# Complexity Analyze

| Landau Symbol | limit | rate of growth |
|---|---|---|
| $f(n)=o(g(n))$ | $\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$ | $f<g$ |
| $f(n)=O(g(n))$ | $\lim_{n\to\infty}\frac{f(n)}{g(n)}<\infty$ | $f\leq g$ |
| $f(n)=\Theta(g(n))$ | $\lim_{n\to\infty}\frac{f(n)}{g(n)}=c, 0<c<\infty$ | $f\sim g$ |
| $f(n)=\omega(g(n))$ | $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$ | $f>g$ |
| $f(n)=\Omega(g(n))$ | $\lim_{n\to\infty}\frac{f(n)}{g(n)}>0$ | $f\geq g$ |

---

- when $p<q$, $np=o(\ln(n)n^p)$, but $\ln(n)n^p=o(n^q)$
- $(\log n)^k=o(n^\varepsilon), \forall k\in Z^+, \varepsilon\in R^+$
- $n!\approx\sqrt{2\pi n}(\frac{n}{e})^n$
  - 或者表达为：$\lim_{n\to\infty}\frac{n!}{\sqrt{2\pi n}(\frac{n}{e})^n}=1$
  - 解释了为什么 $n!<n^n$, but $\log(n!)\sim\log(n^n)$

if-else: assume the longest branch runs (worse case complexity)

## Analyze Recursive algorithms

4 ways to solve a recurrence relation.

### Direction solution

Guess a solution, then prove it by induction.

### Substitution method

一直展开式子，直到$S(n)=S(1)+\dots$

### Recursion tree method

If $T(n)=aT\left(\frac{n}{b}\right)+\Theta(n^d)$ for constants $a>0, b>1, d\geq 0$, then:

计算树的深度和每层上的开销

### Master theorem

$$T(n)=\begin{cases}\Theta(n^d) & if\ d>log_b a\\ \Theta(n^d log n) & if\ d=log_b a\\ \Theta(n^{log_b a}) & if\ d<log_b a\end{cases}$$

- 
  - 可以处理$T(n)=3T(n/4)+n\log n$, 但无法处理 $T(n)=2T(n/2)+n\log n$
  - $T(n)=2T(\sqrt{n})+\Theta(\log n)$: Let $n=2^m, S(m)=T(2^m)$.

> $T(n)=a\cdot T\left(\frac{n}{b}\right)+f(n)$
>
> 其中 $a\geq 1$，$b>1$，$f(n)$ 是一个正函数，则 $T(n)$ 的渐进上界可以根据以下三种情况中的一种确定：
>
> 1. 如果 $f(n)=O(n^{\log_b a-\epsilon})$ 对某个常数 $\epsilon>0$ 成立，则 $T(n)=\Theta(n^{\log_b a})$。
> 2. 如果 $f(n)=\Theta(n^{\log_b a})$，则 $T(n)=\Theta(n^{\log_b a}\log n)$。
> 3. 如果 $f(n)=\Omega(n^{\log_b a+\epsilon})$ 对某个常数 $\epsilon>0$ 成立，且对某个常数 $c<1$ 和足够大的 $n$，有 $a\cdot f\left(\frac{n}{b}\right)\leq c\cdot f(n)$，则 $T(n)=\Theta(f(n))$。**

# Divide and Conquer

## Selection

find the i-th smallest number in a given array.

---

an algorithm that is similar to quick sort:

randomly choose a piviot. on average, each piviot divide the array equally.

$$T(n)=T(n/2)+O(n)$$
$$T(n)=O(n)$$

But if not random, worse case each pivot only divide out one element.

$$T(n)=T(n-1)+O(n)$$
$$T(n)=O(n^2)$$

---

an algorithm introduced in calss (I think it is just a method introduced to avoid the worse case in the former QS without using random)

`select(A, i)`

### Algorithm

### Analysis

$$S(n)=S(n/5)+S(u)+O(n)$$

- $u$ is the size of whichever partition we recurse on.

**show $u\leq 7n/10$**

- n/10 medians less than x
- each group with median less than x, 3 values in this group less than x
- totally at least 3n/10 values less than x
- similarly, at least 3n/10 values larger than x

Therefore, $u\leq 7n/10$ proved, and

$$S(n)\leq S(n/5)+S(7n/10)+O(n)$$

**guess and prove $S(n)=O(n)$**

suppose $S(n)\leq cn$

$$cn=c(n/5)+c(7n/10)+bn$$
$$c=10b$$

## Multiplying complex number – Gauss' s method

$$(a+bi)(c+di)=x+yi$$
$$x=ac-bd$$
$$y=(a+b)(c+d)-ac-bd$$

- less multiplication, more addition -> much faster when used **recursively**

## Long multiplication of two integers – Karatsuba' s algorithm



- not faster, still $\Theta(n^2)$

### use Gauss' s method



- It does 3 multiplications of digit numbers instead of 4

$$S(n)=3S(n/2)+O(n)$$
$$S(n)=\Theta(n^{\log_2 3})=O(n^{1.59})<O(n^2)$$

## Block matrix multiplication

naive, $\Theta(n^3)$



$$S(n)=8S(n/2)+O(n^2)$$
$$S(n)=\Theta(n^3)$$

- not faster

### use Strassen' s algorithm



$$S(n)=7S(n/2)+O(n^2)$$
$$S(n)=\Theta(n^{\log_2 7})=O(n^{2.81})$$

## Counting Inversions

> CS101 notes/Sort/Merge Sort

- divide the array into 2 halves

**total number of inversions = # in the left half + # in the right half + # between the 2 halves**

### count # between the 2 halves:

**naively, 每两个都比，$\frac{n^2}{4}$**

$$S(n)=2S(n/2)+O(n^2)$$
$$S(n)=O(n^2)$$

- not faster

**Sort (which doesn't change the number of inversions)**

merge and count

$$S(n)=2S(n/2)+O(n)$$
$$S(n)=O(n\log n)$$

> 不要哦从各自的头开始比较，从一个的头和另一个的尾。如果从尾开始的某个数已经比从头开始的某个小，那剩下的就全是Inversion。这应该能做到一个常数级的优化？
>
> 不行。一头一尾 没法把sorted 作为结果

## Maximum Subarray

> 在动态规划里也有

**Goal.** Given array A, find with max $\sum_{k=i+1}^{j}A[k]$

### Solution with D&C.

divide the array from the middle, the wanted array is either:

- in the right half
- in the left half
- across the 2 halves

**Find max subarray crossing the midpoint:**

在左右各自找，和mid连续的最大subarray

$O(n)$

$$S(n)=2S(n/2)+O(n)$$
$$S(n)=O(n\log n)$$

### Solution with DP

> CS101/General Resources/DP
>
> CS101/My Notes/DynamicProgramming
>
> 两个里面都有讲

## Closest Point Pair

- 用一条与x轴或y轴垂直的直线，将平面分成两个区域。每个区域点数接近 - $O(n)$
  - > see **Dividing points evenly**
- 最近点可能在：
  - 左
  - 右
  - 横跨左右
- 找横跨左右的最近点 - $O(n)$
  - > see **Sorting R points by y coordinate**
  - $\delta=\min$(左边最近点的距离，右边最近点的距离)



$$S(n)=2S(n/2)+O(n)$$
$$S(n)=O(n\log n)$$

### Sparsity lemma

### Dividing points evenly

在算法最初，先把所有的点以某一个维度sort $O(n\log n)$

然后每次从这个sort好的array，中分数量

- $O(n)$

### Sorting R points by y coordinate

算法初sort $O(n\log n)$

## Polynomial multiplication – FFT

naively, multiplying 2 polynomials each of degree $n$ takes $O(n^2)$ time.

can also be solve using ***Karatsuba's method***, but higher time complexity $\Theta(n^{\log_2 3})$, while ***FFT*** has $\Theta(n\log n)$

> see CS101/PA/PA2/prob1

多项式的系数表示

**point-value representation**：$n-1$次的多项式可以通过$n$个不同的点取值来确定

那么求$C(x)$相当于：求C在 $x=x_k$ 时的取值，那么如果已知A和B 在 $x=x_k$ 的取值，直接 $C(x_k)=A(x_k)B(x_k)$ for all $k$

Therefore, $C(x)=A(x)B(x)$.

# Linear time sorting

基于比较的排序，理论上时间复杂度的下限就是 $O(n\log n)$



而非基于比较的排序理论下限 $O(n)$，因为至少要把输入读一遍并且输出

## Counting sort

**Requirement**: integers in the range 0 to k

- In (a), contains number of occurrences of each input value in A
- 计算prefix sum也就是图(b)中的C
- iterate through in reverse order.



$\Theta(n+k)$

if $k=O(n)$, then complexity is $\Theta(n)$

stable -> can be used for radix sort

## Radix sort

Sort digit by digit, from least to most significant digit.(从最低位起sort，必须stable算法)

- Suppose we sort $n$ $d$-digit numbers, where each digit is between 0 to k-1. Then radix sort takes $O(d(n+k))$ time.
- Given $n$ $b$-bit numbers and $r\leq b$. Radix sort takes $O(\frac{b}{r}(n+2^r))$ time.
  - Break the $b$ bits into blocks of $r$ digits, having values between 0 and $2^r-1$
- Setting $r=\min(\lfloor\log n\rfloor, b)$ minimizes the running time $\Theta(\frac{b}{r}(n+2^r))$
- radix sort is efficient when there are many short numbers, but not when there are a few long numbers.

# Greedy

## Selecting Breakpoints

gas refueling problem

save refuel time

**proof.**

the greedy algorithm always choose to go as far as possible, so OPT can't go further than Greedy.

If $g_{r+1}>f_{r+1}$, than OPT must require one more step than Greedy, then it's not optimal. Contradicts.

## Coin Changing

**Theorem**. Greedy is optimal for U.S. coinage: 1, 5, 10, 25, 100.

**proof.**

the greedy algorithm always choose to go as far as possible, so OPT can't go further than Greedy.

If $g_{r+1} > f_{r+1}$, than OPT must require one more step than Greedy, then it's not optimal. Contradicts.

## Coin Changing

**Theorem**. Greedy is optimal for U.S. coinage: 1, 5, 10, 25, 100.

**proof.**

find max vlaue of coins in any OPT
prove that without the least max coin, can't achieve opt

Such greedy algorithm **doesn't work for all** coin combinations.

**Observation**. Greedy algorithm is sub-optimal for US postal denominations: 1, 10, 21, 34, 70, 100, 350, 1225, 1500 .

## Interval scheduling

**Goal**. Given a set of intervals, pick the largest number of non-overlapping ones.

- sort by finishing time
- choose **earliest finishing** one that doesn't overlap previous selected interval

Overall $O(n \log n)$ time.

**proof.**

greedy is not worse than the optimal solution. prove by **induction**

Greedy solution $S$, optimal solution $T$.
Let $s_k$ and $t_k$ be k'th interval in $S$ and $T$, resp.
Let $fin(i)$ be finishing time of an interval $i$.
**Claim**. $fin(s_k) \leq fin(t_k)$ for all $k$

## Interval coloring problem

**Goal**. schedule all the intervals on some number of resources. Intervals on the same resource cannot overlap. Minimize the number of resources used.

**Observation**. Suppose $k$ intervals intersect at some time point. Then the optimal schedule needs at
least $k$ resources.
**Def**. ***Depth*** of a set of intervals is the max number of intervals that intersect at any time.

**Corollary**. Let $d$ be the depth of a set of intervals, and suppose we find a schedule using $d$ resources. Then the schedule is optimal.

- Sweep through intervals in order of **increasing start time**.
- For each interval, assign it to smallest resource not already assigned to an intersecting interval.

**Claim**. Let the set of intervals have depth d. Then the algorithm uses d resources

## Scheduling to Minimizing Lateness

- **Earliest deadline first** -> optimal

**Observation**. There exists an optimal schedule with no idle time.

**Observation**. The greedy schedule has no idle time.

**Def**. An ***inversion*** in schedule S is a pair of jobs i and j such that: i < j but j scheduled before i. *(I suppose the i and j here means deadline time)*

**Observation**. Greedy schedule has no inversions.

an existing inversion always make the lateness larger.

**Claim**. Swapping two adjacent, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

**Pf. Greedy schedule $S$ is optimal.**

Define $S*$ to be the optimal solution. Then $S*$ must has the fewest number of inversions.

- Can assume $S*$ has no idle time.
- If $S*$ has no inversions, then $S = S*$.
- If S* has an inversion, let i-j be an adjacent inversion.
  - swapping i and j does not increase the maximum lateness and strictly decreases the number of inversions

## Caching

cache: fast, closer to CPU core, not quite large

- store only the most important data in cache
- caching algorithm to decide which data to store

### Optimal Offline Caching

***offline***: at the beginning, **know** which data are going to be accessed

assume only one layer of cache
Intuition. Can transform an unreduced schedule into a reduced one with no
more evictions.

***cache hit***: in cache
***cache miss***: if not find in cache, go to memory, load in into cache, choose to evict some existing item.

**Goal.** Eviction schedule that minimizes number of evictions.

**Farthest-in-future**. Evict item in the cache that is not requested until farthest in the future

- FF is optimal offline eviction algorithm




**Proof.**

**Def**. A ***reduced*** schedule is a schedule that only inserts an item into the cache in a step in which that item is requested.

**Intuition.** Can transform an unreduced schedule into a reduced one with no more evictions. *(There is no need to prepare ahead of time)*

> see proof on ppt

prove by induction

**Lemma.** If the optimal reduced schedule $S$ share the same first $j$ requests as $S_{FF}$, then they also share the same $j+1$ request.

pf: Consider $(j+1)$ st request $d = d_{j+1}$

- **Case 1**: ($d$ is already in the cache). $S' = S$
- **Case 2**: ($d$ is not in the cache and $S$ and $S_{FF}$ evict the same element). $S' = S$
- **Case 3**: ($d$ is not in the cache; $S_{FF}$ evicts $e$; $S$ evicts $f \neq e$).

  From request j+2 onward, we make S' the same as S, but this becomes impossible when e or f is involved
  - **Case 3a**: $g = e$
  - **Case 3b**: $g \neq e, g \neq f$
  - **Case 3c**: $g = f$

### Online Caching

***Online (reality)***: requests are not known in advance.

- **LIFO**. Evict page brought in most recently.
  - LIFO is arbitrarily bad: For $n$ requests, it may do $O(n)$ times more loads than optimal.
- **LRU**. Evict page whose most recent access was earliest
  - LRU is k-competitive: it does times more loads than the optimal eviction algorithm

## Huffman Coding

**prefix-free** codes -> unique decoding
Huffman encoding is an optimal prefix-free code

$n$ different chars to encode -> $O(n^2)$ time

**Pf. optimal**

# Dynamic Programming

## Optimal substructure

After making a decision, the rest of the solution should be optimal for the rest of the problem.

Not all problems have optimal substructure.

## Weighted Interval Scheduling

**Goal**. Pick a set of non-overlapping intervals with the largest combined weight.

- Given interval $I_j$, let $p(j)$ be the maximum index $k$ s.t. $I_k$ finishes before $I_j$ starts.
  finishes before starts.
- Let $OPT(j)$ be the weight of a max weight non-overlapping subset of $I_1, \ldots, I_j$.

## Segmented Least Squares

- Points lie roughly on a sequence of several line segments.
- Given n points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$ , find a sequence of lines that minimizes $E + cL$
  - E: the sum of the sums of the squared errors in each segment
  - L: the number of lines
  - c: some positive constant

$OPT(j)$ = minimum cost for points $p_1, p_{i+1}, \ldots, p_j$ .

$e(i, j)$ = minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$ .

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max_{1 \leq i \leq j} \{e(i,j) + c + OPT(i-1)\} & \text{otherwise} \end{cases}$$

**Running time**. $O(n^3)$

## Subset Sum

背包问题的变种

**Goal**. Each job $i$ takes $w_i$ resource. We have $W$ resources. Find a set of jobs $S \subseteq \{1, \ldots n\}$ to maximize $\sum_{i \in S} w_i$, subject to $\sum_{i \in S} w_i \leq W$.

**Greedy algorithm**. Sort items from largest to smallest. Insert items sequentially, as many as possible. -> Always achieves at least 1/2 the max possible sum.

**DP**. Let $OPT(i, W')$ be max weight of a subset $S \subseteq \{w_1, \ldots, w_i\}$, subject to $\sum_{i \in S} w_i \leq W'$.

$$OPT(i, W') = \max(OPT(i-1, W'), w_i + OPT(i-1, W' - w_i))$$

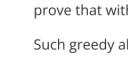
Time complexity and memory complexity are both $O(nW)$.

## Matrix-chain multiplication

choose which point to break on

**Goal**. Given a sequence $A_1, A_2, \ldots, A_n$ of matrices, where $A_i$ has dimensions $p_{i-1} \times p_i$, for $i = 1, \ldots n$, compute the product $A_1 \times A_2 \times \cdots \times A_n$ in a way that minimizes the cost.

Let $M(i, j)$ be the smallest time to multiply matrices of $A_i \times A_{i+1} \times \cdots \times A_j$




$$M(1,n) = \min_{1 \leq i \leq n-1} (M(1,i) + M(i+1,n) + p_0 p_i p_n)$$

Cost of multiplying all the matrices $A_1, \ldots, A_n$ | Choose the best break point $j$ | Cost of the first part | Cost of the second part | Cost of multiplying the matrices produced by the first and second parts

## Table method to solve the DP space

关键在于先找出DP的前后依赖关系

Total time cost: $O(n^3)$

Space complexity: $O(n^2)$

## Problems with(out) optimal substructure

**Shortest paths has optimal substructure**

$$d(x, y) = \min_{(z,y) \in E} (d(x, z) + w(z, y))$$

shortest path from x to y is to take shortest path from x to one of y's neighbors, then go to y

**Longest simple path does not have optimal substructure.**

When sub-solutions aren't combinable, we say the subproblems aren't independent

## Longest common subsequence

Remove the last letter of the long string

- Let $S[1, i]$ be the first i letters of a string S, and $S[i]$ be the $i$'th letter of S.
  - Let $S[i, 0]$ be the empty string, for any i.
- Let $LCS(X[1, i], Y[1, j])$ be the LCS of $X[1, i]$ and $Y[1, j]$.
- Let $c(i, j)$ be the length of $LCS(X[1, i], Y[1, j])$.

$$c[i, 0] = 0 \text{ for all i}$$
$$c[0, j] = 0 \text{ for all j}$$
$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i-1.j, c[i, j-1]]) & \text{if } X[i] \neq Y[j] \end{cases}$$




Time complexity and memory complexity are both $O(mn)$.

# Network Flow

***duality*** between ***max flow*** and ***min cut***

The weight of the edge stands for its capacity.

## Flows

### *s-t flow*

- Each edge can have an amount of flow less than its capacity.
  - $0 \leq f(e) \leq c(e)$ for each $e \in E$
- The sum of in-flow equals the sum of out-flow (except the source and sink)
  - $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ for each $v \in C - \{s, t\}$

### *value* of a flow $f$

$$v(f) = \sum_{e \text{ out of } s} f(e)$$

### Greedy

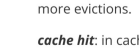
- 单走一条source 到 t 的路径，取上面所有edge的剩余的capacity的最小值作为f(e)
- 重复此步骤直到stuck

not always optimal

- 局部最优无法达到全局最优

undo to go out of the trapped decision

### Residual Graph



## Augmenting Path

a simple $s-t$ path P in the residual graph $G_f$.

## Ford-Fulkerson Algorithm

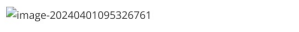
可以反悔的greedy

如果没有$e^R$这条反向edge每次循环后的添加，就等于纯粹的greedy

# Cuts

An ***s-t cut*** is a partition (A, B) of V with $s \in A$ and $t \in B$.

The ***capacity*** of a cut (A, B) is: $cap(A, B) = \sum_{e \text{ out of A}} c(e)$



## Flow value lemma

Let v be any vlow, and let (A, B) be any s-t cut.
Then, the net vlow sent across the cut is equal to the amount leaving s.

$$\sum_{e \text{ out of A}} f(e) - \sum_{e \text{ in to A}} f(e) = v(f)$$

## Weak Duality

Let v be any flow, and let (A, B) be any s-t cut. Then the value of the flow is at most the capacity of the cut.

**Corollary.** Let v be any vlow, and let (A, B) be any cut.
Iv v(v) = cap(A, B), then v is a max vlow and (A, B) is a min cut.

the equivalence of the following three conditions for any flow f:

- There exists a cut (A, B) such that v(v) = cap(A, B).
- Flow v is a max vlow.
- There is no augmenting path relative to v.

# Choosing Good Augmenting Paths

**Assumption.** All capacities are integers between 1 and C.

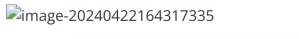
n 是所有edge capacity之和？

Running time of Ford-Fulkerson is $O(mnC)$.

# Applications / reductions

## Bipartite Matching

Given an undirected graph $G = (V, E)$, $M \subseteq E$ is a ***matching*** if each node appears in at most one edge in $M$.
***Max matching***: find a max cardinality matching.



**Theorem**. Max cardinality matching in G = value of max flow in G'

## Perfect Matching (marriage)

A matching M ⊆ E is ***perfect*** if each node appears in exactly one edge in $M$.

**Marriage Theorem**. Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. Then, $G$ has a perfect matching iff $|N(S)| \geq |S|$ for all subsets $S \subseteq L$.

## Edge Disjoint Paths

**Def**. Two paths are ***edge-disjoint*** if they have no edge in common.

**Disjoint path problem**. Given a digraph $G = (V, E)$ and two nodes s and t, find the max number of edge-disjoint s-t paths.

**Max flow formulation**: assign unit capacity to every edge.

**Theorem**. Max number edge-disjoint s-t paths = max flow value

## Network Connectivity

**Network connectivity**. Given a digraph $G = (V, E)$ and two nodes s and t, find min number of edges whose removal disconnects t from s.
**Def**. A set of edges $F \subseteq E$ disconnects t from s if all s-t paths uses at least on edge in F.

**Theorem**. The max number of edge-disjoint s-t paths = the min number of edges whose removal disconnects t from s

## Circulation with Demands

For each $v \in V$: $\sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$

**Circulation problem**: given $(V, E, c, d)$, does there exist a circulation?

**Necessary condition**: sum of supplies = sum of demands.

$\sum_{v:d(v)>0} d(v) = \sum_{v:d(v)<0} -d(v)$

## Max flow formulation.

- Add new source s and sink t.
- For each v with d(v) < 0, add edge (s, v) with capacity -d(v).
- For each v with d(v) > 0, add edge (v, t) with capacity d(v).

**Claim**: G has circulation iff G' has max flow of value D.

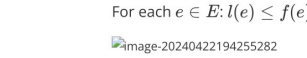
**Integrality theorem**. If all capacities and demands are integers, and there exists a circulation, then there exists one that is integer-valued.

**Characterization**. Given $(V, E, c, d)$, there does not exists a circulation iff there exists a node partition $(A, B)$ such that Σ v∈B d v > cap(A, B)

## Circulation with Demands and Lower Bounds

Edge capacities $c(e)$ and lower bounds $l(e), e \in E$.

For each $e \in E$: $l(e) \leq f(e) \leq c(e)$



**Theorem**. There exists a circulation in G iff there exists a circulation in G'. If all demands, capacities, and lower bounds in G are integers, then there is a circulation in G that is integer-valued.

# NP, reductions

**Def.** P is the set of all problems that can be solved by a polytime algorithm.

# NP = Nondeterministic polynomial time.

**Def**. NP is the class of problems for which the solvability of an instance can be verified in polynomial time. (未知是否有多项式时间的解，但是解的可行性可以在多项式时间内得到验证)

- To show a problem is in P, give an algorithm solving the problem that runs in polynomial time.
- To show a decision problem is in NP, give a polynomial time verifier for the problem satisfying the properties on the previous slide.

1. 4-coloring is in NP
2. Factoring is in NP
3. Traveling salesman is in NP
4. k-Clique is in NP
5. **All problems in P are in NP**

6. Primes is in NP (Pratt certificates https://en.wikipedia.org/wiki/Primality_certificate)

# Incorrect verifiers

1. Always outputs 1, regardless of y.
2. Always output 0, regardless of y.
3. Check all subsets of k nodes. If any form a clique, output 1, else output 0. -> not poly-time

## P vs NP

$P \subseteq NP$

But we don't know whether $NP \subseteq P$.

That is, we don't know whether $P = NP$

## Reduction

Let A and B be two decision problems.
Let X and Y be the set of yes instances for A and B, resp.
If the mapping function from $A$ to $B$ runs in polynomial time, then it's a ***polynomial time reduction***, and we write $A \leq_P B$.

**Thm 1**. Let A, B and C be three problems, and suppose $A \leq_P B$ and $B \leq_P C$. Then $A \leq_P C$

## NP-completeness

**Def**. A problem A is NP-complete (NPC) if the following are true

- $A \in NP$
- Given any other problem $B \in NP, B \leq_P A$

## SAT = satisfiable Boolean formulas

Given a Boolean formula, is there any setting for the variables which makes the formula true?

Steve Cook and Leonid Levin proved around 1970 that **SAT is NP-complete**.

**Cook-Levin theorem** says 2 things:

1. SAT ∈ $NP$
2. Every NP problem reduces to SAT. I.e. every problem A in NP can be mapped to an SAT formula in polytime, such that
   - If A is true, then $\phi$ is satisfiable.
   - If A is false, then $\phi$ is not satisfiable.

For every problem in the picture, if A points to B, it means $A \leq_P B$.



**Thm**. Given two NP problems A and B, suppose A is NP-complete, and $A \leq_P B$. Then B is also NP-complete.

**Thm 2**. Suppose a problem $A$ is NP-complete, and $A \in P$. Then $P = NP$.

**Cor**. Suppose a problem A is NP-complete, and $A \notin P$. Then for any NP-complete problem B, $B \notin P$

## To prove a problem A is NP-complete:

1. Prove that problem A is in NP by showing:
   - poly-size certificate
   - poly-time certifier
2. Choose a problem B in NP-Complete to reduce from
3. Construct poly-time many-one reduction f that maps instances of problem A to instances of problem B
4. Prove the correctness of reduction
   - x is a yes-instance of problem A → f(x) is a yes-instance of problem B
   - f(x) is a yes-instance of problem B → x is a yes-instance of problem A
     - or: x is a no-instance of problem A → f(x) is a no-instance of problem B

# Beyond NP

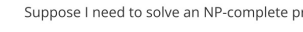
**co-NP.** All problems whose "complement" is in NP

- E.g. GRAPH-ISO∈NP, so GRAPH-NONISO∈co-NP

# Reduction

## Reducing 3-CNF-SAT to CLIQUE

## Reducing 3-CNF-SAT to SUBSET-SUM



# Extending Tractability

Suppose I need to solve an NP-complete problem in poly-time:

**Must sacrifice one of three desired features.**

- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

This lecture. **Solve some special cases of NP-complete problems.**

# finding small vertex covers

**Claim**. Let (u, v) be an edge of G. G has a vertex cover of size ≤ k iff at least one of G − { u } and G − { v } has a vertex cover of size ≤ k − 1.

**Claim**. If G has a vertex cover of size k, it has ≤ k (n − 1) edges

**Claim**. The following algorithm determines **if G has a vertex cover of** $size \leq k$ **in** $O(2^k kn)$ **time.**

```
Vertex-Cover(G, k) {
    if (G contains no edges)    return true
    if (G contains ≥ kn edges)  return false

    let (u, v) be any edge of G
    a = Vertex-Cover(G - {u}, k-1)
    b = Vertex-Cover(G - {v}, k-1)
    return a or b
}
```

Running time:

# Solving NP-hard problems on trees

independent set on general graph is NP-problem.

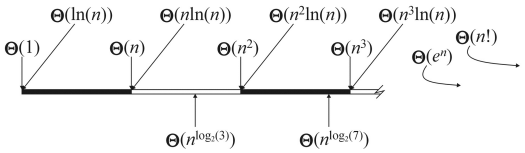
## independent set on trees

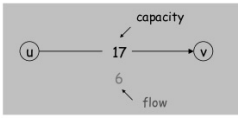
**Independent set on trees.** Given a tree, find a maximum cardinality subset of nodes such that no two share an edge.

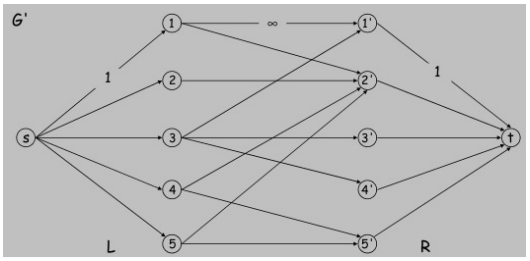
**Key observation.** If v is a leaf, there exists a maximum size independent set containing v.

**Theorem.** The following greedy algorithm finds a maximum cardinality independent set in forests (and hence trees).

```
Independent-Set-In-A-Forest(F) {
    S ← φ
    while (F has at least one edge) {
        Let e = (u, v) be an edge such that v is a leaf
        Add v to S
        Delete from F nodes u and v, and all edges
        incident to them.
    }
    return S
}
```



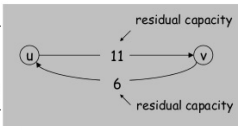



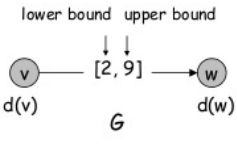

**a** = 10001101 **b** = 11100001

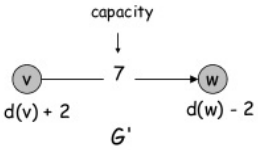
$a_1$ $a_0$ $b_1$ $b_0$

$$a = 2^{n/2} \cdot a_1 + a_0$$
$$b = 2^{n/2} \cdot b_1 + b_0$$
$$ab = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0$$
$$= 2^n \cdot a_1 b_1 + 2^{n/2} \cdot \left((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0\right) + a_0 b_0$$

$$(a + bi)(c + di) = x + yi$$
$$x = ac - bd$$
$$y = (a + b)(c + d) - ac - bd$$



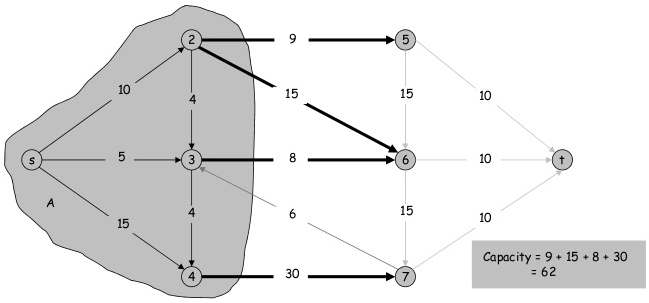




$C_{11}$ $A_{11}$ $A_{12}$ $B_{11}$ $B_{21}$

$$\begin{bmatrix} 152 & 158 & 164 & 170 \\ 504 & 526 & 548 & 570 \\ 856 & 894 & 932 & 970 \\ 1208 & 1262 & 1316 & 1370 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \times \begin{bmatrix} 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 \\ 28 & 29 & 30 & 31 \end{bmatrix}$$

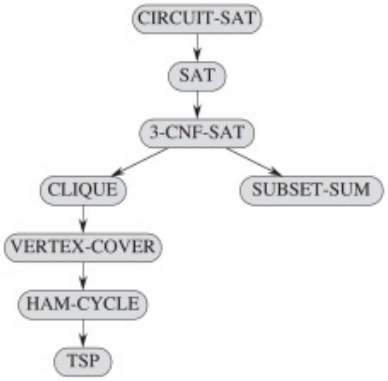
$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}$$

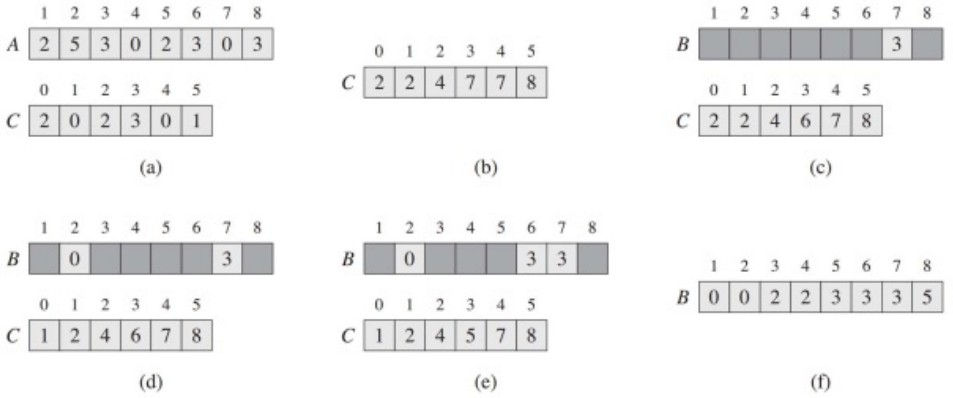



$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

$$\begin{aligned} P_1 &= A_{11} \times (B_{12} - B_{22}) \\ P_2 &= (A_{11} + A_{12}) \times B_{22} \\ P_3 &= (A_{21} + A_{22}) \times B_{11} \\ P_4 &= A_{22} \times (B_{21} - B_{11}) \\ P_5 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 &= (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{aligned}$$




$$M(1,n) = \min_{1 \le j \le n-1} (M(1,j) + M(j+1,n) + p_0 p_j p_n)$$

*Cost of multiplying all the matrices $A_1,\ldots,A_n$.* *Choose the best break point j* *Cost of the first part* *Cost of the second part* *Cost of multiplying the matrices produced by the first and second parts*