Content:

Divide and Conquer　　分治法

Topics:

Karatsuba（大整数乘法 4->3）、Strassen（大矩阵乘法 8->7）、Sorting（归并排序）

Counting Inversions（逆序对计算）、Median-of-the-median（5 个 5 个一组排序然后取中位数）

Closest pair（最近点对）、Fast Fourier Transform（快速傅里叶变换）

Master Theorem 主定理进行算法复杂度递推式分析（样例可以多注意）或者猜后再证

正确性证明：基于归纳法进行证明（Induction）

Graph Algorithms　　图相关算法

Topics:

DFS, BFS, Dijkstra, Bellman-Ford, Floyd-Warshall (DP), Kruskal and Prim(Greedy)

Find SCC or cycle (based on DFS). Traverse the graph (DFS&BFS ok). Cross edge and back edge.

最短路径算法　BFS for unweighted single source. Dijkstra for positive weighted single source.

Bellman-Ford for general weighted single source and Floyd-Warshall for all.

Greedy 贪心算法

Topics:

Exact Algorithms: MST (minimum spanning tree), Task Schedule (earliest first) & Huffman Coding

Prim & Kruskal (Union-Find Set, Path Compression)

Key: Show that the solution at the current iteration is still a part of an optimal solution.

Approximation Algorithms: Make-span Minimizing & set cover/ max-k-coverage

Easy to design but hard to analyze.

Require adequate understanding on the problem's nature. Find a "reference" that your solution can compare with.

Reference: OPT, or lower bound (upper bound) to OPT. and other tricks….

Dynamic Programming　　动态规划

Topics:

从 DAG 入手，需要理解整个状态转移的过程。

LIS (最长不减子序列), Edit Distance (字符串转换且改变次数最小), Knapsack (背包问题)

Floyd-Warshall, Independent Set on Trees (also a greedy algorithm)

Correctness 证明 induction: Validity of recurrence relation is just the validity of inductive step!

Flow and Matching　　流问题与匹配

Topics:

Ford-Fulkerson Algorithm　O(|E|*fmax)

Edmonds-Karp Algorithm　O(|V|*|E|^2)

Dinic's Algorithm　O(|V|^2*|E|)

需要理解这几个算法的核心推导过程以及分析理解的过程。

Linear Programming　　线性规划问题

Topics:

包括计算最优计算 Optimal 等等

Hardness and Approximation Algorithms 困难问题与近似算法

Topics:

NP-complete 只是判定问题！　NP-hard 有更多更复杂的问题。

课上提供的都可以直接拿来使用来证明问题

Divide and Conquer　　分治法

Idea:

大整数乘法一般都需要 a*b 进行逐位运算，若前者长度 m 后者长度 n 则时间复杂度为 O(nm)，得到的方式可以参照竖式相乘法的运算过程。

Result: Divide and conquer for multiplication.

Example: $1234 * 5678$

拆成 1200*5600+1200*78+34*5600+34*78，实际上从 2*4bit->4*2bit

$$xy = \left(a * 10^{\frac{n}{2}} + b\right)\left(c * 10^{\frac{n}{2}} + d\right) = ac * 10^{n} + (ad + bc) * 10^{\frac{n}{2}} + bd$$

运算之后，可以发现，似乎没有任何的效率提升，仍旧需要同样的计算规模。事实上，对于 4 位数字相乘，最后总能转化为 16 次 1 位数字相乘，即运算过程取决于$\log_2 n$。然而，这样的效率由于是基于递归的方式计算，看似复杂度相同，实际运行时耗费的时间比平常的高精度乘法还大，不现实，需要考虑优化?

Question: How to decline the number of items?

Answer: calculate: $a * c,\ b * d,\ (a + b)(c + d)$ which means this just costs 3 items.

$$xy = \left(a * 10^{\frac{n}{2}} + b\right)\left(c * 10^{\frac{n}{2}} + d\right) = ac * 10^{n} + (ad + bc) * 10^{\frac{n}{2}} + bd$$

$$xy = ac * 10^{n} + \left((a + b)(c + d) - ac - bd\right) * 10^{\frac{n}{2}} + bd$$

Result:$3^{\log_2 n} = n^{\log_2 3} \approx n^{1.6}, O(n^{1.59}) \approx O(n^{1.6}) \ll O(n^2)$, great!

Question:如果不是二的倍数怎么办？最简单方法就是补零到二倍。

Better algorithms:

1963	$O(n^{1.465})$	1971	$O(n \log n \, n \log\log n)$
2007	$O(n \log n \ n \log^* n)$	2019	$O(n \log n)$

Extend to matrix multiplication

How to multiply two matrices? For example:

$$X = \begin{vmatrix} A & B \\ C & D \end{vmatrix}, Y = \begin{vmatrix} E & F \\ G & H \end{vmatrix}, XY = \begin{vmatrix} AE + BG & AF + BH \\ CE + DG & BF + DH \end{vmatrix} = \begin{vmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{vmatrix}$$

Can it be done by ***Divide and Conquer***?　Key fact:

$$P_1 = A(F - H), P_2 = (A + B)H, P_3 = (C + D)E, P_4 = D(G - E),$$

$$P_5 = (A + D)(E + H), P_6 = (B - D)(G + H), P_7 = (A - C)(E + F)$$

Result:$7^{\log_2 n} = n^{\log_2 7} \approx n^{2.81}$

Sorting Problem:

Input: A set of n integers x1, x2, …, xn

Output: The same set of n integers in **ascending order**.

I: Insertion Sort

可以用一些高级数据结构如 AVL、红黑树等来进行优化，但是经常得不偿失，不如考虑算法。

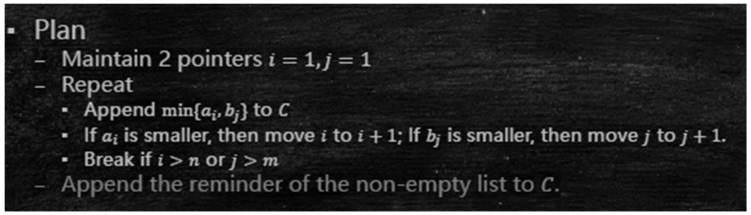
II: Merge Sort

本质上是分治思想：Divide => Recurse => Combine => Basic solver

花费代价在合并过程中最大，考虑有无相应优化？

Input: two sorted lists A,B Output: a sorted list C

关键：插入的过程是单调递增的，即可以用双指针的方式枚举插入变为 O(n)的合并



时间复杂度分析：每次合并 n 长度的 a 数组与 m 长度的 b 数组，耗费代价为 O(n+m)总时间代价为：

设N是2的幂，则

T(1) = 1

T(N) = 2T(N/2) + N

两边除N，得

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1$$

...

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

累加后得：

$$\frac{T(N)}{N} = \frac{T(1)}{1} + \log N$$

即：

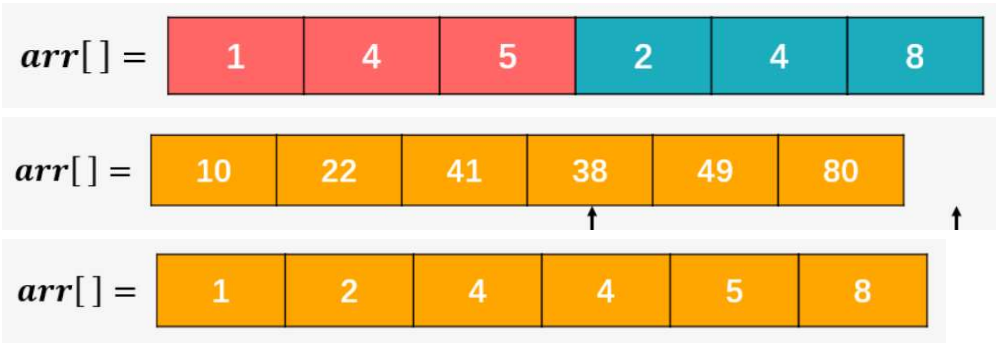
T(N) = NlogN + N = O(NlogN)

时间上是恒定的，因此可以认为是比较稳定的算法

空间上需要额外的空间

→→牺牲时间换空间，牺牲空间换时间，这句话非常重要。

归并排序存在 O(1)额外的空间复杂度方式，即 arr[i] = arr[i] + arr[j] * maxval 同时表示 arr[i] 和 arr[j]



主定理常见计算的应用

(a) $T(n) = 2T(n/3) + 1$ — a) $T(n) = O(n^{\log_3 2}) \approx O(n^{0.63})$

(b) $T(n) = 5T(n/4) + n$ — b) $T(n) = O(n^{\log_4 5}) \approx O(n^{1.16})$

(c) $T(n) = 7T(n/7) + n$ — c) $T(n) = O(n \log n)$

(d) $T(n) = 9T(n/3) + n^2$ — d) $T(n) = O(n^2 \log n)$

(e) $T(n) = 8T(n/2) + n^3$ — e) $T(n) = O(n^3 \log n)$

(f) $T(n) = 49T(n/25) + n^{3/2}\log n$ — f) $T(n) = O(n^{3/2} \log n)$

(g) $T(n) = T(n-1) + 2$ — g) $T(n) = O(n)$

(h) $T(n) = T(n-1) + n^c$，其中常数 $c \geq 1$ — h) $T(n) = O(n^{c+1})$

(i) $T(n) = T(n-1) + c^n$，其中某一常数 $c \geq 1$ — i) $T(n) = O(n^{\log_4 5}) \approx O(n^{1.16})$

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

(k) $T(n) = T(\sqrt{n}) + 1$ — k) $T(n) = O(\log \log n)$

Counting Inversions（逆序对计算）

Input: a list of n integers x1, …, xn Output: number of **inversions**.

Property: i, j are **inverted** if i<j but xi>xj.

Plan1: Brute-force 暴力枚举的方法是 O(n^2)的代价

Plan2: Divide and Conquer 分治思想求解，通过子任务的方式计算逆序对数。

假设两个子任务的逆序对数分别为 c1, c2，合并后产生的逆序对数为 c3

则总逆序对数为 c1 + c2 + c3，但是似乎需要每个子任务之间枚举？子任务的规模 O(n^2)

这样的最终规模是 O(nm)

Plan3: Merge & Count 同时在归并排序的过程中进行计数

关键：i = 1, j = 1 and a counter c = 0

a_i 小则 i++, b_j 小则 j++，i 移动那么答案的贡献为 c+=(j-1)

最终再进行结果累加为 c += m*(n-i+1)

One-by-one Selection

Input: a set S of n integers and an integer k　　Output: The k-th smallest integer x* among.

Plan1: Brute-force

Plan2: **Divide and Conquer** (Merge sort)

Plan3: Divide and Conquer: 将这个序列分成 3 部分，大于 k 小于 k 等于 k 的部分然后再重新合并

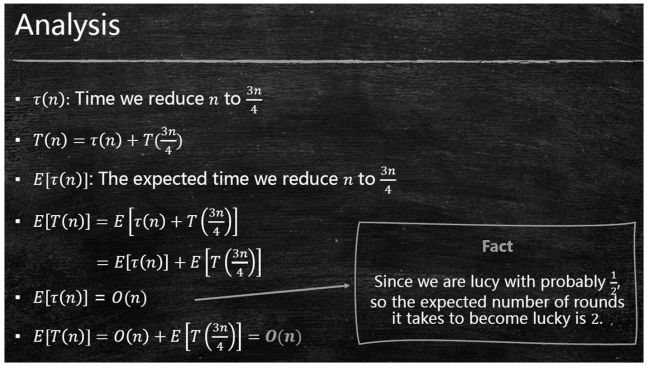
1 2 1 0 3 3 5 4 分成三部分 LMR，然后根据 k 的大小递归调用其中某一部分即可。

L: the k-th in L　　M: actually the M number　　R: the (k-|L|-|M|)th in R

该算法的时间复杂度分析：应该是 O(n^2)的

T(n)<=O(n) + max[T(|L|), T(|R|)]<=O(n)+T(n-1)<=O(n)+O(n-1)+…+O(1)=O(n^2)

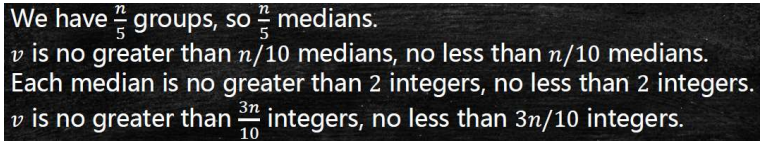
最坏情况：O(n^2)　　最好情况：O(n)



现在不想随机了，想每次都选到很好的结果怎么办呢（找中位数的中位数）？

为什么这个方法更加优秀呢？它每次找到**中位数的中位数**，都能满足尽可能在“中间”的情况。

每个中位数都应该不大于并且不小于 3 个数，因此最后是 3*n/10。



$$T(n) = T(0.2n) + T(0.7n) + O(n)$$

（用试的方式发现）T(n)=O(n)

Question: 如果在分组的时候通过 2、3、4、6、7、8……会怎么样呢？为什么选 5 更好？

首先，选的应该是奇数。假设该奇数为 2k+1，那么：

$$Groups: \frac{n}{2k+1} \rightarrow \frac{n}{2k+1} medians$$

v is no greater than $\frac{n}{4k+2}$ medians, no less than $\frac{n}{4k+2}$ medians

Each median is no greater than k+1 integers, no less than k+1 integers.

v is no greater than $\frac{(k+1)n}{4k+2}$ integers, no less than $\frac{(k+1)n}{4k+2}$ integers.

$$T(n) = T\left(\frac{n}{2k+1}\right) + T\left(\frac{(3k+1)n}{4k+2}\right) + O(n) \Rightarrow 3k+3 < 4k+2 \rightarrow k > 1$$

Closest Pair

Input: a set of n points (xi,yi)　　Output: A pair of distinct points whose distance is smallest.

Plan1: Brute-force　O(n^2)

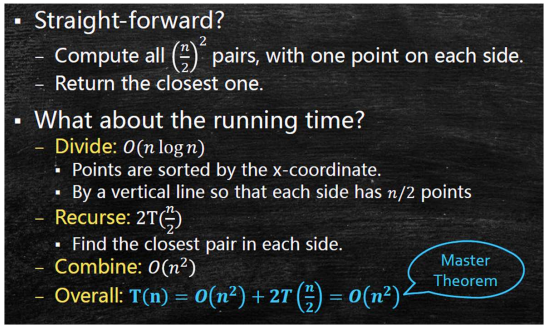
计算所有点对距离，输出最小距离。

Plan2: 排序优化

如果是一条线上，好办，耗费代价即排序代价。二维平面，不好办。

Plan3: 分治优化

按 x 轴排序后，将点对分成左右两部分，左右分别解决，再解决中央可能存在的交集部分。



关键在于交集部分的选取定义，我们选取中间带，中间带的划分是左右两边结果的最小值。

然而，这样一来，万一中间部分太过密集会怎么办呢？太过密集仍旧导致 O(n^2)。把中间带划分成正方形的形式可以大大优化此类的情况。而且，由于其中两个点之间的距离应该满足小于最小值，那么点数是有限的（至多 4 个）。

这里解释了为什么实际上选取的是 7 个点。

- Can we improve divide and combine to $O(n)$?
  - If we success, then $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$
- Tips
  - Do we actually need sorting every time?
  - What happens if do sorting before divide and conquer?
- Even more
  - A randomized algorithm achieves $O(n)$.
    - **Samir Khuller and Yossi Matias** (1995).
    - A simple randomized sieve algorithm for the closest-pair problem.

Episode => Sorting Lower Bound

$\Theta(n^2)$	$\Theta(n \log n)$
Selection Sort	Quick Sort
Bubble Sort	Merge Sort
Insertion Sort	Balance Tree

随机化的排序算法的 bound 也是 n log n

如果想要实现 O(n)的排序算法，那么应该有一定的限制条件。例如数值只有 0、1，但这违背了“盲盒”的一个概念，“盲盒”概念仅提供比较。

Master Theorem:

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$$

$$T(n) = O(n^d)\left(1 + \frac{a}{b^d} + \cdots + \left(\frac{a}{b^d}\right)^{\log_b n}\right) = \begin{cases} O(n^d), a < b^d \\ O\left(n^d \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O(n^d \frac{a^{\log_b n}}{n^d}) = O(n^{\log_b a}), a > b^d \\ O(n^d \log_b n), a = b^d \end{cases}$$

a:拆分成几个子问题　　b:子问题的分割情况　　d:子问题的规模

第一种情况是由于$\frac{a}{b^d} < 1$因此最后的结果应该是 1 的常数倍。第二种情况由于大于 1，因此最后的结果应该是与最后一项紧密相关。第三种情况由于是 1，因此有且仅与项数有关。

Graph Algorithms　图相关算法

Depth First Search and Its Applications

连通性问题：Reachability Problem（利用邻接矩阵表示的输入图）

直觉性做法：探究并拓展邻居是绝佳的思路

注意：容易出现环从而进入死循环（因此加入了访问到则不再访问的一个状态）

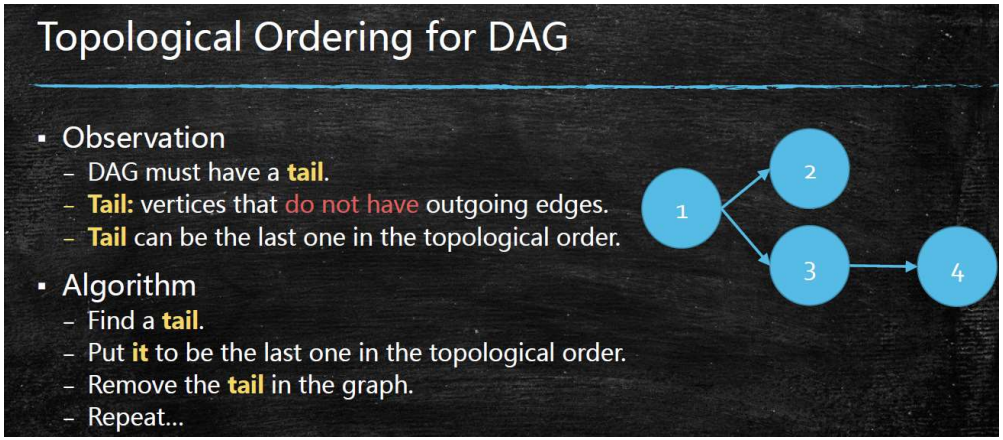
Cycle 环

发现，在 DFS 遍历的时候生成的实际上是一个生成树，因此如果在 DFS 遍历的时候出现了 back edge 那么这就是一个环。

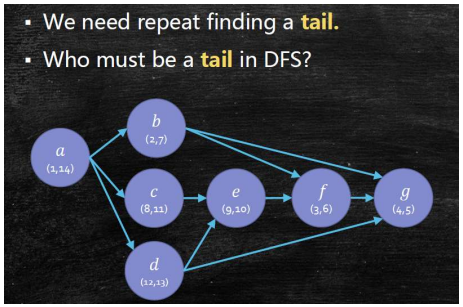
Top order 拓扑

Directed Acyclic Graph (DAG)：　有向无环图

DAG 一定有一个 tail 中点，进而 DAG 总能找到相应的拓扑排序



首先，通过点的遍历，每次都把队伍中的 tail 删除即可。但是这样的复杂度是 V^2 显然我们不想要的。



其次，我们通过 DFS 并且存入 start-time 和 end-time 作为数据标识，然后通过对 end-time(finish-time)进行排序则可以找到拓扑排序。复杂度 O(VlogV + E)

实际上，可以在 DFS 的时候就输出拓扑排序，这样省去了排序过程，复杂度 O(V+E)

Scc: Strongly connected component 强连通分量

Weak connected component:弱连通分量则当有向边变成无向边的时候连通

强连通分量是一个很不错的划分方式（good partition）

找强连通分量的个数、分量：

首先：发现如果利用先后访问次序，每次都能找到一个”head”

那么：我们逆置这张图可以得到相应的 tail

Basic Plan

1. Construct G^R
2. DFS G^R with **finish time**.
3. Choose v with the largest **finish time**.
4. Explore(v) in G.
5. When it returns, reached vertices form one SCC.
6. Remove them in both G and G^R.
7. Repeat from 2.

更加有效率的方式：

Super Plan

1. DFS G^R and maintain a **sorted list** by the finish time.
2. DFS G by the **descending order** of the finish time.
   1. Keep explore vertices by the descending order.
3. Each explore() forms a SCC.

Shortest Path (Negative)　最短路径问题（带负权边）

利用 Bellman-Ford 算法进行判断负环的操作

Bellman-Ford

Function bellman_ford(G, s)

$dist[s] = 0, dist[x] = \infty$ for other $x \in V$

while $\exists dist[x]$ is updated

for each $(u, v) \in E$

$dist[v] = \min\{dist[v], dist[u] + d(u, v)\}$

需要利用几个思想：1. k 轮后 dist(v)已经是所有带有 k 条路径中的最小路径(归纳法,0 轮显然正确,k-1 轮假设正确,第 k 轮(s,u1,u2,…,uk-1,v)确保 dist(uk-1)<=d(s 到 k-1)并且 dist(uk)<= d(s 到 k-1)+d(k-1 到 v)|2. 所有|V|边路径不可能比|V|-1 边小除非存在负环

利用 Dijkstra 算法进行最短路的计算

Dijkstra($G = (V, E), s$)

1. **Initialize**
   - $T \leftarrow \{s\}$
   - $tdist[v] \leftarrow w(s, v)$, $pre[v] \leftarrow s$ for all $(s, v) \in E$.
2. **Explore**
   - Find $v \notin T$ with smallest $tdist[v]$.
   - $T \leftarrow T + \{v\}$
3. **Update $tdist[u]$**
   - $tdist[u] = \min\{tdist[u], tdist[v] + w(v, u)\}$ for all $(v, u) \in E$.
   - If $tdist[u]$ is updated, then $pre[u] \leftarrow v$.

- Find Min
  - $|V|$ rounds
- Update
  - $|E|$ rounds
- If we use simple array, then
  - First round find min: $|V| - 1$
  - Second round find min: $|V| - 2$
  - ...
  - Find min totally: $O(|V|^2)$
  - Each update: $O(1)$
  - Update totally: $O(|E|)$
  - Algorithm totally: $O(|V|^2 + |E|)$

$|V|$ rounds

$|E|$ rounds

$|E|$ rounds

BFS for unweighted single source.

Dijkstra for positive weighted single source.

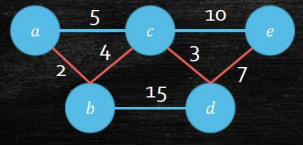
Bellman-Ford for general weighted single source and Floyd-Warshall for all.

Greedy　贪心算法

最小生成树的 Prim 与 Kruskal 算法

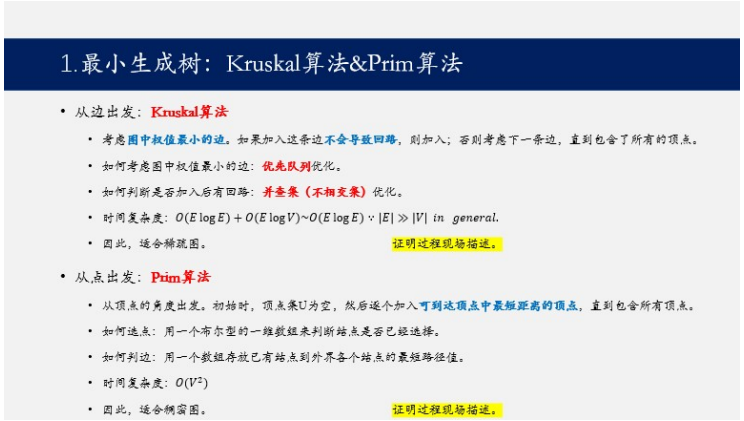
易知，单纯为了找生成树，可以很容易得到 BFS 与 DFS 的算法（只要遍历全部即可）

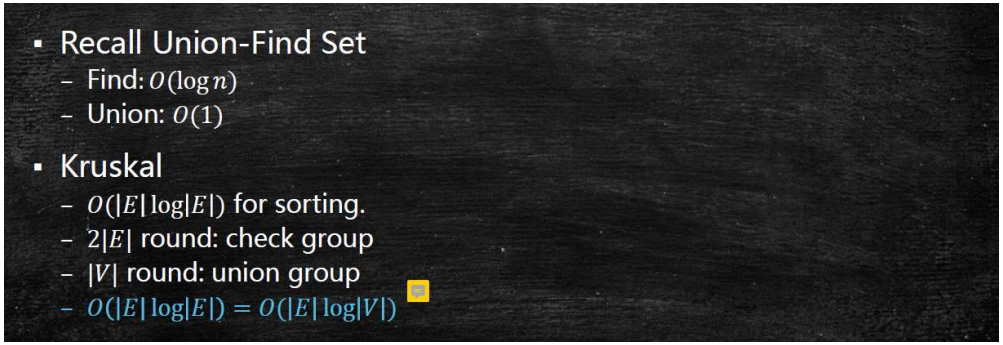
- **Input:** Given a connected undirected graph $G = (V, E)$, and a weight function $w(e)$ for each $e \in E$.
- **Output:** A spanning tree of G is, i.e., a subset of edges, with minimized total weight.
- Applications
  - Building a network, connecting all hubs via minimum number of cables.



Prim 与 Kruskal 算法的不同之处仅在考虑的方向不同，前者考虑从点维护优化，后者考虑从边维护

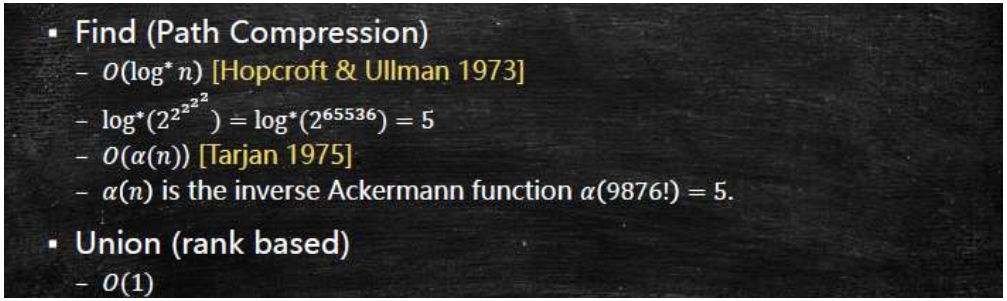
优化，从而生成完整的生成树。





|E|=|V|^2因此log|E|=2log|V|

并查集



最早接触的样题：贪心作业安排

输入：N 项作业且每项作业有不同的 s_j 以及 deadline d_j。

输出：一个安排表。

贪心（这里实际上就是最优）思想就是每次找最近的 deadline 去完成即可。

对这个样题进行泛化：当可能存在冲突时间的作业，我们想要尽可能多的完成作业。有三种选法：

1. 以开始时间为选择（直接拉满的初始点显然不行）
2. 以最短时间为选择（卡在两项任务的交界处的短时间显然导致了 1 < 2）
3. **以结束时间为选择（正确的想法）**

通过与之前的 Dijkstra 算法和最小生成树算法对比，容易发现这样的过程始终都不会破坏最优的性质。

Assumptions: the selected k-1 activities are in an OPT.

Induction: After adding the k-th activity, we are still in an OPT.

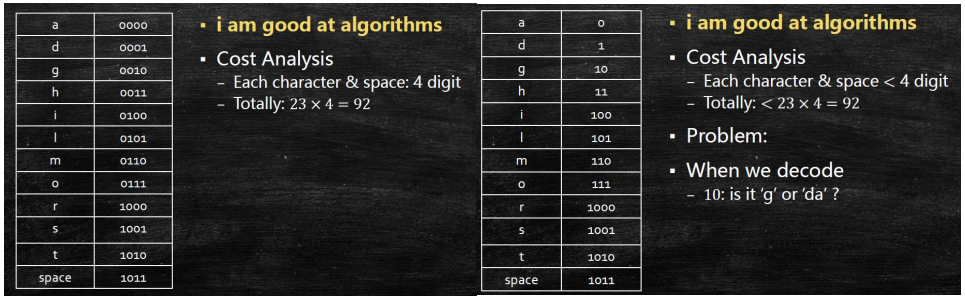
我如果选择的不是第 K 早，那么总有一个 OPT 可以选择第 K 早的。

假设第 k 次选择我们选取了第 k 早的完成时间，而破坏了 OPT 的结构，那么这意味着至少存在两项完成时间不大于第 k 小的且没有被选择且满足 a<=b<=k，那么这样一来就违背了我们选取的是第 k 早的一个命题，从而命题是错的

下一个问题：编码书本的最优策略（实际上就是哈夫曼编码策略）

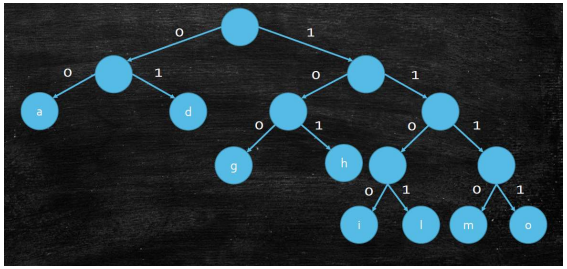
如何编码书本：首先提供字母表的编码服务

其次编码所有语句即可。

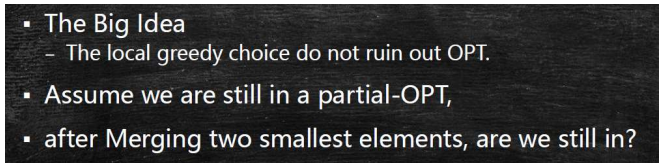


根据上图可以发现，前缀编码是非常优秀的。

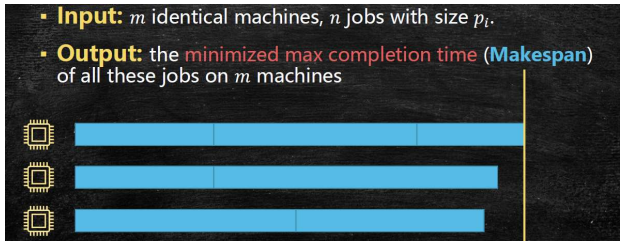
而事实上，前缀编码树就是一棵树，具体的编码结果就是树到叶子的路径。



贪心思想：每次选择最小的节点合并起来。



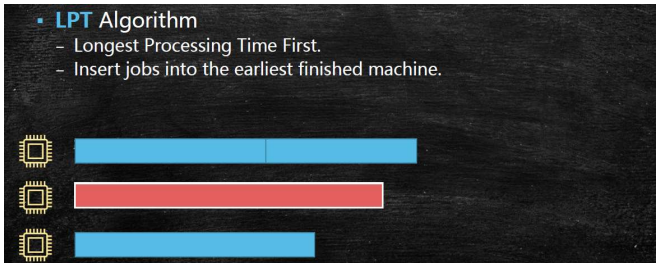
Makespan Minimization 机器任务分配问题



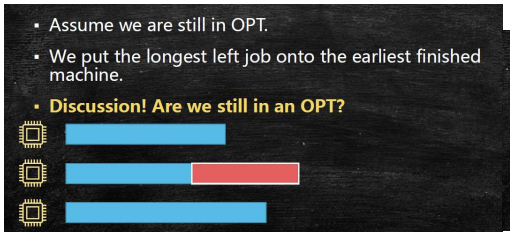
简单来说就是多个处理器的时候怎么样能够最早完成，这个可以理解为作业有好几个同学帮你一起做的时候怎么做能够最早完成。实际上这个问题可能局部最优未必是全局最优的。

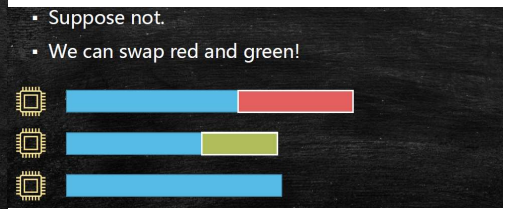
首先利用前面的 finish_time 进行排序，可以发现可能存在问题（因为局部的可能影响到了全局）

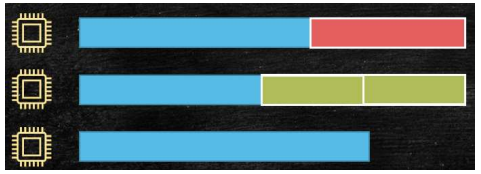
然后引入了 LPT 算法：最长处理时间优先算法



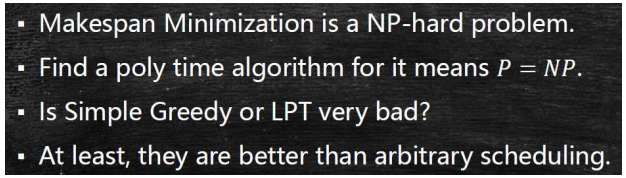
尝试进行贪心的证明：







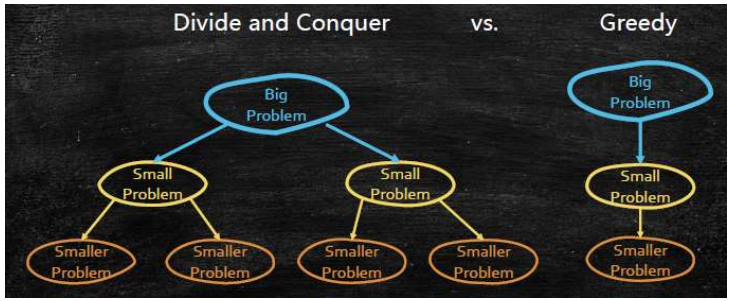
因此，可以发现，永远放最长的局部是最优的，但是全局看来不是最优的。



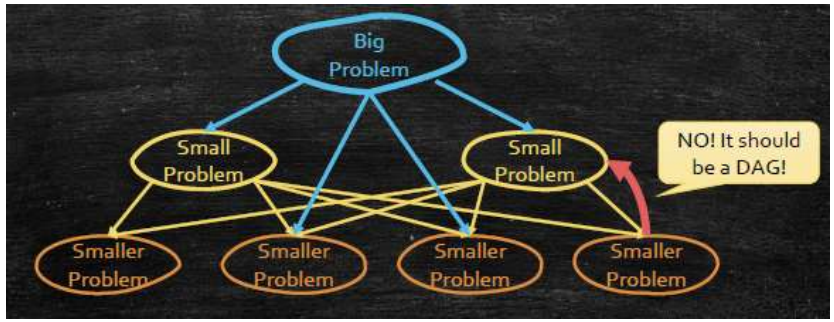
这一个题目是 NP 问题，而贪心至少能得到较好的结果（**分析这个“较好”的程度参照课件与作业**）。

Dynamic Programming　　动态规划

分治与贪心算法的比较



动态规划的直观化说明 唯一要求：不能存在“环”的状态，即不能出现如下图所示的一个情况：

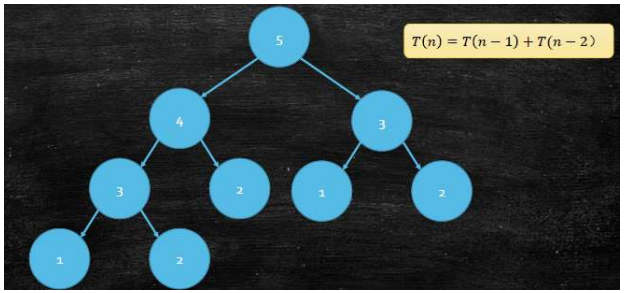


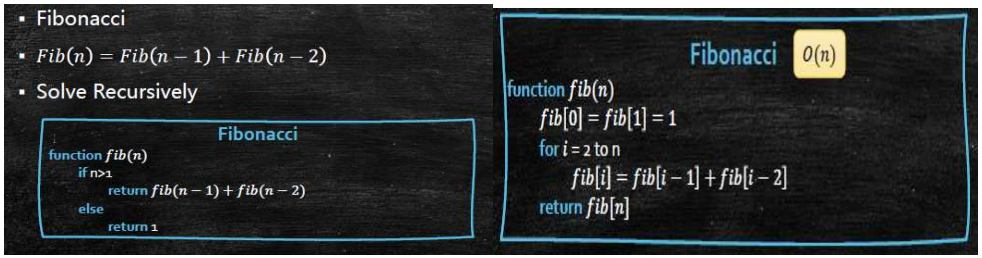
即应该是一个 DAG:有向无环图

非常经典的动态规划的例子：Fibonacci 斐波那契数列

分析数列第 n 项计算由递归到递推的过程（关键是有重复状态可以重复利用的）：

对于递归情况，当树的高度为 n 的时候，元素个数为 2^n -1（由完全树性质可知）





对于 DP 问题的设计思路：

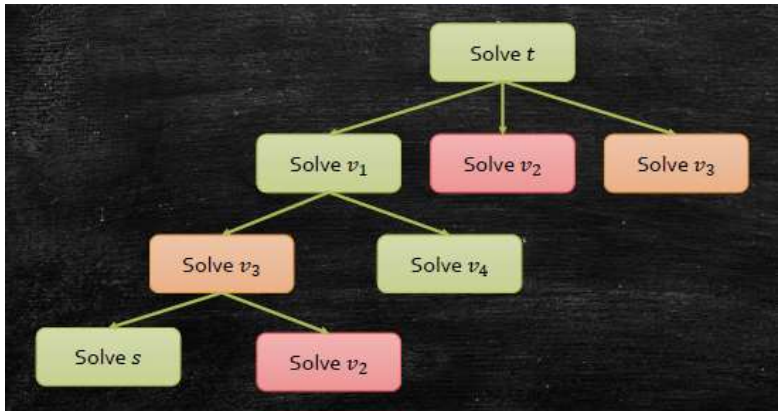
设计递归算法

合并相同的子问题

判断是否已经形成了 DAG 有向无环图，如果形成了就可以通过拓扑序去运行

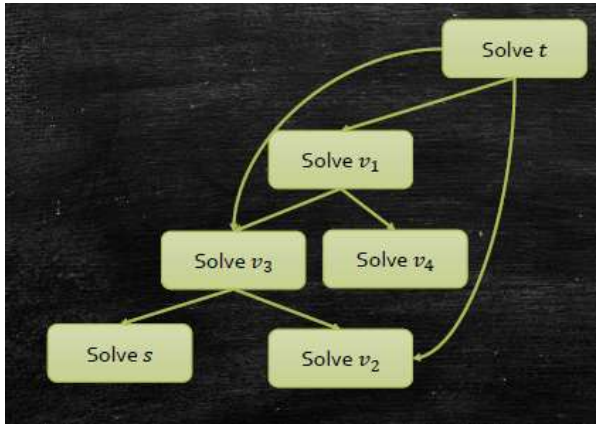
按照拓扑序解决并存储子问题的值

样例 1：有向无环图中的最短路径问题



将这个图转换为拓扑 DAG 图，显然只要用拓扑序按序运行就可以很容易得到结果。

我们可以确保这张图就是 DAG 因为原图告诉我们它就叫做 DAG。



所以 DP 的算法就出来了：

1. 找到各个顶点的拓扑序 O(V+E)
2. 初始化初始距离 dist[s]=0
3. 按照拓扑序计算所有点的路径，并且计算相应的最小值，则为 O(V+E)

对于 DP 问题的更简单的设计思路：

找到子问题

判断我们构造出的联系是否是 DAG，然后找到相应的拓扑序

利用拓扑序解决并存储子问题的值

通过例子加深理解：

最长单调增子序列(LIS)问题

关键：中间可断（只要找到一组序列中的数并且最长且单调不递减即可）

LIS[k]：最长不递减子序列且结尾为 k 序号的元素。

LIS[k] = max(LIS[I] +1 if a[i]<=a[k])　　复杂度 O(n^2)

如果想要优化，可以记录 dis[i]表示以 i 为长度的最小元素为 dis[i]，容易清楚的是这个长度应该是递增的，那么通过二分搜索可以 logn 时间就确定到恰好夹中间的位置，那么可以直接找到 max(LIS[I])

字符串修改距离 Edit Distance：分辨两个字符串的相似程度

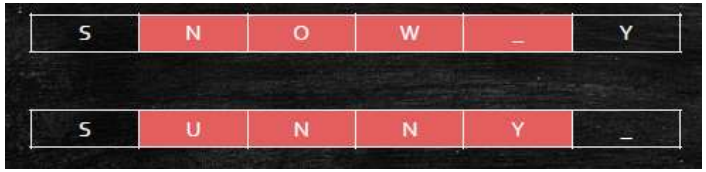
如何用最小的操作次数从一个字符串变换到另一个字符串呢？

允许操作：插入、删除、替换　　SNOWY->SNNWY->SNNY->SUNNY　　3 步

换一个角度看：首先是 0 代价的插入过程去“对齐”，然后插入、删除、替换都变成了重写

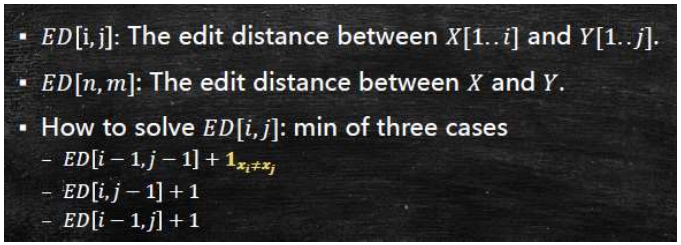
事实上，对齐之后可以发现对齐后的最小数目，恰好是最终解。

这个对齐方式，5 个不同因此要至少 5 步



这个对齐方式，3 个不同因此至少要 3 步

所以转化为了：找到最长相同子序列？



Knapsack Problems 背包问题

给定 n 物品，每件物品要占用 c 空间有价值 v 以及总容量 W,找到最优策略能够以 W 容纳的最大价值。

F[i]表示在 i 容量下得到的最大价值，那么 F[j]=max(F[j-c[i]] + v[i])以此求得 F[W]

直观来看，贪心似乎是不错的一个求解方式？但是往往是错的，举例如下

我们如果先买了便宜的东西后面可能就没有额外的空间装更丰富的东西了。

(贪心算法只适用于普通背包问题，**物品可以任意分割，可以得到最优解。如果不能任意分割，贪心法得到的不一定是最优解，而是一个可行解。**)

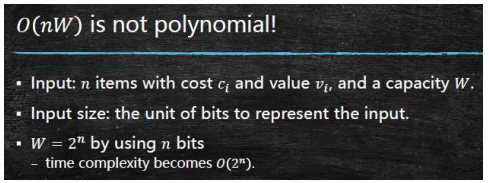
事实上，背包问题是 NP-hard 问题，但是我们可以用动态规划算法在合适的时间复杂度下解决问题。

当然，可以升维变成（结果直接从 f[n,w]获取即可）：

$$f[i,w]=\max\{f[i-1,w],f[i-1,w-c_i]+v_i,f[i,w-c_i]+v_i\}$$

看似这个只需要 O(N^2)的复杂度，但是需要注意的是我们的数组下标与仓库的容量 W 有着密切关系，

也与物品的种类、个数有着重要关系，时间复杂度可能是非多项式的！



背包问题有着非常多的变种情况（例如**背包九讲**）

① Surplus Supply 当出现充足供应的时候那是不同的变种，不同地方在于：

F[i]表示在 i 容量下得到的最大价值，那么 F[j]=max(F[j- **k** * c[i]] + v[i])以此求得 F[W]

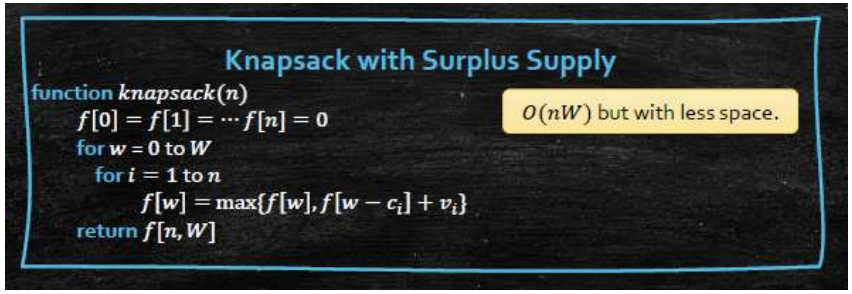
其中 k 表示最多可以选取的数目，因此 k = 1 -> W/c[i]

② 如果我想知道我选了哪几个背包怎么办？

开一个等大小的矩阵来记录我的值从哪里来比较好。

即 G[j]表示 j 容量的优秀值是从前面的哪一个 G[k]更新而来的。

G[j] = k 满足 F[j] = F[k] +j-k　　如果想再记录点东西，那再开数组即可。



需要注意的是，O(nW)并不是一个多项式的时间复杂度解。

举个例子，n 件物品，每件花费 c_i 有价值 v_i 且总容量为 W。我们如果用二进制表示输入。

那么 W=2^n 那么时间复杂度就变成了 O(2^n)

【REVIEW】对于 DP 问题的更简单的设计思路：

找到子问题

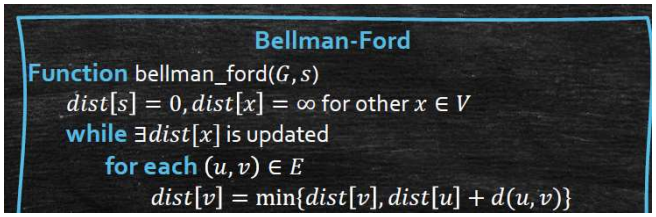
判断我们构造出的联系是否是 DAG，然后找到相应的拓扑序

利用拓扑序解决并存储子问题的值

【REVIEW】最长不减子序列、修改距离、背包问题（子问题可以从递归入手查找）

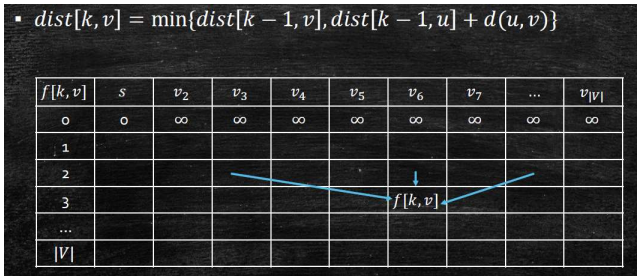
这三种 DP 问题都可以理解为走一步就到达的问题。Continue DP problems.

Understand Bellman-Ford as A DP



引理 1：k 轮后 dist(v)一定是 k 条边的路径上的最短路

那么定义子问题为 dist[k,v]视为从 s 到 v 在经过至多 k 条边之后的最短路



一行一行算，实际上也是 DAG 和拓扑序，那么本质上也可以认为是个 DP 问题。

现在我们想考虑 All Pair Shortest Path.多源最短路问题。全部点对的最短路关系

初步想法：跑|V|次 Bellman-Ford 也可以，复杂度为$O(|V|^2|E|)$

可以通过：Floyd-Warshall Algorithm 优化为$O(|V|^3)$，关键在于子问题的定义。

Bellman-Ford：dist[k,v]视为从 s 到 v 在经过至多 k 条边之后的最短路

初步泛化：dist[k,u,v]视为从 u 到 v 在经过至多 k 条边之后的最短路，转换方式为：

$$dist[k,u,v] = \min_{(s,v)\in E}\{dist[k-1,u,s] + d(s,v)\}$$

时间分析：总共|V|轮，每次轮次，一条边可以被用于更新|V|次距离，总共$O(|V|^2|E|)$

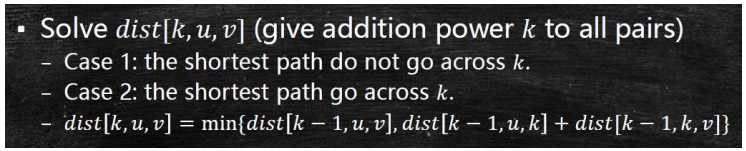
似乎没优化？

Floyd-Warshall Algorithm：换了一个子问题来定义

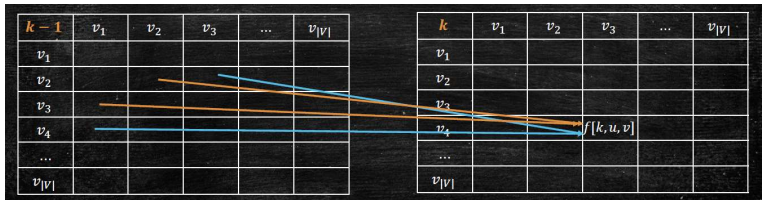
$dist[k,u,v]$:从 u 到 v 并且只经过了$\{v_1, v_2, \dots, v_k\}$的节点集合。

$dist[0,u,v]$:实际上就是从 u 到 v 的起始边的长度，要么 d(u,v)要么∞

$dist[|V|,u,v]$:我们最终想要的结果。

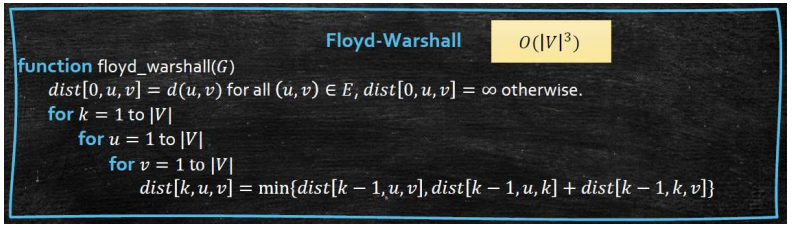


这种方法也可以用于判断负环（如果更新了不止一次）。



这很明显是个 DAG，结果当然也是非常理想的 DP 问题。

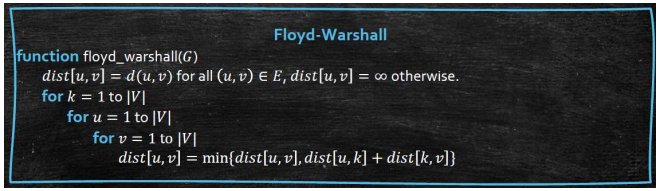
$dist[k,u,v]$仅取决于$dist[k-1,u,v]$，$dist[k-1,u,k]$，$dist[k-1,k,v]$



事实上可以进一步被优化（省掉一维空间复杂度）。***有什么合理解释吗？***

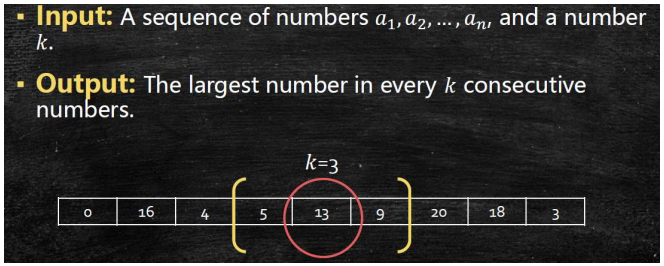
二维的情况可以这么解释：假设任意两点 I,j 之间路径可选择经过的节点集合中座号最大的 x（即，当 k=x 时 d[i][j]恰好取得最小值）|那么当 I,j 两点之间没有节点（直连）或只有一个可经过节点时显然成立|当更大的情况下，总能将其分为两部分 i-x 与 x-j，其中假设前者最大座号为 x_1 后者为 x_2 那么 $x>x_1$ 且 $x>x_2$。由前面已知此时得到的一定满足 $k=x_1$ 和 $k=x_2$ 都有 d[i][x]和 d[x][j]最小，那么 k=x 时自然

e[i][x]+e[x][j]最小必然能够得到最小值，所以 k 循环后有最小值



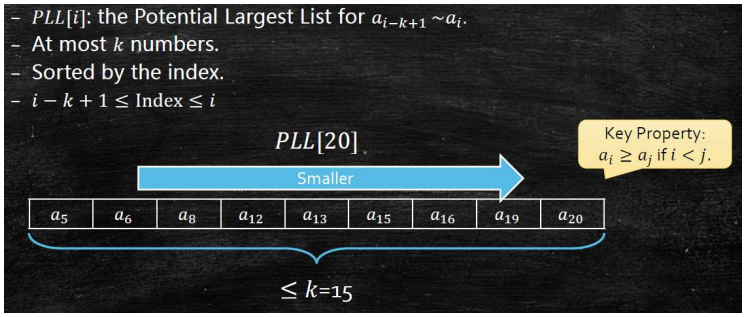
单调队列

应用 1：求连续 k 个数中最大数



暴力方法显然是对于每一个位置枚举属于他的区间即可。如果想要优化，那最好的方法似乎是维护一个队列，这个队列中保存的是每一个可能成为最大值的一个存在，如果不可能再成为最大值了那么没必要存在，直接弹出即可。

但是如果用子问题的方式思考怎么办呢？：与背包问题相联系：需要思考对于 large[i]（从 i-k+1 到 i 的最大数）而言 large[i-1]的影响情况是什么？***关键想法：Potential Largest List(PLL)***



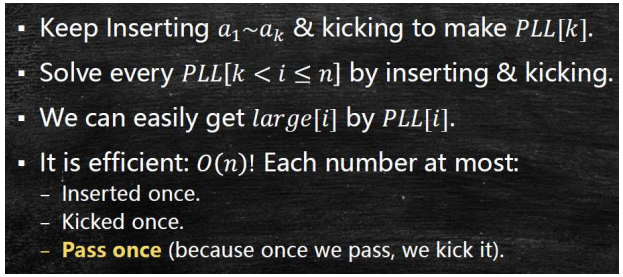
实际上就是构造一个单调队列，这个单调队列中的所有数都是单调的。

对于最大值而言（最小值则反之亦然）：

只要加入进来的数比前面的那个数大，那么前面的那个数就不可能为后面的答案做贡献，大胆踢出。

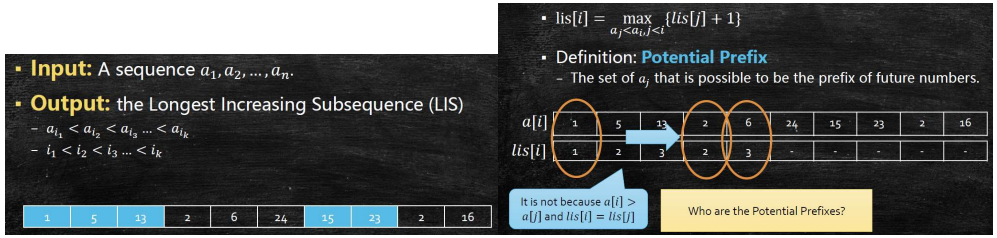
但是，如果加进来的数比前面那个小，那么这个小的数还是有可能为后面的答案做贡献，需要保留。

核心想法：构造单调队列并每次输出头结点。

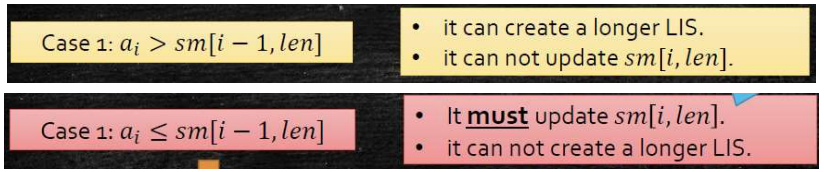


应用 2：LIS 问题改进

学到单调队列后，我们试图用这个方法优化最长单调不减子序列问题(LIS)

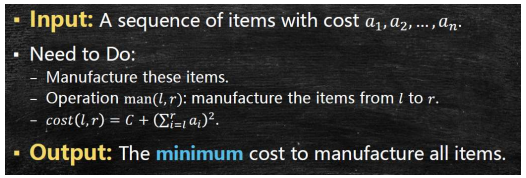


关键在于，有些性质是未来用得到的而有些是未来用不到的。那么可以通过 sm 的一个数组（队列）来维护当序列到达第 i 项之后 len 长度下最小值的情况。（我们当然希望这个值越小越好，越小给后面留出的可操作范围数更大）



这里需要找到 a_i 在 sm 的最佳位置。那么要得到 sm[i,len]需要用 logn 时间找到 sm[i-1,len]的位置然后更新即可。所以最后复杂度为 O(n log n)

应用 3：最小化生成代价 Minimizing Manufacturing Cost



需要尽可能优化 cost(l,r)=C+(a_l+…+a_r)^2，直观而言应该利用 DP 想法去做

假设我们定义 f[i]为从第 1 件到第 i 件的合并代价，那么本质上应该是找到切分点去做：

$$f[i] = \min_{j<i}\left\{f[j] + C + \left(\Sigma_{k=j+1}^{i} a_k\right)^2\right\}$$

这个方法直观来看就是$O(n^2)$的，而且需要注意到似乎这个切分点对未来有影响？

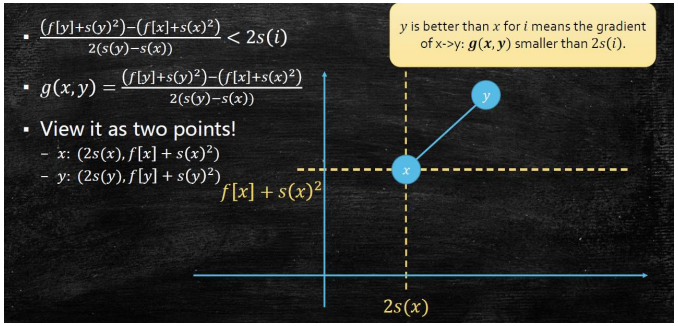
不妨从数学的角度分析一下这几个数据切分点的关系：

$$f[x] + C + \left(\Sigma_{k=x+1}^{i} a_k\right)^2 \; and \; f[y] + C + \left(\Sigma_{k=y+1}^{i} a_k\right)^2$$

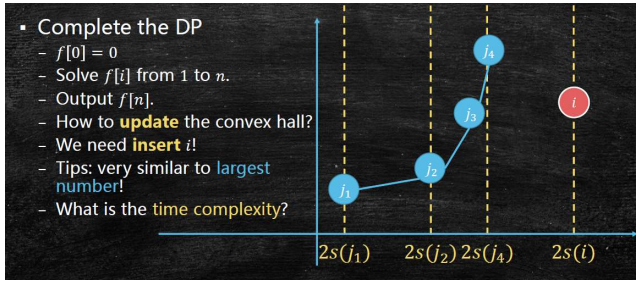
不妨令$s(i) = \Sigma_{j=1}^{i} a_j$那么移项后可以得到的是：

$$f[x] - f[y] > \left(s(i) - s(y)\right)^2 - \left(s(i) - s(x)\right)^2 = s(y)^2 - s(x)^2 - 2s(i)(s(y) - s(x))$$

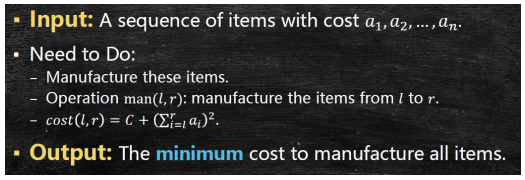
$$\frac{(f[y] + s(y)^2) - (f[x] + s(x)^2)}{2(s(y) - s(x))} < 2s(i)$$



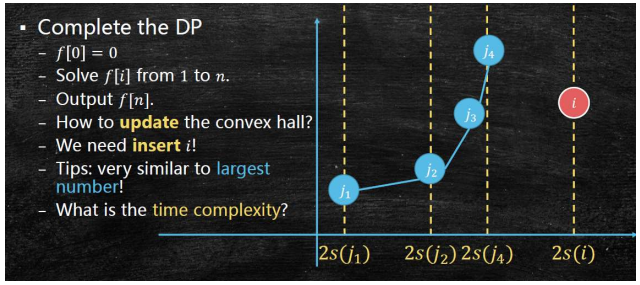
最后，本质上就是构造凸集，则确保这个凸集足够完美就可以了。



应用 3：最小化生成代价 Minimizing Manufacturing Cost【继续】

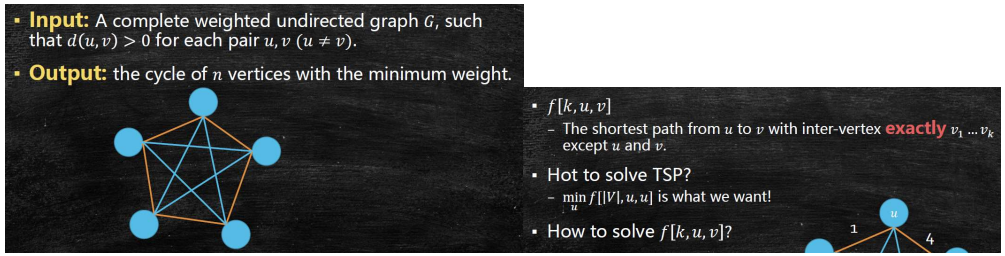


最后，本质上就是构造凸集，则确保这个凸集足够完美就可以了。

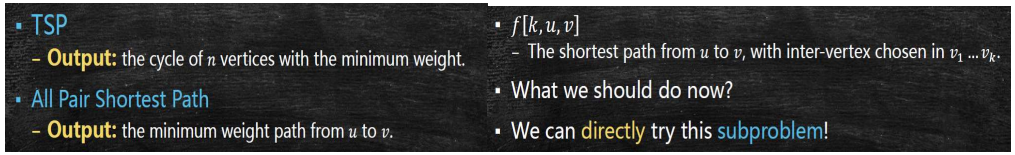


比较低效的 DP 问题：

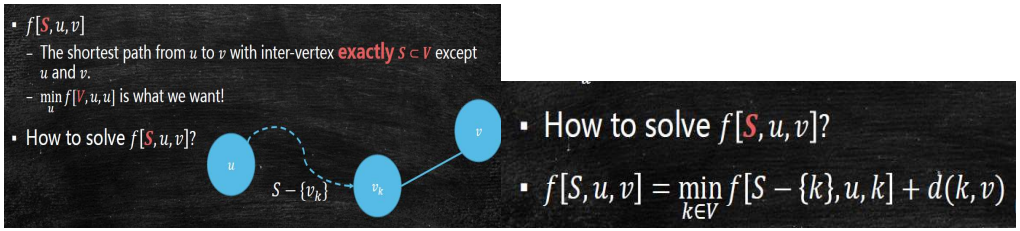
例题 1：Traveling Salesman Problem(TSP)



给一张完全无向无负权图，找到最小权重的哈密顿回路。【如果暴力做，怎么办呢？】



计划 A 似乎不太行？计划 B：



这题的正解应考虑状压 DP

所有可能路线有$(n-1)!$种(暴力 DFS 就是这个结果)，所以直接做是行不通的。使用状压 DP 很好解决

假设已经访问过的顶点集合 S(初始起点 0 也视为未访问)和当前所在节点 v，$dp[S][v]$表示从 v 出发访问

剩余的所有结点，最终回到顶点 0 的哈密顿路径权重最小和，有递推式:

$$dp[v][0] = 0, dp[S][v] = \min\{dp[S \cup \{u\}][u] + d(v,u)\}, u \notin S$$

通过状态压缩的方式尝试记忆化搜搜求解，对应到二进制位中的选取即可状压 DP 复杂度$2^n n^2$

```
//已经访问过的节点集合为S，当前位置为v
//用d[S][v]从v出发访问剩余的所有顶点，最终回到顶点0的路径的权重总和的最小值
int dp(int S, int v)
{
    int& ans = d[S][v];
    if (ans >= 0)  return  ans;             //已经求出来了，直接返回
    if (S == (1 << n) - 1 && v == 0)  return ans = 0;     //已经访问过所有节点并回到0号点
    ans = INF;
    for (int u = 0; u < n; u++)  if ((!((S >> u) & 1)) && G[v][u])
        ans = min(ans, dp(S | (1 << u), u) + G[v][u]);
    return ans;
}
```

搜索

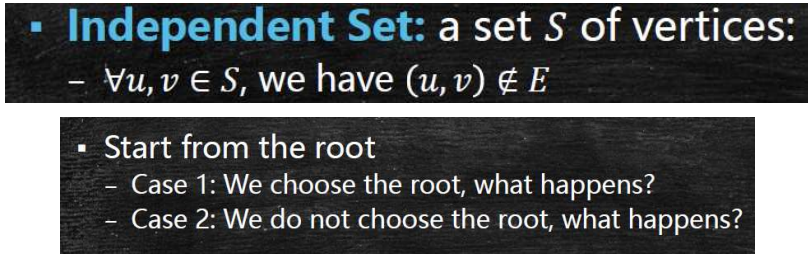
```
void solve()
{
    //用足够大的值初始化数组
    for (int S = 0; S < (1 << n); S++)
        fill(d[S], d[S] + n, INF);
    d[(1 << n) - 1][0] = 0;

    //根据递推式依次计算
    for (int S = (1 << n) - 2; S >= 0; S--)
        for (int v = 0; v < n; v++)
            for (int u = 0; u < n; u++)
                if ((!(S >> u & 1)) && G[v][u])
                    d[S][v] = min(d[S][v], d[S | 1 << u][u] + G[v][u]);
    printf("%d\n", d[0][0]);
}
```

DP 递推

例题 2：最大独立集问题 Maximize Independent Set On Trees

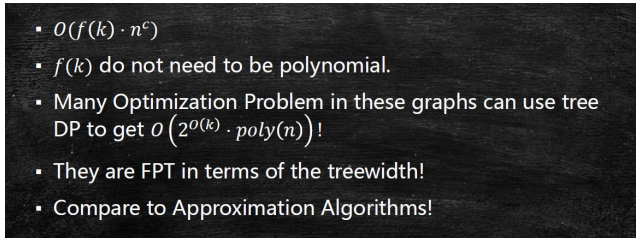
输入一棵无向树，输出最大独立集



这个实际上可以使用贪心算法进行解决。当然用树形 DP 也可以解决。

如果输入的并不是树，而是个图（有成环），那么尝试找到 supernode，对于 k 个点的 supernode 它的最大

分配情况是 2^k。所以对于 DP 可以构造 f[i,way]到达时间复杂度 O(2^(k*k)*n)



用 d(i)表示以 i 为根节点的子树的最大独立集大小。节点 i 只有选或者不选的可能，如果选了那么就是

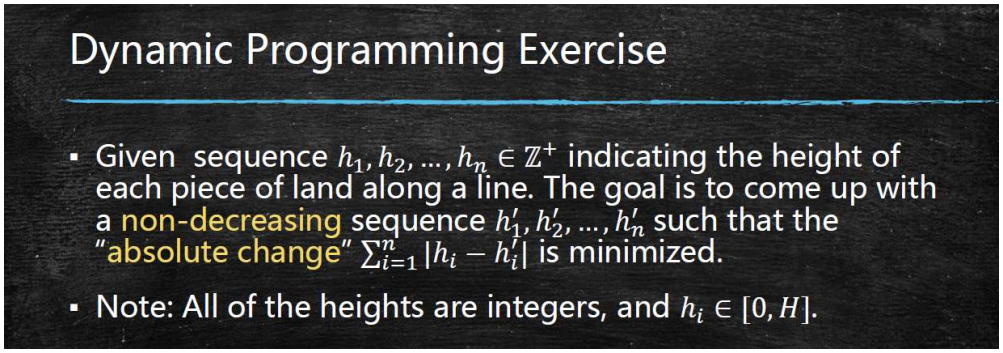
求 i 节点孙子的 d 值，不选就是儿子的 d 值，转移方程:$d(i) = \max(gs(i) + 1, s(i))$

详细一点:$dp[i][0]$ 与$dp[i][1]$是不选 i 与选 i 的最优解，那么转移方程就是

$$dp[i][1] += dp[v][0], v \text{ is the son of } i$$

$$dp[i][0] += \max(dp[v][0], dp[v][1])$$

例题:



两种可能：垒高或者砍低
非递减:
如果遇到了5 4 2　4我只能垒高
　　3我可以垒高也可以拉低
5 4 2 4 5
[N*MIN, N*MAX]
f[i][j]表示到第i位高度为j的最优情况
j:0->a[i]
f[i][j] = f[i-1][j] + (a[i]-j) when a[i]<=j
f[i][j] = f[i-1][j] when a[i]>j
f[i][j] = f[i-1][j]
if(a[i] > j) //4>3可以砍可以加
　　f[i][j] = f[i-1][j] + (a[i]-j)
else　　//4<5只能垒高
　　f[i][j] = f[i-1][j] + (j-a[i])
Ans = max(f[n][j])

f(i,h)=|h-hi|+min(f[i-1,x])其中 x<=h

Flow and Matching　　流问题与匹配

- Ford-Fulkerson Method: $O(|E| \cdot f_{max})$
- Edmonds-Karp Algorithm: $O(|V| \cdot |E|^2)$
- Dinic's Algorithm: $O(|V|^2 \cdot |E|)$

需要理解这几个算法的核心推导过程以及分析理解的过程。

最大流问题可以认为是铁路传递，能够传递运输最大的乘客数量是多少？流问题定义：

Flow – Formal Definition

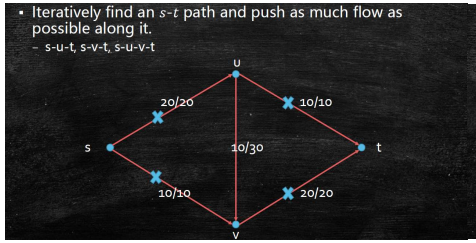
- Given a directed graph $G = (V, E)$ with a source $s \in V$ and a sink $t \in V$, and a capacity assigned to each edge $c: E \to \mathbb{R}^+$, a flow is a map $f: E \to \mathbb{R}_{\geq 0}$ satisfying the followings:
  - **Capacity Constraint**: for each $e \in E$, $f(e) \leq c(e)$, and
  - **Flow Conservation**: for each $u \in V \setminus \{s, t\}$,

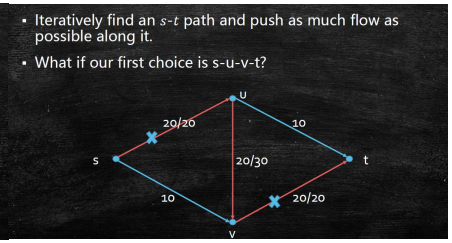
$$\sum_{v:(u,v)\in E} f(v,u) = \sum_{w:(u,w)\in E} f(u,w).$$

The value of the flow is defined as

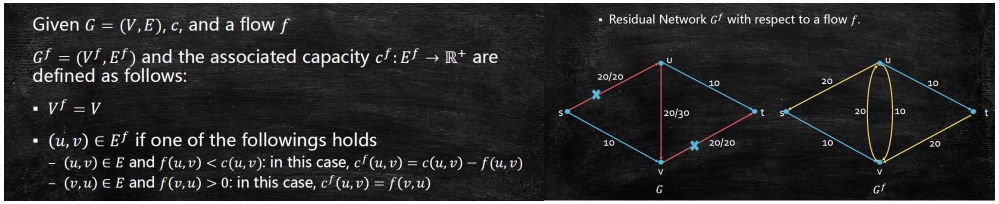
$$v(f) = \sum_{v:(s,v)\in E} f(s,v).$$

关键：流量、约束　　最初想法：贪心找路径　　似乎正确，但是反例右图所示：





残差网络 Residual Network



这就是 Ford-Fulkerson Algorithm:

Ford-Fulkerson Algorithm

FordFulkerson$(G = (V, E), s, t, c)$:

1. initialize f such that $\forall e \in E: f(e) = 0$; initialize $G^f \leftarrow G$;
2. while there is an s-t path p on G^f:
3. find an edge $e \in p$ with minimum capacity b;
4. for each $e = (u, v) \in p$:
5. if $(u, v) \in E$: update $f(e) \leftarrow f(e) + b$;
6. if $(v, u) \in E$: update $f(e) \leftarrow f(e) - b$;
7. endfor
8. update G^f;
9. endwhile
10. return f

Correctness: **Max Flow Min Cut Theorem**

接下来引入了割 cut 的概念 In fact, every "cut" gives an upper bound to $v(f)$

The Minimum Cut Problem

- We want to find a tightest upper-bound to $v(f)$ by a carefully chosen cut.
- Given weighted graph $G = (V, E, w)$ and $s, t \in V$, an s-t cut is a partition of V to L, R such that $s \in L$ and $t \in R$.
- The value of the cut is defined by

$$c(L, R) = \sum_{(u,v)\in E, u\in L, v\in R} w(u, v)$$

- **Min-Cut Problem**: Given $G = (V, E, w)$ and $s, t \in V$, find the s-t cut with the minimum value.

最大流最小割定理：一个问题的最大流就等于最小割 max v(f) = min c(L,R)

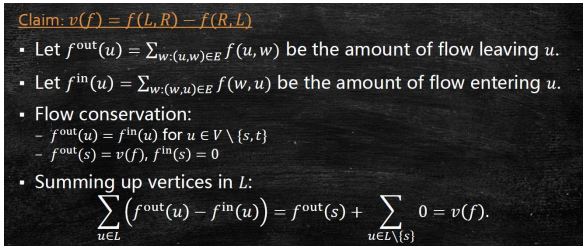
证明该定理首先从两方面的引理入手:

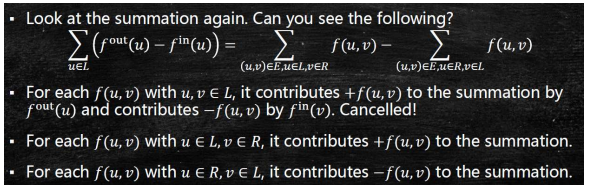
Lemma1:任意的流与割，总能满足流 v(f)≤割 c(L,R)

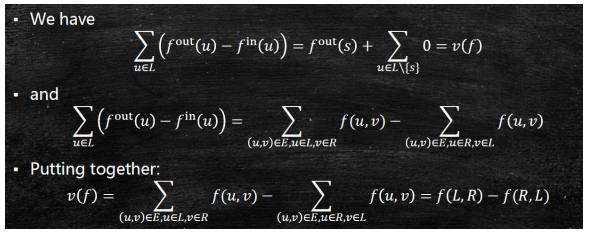
定义$f(L,R) = \Sigma_{(u,v)\in E, u\in L, v\in R} f(u,v)$是从割 L 到 R 的一个流，目标证明：$v(f) = f(L,R) - f(R,L)$

假设该设想成立，则

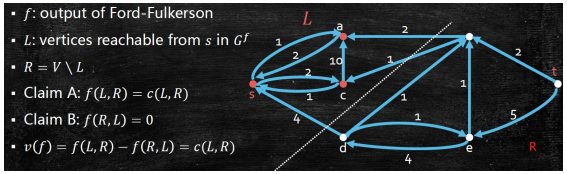
$$v(f) = f(L,R) - f(R,L) \leq f(L,R) = \Sigma_{(u,v)\in E, u\in L, v\in R} f(u,v) \leq \Sigma_{(u,v)\in E, u\in L, v\in R} c(u,v) = c(L,R)$$

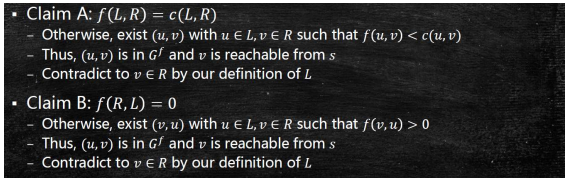






Lemma2:存在一个割与通过 Ford-Fulkerson 定理得到的流（最大流）相等,v(f)=c(L,R)





这个思想可以很好地用来解决 minimum cut 问题（最小割问题），因为 Ford- Fulkerson 算法中止时已经将

图分成了可到达的 L 以及其他点 R，只需要遍历一次残差网络就可以很清楚知道了

Time Complexity

Question 1: Does the algorithm always halt?

有理数中止而无理数可能不会终止/wiki

Question 2: If so, what is the time complexity?

Each iteration requires $O(|E|)$ time: $O(|E|)$is sufficient for finding p, updating $f$and Gf

There are at most f_{max}iterations. Overall: $O(E \cdot f_{max})$

Applications of Integrality Theorem

Theorem: If each $c(e)$is an integer, then the value of the maximum flow f is an integer.

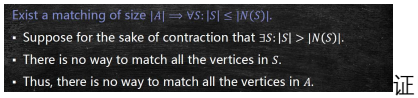
最经典应用：二部图匹配（男女婚配问题）

这里引入了 Hall's Marriage Theorem(考虑二分图 bipartite graph G)

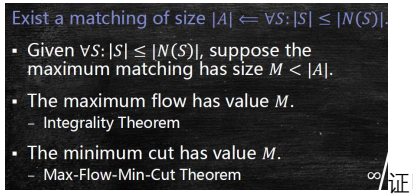
Consider the matching problem on a bipartite graph $G=(A,B,E)$.

For a subset $S \subseteq A$, let $N(S) \subseteq B$ be the set of vertices that are incident to vertices in S.

Hall's Marriage Theorem: There exists a matching of size |A| if and only if |S|≤|N(S)| for every $S \subseteq A$.

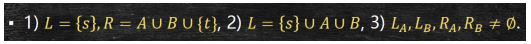


证明充分性（往右推导）



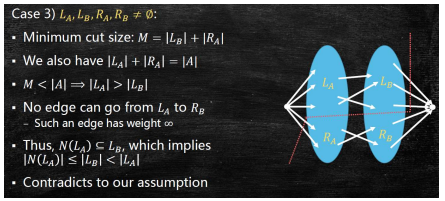
∞证明必要性（往左推导）

其中，对于最小割{L,R}有几种情况



第一种情况最小割的大小已经是|A|了（我们还假设了这个情况不可能）所以矛盾

第二种情况是反过来的第一种情况，自然也矛盾了所以也不可能　　第三种情况如下图所示



A graph is regular if all the vertices have the same degree.

A matching is perfect if all the vertices are matched.

Theorem (**Bipartite regular graph**): A regular bipartite graph always has a perfect matching.

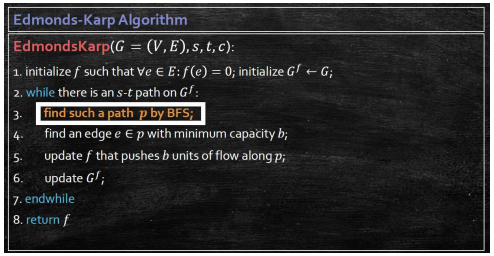
5. ([R], Page 666, Exercise 31)Suppose there is an integer k such that every man on a desert island is willing to marry exactly k of the women on the island and every woman on the island is willing to marry exactly k of the men. Also, suppose that a man is willing to marry a woman if and only if she is willing to marry him. Show that it is possible to match the men and women on the island so that everyone is matched with someone that they are willing to marry.

Answer Area:

Construct a graph $G = (R \cup T, E)$. Let R (respectively, T) be the set of nodes that denote men (respectively, women) on the island. For any $r \in R, t \in T$, (r, t) belongs to E if and only if r is willing to marry t. By definition each nodes' degree is k and for any $r_1, r_2 \in R, t_1, t_2 \in T$, $(r_1, r_2), (t_1, t_2) \notin E$.

It's obviously that $|R| = |T|$ since $k|R| = k|T|$. For any $A \subseteq R$, we have $k|A| \leq kN(A)$, i.e. $N(A) \geq A$. By Hall's theorem, there's a complete matching from R to T, thus it's possible to match the men and women on the island so that everyone is matched with someone that they're willing to marry.

讲完了 Ford-Fulkerson 算法后，在找路径方面 Edmonds-Karp 算法在路径上提出了更优的方式:

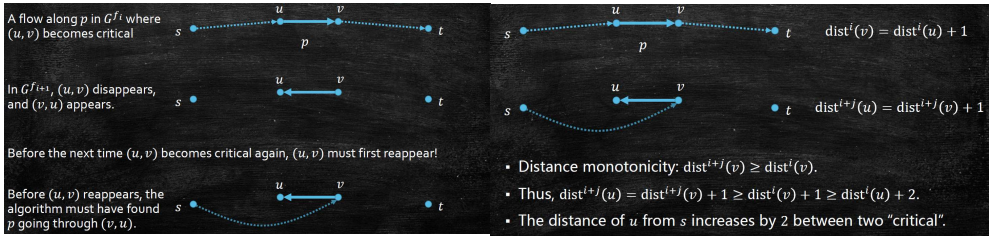


在理解这个算法的 BFS 思路时需要特别注意:过程中任何一个节点的 *dist(u)*实际上永远都是非递减的

在这个 BFS 思想中，它能够考虑到距离的因素(层次遍历的感觉)，因为一条新产生的边(残差网络边)只会从一个距离大的往一个距离更大的跑(例如已知 3->4->5)那么下一次只会有(7/3<-6/4<-5/5)实现了非递减功能。

事实上，非递减还不够，因为如果非递减那么还是可能出现非多项式时间（取决于流）的算法。

注意到一件事情：在一条路径上至少有一条边是饱和的（会被删除）那么每次迭代都会删除一个从距离 i 到距离 i+1 的点，显然不可能只删除边而不影响所有点间的距离。假设我们现在在第(i+1)次迭代，fi 是当前流量 p 是在当前流量的残差网络的第(i+1)次迭代中找到的路径。假定 critical 的概念是当前路径的最大容量所在的边(u,v)，A critical edge 可能在这次迭代中被删除但是在未来可能会出现，我们尝试约束这条边(u,v)变成关键边的次数的一个上下界。

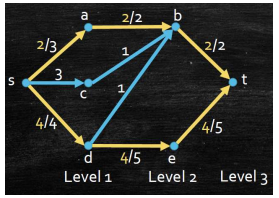


在两次 critical 之间距离每次都**至少**+1，意味着如果这条边再次在未来出现，那么至少是+2 的。

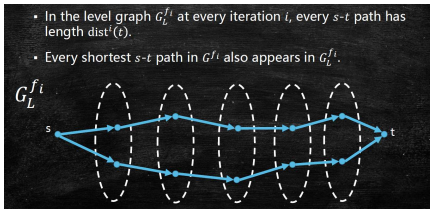
所以，通过 BFS 找到的这个更新次数（或者说迭代次数）最多只会有|V|次因为**距离**只会在{0,1,…,|V|,∞}中取值那么每条边最多会 critical O(|V|)次，所有边 critical 最多 O(|V||E|)次。注意到每次找 BFS 路径的迭代过程需要 O(|E|)从而最终的代价为 O(|V|*|E|^2)

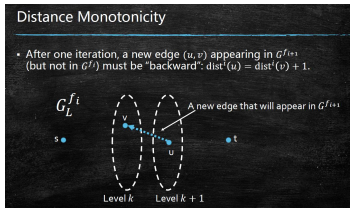
然后，我们发现，在 BFS 找迭代的时候似乎耗费了很长的时间？O(|E|)的时间但是找到的长度最多就|V|因此，想到了多路推广最短路的想法，通过多路并进的方式使得这个迭代的方法得到了优化->**Dinic's**

Dinic's 的算法关键在于建立了一个层次图，在第 i 层的节点它的距离就是 i。分层建立的过程通过树的层次遍历也可以很清楚的明白其实 O(|E|)就能够建立相应的一张 level graph。接下去尝试在 level graph 上找 blocking flow（多层推进的思想）、每条 s-t 的路径都应该有**一条关键边**并且及时删除:



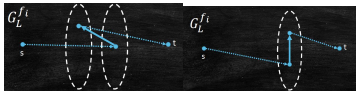
Dinic's 算法只剩下了两个问题：1.需要做多少次迭代？2.怎么找到 blocking flow?





在迭代次数方面的分析，其实与前面是类似的，都是从单调性的角度出发，确保增长的过程

第 i 次迭代后在流 f_{i+1} 图中的新边(u,v)必须满足了 $\text{dist}^i(u)=\text{dist}^i(v)+1$ 的一个思想，并且在这一次迭代中，所有 $\text{dist}^i(t)$都被"blocked"了那么必然有一些下一次的边不会出现在原来的 level graph 中，这样一来距离至少增加了 2(如下图左所示的两层之间)或增加 1(单层之间互连边) 可以满足的是严格单调增的



严格证明过程利用反证法的思想：假设两次迭代得到的距离值一样

考虑在第 i+1 次迭代中形成的 level graph 的任意一条 s-t 的路径 p 生成了 $dist^{i+1}(t)$的路径长度

Case 1: all edges in p also appear in $G_L^{f_i}$. Then p is a shortest path containing no critical edges in $G_L^{f_i}$. Contracting to the definition of blocking flow!

Case 2: p contains an edge (u,v) that is not in $G_L^{f_i}$

如果这条边不在G^{f_i}那么(v,u)必然是上一次的关建边至少能说明$dist^i(u) = dist^i(v) + 1$

如果这条边在G^{f_i}而不在它的层次图，那么通过 level graph 的定义可以知道$dist^i(u) \geq dist^i(v)$

根据 dist 的单调性也很清楚$dist^{i+1}(u) \geq dist^i(u)$，将这些性质融合起来可以得到

Putting together:

$$\begin{aligned} \text{dist}^{i+1}(t) &= \text{dist}^{i+1}(u) + 1 + \text{dist}^{i+1}(v,t) \\ &\geq \text{dist}^{i}(u) + 1 + \text{dist}^{i}(v,t) && \text{(Fact ii and iii)} \\ &\geq \text{dist}^{i}(v) + 1 + \text{dist}^{i}(v,t) && \text{(Fact i)} \\ &\geq \text{dist}^{i}(t) + 1 && \text{(triangle inequality)} \end{aligned}$$

由于 dist(t)只会在{0,1,…,|V|}中取值，它最坏情况就是 O(|V|)的从而迭代次数在 O(|V|)。

接下来是 blocking flow 的问题，事实上它很简单，就是对 level graph 中的每个顶点都去 DFS 找到路径

如果能够到达 t 那么就删除这条路径上的关建边并且更新 flow 的值

如果陷入了思路那么回退一步删除这条边（这条边没用）

每次搜索至少删除了一条边因此最坏 O(|E|)并且每次搜索需要最多|V|次所以最终算法复杂度$O(|V|^2|E|)$

后面还有很多对最大流问题的优化问题，不是重点故忽略。

对于找二分最大匹配问题有一个算法 Hopcroft-Karp-Karzanov Algorithm 时间复杂度$O(|E|\sqrt{|V|})$ 可以认为是 Dinic's Algorithm 的一个特例**（作业有分析过程可以着重看）** 关键在于证明 1.找 blocking flow 只需要 O(|E|)【做一遍 DFS 并且删除所有 s-t 的边、走死路就将前条边删除则只需要所有边遍历一次】并且 2.最大迭代次数至多是$2\sqrt{|V|}$（假设我们已经运行了$\sqrt{|V|}$的迭代得到了流，并且要证明在残差网络中的最大流最多只有$\sqrt{|V|}$）【对于每次迭代，任意一个非源汇点的点入度或出度必然是 1：每次迭代中经过一个节点 v 的流量只会是 0 或者 1，如果是 0 那么入度和出度保持不变，如果是 1 那么必然存在一条入边和一条出边使得这个性质还是成立的||Integrality Theorem:在残差网络中存在最大完整流包含了完全不相交边的路径，边不相交意味着点也不相交除了源汇点】

- Max-flow on G^f, f', is integral and consists of edge-disjoint paths.
- By our analysis to Dinic's algorithm, $\text{dist}^{G^f}(s,t) \geq \sqrt{|V|}$.
- Each path in f' has length at least $\sqrt{|V|}$.
- There are at most $\frac{|V|}{\sqrt{|V|}} = \sqrt{|V|}$ paths in f' by vertex-disjointness.
- $v(f') \leq \sqrt{|V|}$

那么我们已经说明了在残差网络中最大流最多是$\sqrt{|V|}$（$\sqrt{|V|}$次迭代），并且每次迭代流至少增加1，因此这个算法会在至多$\sqrt{|V|}$次之后停止所以总共是$2\sqrt{|V|}$，从而最终的复杂度是$O(|E| * \sqrt{|V|})$

Maximum Flow Problem:

- Edmonds-Karp Algorithm
  - Implement Ford-Fulkerson method by BFS
  - $O(|V| \cdot |E|^2)$
- Dinic's Algorithm
  - Push flow on multiple paths at one iteration
  - Level graph and blocking flow
  - $O(|V|^2 \cdot |E|)$

Maximum Bipartite Matching Problem:

- Hopcroft–Karp–Karzanov algorithm
  - Apply Dinic's algorithm
  - $O\left(|E| \cdot \sqrt{|V|}\right)$

Linear Programming　线性规划问题

LP 问题是在线性方程组或线性不等式的集合约束条件下对给定的线性函数求最值的问题。

1. 容易发现，对于 LP 问题，它的最优解如果存在那么必然在某个顶点处可以取到
2. 可解区域永远是凸的（因为所有线性约束都形成了超平面而可解集被超平面约束）。
3. 局部最大值必然是全局最大值（由凸函数的定义可知）

单纯形法 Simplex Method

选定任意节点，每次移动到邻边的端点上，当找到局部最大值中止。怎么移动到相邻端点？通过松弛其中一个约束并且引入新的约束就可以了，新的顶点可以通过解新的线性方程组得到。单纯形法一般去做复杂度很高，但是引入了高斯噪音之后收敛效果会很不错。

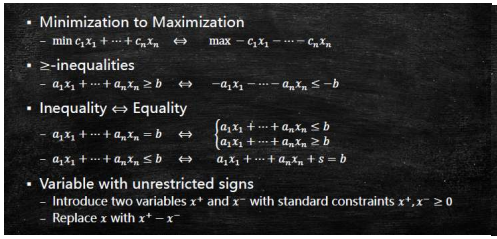
还有其他方法如椭圆法 Ellipsoid Method 和内点法 Interior Point Method 都是差不多类似的。

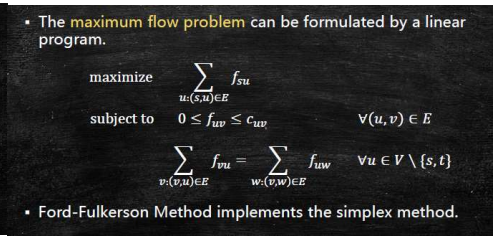
LP 问题的标准型：约束条件都是非正且所有变量都是非负

$$\begin{aligned} &\text{maximize} && \mathbf{c}^{\mathsf{T}}\mathbf{x} \\ &\text{subject to} && A\mathbf{x} \leq \mathbf{b} \\ & && \mathbf{x} \geq \mathbf{0} \end{aligned}$$

转化方式：

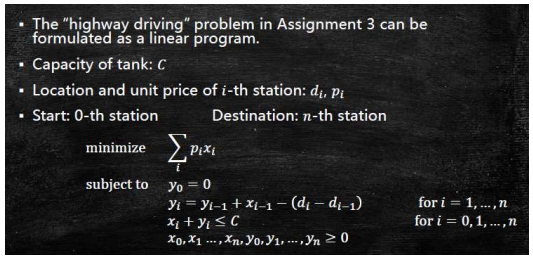
1.求最小值变成求最大值；2.大于等于的不等式转化为负数形式的小于等于不等式；3.等式转化为大于等于和小于等于同时存在的不等式；4.无约束条件变量或者不是我们想要的条件变量可以拆分变量。





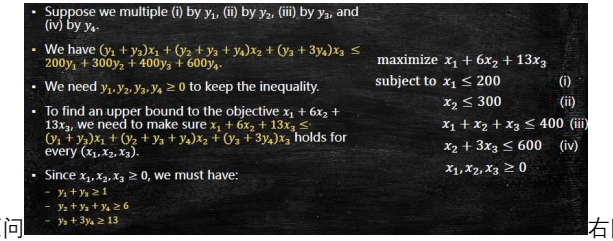
其实，最大流问题可以转化为一个 LP 问题，Ford-Fulkerson Method 实际上就实现了单纯形法。

把一个问题转化为线性规划问题的方式就是明确它的目标函数和约束条件（加油问题）：

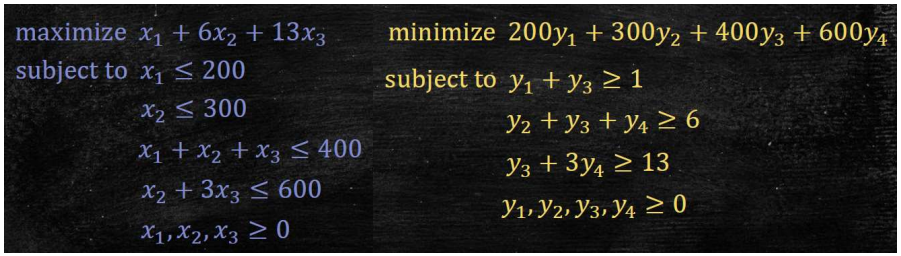


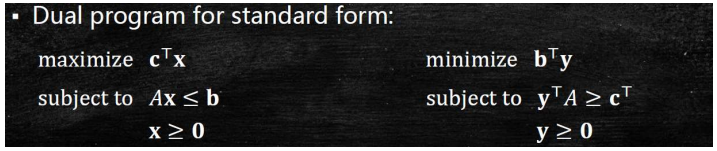
LP 对偶问题：最初的想法就是因为如果调整约束变量的区间可以把最值限制住，甚至可能到最值。

LP 的转化就是将所有约束条件前面乘上一个因子后放缩，并且因子之间也有所限制，如下图所示：



左图原问右图对偶





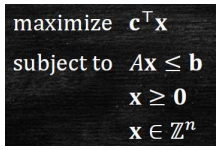
Weak duality theorem:如果原问题的可解值为 x 而对偶问题的可解值为 $\boldsymbol{y}$ 那么原最大≤对偶最小

Strong duality theorem:如果原问题的可解值为 x 而对偶问题的可解值为 $\boldsymbol{y}$ 那么原最大=对偶最小

强对偶定理的应用范围：

最大流最小割定理、Minimax 定理、匈牙利定理、近似算法的对偶拟合、经济学资源配置与估价

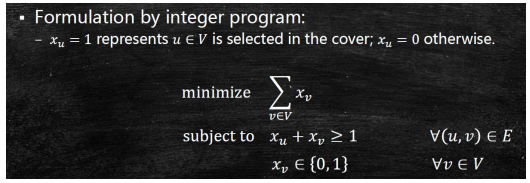
LP-松弛/Relaxation 对于整数（线性）规划问题 I(L)P 的标准形式如下

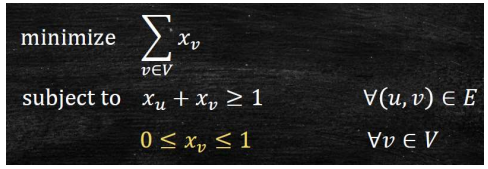


整数线性规划问题是 NPC 问题，即便是对于只有两种取值 0、1 的情况也是一样的。

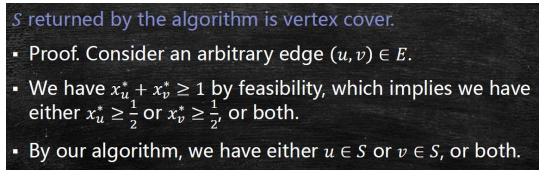
对于$x_i \in \{0,1\}$我们可以松弛到$0 \leq x_i \leq 1$后通过四舍五入取整的方式得到整数解；只要能够证明这个四舍五入的方式是可解的并且能够保证最优拟合那么就非常不错。在这里引入了 Vertex Cover 的概念

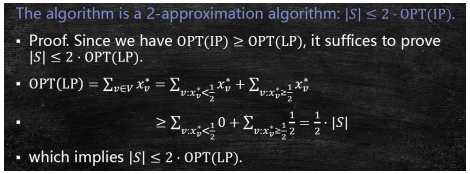
而最小顶点覆盖问题 **Minimum Vertex Cover** 恰好可以用整数规划来进行处理，右式是松弛式：





原问题 IP 的最优解和松弛后的 LP 问题最优解形式是一样的，但是显然前者的最优解应该大于等于后者因为后者有着更大的可解域。Vertex Cover 在这种情况下的一种近似算法就是解后边的等式并且返回的解集是所有概率大于 1/2 的点的集合，这个近似算法返回的确实是一个顶点覆盖集合。证明：





而且这个算法好在他是二倍近似的，因此效果还不错。这里可以好好体会一下近似算法的应用。

强对偶定理的应用范围：

最大流最小割定理、Minimax 定理、匈牙利定理、近似算法的对偶拟合、经济学资源配置与估价

用强对偶定理怎么证明最大流最小割定理呢：总共有 4 个步骤

1. 写下最大流问题的 LP 形式（并且转换成标准型）

- The maximum flow problem can be formulated by a linear program.

$$\text{maximize} \quad \sum_{u:(s,u)\in E} f_{su}$$

$$\text{subject to} \quad 0 \le f_{uv} \le c_{uv} \qquad \forall (u,v) \in E$$

$$\sum_{v:(v,u)\in E} f_{vu} = \sum_{w:(v,w)\in E} f_{uw} \qquad \forall u \in V \setminus \{s,t\}$$

$$\begin{aligned}
\text{maximize} \quad & \sum_{u:(s,u)\in E} f_{su} \\
\text{subject to} \quad & f_{uv} \le c_{uv} && \forall (u,v) \in E \\
& \sum_{v:(v,u)\in E} f_{vu} - \sum_{w:(v,w)\in E} f_{uw} \le 0 && \forall u \in V \setminus \{s,t\} \\
& -\sum_{v:(v,u)\in E} f_{vu} + \sum_{w:(v,w)\in E} f_{uw} \le 0 && \forall u \in V \setminus \{s,t\} \\
& f_{uv} \ge 0 && \forall (u,v) \in E
\end{aligned}$$

2. 证明对偶问题描述了最小割的有理数版本

首先写出最大流问题的对偶形式，我们想要说明这个对偶问题恰好就是最小割问题。

$$\begin{aligned}
\text{minimize} \quad & \sum_{(u,v)\in E} c_{uv} y_{uv} \\
\text{subject to} \quad & y_{su} + z_u \ge 1 && \forall u: (s,u) \in E \\
& y_{vt} - z_v \ge 0 && \forall v: (v,t) \in E \\
& y_{uv} - z_u + z_v \ge 0 && \forall (u,v) \in E, u \ne s, v \ne t \\
& y_{uv} \ge 0 && \forall (u,v) \in E
\end{aligned}$$

- We aim to show the LP above describes the min-cut problem.
- Let OPT_{dual} be its optimal objective value. We need to show OPT_{dual} is the size of the min-cut.

- y_{uv} describes if edge (u,v) is cut:

$$y_{uv} = \begin{cases} 1 & \text{if } (u,v) \text{ is cut} \\ 0 & \text{otherwise} \end{cases}$$

- z_u describes u's "side":

$$z_u = \begin{cases} 1 & \text{if } u \text{ is on the } s-\text{side} \\ 0 & \text{if } u \text{ is on the } t-\text{side} \end{cases}$$

3. 证明对偶问题永远存在最优整数解（从而它的最优解就是最小割的大小）

然后证明，最优解的值必然只会在$y_{uv} = \{0,1\}$中取到，即当$y_{uv} \ge 2$时取不到最优解

接下来需要证明我们提出的这几个变量的含义是正确的，这样才能说明这个对偶问题能给出最小割：

首先证明确实存在$y_{uv}, z_u \in Z$的最优解：利用 unimodularity 幺模性进行证明

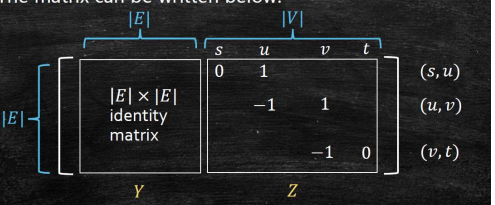
关于幺模矩阵性质：If $\boldsymbol{A} \in \boldsymbol{R^{m\times n}}$ is totally unimodular and $\boldsymbol{b}$ is an integer vector, then the polytope $\boldsymbol{P} = \{\boldsymbol{x}: \boldsymbol{Ax} \le \boldsymbol{b}\}$ has integer vertices.证明也很简单，关键就在于 Cramer's Rule 中对于强行暴力求解每个余子式的行列式值都为±1（否则没意义）因此自然是整数解（因为运行的计算都是整数）。

Proving Integrality of y_{uv}, z_u

$$\begin{aligned}
\text{minimize} \quad & \sum_{(u,v)\in E} c_{uv} y_{uv} \\
\text{subject to} \quad & y_{su} + z_u \ge 1 && \forall u: (s,u) \in E \\
& y_{vt} - z_v \ge 0 && \forall v: (v,t) \in E \\
& y_{uv} - z_u + z_v \ge 0 && \forall (u,v) \in E, u \ne s, v \ne t \\
& y_{uv} \ge 0 && \forall (u,v) \in E
\end{aligned}$$

- Now, we show that the matrix describing the first three rows of the constraints is totally unimodular.

- The matrix can be written below:



- Let the matrix be $[Y\ Z]$. Y is the identity matrix. We only need to show Z is totally unimodular.

Proving Z is totally unimodular by Induction...

- Base Step: Each cell of Z belongs to $\{0,1,-1\}$.
- Inductive Step: Suppose every $k \times k$ submatrix of Z has determinant belongs to $\{0,1,-1\}$. Consider any $(k+1)\times(k+1)$ submatrix Z'.
- Case 1: If a row of Z' is all-zero, then $\det(Z') = 0$.
- Case 2: If a row of Z' contains only one non-zero entry, then $\det(Z')$ equals to ± 1 times the determinant of a $k \times k$ submatrix. $\det(Z') \in \{0,1,-1\}$ by induction hypothesis.
- Case 3: If every row of Z' has two non-zero entries (one of them is -1 and the other is 1), then $\det(Z') = 0$:
  - Adding all the column vectors, we get a zero vector.

Some Intuitions

- Consider an arbitrary s-t path $s - v_1 - v_2 - \cdots - v_{\ell-1} - t$.
- Sum up all the constraints for the edges on the path:

$$(y_{sv_1} + z_{v_1}) + (y_{v_{\ell-1}t} - z_{v_{\ell-1}}) + \sum_{i=1}^{\ell-2} (y_{u_i u_{i+1}} - z_{u_i} + z_{u_{i+1}}) \ge 1$$

$$\Rightarrow \quad y_{sv_1} + y_{v_{\ell-1}t} + \sum_{i=1}^{\ell-2} y_{u_i u_{i+1}} \ge 1$$

- Conclusion: We must have $y_{uv} \ge 1$ for at least one edge (u,v) on the path.
- Removing $\{(u,v): y_{uv} \ge 1\}$ disconnects t from s.

证明对偶问题的解恰好是最小割的整数解：

- **Lemma 1.** OPT_{dual} is an upper-bound to min-cut.
- Proof. Let $(\mathbf{y}^*, \mathbf{z}^*)$ be an integral optimal solution.
- Let $C = \{(u,v) \in E: y^*_{uv} \ge 1\}$. We have shown removing C disconnect t from s.
- Let $L \subseteq V$ be the vertices reachable from s after removing C, and $R = V \setminus L$. Then $\{L,R\}$ is an s-t cut.
- For min-cut $\{L^*,R^*\}$, we have

$$c(L^*,R^*) \le c(L,R) = \sum_{(u,v)\in E: u\in L, v\in R} c_{uv} \le \sum_{(u,v)\in E: u\in L, v\in R} c_{uv} y^*_{uv} = \text{OPT}_{\text{dual}}$$

- **Lemma 2.** OPT_{dual} is a lower-bound to min-cut.
- Proof. Let $\{L^*,R^*\}$ be a min-cut. We construct a LP solution:
- $y_{uv} = \begin{cases} 1 & \text{if } u \in L^*, v \in R^* \\ 0 & \text{otherwise} \end{cases}$ and $z_u = \begin{cases} 1 & \text{if } u \in L^* \\ 0 & \text{if } u \in R^* \end{cases}$
- It is easy to verify that the solution is feasible...
- Then,

$$\text{OPT}_{\text{dual}} \le \sum_{(u,v)\in E} c_{uv} y_{uv} = \sum_{(u,v)\in E: u\in L^*, v\in R^*} c_{uv} = c(L^*,R^*)$$

4. 使用强对偶定理证明最大流=最小割

通过强对偶定理容易明白$OPT_{dual} = OPT_{maxflow}$因此二者相等。

所以对于利用强对偶问题的证明方式，可以参考这样的步骤：

1. **写下原问题和对偶问题的 LP 标准形式**
2. **判断这两个形式是否描述同一个问题**
3. **如果描述的是离散问题（不是连续的）那么证明他们对应的 LP 问题总能给出整数最优解（幺模）**
4. **使用强对偶证明**

冯·诺依曼的极大极小定理：由零和游戏引入。

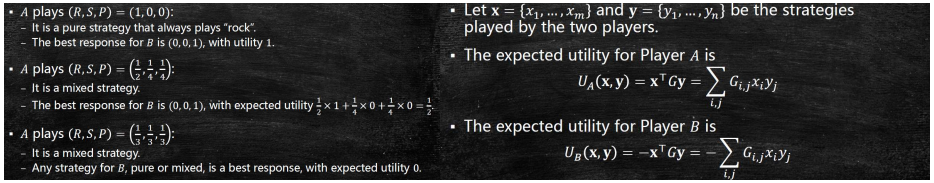
- Two players: A and B
- Each player has a set of actions that (s)he can play.
  - Set of actions A can play: $\mathbf{a} = \{a_1, a_2, \ldots, a_m\}$
  - Set of actions B can play: $\mathbf{b} = \{b_1, b_2, \ldots, b_n\}$
- For each pair of actions (a_i, b_j) that two players play, an utility is assigned to each player: $u_A(a_i,b_j), u_B(a_i,b_j)$.
- A game is a zero-sum game if $\forall x_i, y_j: u_A(a_i,b_j) + u_B(a_i,b_j) = 0$.
- Payoff Matrix $G \in \mathbb{R}^{m\times n}$, where $G_{i,j}$ is the utility gain for A, or the utility loss for B, when (a_i,b_j) is played.

- The payoff matrix for the Rock-Scissors-Paper game:

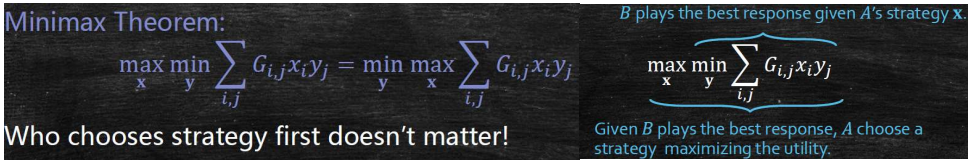
		Player B		
		Rock	Scissors	Paper
Player A	Rock	0	1	-1
	Scissors	-1	0	1
	Paper	1	-1	0

策略 strategy 是对所有可能操作的一个概率分布，纯策略就是选取某一个行为（即其他行为概率为 0）

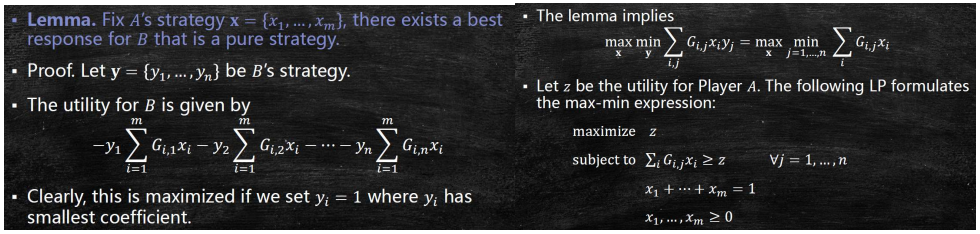
混合策略意味着有至少两个概率不为零的可能行为，对 B 玩家而言，他的策略就是确定 A 策略之后找到自己在 A 的策略下能获取的最大值、最大 utility 效用。对剪刀石头布而言如下（包括期望）：



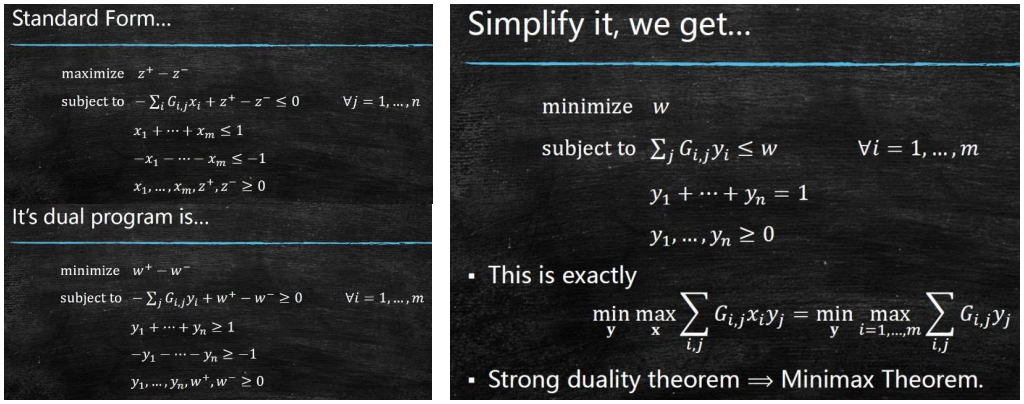
事实上，在零和游戏中，似乎谁先选择决策，期望的效用总是相同的，即冯诺依曼 Minimax Theorem.



证明方法如下：首先通过引理说明存在纯策略的最佳反应以及 A 玩家的应对措施(LP 问题):



然后将这个得到的 LP 问题转化为标准型并产生对偶问题，进行同样的推导可以发现确实如此：



因此，这里很好地把极大极小值定理通过强对偶定理的方式证明出来了。

Hardness and Approximation Algorithms 困难问题与近似算法

非常经典的 NP 困难问题：SAT + Vertex Cover + Independent Set + Subset Sum + Hamiltonian Path

SAT：由变量以及操作符(AND, OR, NOT)、括号组成的布尔表达式

SAT 问题：给定合取范式φ（一堆 clause 的合取范式），确定是否存在一组解满足φ=true

Eg. $(x_1\vee x_3\vee\neg x_4)\wedge(x_2\vee\neg x_3)\wedge(\neg x_1\vee\neg x_2)\rightarrow$ Solution: $x_1=true, x_2=false, x_3=false$

Vertex Cover：无向图中，一个顶点集的子集能够包含所有边的至少一端就叫做顶点覆盖集合

Vertex Cover 问题：给定无向图$G=(V,E)$以及$k\in Z^+$判断是否存在大小为 k 的顶点覆盖

例如对于长度为 5 的线段显然存在 k=2,3,4,5 的顶点覆盖

Independent Set：无向图中，一个顶点集的子集能够确保任何两点都没有边就是独立集

Independent Set 问题：给定无向图$G=(V,E)$以及$k\in Z^+$判断是否存在大小为 k 的独立集

还是长度为 5 的线段显然存在 k=1,2,3 的独立集，更大显然不成立

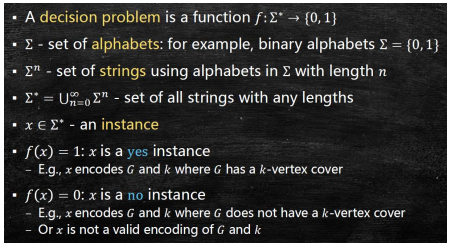
Subset Sum 问题：给定一组整数集合$S=\{a_1,\dots,a_n\}$以及$k\in Z^+$判断是否存在S的一个子集$T\subseteq S$且子集中的所有元素之和恰好为k。

例如对于$S=\{1,1,6,13,27\}$ and $k=21\rightarrow yes,\ S=\{1,1,6,13,27\}$ and $k=22\rightarrow no.$

Hamiltonian Path：无向图中经过每个节点一次的路径（回路则包括回到起点）

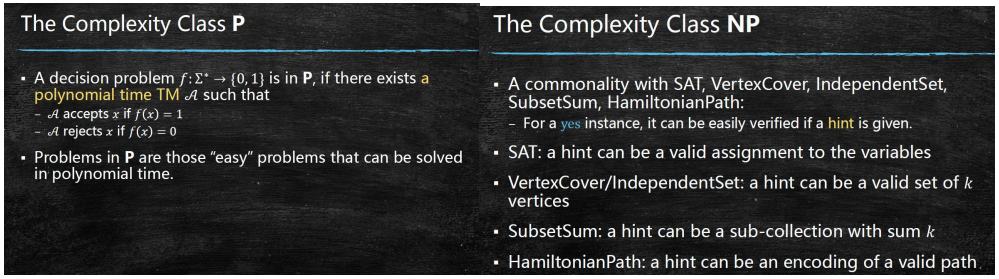
Hamiltonian Path 问题：给定无向图$G=(V,E)$判断是否存在哈密顿路径

这几个问题的侧重点都在于给出**判断**是 yes 或 no 以及运行时间是否是多项式的。引入了决策问题概念



一个决策问题我们认为他是简单的，当存在一个多项式时间的算法去计算这个函数的值时。（图灵机）

图灵机简单来理解就是能够提供"accept"或者"reject"的回应，如果一个算法或程序是多项式时间的那么它在图灵机上也一定能够经过多项式时间计算完毕，进而引入了 **P** 问题和 **NP** 问题的概念：



Examples for Problems in P

- [PATH] Given a graph $G = (V, E)$ and $s, t \in V$, decide if there is a path from s to t.
  - Build a TM that runs BFS or DFS at s; accept if t is reached; reject if the search terminates without reaching t.
  - PATH ∈ **P**
- [k-FLOW] Given a directed graph $G = (V, E)$, $s, t \in V$, a capacity function $c: E \to \mathbb{R}^+$, and $k \in \mathbb{R}^+$, decide if there is a flow with value at least k.
  - Build a TM that implements Edmonds-Karp, Dinic's, or other algorithms.
  - k-FLOW ∈ **P**
- [PRIME] Given $k \in \mathbb{Z}^+$ encoded in binary string, decide if k is a prime number.
  - [Agrawal, Kayal & Saxena, 2004] PRIME ∈ **P**

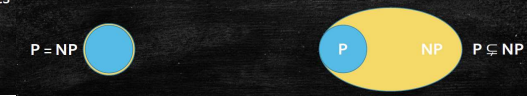
- **NP**: Problems whose yes instances can be efficiently verified if hints are given.
- Formal Definition. A decision problem $f: \Sigma^* \to \{0,1\}$ is in **NP** if there exist a polynomial q and a polynomial time TM $\mathcal{A}$ such that
  - If x is a yes instance ($f(x) = 1$), there exists $y \in \Sigma^*$ with $|y| \le q(|x|)$ such that $\mathcal{A}$ accepts the input (x, y)
  - If x is a no instance ($f(x) = 0$), for all $y \in \Sigma^*$ with $|y| \le q(|x|)$ such that $\mathcal{A}$ rejects the input (x, y)
- The string y is called a certificate.
- SAT, VertexCover, IndependentSet, SubsetSum, HamiltonianPath are all in **NP**.

P ⊆ NP

- Proof. If a decision problem $f: \Sigma^* \to \{0,1\}$ is in **P**, we will show it is in **NP**.
- By definition of **P**, there exists a polynomial time TM $\mathcal{A}$ such that $\mathcal{A}$ accepts x if and only if $f(x) = 1$.
- Let $\mathcal{A}'$ be a TM such that it outputs $\mathcal{A}(x)$ on input (x, y). That is, $\mathcal{A}'$ implements $\mathcal{A}$ and ignore y.
- If $f(x) = 1$, there exists y, say, $y = \emptyset$, such that $\mathcal{A}'$ accepts (x, y).
- If $f(x) = 0$, for all y, $\mathcal{A}'$ rejects (x, y).
- Thus, $f \in$ **NP**.

Central Open Problem: P vs. NP

- Central Open Problem: Does **P** equals **NP**?
- Most research believes no...
  - If **P** = **NP**, we do not need the certificate: we can just "guess" it correctly and efficiently... This doesn't seem possible.
  - Given an exam question, do you believe solving the question is much harder than checking if someone's solution to the question is correct? **P** = **NP** would suggest they are equally easy...



很多 NP 问题在 P 问题中都是算**"未知"**的，如我们上文提到的 5 种 NP 困难问题。接下来想探究的是，是否这些问题中有的问题"更难"？还是它们一样难？

3SAT 布尔表达式：CNF formula that each clause contains at most three literals 每个 clause 至多 3 个变量

3SAT 问题：给定一个 3-CNF 式子φ判断是否能够找到使得φ为 true 的变量取值

显然，3SAT 问题至多和 SAT 一样难毕竟它是 SAT 的一种特例，现在要证明 3SAT 至少比 SAT 难

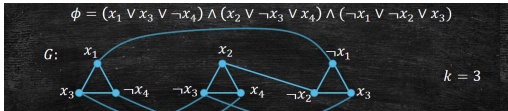
想法是：任何一个 CNF formula φ都可以多项式时间建立一个 3-CNF formula φ'使得$\phi_{yes} \Leftrightarrow \phi'_{yes}$

关键在于任何一个 long clause 都可以被拆分成 shorter clause，任何一个 shorter clause 可以被补长

- If we have a polynomial time algorithm for 3SAT, we have a polynomial time algorithm for SAT:
  - Given input ϕ, compute ϕ'
  - Solve 3SAT instance ϕ' and obtain answer yes or no
  - Output the same answer for ϕ

Independent Set 是"weakly harder" than 3SAT.

想法是：只要证明任何一个 3SAT instance 都可以在多项式的时间内规约到一个 Independent Set instance$(G = (V, E), k)$使得$\phi_{yes} \Leftrightarrow (G = (V, E), k)_{yes}$。其实这个就是作业中的一道题目（2SAT）的实现方式，关键就是建立图，最终要找到一个大小为 k 的独立集即可（k 是变量元素个数）。



- If ϕ is a yes instance, each clause must have a literal with value true.
- For each triangle in G, pick exactly one vertex representing a true literal in S.
- S is an independent set and $|S| = k$. So (G, k) is a yes instance.

- If ϕ is a no instance, for contradiction, assume (G, k) is a yes instance. Let S with $|S| = k$ be the independent set.
- S must contain exactly one vertex in each triangle.
  - because any two vertices in a triangle is connected
- Assign true to the literals representing the chosen vertices.
  - We will not assign both true and false to a same literal, as x_i and $\neg x_i$ is connected.
- For variables not yet assigned a value, assign values to them arbitrarily.
- The resultant assignment makes ϕ true (as each clause has a true literal), contradicting to that ϕ is a no instance!

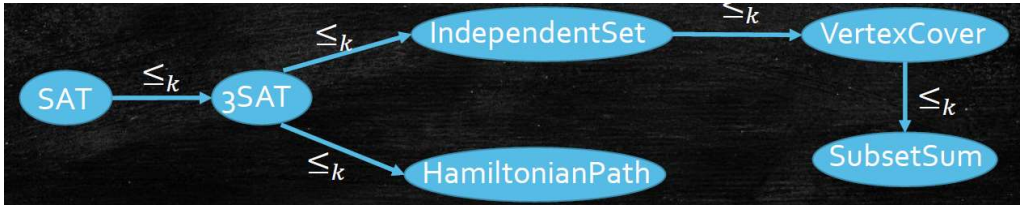
No instance 可以用 Pigeonhole principle（抽屉原理）理解，每个三角形都选择一个不然必然会冲突

Reduction 规约：一个决策问题f Karp reduce to a decision problem g当且仅当存在多项式图灵机 A 使

- 如果 f 输入了 yes instance 那么 g 的输出也是 yes
- 如果 f 输入了 no instance 那么 g 的输出也是 no

用$f \le_k g$表示 f 比 g 弱一些。一般第二个条件可以换做 g 的输出是 yes 推导出 f 的输出是 yes（逆否）

在前面的内容中我们已经证明了$SAT \le_k 3SAT \le_k Independent\ Set$，还有更多很类似的问题



接下来，对于所有 NP 问题里面，我们想找**最难**的问题，因此定义了如下 NPC 和 NP-hard 问题：

1. A decision problem f is NP − hard if $g \le_k f$ for any problem $g \in$ **NP**.
2. A decision problem f is NP − complete if $f \in$ **NP** and $g \le_k f$ for any problem $g \in$ **NP**.

Cook-Levin Theorem: SAT is NP-complete and this implies that each problem is NP-complete by transitivity.

These problems are "equally hard", and are the hardest problems in **NP.**

Intuition behind Cook-Levin Theorem

- We have seen SAT is in **NP**.
- Consider an arbitrary **NP** problem f. We will show $f \le_k$ SAT.
- For a yes instance x, there exist a polynomial time TM $\mathcal{A}$ and a polynomial length certificate y such that $\mathcal{A}$ accepts (x, y).
- Consider a computation tableau that records the tape at every step of $\mathcal{A}$'s execution.

	x					y				
	x_0	x_1	x_2	...	x_n	y_0	y_1	y_2	...	y_m
Step 0	1	1	0	0	0	1	1	1	1	0
Step 1	1	1	1	0	0	1	1	1	1	0
Step 2	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Final Step	0	1	1	0	0	0	1	1	0	0

Intuition behind Cook-Levin Theorem

	x					y				
	x_0	x_1	x_2	...	x_n	y_0	y_1	y_2	...	y_m
Step 0	1	1	0	0	0	1	1	1	1	0
Step 1	1	1	1	0	0	1	1	1	1	0
Step 2	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
Final Step	0	1	1	0	0	0	1	1	0	0

- For each y_i and each cell in the tape from Step 1 to the final step, construct a Boolean variable for the SAT instance.
- We can use clauses to ensure the tableau gives a valid TM computation.
- E.g., we can use two clauses $(x \vee \neg y) \wedge (\neg x \vee y)$ to enforce $x = y$.

NP 困难和 NPC 怎么区分？对于决策问题而言，NPC 实际上是在 NP 中的 NP 困难问题。

NP 困难问题可以形容最优化问题：最大独立集、最小顶点覆盖、最大 3SAT、最长简单路径等等

VectorSubsetSum:任意向量集合$S = \{a_1, \dots, a_n : a_i \in Z^m\}$ and a vector $k \in Z^m$确定是否存在子集求和为 k

接下来尝试证明$VertexCover \leq_k VectorSubsetSum \leq_k SubsetSum$

VertexCover $\leq_k$ VectorSubsetSum

- Given a VertexCover instance $(G = (V, E), k)$, we will construct a VectorSubsetSum instance $(S, \mathbf{k})$.
- First, we label the edges with $1, 2, \dots, |E|$ (in arbitrary order).
- For each $v_i \in V$, construct a $(|E| + 1)$-dimensional vector $\mathbf{a}_i \in S$ such that $\mathbf{a}_i[0] = 1$ and for each $j = 1, \dots, |E|$:
  $$\mathbf{a}_i[j] = \begin{cases} 1 & \text{if } v_i \text{ is an endpoint of edge } j \\ 0 & \text{otherwise} \end{cases}$$
- For each edge j, construct $\mathbf{b}_j \in S$ where $\mathbf{b}_j[j] = 1$ is the only non-zero entry.
- Let $\mathbf{k} = (k, 2, 2, \dots, 2)$.

Ideas Behind the Reduction

- Picking $\mathbf{a}_i \in T$ represents picking v_i in the vertex cover.
- The 0-th entry of $\mathbf{k}$ is set to k, enforcing exactly k vertices must be picked.
- The j-th entry of $\mathbf{k}$ is set to 2 enforcing edge j must be covered:
  - Say, edge j is (v_{i_1}, v_{i_2})
  - If $\mathbf{a}_{i_1}, \mathbf{a}_{i_2} \in T$, we are fine, as the j-th entries already add up to 2.
  - If one of $\mathbf{a}_{i_1}, \mathbf{a}_{i_2}$ is chosen in T, we are also fine, as we can include $\mathbf{b}_j \in T$.
  - If $\mathbf{a}_{i_1}, \mathbf{a}_{i_2} \notin T$, we are not fine: the j-th entries add up to at most 1 even if we include $\mathbf{b}_j \in T$.
- We are done! VertexCover $\leq_k$ VectorSubsetSum

VectorSubsetSum $\leq_k$ SubsetSum

- We can convert a vector $\mathbf{a} = (\mathbf{a}[0], \dots, \mathbf{a}[m])$ to a large number.
- For example, convert $\mathbf{a} = (1, 4, 5, 3)$ to number 1453
  - $1453 = \mathbf{a}[0] \times 1000 + \mathbf{a}[1] \times 100 + \mathbf{a}[2] \times 10 + \mathbf{a}[3] \times 1$
- We are using decimal representation in the above example...
- To avoid carry, use N-ary representation instead (for sufficiently large N)?
- Additions with vectors are now equivalent to additions with numbers, since we do not have carry issue.
- VectorSubsetSum $\leq_k$ SubsetSum

SubsetSum is NP-complete

- We have seen SubsetSum is in NP.
- We have proved
  1. VertexCover $\leq_k$ VectorSubsetSum
  2. VectorSubsetSum $\leq_k$ SubsetSum

接下来开始学习如何写一个 NPC 的正式证明以及更多重要的 NPC 问题。

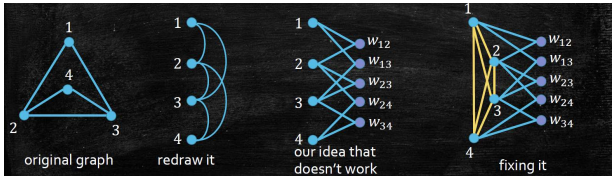
1. 找正确的规约问题。最好的情况就是找最像的方向。

Dominating Set:无向图$G = (V, E)$，存在一个顶点子集使得它里面总能找到一个点与补集中的点邻接。

Dominating Set 问题：$(G = (V, E), k \in Z^+)$判断是否存在一个大小为 k 的 Dominating Set

2. 尝试修复你的规约如果它不成立

Dominating Set 问题与 Vertex Cover 问题有很多相似之处，但是单纯加中间点好像还不够，事实上应该要在所有点集之间连边才是最安全的方式。



NPC 问题的证明：

Four Parts for proving f is NP-complete:

1. Prove that f is in **NP**
2. Present the reduction $g \leq_k f$ for an NP-complete problem g
3. Show that yes instances of g are mapped to yes instances of f
4. Show that no instances of g are mapped to no instances of f
   - Most of the time, it is easier to prove its contrapositive: if an instance x of g is mapped to a yes instance of f, then x is a yes instance of g.

Proof. First of all, DominatingSet is in **NP**, as a dominating set S can be served as a certificate, and it can be verified in polynomial time whether S is a dominating set and whether $|S| = k$.

To show that DominatingSet is NP-complete, we present a reduction from VertexCover. Given a VertexCover instance $(G = (V, E), k)$, we construct a DominatingSet instance $(G' = (V', E'), k')$ as follows.

The vertex set is $V' = V \cup E$, which is defined as follows. For each vertex $v \in V$ in the VertexCover instance, construct a vertex $\overline{v} \in \overline{V} \subseteq V'$; for each edge $e \in E$ in the VertexCover instance, construct a vertex $w_e \in \overline{E} \subseteq V'$.

The edge set E' is defined as follows. For each edge $e = (u, v)$ in the VertexCover instance, build two edges $(\overline{u}, w_e), (\overline{v}, w_e) \in E'$. For any two vertices $\overline{u}, \overline{v}$ in $\overline{V}$, build an edge $(\overline{u}, \overline{v})$.

Define $k' = k$.

Proof (Continued).

Suppose $(G = (V, E), k)$ is a yes VertexCover instance. There exists a vertex cover $S \subseteq V$ with $|S| = k$. We will prove $\overline{S}$ corresponding S is a dominating set in G'.

For each vertex in $\overline{V}$, it is covered by any vertex in $\overline{S}$ as $\overline{V}$ forms a clique.

For each vertex w_e in $\overline{E}$, let $e = (u, v) \in E$ be the corresponding edge in the VertexCover instance. We have either $u \in S$ or $v \in S$ (or both), as S is a vertex cover. This implies either $\overline{u} \in \overline{S}$ or $\overline{v} \in \overline{S}$ (or both), which further implies w_e is covered as $(\overline{u}, w_e), (\overline{v}, w_e) \in \overline{E}$ by our construction.

Since $|\overline{S}| = |S| = k = k'$, the DominatingSet instance we constructed is a yes instance.

Suppose $(G' = (V', E'), k')$ is a yes DominatingSet instance. There exists a dominating set $S' \subseteq V' = \overline{V} \cup \overline{E}$ with $|S'| = k' = k$. We aim to show that $(G = (V, E), k)$ is a yes VertexCover instance.
First of all, we can assume $S' \subseteq \overline{V}$ without loss of generality. If we have $w_e \in S'$ for some $w_e \in \overline{E}$, we can replace w_e with either $\overline{u}$ or $\overline{v}$ for the edge $e = (u, v)$ in the VertexCover instance. (In the case $\overline{u}$ and $\overline{v}$ have already been included in S', we can replace w_e with any unpicked vertex in $\overline{V}$.) It is easy to see that S' is still a dominating set after the change, as the set of vertices covered by either $\overline{u}$ or $\overline{v}$ is a superset of the set of vertices covered by w_e (which is just $\{\overline{u}, \overline{v}\}$).
Next, since $S' \subseteq \overline{V}$, S' corresponds to a vertex set $S \subseteq V$ in the VertexCover instance with $|S| = |S'| = k' = k$. It remains to show S is a vertex cover.
For any edge $e = (u, v)$, we have either $\overline{u} \in S'$ or $\overline{v} \in S'$ (or both) since S' is a dominating set and $\overline{u}, \overline{v}$ are the only two vertices that can cover w_e. This implies $u \in S$ or $v \in S$ (or both), so S is a vertex cover.

3. 证明 no instance 匹配 no instance 实际上可以变成逆否命题证明

4. 千万不要弄错证明方向，严格明晰方向是：任意的 f 能够 reduce to g 而不是 g reduce to f。

团 Clique 问题：无向图判断是否存在一个完全子图且大小为 k

再例如$SubsetSum$ +约束了正整数$SubsetSum$ +$\leq_k SubsetSum$

再例如$Partition$ +约束了正整数也是同样的$Partition$ +$\leq_k Partition$

5. 找到一个中间状态也是不错的方式即$g \leq_k h \leq_k f$

   哈密顿路径问题是 NPC 的，我们通过有向哈密顿路径这个中间值来进行选取

DirectedHamiltonianPath 问题：给定有向图与源汇点，判断是否存在从源点到汇点的哈密顿路径

3SAT $\leq_k$ DirectedHamiltonianPath $\leq_k$ HamiltonianPath

6. 构造小工具（证明**3SAT $\leq_k$ DirectedHamiltonianPath**）

If ϕ is a yes instance, the graph has a Hamiltonian path:

- For each clause, choose a representative true literature.
- Go from s to t, and visit each v_j from its representative by taking a detour.

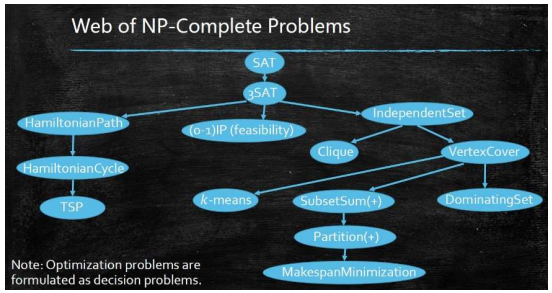
If the graph has a Hamiltonian path, ϕ is a yes instance:

- The Hamiltonian path has to go from s to t.
- Each v_j has to be visited by a detour from a variable.
- The variable's value is then determined.

DirectedHamiltonianPath $\leq_k$ HamiltonianPath

- Vertex Gadget:

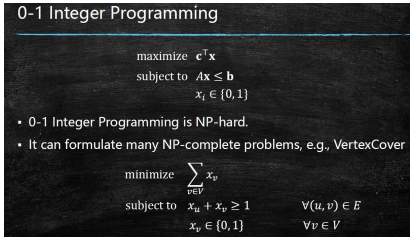
哈密顿回路实际上也是 NPC 的问题



NP 困难优化问题：例如 Makespan 就是从 Partition 规约来|旅行商问题 TSP 是哈密顿环但是最小值

- A maximization problem is NP-hard if there exists $k \in \mathbb{R}$ such that deciding whether $\mathrm{OPT} \geq k$ is NP-hard.
- A minimization problem is NP-hard if there exists $k \in \mathbb{R}$ such that deciding whether $\mathrm{OPT} \leq k$ is NP-hard.

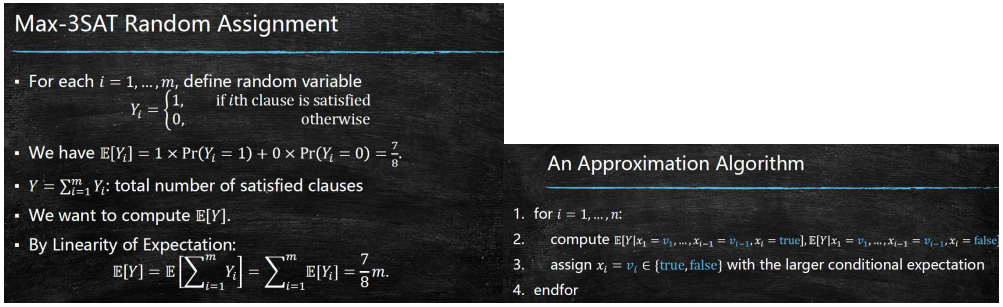
k-Means（k 聚类）、01 整数规划（可以变成 Vertex Cover 的 LP 形式）、



近似算法：用来解决 NP-hard 问题的一种方式。NP 困难问题分析时采取**最坏情况分析**。

- $\text{IntegralityGap} = \frac{\text{OPT(IP)}}{\text{OPT(LP)}}$
- If you analyze your approximation algorithm based on OPT(LP)...
- the best approximation ratio you can ever get is the integrality gap!

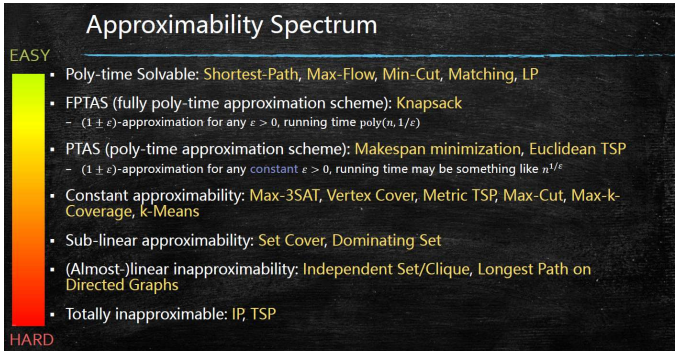
MAX-3SAT：给定一个 3CNF 式输出最大能达成的 clauses.



事实上很多算法的近似算法都是贪心的近似（例如任务调度 Makespan 问题）

在最大 k 覆盖问题中近似算法如下：

- 初始化空集合 S，并且重复做：
  - 找任何一个$A \in T \backslash S$使得$f(S \cup \{A\}) - f(S)$最大
  - 更新此时的空集合$S \leftarrow S \cup \{A\}$
- 直到$f(S) = |U| = n(\text{for set cover})$ and $|S| = k(\text{for max} - \text{k} - \text{coverage})$



常见近似算法：

Vertex Cover:2 倍近似　Metric TSP:1.5 倍近似　Max-3SAT:0.875 倍近似 Set Cover: ln n 倍近似

Max-k-Coverage: (1-1/e)倍近似　Max-Cut: 0.5 倍近似

常见近似算法技巧：

贪心：makespan minimization, set cover, max-k-coverage　　局部搜索: max-cut，作业 6.1

LP 松弛: Vertex cover, metric facility location(度量设施选址).　条件期望和解随机处理(max-3SAT)