 确实是 v 的长辈。

4. Cross Edge (横向边) —— "跨部门串门"

- **定义：** 连接两个没有祖孙关系的节点的边。通常是从一个分支跳到另一个已经处理完的分支。
- **判断标准：**
  1. 当你遇到 v 时，v 的颜色是 **BLACK** (已完成)。
  2. 时间戳检查：u 的发现时间晚于 v ($d[u] > d[v]$)。
- **直观理解：** u 和 v 可能是兄弟分支，或者隔壁分支。u 访问 v 时，发现 v 早就收工回家了 (Black)，而且 u 既不是 v 的祖先也不是后代。

⚡ **快速判断口诀 (Cheatsheet)**

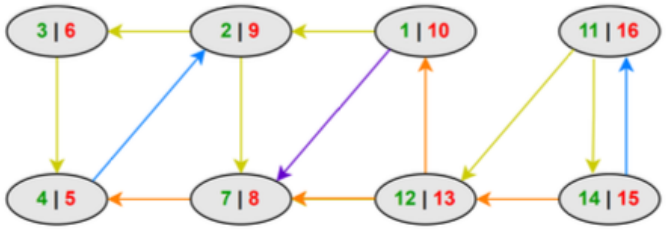
在做题或分析时，你只需要看目标点 v 的颜色：

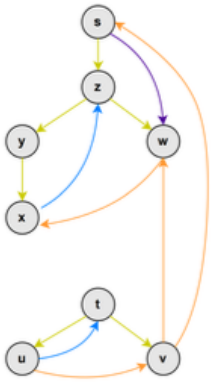
1. **遇到 WHITE** $\rightarrow$ **Tree Edge** (正常探索)
2. **遇到 GRAY** $\rightarrow$ **Back Edge** (有环！)
3. **遇到 BLACK** $\rightarrow$ **比较 d 值**：
   - $d[u] < d[v] \rightarrow$ **Forward Edge** (指向后代)
   - $d[u] > d[v] \rightarrow$ **Cross Edge** (指向别人家)

需要我用一个简单的 3 个节点的图来举例说明这几种边吗?

Topological Sort

tree / back / forward / cross





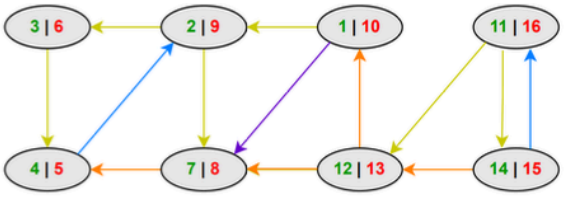
1: **function** TopologicalSort(G)
2: DFS(G)
3: Output vertices in decreasing order of finish time

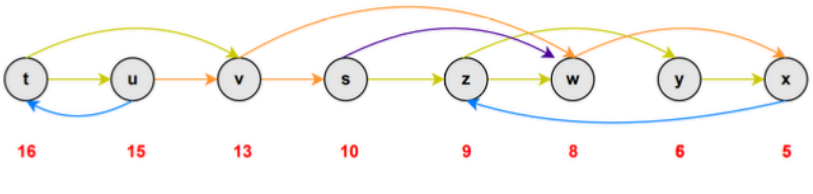
UNIVERSITY OF TORONTO

Winston (Yuntao) Wu (UofT) ECE345 Tutorial9 3 / 24

Example

This is not valid for TopoSort. To make it valid, remove all the back edges.





UNIVERSITY OF TORONTO

A directed graph $G = (V, E)$ is said to be semiconnected if $\forall u, v \in V$ either $u \to v$ or $v \to u$. Give an $\mathcal{O}(V + E)$ algorithm to determine whether a given directed graph is semiconnected.

Algorithm: Find G^{SCC} the strongly connected components of G using CLRS 20.5. Then run Toposort on G^{SCC}. (G^{SCC} is a DAG), G is semiconnected if there exists edge between all adjacent vertices in the topological order of G^{SCC}.

Runtime: $\mathcal{O}(V + E)$
Correctness:
Claim:
Let $v_1, v_2, \ldots, v_n \in V^{SCC}$ be the topological ordering
$(v_i, v_{i+1}) \in E^{SCC}$ $\forall i \Leftrightarrow G^{SCC}$ semiconnected.
Proof:
$\Rightarrow \exists$ path $v_i \to v_{i+1}$ $\forall i$
so take $i < j$, $\forall v_i \to v_j$ via path $v_i \to v_{i+1} \to \cdots \to v_{j-1} \to v_j$
$\Leftarrow$ either exists path $v_i \to v_{i+1}$ or $v_{i+1} \to v_i$
$v_{i+1} \to v_i$ cannot exist because that breaks topological ordering.
consider $v_i \to v_k \to v_{i+1}$, which cannot exist because then $k > i$ and $k < i + 1$, but k integer.
So $v_i \to v_{i+1}$ is the edge (v_i, v_{i+1})

Minimum Spanning Tree:

1.0 Spanning Tree definition:

🔷 1. Spanning Tree

A **spanning tree** of a connected, undirected graph $G = (V, E)$ is a **subgraph** that:

1. Includes **all the vertices** of G.
2. Is a **tree** — meaning it's connected and has **no cycles**.
3. Has exactly $|V| - 1$ edges.

So, it "spans" all vertices while removing just enough edges to eliminate cycles but still keep the graph connected.

1.1 Minimum Spanning tree (MST):

🔷 2. Minimum Spanning Tree (MST)

If each edge in the graph has a **weight** (like cost, distance, or time), then a **minimum spanning tree (MST)** is a spanning tree whose total edge weight is **as small as possible** among all spanning trees.

Formally:

$$T_{\text{MST}} = \arg\min_{T \subseteq E, \text{spanning tree}} \sum_{(u,v) \in T} w(u, v)$$

So the MST connects all vertices with the minimum total edge cost, without forming cycles.

Prim's Algo:

MST-PRIM(G, w, r) 1 **for** each vertex $u \in G.V$ 2 $u.key = \infty$ 3 $u.\pi$ = NIL 4 $r.key = 0$ 5 $Q = \emptyset$	initialize all the graph nodes to have a key of infinity (v.key = inf). Only set root node's key to 0
6 **for** each vertex $u \in G.V$ 7 INSERT(Q, u)	insert all the nodes to a min heap

```
8 while Q ≠ Ø
9     u = EXTRACT-MIN(Q)   // add u to the tree
10    for each vertex v in// update keys of u's non-tree
          G.Adj[u]                neighbors
11        if v ∈ Q and w(u, v) < v.key
12            v.π = u
13            v.key = w(u, v)
14            DECREASE-KEY(Q, v, w(u, v))
```

every time we pop the node with min key, we relax its neighbouring nodes (so that next time, the min among them are popped out)

TIME COMPLEXITY: O(Elog(V))

MST

Key Take-Aways

- Spanning Tree: $T \subseteq E$ such that $\forall v \in V$, $\exists (u, v) \in T$ or $\exists (v, u) \in T$. (contains all vertices)
- Minimum Spanning Tree: Spanning tree $T \subseteq E$ such that $w(T) = \sum_{(u,v)\in T} w(u, v)$ is minimized.
- Prim's Algorithm: $\mathcal{O}(E \log V)$ with binary heap (and $\mathcal{O}(V \log V + E)$ with Fibonacci heap)
- An MST always has $|V|$ vertices and $|V| - 1$ edges ($|E_{\text{MST}}| = |V_{\text{MST}}| - 1$)
- Consider $(u, v) \in T$ and $(x, y) \notin T$, then $T \cup \{(x, y)\}$ always contains a cycle and $T \setminus \{(u, v)\} \cup \{(x, y)\}$ is a spanning tree.

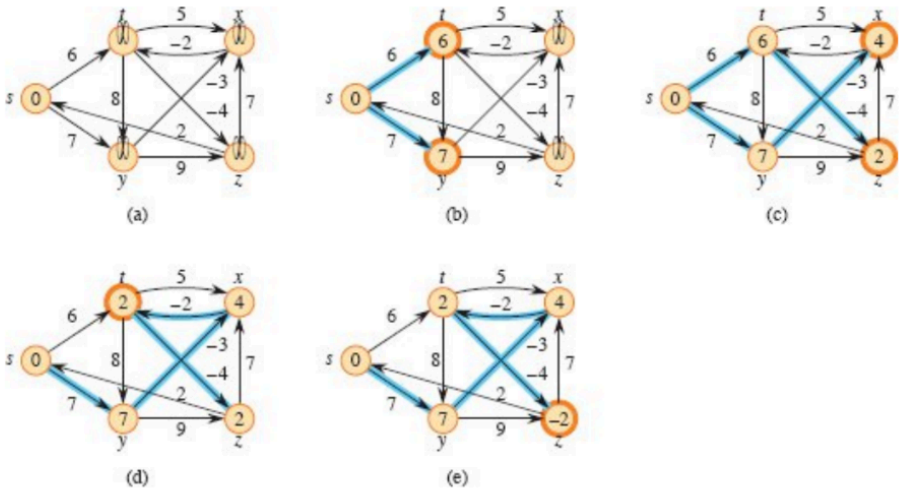
UNIVERSITY OF TORONTO

Shortest path Algorithms:

Bellman Ford Algorithm:

```
BELLMAN-FORD(G, w, s)
1 INITIALIZE-SINGLE-SOURCE(G, s)
2 for i = 1 to |G.V| − 1
3     for each edge (u, v) ∈ G.E
4         RELAX(u, v, w)
5 for each edge (u, v) = G.E
6     if v.d > u.d + w(u, v)
7         return FALSE
8 return TRUE
```

TIME COMPLEXITY: O(EV)



Relatation:

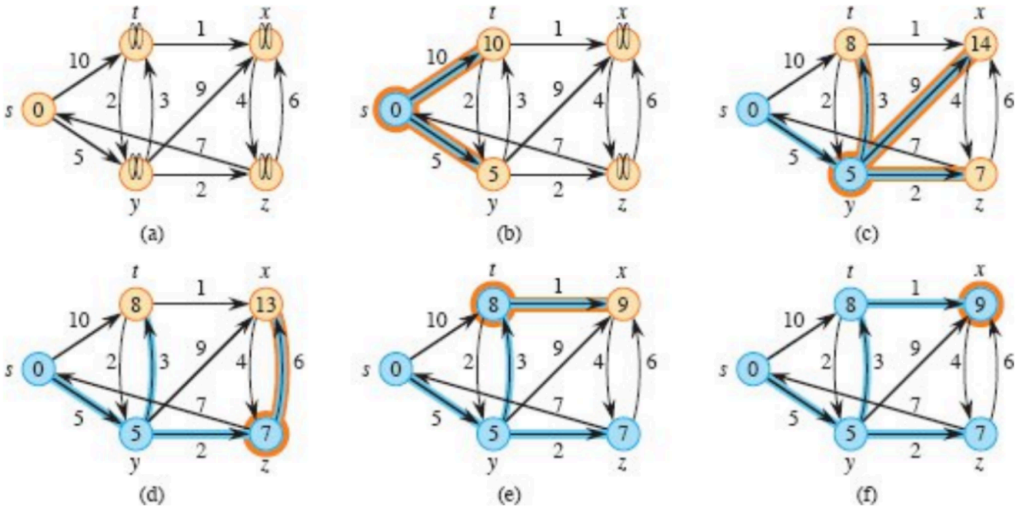
```
RELAX(u, v, w)

1 if v.d > u.d + w(u, v)
2     v.d = u.d + w(u, v)
3     v.π = u
```

Other Properities:

1. Allow edges with negative weights
2. Can correctly report Negative cycles in the resulting graph
3. Cannot guarantee correctness if NEGATIVE CYCLE reachable from the source

Dijkstra's Algorithm (very similar with Prim's Algorithm):

```
DIJKSTRA(G, w, s)
 1 INITIALIZE-SINGLE-SOURCE(G, s)
 2 S = Ø
 3 Q = Ø
 4 for each vertex u ∈ G.V
 5     INSERT(Q, u)
 6 while Q ≠ Ø
 7     u = EXTRACT-MIN(Q)
 8     S = S ∪ {u}
 9     for each vertex v in G.Adj[u]
10         RELAX(u, v, w)
11         if the call of RELAX decreased v.d
12             DECREASE-KEY(Q, v, v.d)
```



TIME COMPLEXITY: O(Elog(V))

Other Properities:

4. DOES NOT allow edges with negative weights

Example

Find the minimum weight cycle in $G = (V, E)$ in $\mathcal{O}(VE^2)$ time (assuming no negative weight cycles)
Solution: Remove (u, v) from the graph, find the shortest path from v to u using Bellman Ford
Do this for each edge, keeping track of minimum cycle $d(v, u) + w(u, v)$
Runtime: $E \cdot \mathcal{O}(VE) = \mathcal{O}(VE^2)$

Find the minimum weight cycle in $G = (V, E)$ in $\mathcal{O}(VE \log V)$ time. (assuming no negative weights)
Solution: $\forall v \in V$ call Dijkstra, store results in a matrix where $D[u, v] = d(u, v)$ (we solved the all-pairs shortest path problem)
Compute $\min_{u,v \in V}(d(u, v) + d(v, u))$
Runtime: $V \cdot \mathcal{O}(E \log V) + \mathcal{O}(V^2) = \mathcal{O}(VE \log V)$

Difference Constraints

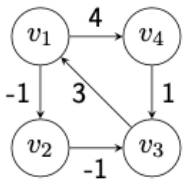
To solve a difference constraint problem with n variables, x_1,...,x_n,

1. Build constraint graph (weighted, directed) $G = (V, E)$
   - $V = \{v_0, v_1, ..., v_n\}$: one vertex per variable. Define v_0 as the pseudo-start.
   - $E = \{(v_i, v_j) : x_j - x_i \leq b_k$ is a constraint.$\} \cup \{(v_0, v_1), (v_0, v_2), ..., (v_0, v_n)\}$: one edge per constraint. Direction is from the subtrahend to the minuend[4]. Also connect pseudo-start to all vertices in the constraints.
2. Assign weights
   - $w(v_0, v_i) = 0$, for all i
   - $w(v_i, v_j) = b_k$, for all constraints $x_j - x_i \leq b_k$
3. Theorem:
   - If G has no negative weight cycle, then $x_1 = \delta(v_0, v_1)$, ..., $x_n = \delta(v_0, v_n)$ is a feasible solution
   - If G has a negative weight cycle, then there is no feasible solution
4. Build graph and run Bellman-Ford to find the shortest path from v_0 to all of $v_1, ..., v_n$, which gives the solution, or detect negative cycle (no solution).

example

Find a feasible solution or determine that no solution exists for the following system of difference constraints:

$x_1 - x_3 \leq 3$
$x_2 - x_1 \leq -1$
$x_3 - x_2 \leq -1$
$x_3 - x_4 \leq 1$
$x_4 - x_1 \leq 4$



Running Bellman-Ford gives:
$\delta(v_0, v_1) = 0$, $\delta(v_0, v_2) = -1$, $\delta(v_0, v_3) = -2$, $\delta(v_0, v_4) = 0$

	v_0	v_1	v_2	v_3	v_4
iter0	0	0	0	0	0
iter1	0	0	-1	-1	0
iter2,3,4	0	0	-1	-2	0

A feasible solution is:

$x_1 = 0$, $x_2 = -1$, $x_3 = -2$, $x_4 = 0$.

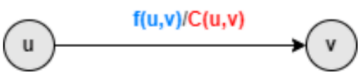
UNIVERSITY OF TORONTO

Maximum Flow

Definitions

- Flow Network: a directed graph $G = (V, E)$ with the following properties.
  - all edges $(u, v) \in E$ have a non-negative capacity, i.e. $c : V \times V \to \mathbb{R}_{\geq 0}$ with $c(u, v) \geq 0$.
  - $(u, v) \in E \Rightarrow (v, u) \notin E$ (no anti-parallel edges)
  - $(u, v) \notin E \Rightarrow c(u, v) = 0$
  - $\forall v \in V$ $(v, v) \notin E$ (no self-loops)
  - $\forall v \in V$ $\exists$ path $s \to v \to t$ (G is connected)

  Note: i.e. a flow network is a directed, connected, positively-weighted graph with no anti-parallel edges.
- Flow: $f : V \times V \to \mathbb{R}$ with 2 properties.
  - Capacity Constraint: $\forall u, v \in V$, $0 \leq f(u, v) \leq c(u, v)$
  - Flow Conservation: $\forall u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$ (flow entering a vertex = flow exiting a vertex)
- Maximum Flow: $|f^*| = \max_f |f|$ subject to the capacity constraint and flow conservation.
- Basically all edges look like this



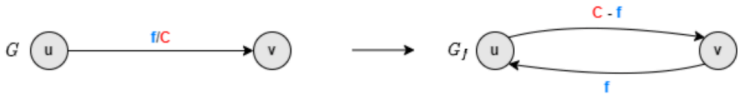
UNIVERSITY OF TORONTO

Residual Network

Residual Capacity: represents how much the flow can change in the direction $u \to v$

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{if } (u,v) \in E \\ f(v,u) & \text{if } (v,u) \in E \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

$G_f = (V, E_f)$ where $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$ $(|E_f| \leq 2|E|)$



Cut

A cut (S,T) of G is a partition of V into S, $T = V - S$ such that $s \in S$, $t \in T$.

Capacity of (S,T): $c(S,T)$, it is directional, and only consider the path from S to T.

Net flow across (S,T): $f(S,T)$, sum of flow across the cut,
$=$ $\sum$ flow from S to $T - \sum$ flow from T to S.

Lemma: for any cut (S,T), $f(S,T) = |f|$ in each iteration.

Corollary (Max Flow Min Cut): the value of any flow $\leq$ capacity of any cut $|f| \leq c(S,T)$,
$\forall S, T, f$, and $|f^*| = \min c(S,T)$

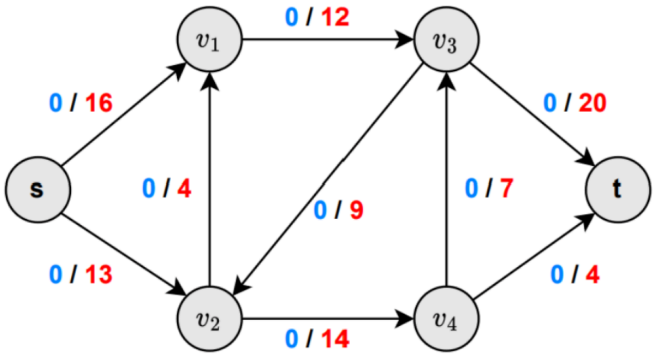
UNIVERSITY OF TORONTO

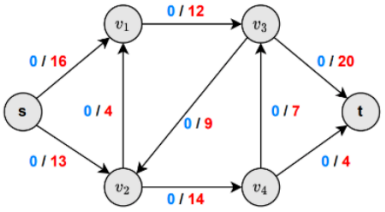
Ford-Fulkerson Algorithm

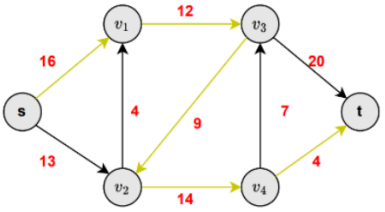
```
FORD-FULKERSON (G, s, t)
1 for each edge (u, v) ∈ G.E
2     (u, v).f = 0
3 while there exists a path p from s to t in the residual network G_f
4     c_f(p) = min {c_f(u, v) : (u, v) is in p}
5     for each edge (u, v) in p
6         if (u, v) ∈ G.E
7             (u, v).f = (u, v).f + c_f(p)
8         else (v, u).f = (v, u).f − c_f(p)
9 return f
```

Ford-Fulkerson Example

Choose augemented paths randomly.
For some augmented path p, $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$
Runtime: $\mathcal{O}(E|f^*|)$



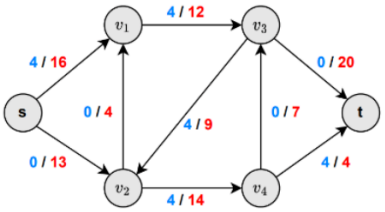


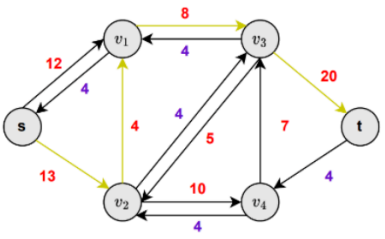


$p : s \to v_1 \to v_3 \to v_2 \to v_4 \to t$

$|f_p^*| = C_f(p) = 4$

$|f| \leftarrow |f| + |f_p^*| = 0 + 4 = 4$

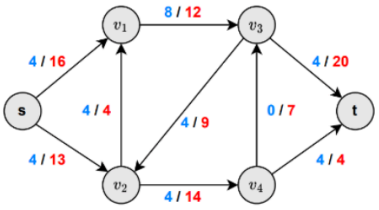


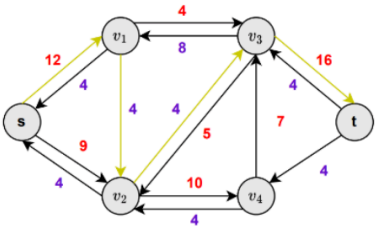


$p : s \to v_2 \to v_1 \to v_3 \to t$

$|f_p^*| = C_f(p) = 4$

$|f| \leftarrow |f| + |f_p^*| = 4 + 4 = 8$

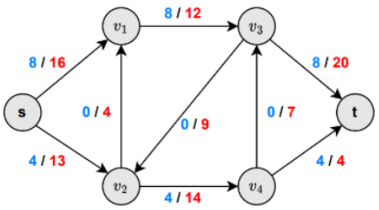


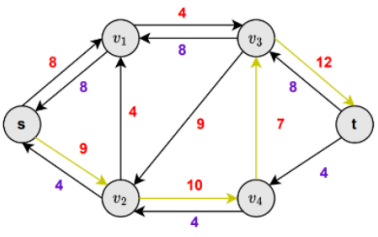


$p : s \to v_1 \to v_2 \to v_3 \to t$

$|f_p^*| = C_f(p) = 4$

$|f| \leftarrow |f| + |f_p^*| = 8 + 4 = 12$

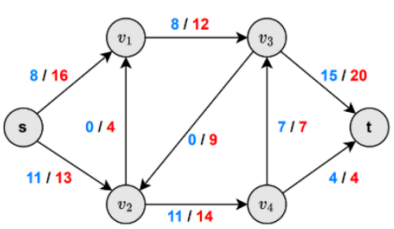


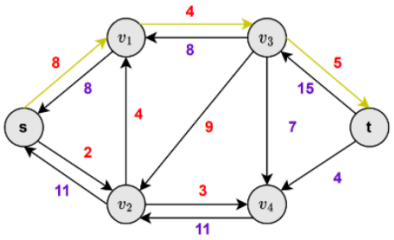


$p : s \to v_2 \to v_4 \to v_3 \to t$

$|f_p^*| = C_f(p) = 7$

$|f| \leftarrow |f| + |f_p^*| = 12 + 7 = 19$

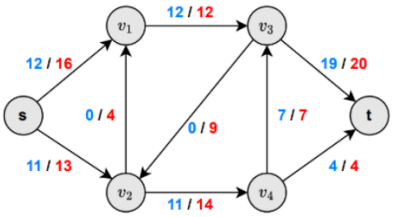


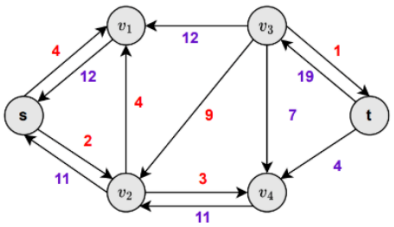


$p : s \to v_1 v_3 \to t$

$|f_p^*| = C_f(p) = 4$

$|f| \leftarrow |f| + |f_p^*| = 19 + 4 = 23$





There are no more paths in the residual graph Gf from s to t

$|f^*| = 23$

Edmond Karp Algorithm:

- It's one implementation of the Ford Fulkerson, by using BFS to find the optimizing path
- **Ford Fulkerson:** DFS for path finding
- **Edmond Karp:** BFS (always look for the current shortest path)
- **Motivation:** long paths are more likely to have a limiting bottleneck (not ideal)

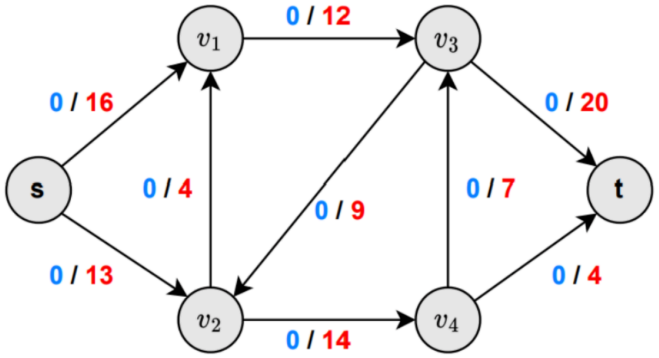
我们只在乎从起点 s 到终点 t 经过几根管子。经过管子越少越好。

```
def edmonds_karp(self, source, sink):
    parent = [-1] * self.size
    max_flow = 0

    while self.bfs(source, sink, parent):
        path_flow = float("Inf")
        s = sink
        while(s != source):
            path_flow = min(path_flow, self.adj_matrix[parent[s]][s])
            s = parent[s]

        max_flow += path_flow
        v = sink
        while(v != source):
            u = parent[v]
            self.adj_matrix[u][v] -= path_flow
            self.adj_matrix[v][u] += path_flow
            v = parent[v]
```

```
        path = []
        v = sink
        while(v != source):
            path.append(v)
            v = parent[v]
        path.append(source)
        path.reverse()
        path_names = [self.vertex_data[node] for node in path]
        print("Path:", " -> ".join(path_names), ", Flow:", path_flow)

    return max_flow
```

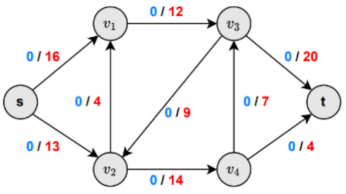
Edmonds-Karp Example

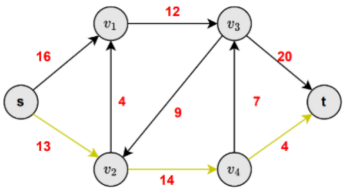
Choose the shortest path $s \to t$ in G, where shortest path defined as minimum number of edges.
Runtime: $\mathcal{O}(VE^2)$



UNIVERSITY OF TORONTO

Edmonds-Karp Example

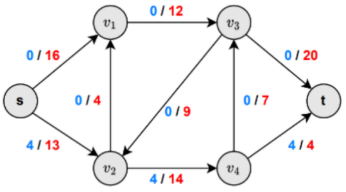


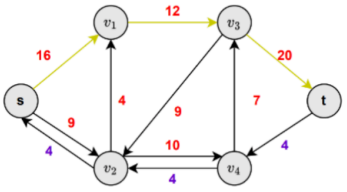


$p : s \to v_2 \to v_4 \to t$

$|f_p^*| = C_f(p) = 4$

$|f| \leftarrow |f| + |f_p^*| = 0 + 4 = 4$

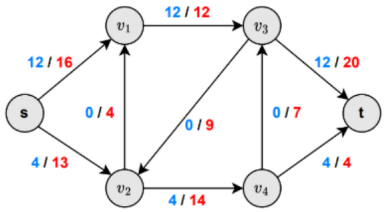


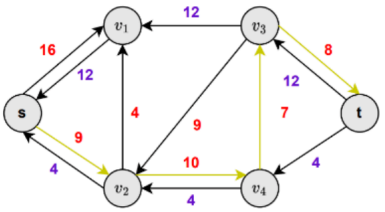


$p : s \to v_1 \to v_3 \to t$

$|f_p^*| = C_f(p) = 12$

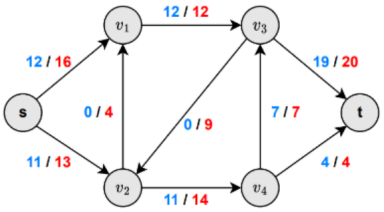
$|f| \leftarrow |f| + |f_p^*| = 4 + 12 = 16$

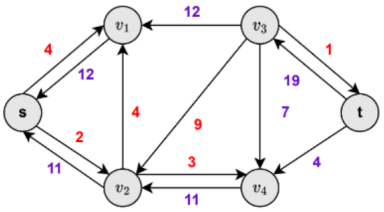




$|f_p^*| = C_f(p) = 7$

$|f| \leftarrow |f| + |f_p^*| = 16 + 7 = 23$





There are no more paths in the residual graph Gf from s to t

$|f^*| = 23$

4.0 Useful results for Max Flow:

Useful Results

The maximum flow $|f^*|$ of any flow network is upper bounded by the sum of capacities of any cut.
Proof: The flow going across that cut cannot be any larger, so it bottlenecks the flow of the graph.

In the max flow network, the min-cut has capacity $|f^*|$.
Proof: By Max-Flow Min-Cut Theorem.

Let $G = (V, E)$ be a flow network with all edge capacities of 1. $(c(u, v) = 1, \forall u, v \in V)$, then $|f^*| =$ the number of edge-disjoint paths in G
The min cut has capacity $|f^*|$, since all edge weighs 1, each path through the cut accounts for 1 capacity.
Therefore, there are net $|f^*|$ independent paths across the cut.

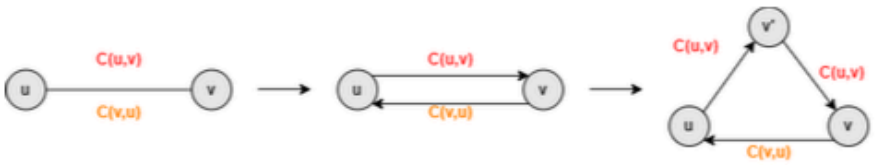
To find the number of independent paths in G
We just find the max flow in G.

UNIVERSITY OF TORONTO

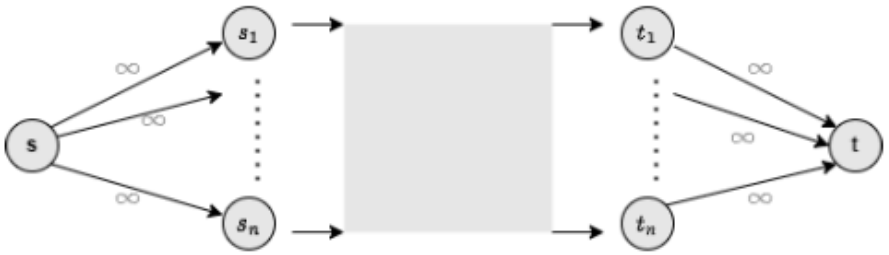
Other Variations:

Convert any connected graph into a flow network

Undirected/antiparallel case:

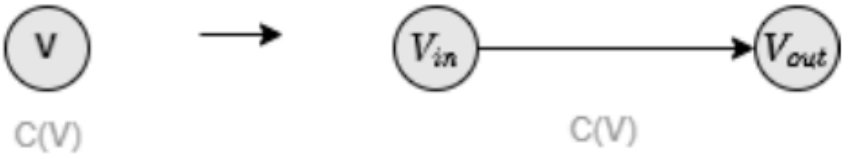


Multiple sources and sinks:



Vertex capacities

Suppose each vertex has capacity $c(v)$. Find the maximum flow from s to t.
Transformation:



Formally: $G' = (V', E')$ s.t.

- $V' = \{v_{\text{in}} : v \in V\} \cup \{v_{\text{out}} : v \in V\}$
- $E' = \{(v_{\text{in}}, v_{\text{out}}) : v \in V\} \cup \{(u_{\text{out}}, v_{\text{in}}) : (u, v) \in E\}$
- $c(u_{\text{out}}, v_{\text{in}}) = c(u, v),\ \forall (u, v) \in E$
- $c(v_{\text{in}}, v_{\text{out}}) = c(v),\ \forall v \in V$

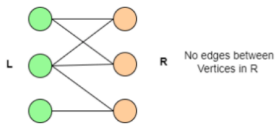
- **场景：** 如果图里**每一条边**的容量全都是 **1**。
- **结论：** 此时的**最大流数值** ($|f^*|$)，就等于 s 到 t 的**边独立路径 (Edge-disjoint paths)** 的数量。

Bipartide Graph:

1.0 Bipartide Definition:

Definitions

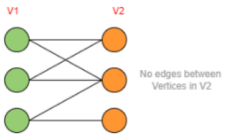
Bipartite graphs: V can be divided into 2 sets L and R, s.t. $L \cap R = \emptyset$, $L \cup R = V$, and edges only exist between vertices in L to vertices in R.



1.1 Check bipartide Algorithm by coloring

Determine if a graph is bipartite.

Bipartite graphs:
V can be divided into 2 sets V_1 and V_2 such that $V_1 \cap V_2 = \emptyset, V_1 \cup V_2 = V$, and edges only exist between vertices in V_1 to vertices in V_2. (or vice versa.)



Solution:Use color GREEN and ORANGE.
Run DFS (or BFS), when you discover a new vertex, color is the opposite color of its parents (if parent is GREEN, assign ORANGE; if parent is ORANGE, asign GREEN), iterate over edges to check if an edge connects two vertices of the same color.
Correctness: Need to show that G bipartite $\Rightarrow$ DFS will color all pairs of vertices different colors.
Assume that G bipartite, but $\exists (u, v) \in E$ such that DFS colored u and v the same color, say GREEN. So v was discovered previously via another edge $\Rightarrow (w, v) \in E \Rightarrow$ color[w]=ORANGE.
But then G is not bipartite because there exists $u \rightarrow v \leftarrow w$. Contradiction.

有些图画得乱七八糟，不像左右两排那么整齐，我们怎么知道它能不能分成两拨人？ **方法：染色法 (2-Coloring via DFS/BFS)**

- **规则：** 只有两种颜色的帽子，比如 **绿色 (GREEN)** 和 **橙色 (ORANGE)**。
- **操作：**
  1. 随便抓一个点，戴上**绿**帽子。
  2. 根据规则，它的所有邻居（也就是它相亲的对象）必须戴**橙**帽子。
  3. 邻居的邻居（对象的对象）必须戴**绿**帽子。
  4. 以此类推，**相邻的节点颜色必须相反**。
- **判定：**
  - **成功：** 如果你把全图都染完了，没有发生任何冲突，那就是二分图。
  - **失败：** 如果你发现某个点，它的邻居本来应该染橙色，结果发现它被另一个邻居染成了绿色（即**两个绿色的人连在了一起**），说明冲突了。这就不是二分图（说明图里有奇数长度的环）。

1.2 Bipartite Matching

⭐ Matching (also called an *independent edge set*)

A **matching** in a graph $G = (V, E)$ is a set of edges $M \subseteq E$ such that **no two edges in M share a common endpoint**.

In other words:

- each vertex in the graph can be incident to **at most one** edge from the matching
- edges in the matching are pairwise disjoint in terms of vertices

---

⭐ Maximum vs. Maximal (different!)

These terms get mixed up, so here's the distinction:

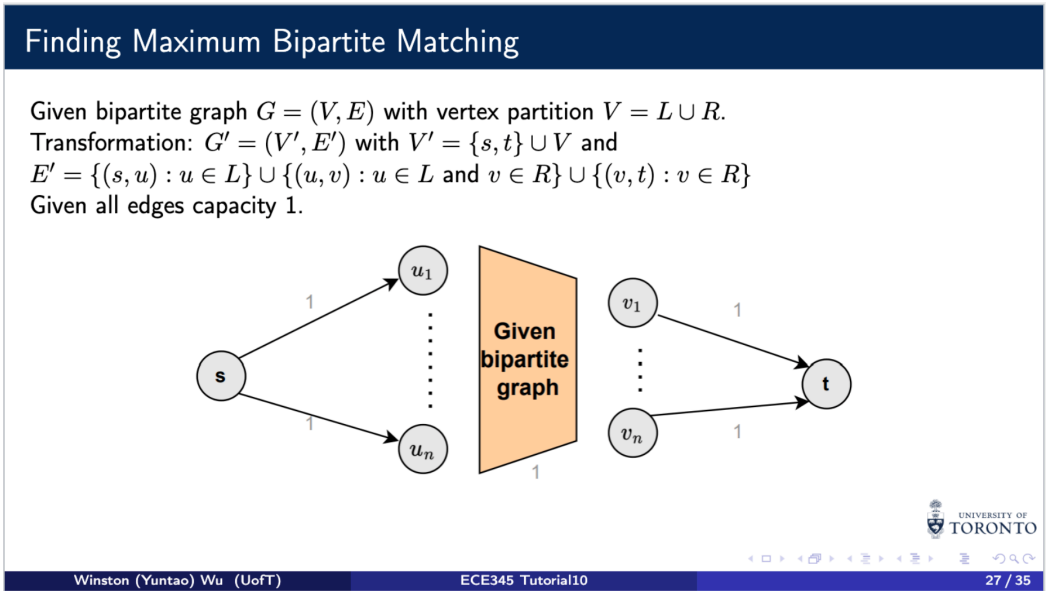
Maximal Matching

- A matching that **cannot be enlarged by adding another edge**
- But it might **not be the largest possible**
- Local optimum

Maximum Matching

- A matching with the **largest number of edges possible**
- Global optimum

Every **maximum** matching is **maximal**, but not every maximal matching is maximum.



1.3 Maximum Bipartite matching problem:

⭐ Goal

Given a bipartite graph

$$G = (U, V, E)$$

find a **maximum matching**.

We will turn this into a **flow network** and run **Max-Flow**.

⭐ Step 1 — Create a flow network

Start with a directed graph.

1. Add a super-source s

Add a new node s and connect it to **every vertex in** U.

2. Add a super-sink t

Add a new node t and connect **every vertex in** V to t.

3. Direct edges from U to V

For each edge $(u, v) \in E$ in the original bipartite graph, add a **directed** edge:

$$u \to v$$

⭐ Step 2 — Assign capacities

Give **every edge capacity 1**.

Why?

Because:

- a node in a matching can only participate in **one** matching edge
- capacity 1 enforces "at most one unit of flow passes through a vertex"

Capacities:

- $c(s, u) = 1$ for all $u \in U$
- $c(u, v) = 1$ for all edges $(u, v) \in E$
- $c(v, t) = 1$ for all $v \in V$

第三部分：如何用最大流解决“最大匹配”？(核心考点)

来源： 上半部分

既然我们想要“牵手对数最多”，这听起来很像“流量最大”。我们可以把相亲问题改造成水管流问题。

改造步骤 (Transformation):

1. **建立超级源点 (Source s):** 代表“月老”。
   - 让 s 连向所有左边的节点（男生 $u_1 \dots u_n$）。
   - **关键点：** 这些边的容量（Capacity）全部设为 **1**。
   - *含义：* 每个男生只能从月老那里领到 1 份“恋爱许可”，保证了他只能参与一次配对。
2. **建立超级汇点 (Sink t):** 代表“民政局”。
   - 让所有右边的节点（女生 $v_1 \dots v_n$）都连向 t。
   - **关键点：** 这些边的容量也全部设为 **1**。
   - *含义：* 每个女生最终只能算 1 个配对名额。
3. **中间的连线：**
   - 把原来图里男生 u 和女生 v 的连线，变成从左指向右的有向边 ($u \to v$)。
   - **关键点：** 容量设为 **1**（或者无穷大也可以，因为瓶颈在源头和尽头）。

结论： 这个时候，你跑一遍 **Max Flow** 算法。

- **最大流量的值 = 最大匹配的对数。**
- 因为容量为 1 的限制，强行保证了每个人只能和一个异性配对。

第四部分：一个特殊的限制题 (Example)

来源： 下半部分

题目： 假设我们要找一个最大匹配，但是有一个额外的总限制：**总共最多只允许 N 对成功。**（虽然题目写的是 "at most N edges"，结合图示，它的意思是给总流量加了一个总阀门）。

解法： 这用到了我们之前学的 **“拆点限流”** 技巧。

1. 把超级源点 s 拆成两个：S_{in} 和 S_{out}。
2. 中间连一条边 $S_{in} \to S_{out}$。
3. 把这条边的容量设为 N。
4. 其他结构不变。

原理： 这就好比在自来水厂的总出水口装了一个总阀门，最大只能流出 N 吨水。那么无论后面有多少对男女情投意合，最终能成的对数绝不会超过 N。

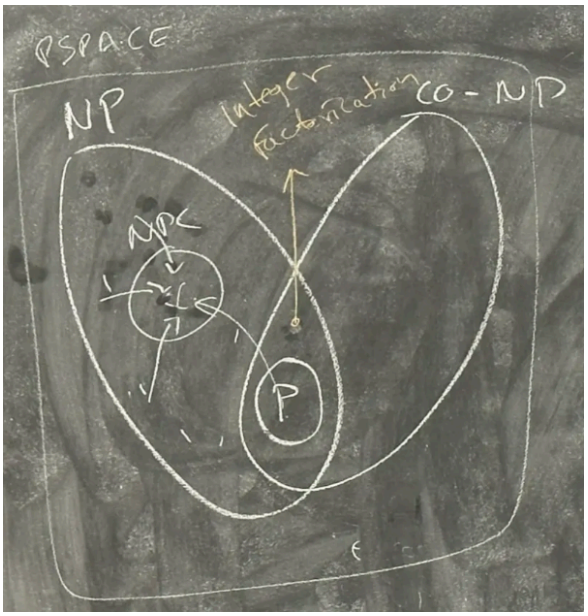
总结

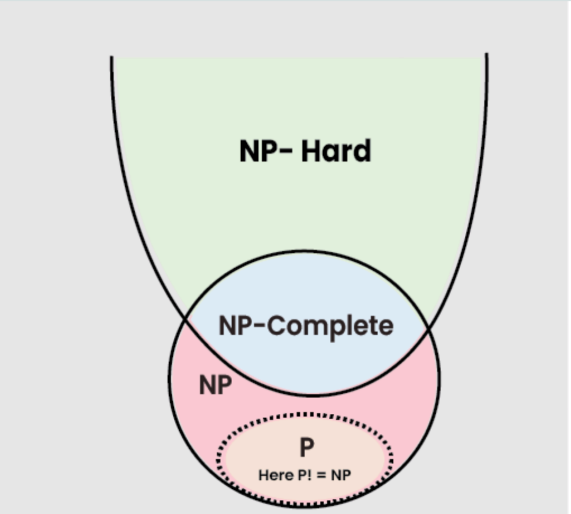
- **二分图** = 男生一拨，女生一拨，内部没连线。
- **判断** = 染色法（黑白/绿橙），不冲突就是二分图。
- **求最大匹配** = 加个 s 和 t，所有边容量设为 1，跑 Max Flow。

Complexity & NP-Completeness

Transformation / runtime / claim / proof<- / proof->

1.0 Complexity Classes:





P Class

The P in the P class stands for **Polynomial Time.** It is the collection of decision problems(problems with a "yes" or "no" answer) that can be solved by a deterministic machine (our computers) in polynomial time.

Features:

- The solution to **P problem**s is easy to find.
- **P** is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

NP Class

The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine (note that our computers are deterministic) in polynomial time.

Features:

- The solutions of the NP class might be hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
- Problems of NP can be verified by a deterministic machine in polynomial time.

Co-NP Class

Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.

Features:

- If a problem X is in NP, then its complement X' is also in CoNP.
- For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer "yes" or "no" in polynomial time for a problem to be in NP or CoNP.

NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

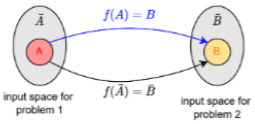
Features:

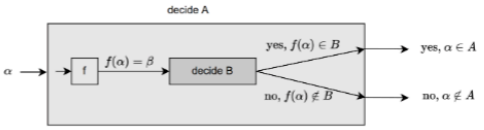
- NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
- If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

2.0 Problem Reduction:

Meaning of $A \leq_p B$

- B is at least as hard as A
- B is harder than A
- A is poly time reducible to B
- $\exists$ poly time computable function f s.t. $\alpha \in A \Leftrightarrow f(\alpha) \in B$
- we can decide $B \Leftrightarrow$ we can decide A
- $A \leq_p B \Leftrightarrow$ running time of $A \leq$ running time of $B + \mathcal{O}(n^k)$





2.5 Three theorems

一、定理 1：NP = coNP 的等价条件

黑板最上面那行：

> THM NP = coNP ⇔ ∃ L ∈ NP-C s.t. L ∈ coNP

翻译成人话：

> NP = coNP 当且仅当：
> 至少存在一个 NP-Complete 的问题 L，
> 它同时也属于 coNP。

二、Cook 定理：CIRCUIT-SAT 是 NP-Complete

黑板中间那块写着：

> Cook's THM (1971)
> CIRCUITSAT ∈ NPC
> "Is there an input assignment so f(C)=1?"

图上画了一个"盒子电路"：左边很多输入线 in，右边一个输出 out。

这部分在说：

> 第一个被证明 NP-Complete 的问题是「电路可满足性」（CIRCUIT-SAT）。

三、定理 2：如果有一个 NP-Complete 在 P 里 ⇒ P = NP

黑板右侧写着：

> THM If ∃ L ∈ NP-C & L ∈ P, then NP = P.
> A ∈ NP, L ∈ NPC any NP language $\phi \leq_p$ L, and solve in poly time.

翻译：

> 如果有某个 NP-Complete 问题 L 竟然可以在多项式时间里解决（L ∈ P），那所有 NP 问题都可以在多项式时间解决，于是 P = NP。

2.1 Proving NPC:

Given decision problem A whose complexity class is unknown

1. Show that $A \in NP$ (i.e. that A can be verified in polynomial time)
   - provide a certificate: the evidence that the solution is an instance of A
   - provide a verification procedure that checks if the certificate is valid
2. Show that $A \in$NP-Hard (i.e. A is at least as hard as all other problems in NP)
   - determine problem $L' \in$NPC you will reduce from (often given)
   - show $L' \leq_p A$, i.e. find polynomial reduction f s.t. $l \in L' \Leftrightarrow \alpha \in A$

Note: we reduce known to unknown, i.e. start with an instance of a known NPC problem, and tranform it to unknown problem. You should not start with the unknown problem and reduce it to known problem.

3.0 Properties illustrated by examples:

Using reductions:
Suppose $A \leq_p B$, then $A \leq B + O(n^k)$, $\exists k \in \mathbb{N}$
if $A \in P$, then $B \in$? we don't know
if $A \in$NPC, then $L' \leq_p A \ \ \forall L' \in$NPC. Also $L' \leq_p A \leq_p B \ \ \forall L' \in$NPC, which implies $B \in$NP-Hard
if $B \in P$, then $A \in P$
if $B \in$NPC, then $A \in NP$

🌟 **用一个超级形象的类比总结：**

✓ **Literal = LEGO 单块**

✓ **Clause = 用 OR 拼成的小积木**

✓ **CNF = 用 AND 把这些小积木叠起来的大建筑**

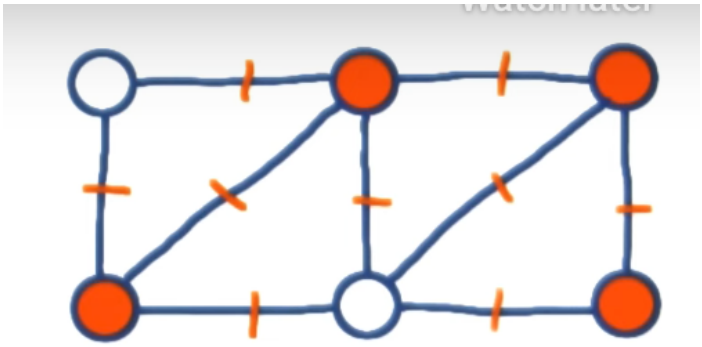
✓ **SAT = 问"建筑是否能搭建成功？"**

✓ **3-CNF-SAT = 建筑规则变更：每个积木必须恰好用 3 块 LEGO 组成**

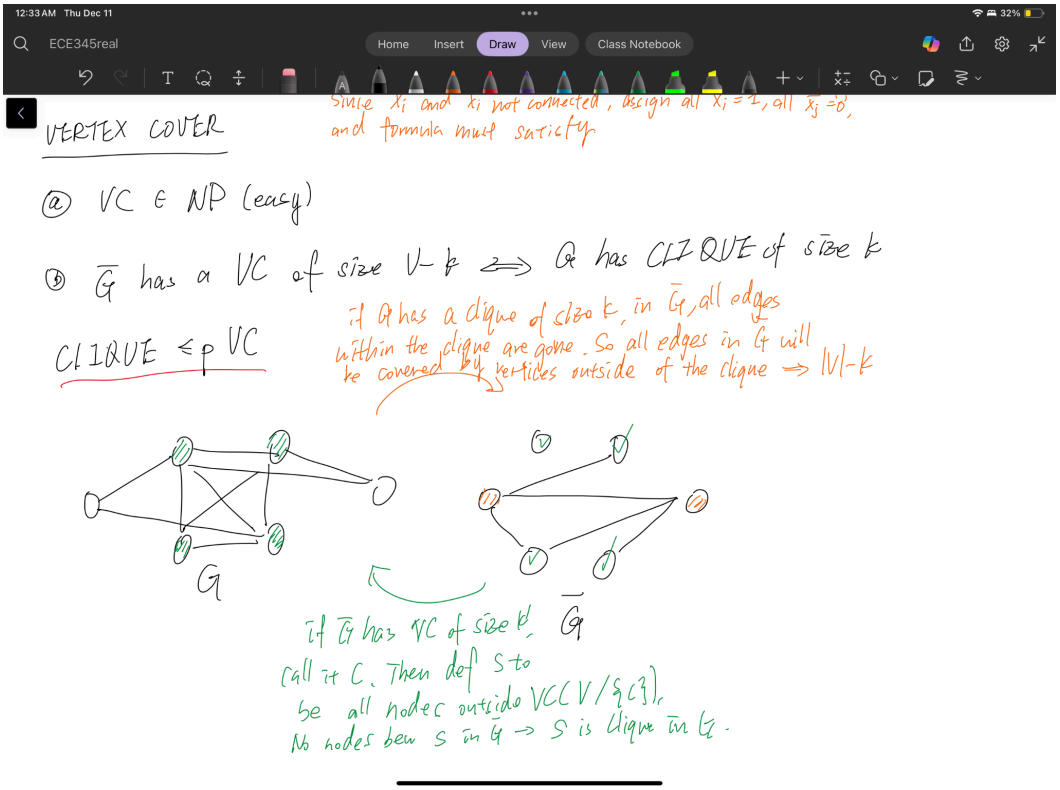
CLIQUE Definition: In graph theory, a **clique** (/ˈkliːk/ or /ˈklɪk/) is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent. That is, a clique of a graph G is an induced subgraph of G that is complete.

Vertex Cover: a vertex cover (sometimes node cover) of a graph is a set of vertices that includes at least one endpoint of every edge of the graph.

Example: the set of orange vertices form a vertex cover



Examples:


12:33 AM Thu Dec 11
ECE345real
Home Insert Draw View Class Notebook
VERTEX COVER
Since x_i and x_i not connected, assign all $x_i = 1$, all $x_j = 0$, and formula must satisfy
(a) VC ∈ NP (easy)
(b) $\bar{G}$ has a VC of size V − k ⟹ G has CLIQUE of size k
CLIQUE ≤p VC
if G has a clique of size k, in $\bar{G}$, all edges within the clique are gone. So all edges in $\bar{G}$ will be covered by vertices outside of the clique ⇒ |V|-k
G
$\bar{G}$
if $\bar{G}$ has VC of size k, call it C. Then def S to be all nodes outside VC (V/{C}). No nodes bew S in $\bar{G}$ → S is clique in G.

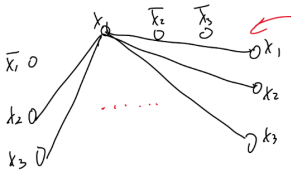
THM: Approximation Clique is NPC

@. Clique $\in$ NP (easy)

Optimization vs Reduction ??

Ⓑ. 3-CNF-SAT $\leq_p$ CLIQUE

$\phi = (\bar{x}_1 \vee x_2 \vee x_2) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee x_2 \vee x_3) \wedge (\cdots) \wedge \cdots$



connect Everything except ones in same partite, ones with its complement

for each clause to be '1', at least 1 term in each clause = 1. If $x_i = 1$, $\bar{x}_i$ is garenteed a '0', which can't be used to satisfy other clauses. $\therefore$ if k clauses all → '1',

→ ϕ has satisfying Assignment iff G has a clique of size (# clause) the satisfying vertices must be connected.

← iff G has clique of size k, means $x_1, x_2 \cdots x_k$ each = '1', Since x_i and $\bar{x}_i$ not connected, assign all $x_i = 1$, all $\bar{x}_j = 0$, and formula must satisfy

VERTEX COVER

Open Addressing

Use hash function of the form $h(k, i)$, try to insert k at $h(k, 0)$, $h(k, 1)$, ..., until we find an empty slot

Linear probing: $h(k, i) = (h'(k) + i) \mod m$

Quadratic probing: $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \mod m$ (slightly better performance, but still m probing sequences)

Double hashing: $h(k, i) = (h_1(k) + i h_2(k)) \mod m$

- No clusters
- $h_2(k)$ must be relatively prime to m (gcd$(h_2(k), m) = 1$) for all k
- Can choose m prime, $h_2(k) < m$, $\forall k$, also require $h_2(k) \geq 1$.
- $\Theta(m^2)$ different probe sequences

Definition

Let U be the universe, $K \subset U$ a set of keys. T a table of size m with indices $\{0, ..., m-1\}$
A hash function $h \to \{0, ..., m-1\}$ maps objects in the universe to the indices (hashes key $k \in K$ to index $h(k)$)

Good hashing scheme:

- Simple uniform hashing: any given element is equally likely to be hashed into any of the m slots.
- Good mechanism for collision resolution (since $m < |U|$, there could be collision)

Load Factor (装载因子) 是衡量 Hash Table **"有多满"** 或者 **"有多拥挤"** 的一个核心指标。

简单直接的公式是：

$$\alpha = \frac{n}{m}$$

- n **(Number of elements):** 表里现在已经存了多少个数据。
- m **(Size of table):** 表里总共有多少个坑位（Buckets/Slots）。

2. **避免死循环与分布均匀:**
   - 数字 6 是合数 (2×3)。如果你使用的探测步长（比如在开放寻址法中）与 2 或 3 有公约数，你可能永远跳不到某些空格子，或者导致分布很不均匀。
   - 数字 7 是素数。素数能最大程度减少模运算 ($k \mod m$) 产生的规律性冲突，并且能让探测序列更容易覆盖整个表。

Primary Clustering:

🔷 2️⃣ Primary Clustering (Linear Probing)

- **Happens in: Linear probing**
  $h_i(k) = (h(k) + i) \bmod m$
- When two keys hash into the *same cluster* (contiguous block of filled slots),
  they share the **same probe sequence pattern** after that point.
- So long clusters tend to grow even longer — new keys join the end of the cluster.

📉 **Effect:**

- The "dense cluster" phenomenon slows insertion and search dramatically.
- Average probes per search increase rapidly as load factor α rises.

✅ **Analogy:**

Once a line starts forming, everyone new joins the same line, making it longer.

Secondary Clustering: if first hash function collide, later attempts will too

🔷 3️⃣ But why secondary clustering still happens

Let's say two different keys, k_1 and k_2, happen to hash to the **same base index**:

$$h(k_1) = h(k_2)$$

That means their probe sequences will be **identical**:

$$h_i(k_1) = (h(k_1) + c_1 i + c_2 i^2) \bmod m = (h(k_2) + c_1 i + c_2 i^2) \bmod m = h_i(k_2)$$

So if k_1 and k_2 collide at the first slot, they'll continue to collide at every subsequent slot in the exact same order.
They'll follow the *same path*, forming a little "mini-cluster" among themselves.

That's **secondary clustering** — clustering caused by keys with *the same hash value* following *identical probe patterns*.

Resolution: Double hashing, second hash doesn't depend on result from first hash

🔷 5️⃣ How to eliminate it entirely: Double hashing

Double hashing uses a second hash function:

$$h_i(k) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

Here $h_2(k)$ depends on the key, so even if $h_1(k_1) = h_1(k_2)$,
their step sizes differ ($h_2(k_1) \neq h_2(k_2)$).

Thus:

- Different probe sequences for different keys → no clustering (beyond random chance).
- That's why **double hashing** removes *both* primary and secondary clustering.

Credit Invariant (存款守恒定律/信用不变量) 是 **Accounting Method (核算法)** 中最核心的一个概念。

如果在考试中题目让你写 "Credit Invariant"，它其实是让你**列出一张"状态对应存款"的清单**。

你需要回答的是：**"当数据结构处于某种特定状态时，它身上必须存有多少钱（Credits）。"**

---

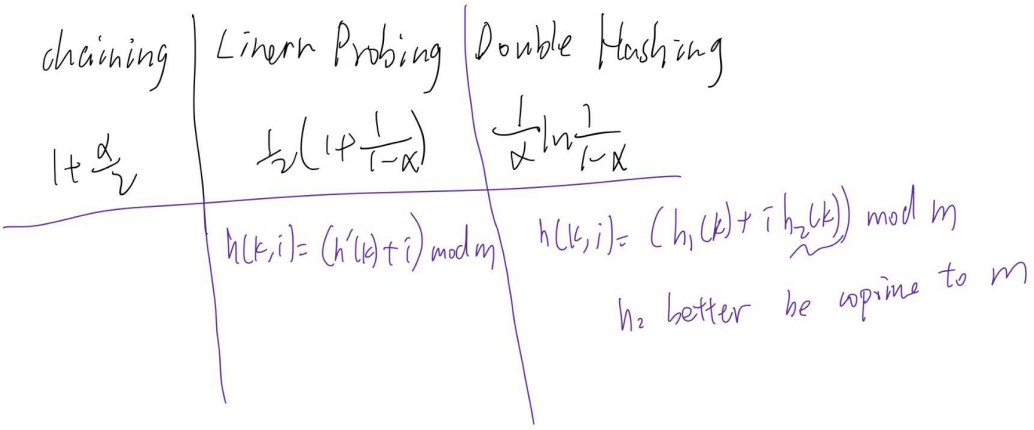
1. 考试标准答案格式

对于这道 **Ternary Counter (三进制计数器)** 的题目，如果考卷上问 "Provide an appropriate credit invariant"，你需要写：

> **Credit Invariant:**
>
> - A trit with value **0** has **$0** credit.
> - A trit with value **1** has **$2** credits.
> - A trit with value **2** has **$1** credit.

(或者用数学符号写：Let $C(d)$ be the credit on a digit d. Then $C(0) = 0, C(1) = 2, C(2) = 1$.)

此信息直接来自题目答案。



Method	Mechanism	Typical choice of table size (m)	Why that choice
Chaining	Each bucket stores a *linked list* (chain) of items that hash to same index	Any (m), ideally a prime not close to power of 2	Helps uniform distribution for modular hash functions
Linear probing	On collision, probe next slot: ($h_i(k)$ = (h(k) + i) mod m)	(m) **prime** and not power of 2	Prevents patterns causing repeated collision cycles

Quadratic probing	$(h_i(k) = (h(k) + c_1\, i + c_2\, i^2) \bmod m)$	(m) **prime**; ensure probe sequence covers all slots	Avoids primary clustering
Double hashing	$(h_i(k) = (h_1(k) + i\, h_2(k)) \bmod m)$	(m) **prime**, ensure $(h_2(k))$ coprime to (m)	Ensures all slots are visited; minimizes clustering