

# Divide and Conquer 分治法解决大整数乘法

## 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  $2*4\text{bit} \rightarrow 4*2\text{bit}$

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

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.

$$\begin{aligned} xy &= (a * 10^{\frac{n}{2}} + b)(c * 10^{\frac{n}{2}} + d) = ac * 10^n + (ad + bc) * 10^{\frac{n}{2}} + bd \\ xy &= ac * 10^n + ((a + b)(c + d) - ac - bd) * 10^{\frac{n}{2}} + bd \end{aligned}$$

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

## Divide and Conquer      分治法解决 sorting(排序)

### Overview:

- ① Invert on sort.
- ② Merge sort
  - Framework
  - Combine
- ③ Master Theorem
- ④ Count Inversions

### Sorting Problem:

Input: A set of n integers  $x_1, x_2, \dots, x_n$

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

```
• Plan
  - Maintain 2 pointers  $i = 1, j = 1$ 
  - Repeat
    • Append  $\min\{a_i, b_j\}$  to C
    • If  $a_i$  is smaller, then move  $i$  to  $i + 1$ ; If  $b_j$  is smaller, then move  $j$  to  $j + 1$ .
    • Break if  $i > n$  or  $j > m$ 
  - Append the remainder of the non-empty list to C.
```

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：

$$T(N) = N \log N + N = O(N \log N)$$

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。

## Master Theorem:

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

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

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。

## Counting Inversions (逆序对计算)

Input: a list of n integers  $x_1, \dots, x_n$

Output: number of **inversions**.

Property:  $i, j$  are **inverted** if  $i < j$  but  $x_i > x_j$ .

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  $c_1, c_2$ ，合并后产生的逆序对数为  $c_3$

则总逆序对数为  $c_1 + c_2 + c_3$ ，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

$$T(n) \leq O(n) + \max[T(|L|), T(|R|)] \leq O(n) + T(n-1) \leq O(n) + O(n-1) + \dots + O(1) = O(n^2)$$

最坏情况:  $O(n^2)$       最好情况:  $O(n)$

## Divide and Conquer      分治法解决 Selecting Problem

### One-by-one Selection

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

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

$$T(n) \leq O(n) + \max[T(|L|), T(|R|)] \leq O(n) + T(n-1) \leq O(n) + O(n-1) + \dots + O(1) = O(n^2)$$

最坏情况:  $O(n^2)$       最好情况:  $O(n)$

似乎很 bad, 因此需要进行一定的优化? (可能能够优化?)

如果是选择随机的, 比较 lucky 的情况下我们每次选择并分类序列需要  $O(n)$  的期望:

## Analysis

- $\tau(n)$ : Time we reduce  $n$  to  $\frac{3n}{4}$
- $T(n) = \tau(n) + T\left(\frac{3n}{4}\right)$
- $E[\tau(n)]$ : The expected time we reduce  $n$  to  $\frac{3n}{4}$
- $$E[T(n)] = E\left[\tau(n) + T\left(\frac{3n}{4}\right)\right] \\ = E[\tau(n)] + E\left[T\left(\frac{3n}{4}\right)\right]$$
- $E[\tau(n)] = O(n)$
- $E[T(n)] = O(n) + E\left[T\left(\frac{3n}{4}\right)\right] = O(n)$

Fact

Since we are lucky with probability  $\frac{1}{2}$ ,  
so the expected number of rounds  
it takes to become lucky is 2.

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）？

### Median of medians (1973)

Blum, M.; Floyd, R. W.; Pratt, V. R.;  
Rivest, R. L.; Tarjan, R. E.

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  $3*n/10$ 。

We have  $\frac{n}{5}$  groups, so  $\frac{n}{5}$  medians.

$v$  is no greater than  $n/10$  medians, no less than  $n/10$  medians.

Each median is no greater than 2 integers, no less than 2 integers.

$v$  is no greater than  $\frac{3n}{10}$  integers, no less than  $3n/10$  integers.

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

(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：

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

$v$  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  $(x_i, y_i)$

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。

- Straight-forward?
  - Compute all  $\binom{n}{2}^2$  pairs, with one point on each side.
  - Return the closest one.
- What about the running time?
  - Divide:  $O(n \log n)$ 
    - Points are sorted by the x-coordinate.
    - By a vertical line so that each side has  $n/2$  points
  - Recurse:  $2T\left(\frac{n}{2}\right)$ 
    - Find the closest pair in each side.
  - Combine:  $O(n^2)$
  - Overall:  $T(n) = O(n^2) + 2T\left(\frac{n}{2}\right) = O(n^2)$

Master  
Theorem

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

- $\Theta(n^2)$ 
  - Selection Sort
  - Bubble Sort
  - Insertion Sort

- $\Theta(n \log n)$ 
  - Quick Sort
  - Merge Sort
  - Balance Tree

## Spaghetti Sort

The diagram shows a hand holding a vertical stack of spaghetti strands against a table edge. One strand is being pulled down by another hand, representing the process of breaking spaghetti of length  $n$  and pushing them against the table.

- For each number, **break** a piece of spaghetti whose length is that number.  $O(n)$
- Take all spaghetti and **push** them against the table!  $O(1)$
- Keep **touching** and **removing** spaghetti from the top by your other hand.  $O(n)$

天才般的意大利面  $O(n)$ 排序算法，可是电脑不会。

电脑会的，应该是提供了 Comparison 大小关系的前提下进行的比较模型。对于 Merge sort 而言它恰恰是需要比较次数最小的一种排序算法，太棒了！这种只提供大小比较的排序方式可以参照第一次作业的交互题。

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。

## Basic Graph Algorithm 基本图论算法

图：由顶点集和边集构成的集合。

有向图与无向图都需要考虑连通性的一个情况。其实无向图可以被认为是特殊的有向图。

图存储的方式有邻接矩阵和邻接表，但是常见是邻接表因为图的点规模一般都非常大。

邻接矩阵的空间大小为  $O(n^2)$ 而邻接表的空间大小为  $O(V+E)$ 因此更优异。

Reachability：连通性， $u-v$  节点之间是否连通

Connected component: 连通分量，满足任意两点互达的最大子图（无向图）

Strongly connected component: 有向图强连通分量。

# 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

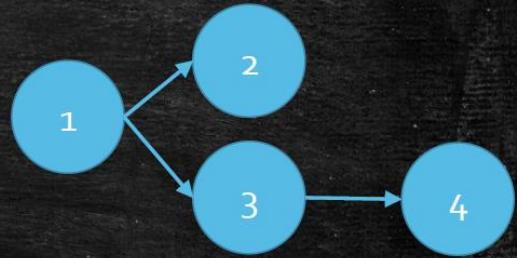
## Topological Ordering for DAG

### ▪ Observation

- DAG must have a **tail**.
- **Tail:** vertices that **do not have** outgoing edges.
- **Tail** can be the last one in the topological order.

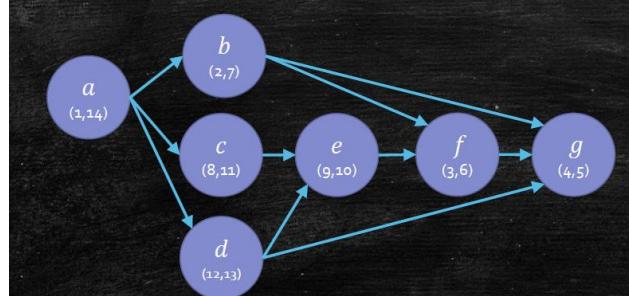
### ▪ Algorithm

- Find a **tail**.
- Put **it** to be the last one in the topological order.
- Remove the **tail** in the graph.
- Repeat...



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。

- We need repeat finding a **tail**.
- Who must be a **tail** in DFS?



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  $O(V \log V + E)$

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

利用 Dijkstra 算法进行最短路的计算

- 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:  $O(|V|^2 + |E|)$

## 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  $tdist[v]$ .
- $T \leftarrow T + \{v\}$

$|V|$  rounds

### 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  $pre[u] \leftarrow v$ .

$|E|$  rounds

$|E|$  rounds

当然，利用 SPFA 也是可以考虑的（需要现学）

## Greedy 贪心思想

Input:  $n$  homework, each homework  $j$  has a size  $s_j$ , and a deadline  $d_j$ .

Output: output a time schedule of doing homework!

最经典的贪心问题：最小生成树的 Prim 与 Kruskal 算法

## 最小生成树的 Prim 与 Kruskal 算法

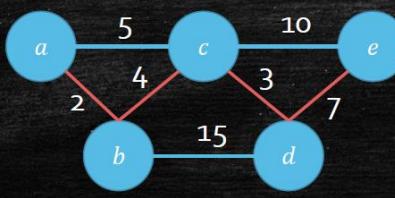
易知，单纯为了找生成树，可以很容易得到 BFS 与 DFS 的算法（只要遍历全部即可）

• **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。

## 1. 最小生成树：Kruskal 算法&Prim 算法

### • 从边出发：**Kruskal 算法**

- 考虑图中权值最小的边。如果加入这条边不会导致回路，则加入；否则考虑下一条边，直到包含了所有的顶点。
- 如何考虑图中权值最小的边：**优先队列**优化。
- 如何判断是否加入后有回路：**并查集（不相交集）**优化。
- 时间复杂度： $O(E \log E) + O(E \log V) \sim O(E \log E)$  ∵  $|E| \gg |V|$  in general.
- 因此，适合稀疏图。证明过程现场描述。

### • 从点出发：**Prim 算法**

- 从顶点的角度出发。初始时，顶点集  $U$  为空，然后逐个加入可到达顶点中最短距离的顶点，直到包含所有顶点。
- 如何选点：用一个布尔型的一维数组来判断站点是否已经选择。
- 如何判边：用一个数组存放已有站点到外界各个站点的最短路径值。
- 时间复杂度： $O(V^2)$
- 因此，适合稠密图。证明过程现场描述。

## ▪ Recall Union-Find Set

- Find:  $O(\log n)$
- Union:  $O(1)$

## ▪ Kruskal

- $O(|E| \log |E|)$  for sorting.
- $2|E|$  round: check group
- $|V|$  round: union group ?
- $O(|E| \log |E|) = O(|E| \log |V|)$

$|E|=|V|^2$  因此  $\log |E|=2\log |V|$

## More Greedy Problems 更多的贪心算法题目

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  $s_i$  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  $a \leq b \leq k$ ，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）

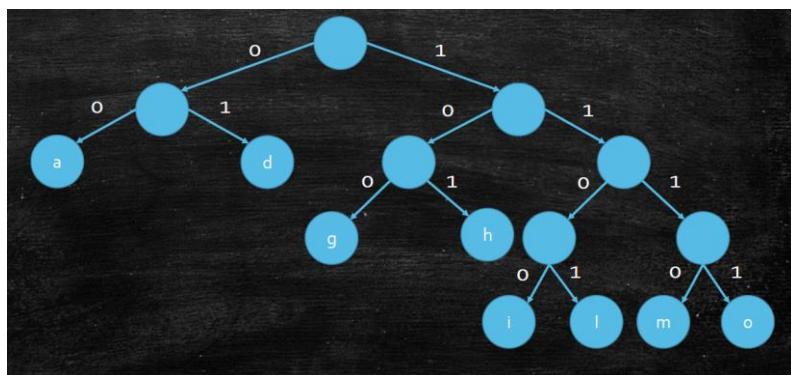
如何编码书本：首先提供字母表的编码服务

其次编码所有语句即可。

a	oooo	a	o
d	ooo1	d	1
g	oo10	g	10
h	oo11	h	11
i	o100	i	100
l	o101	l	101
m	o110	m	110
o	o111	o	111
r	1000	r	1000
s	1001	s	1001
t	1010	t	1010
space	1011	space	1011

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。

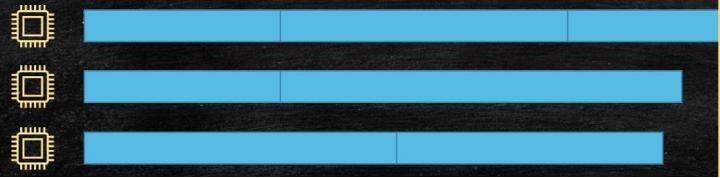


贪心思想：每次选择最小的节点合并起来。

- The Big Idea
  - The local greedy choice do not ruin out OPT.
- Assume we are still in a partial-OPT,
- after Merging two smallest elements, are we still in?

Makespan Minimization 机器任务分配问题

- **Input:**  $m$  identical machines,  $n$  jobs with size  $p_i$ .
- **Output:** the minimized max completion time (**Makespan**) of all these jobs on  $m$  machines

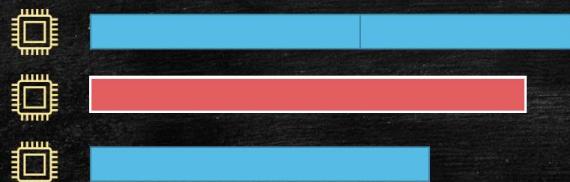


简单来说就是多个处理器的时候怎么样能够最早完成，这个可以理解为作业有好几个同学帮你一起做  
的时候怎么做能够最早完成。实际上这个问题可能局部最优未必是全局最优的。

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

- **LPT Algorithm**
  - Longest Processing Time First.
  - Insert jobs into the earliest finished machine.



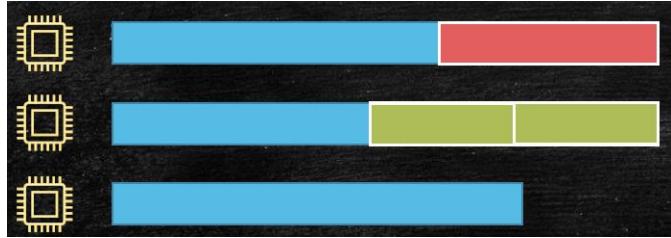
尝试进行贪心的证明：

- Assume we are still in OPT.
- We put the longest left job onto the earliest finished machine.
- **Discussion! Are we still in an OPT?**



- Suppose not.
- We can swap red and green!





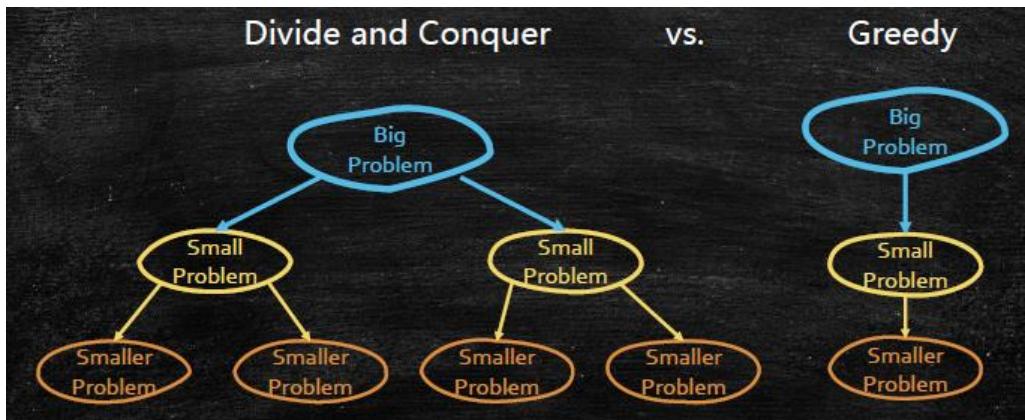
因此，可以发现，永远放最长的局部是最优的，但是全局看来不是最优的。

- Makespan Minimization is a NP-hard problem.
- Find a poly time algorithm for it means  $P = NP$ .
- Is Simple Greedy or LPT very bad?
- At least, they are better than arbitrary scheduling.

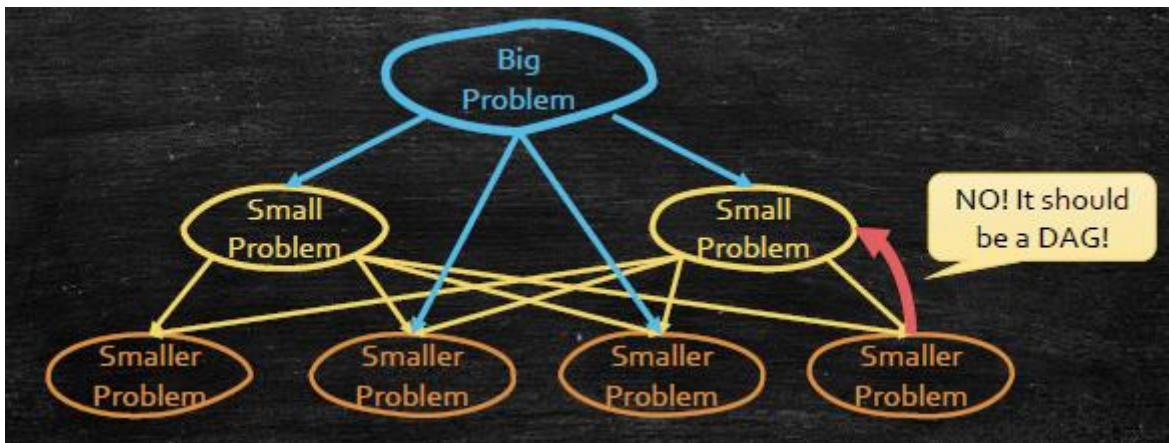
这一个题目是 NP 问题，而贪心至少能得到较好的结果（分析这个“较好”的程度参照课件）。

## 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：

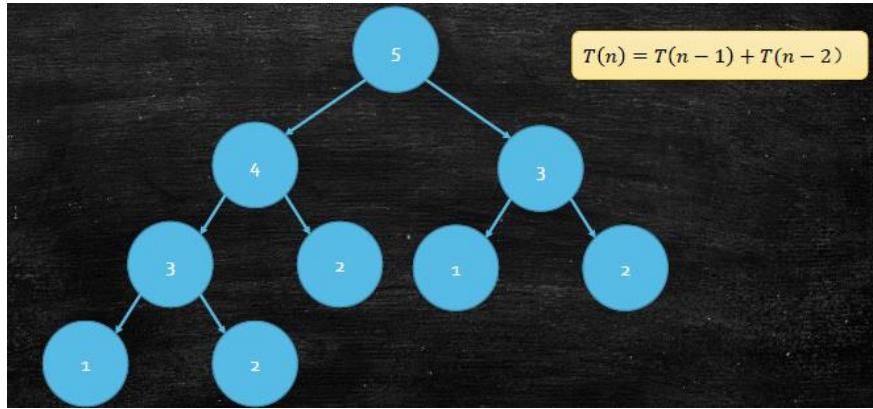


即应该是一个 DAG:有向无环图

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)



- Fibonacci

- $Fib(n) = Fib(n-1) + Fib(n-2)$

- Solve Recursively

### Fibonacci

```
function fib(n)
    if n>1
        return fib(n-1) + fib(n-2)
    else
        return 1
```

### Fibonacci

$\Theta(n)$

```
function fib(n)
    fib[0] = fib[1] = 1
    for i = 2 to n
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

对于 DP 问题的设计思路:

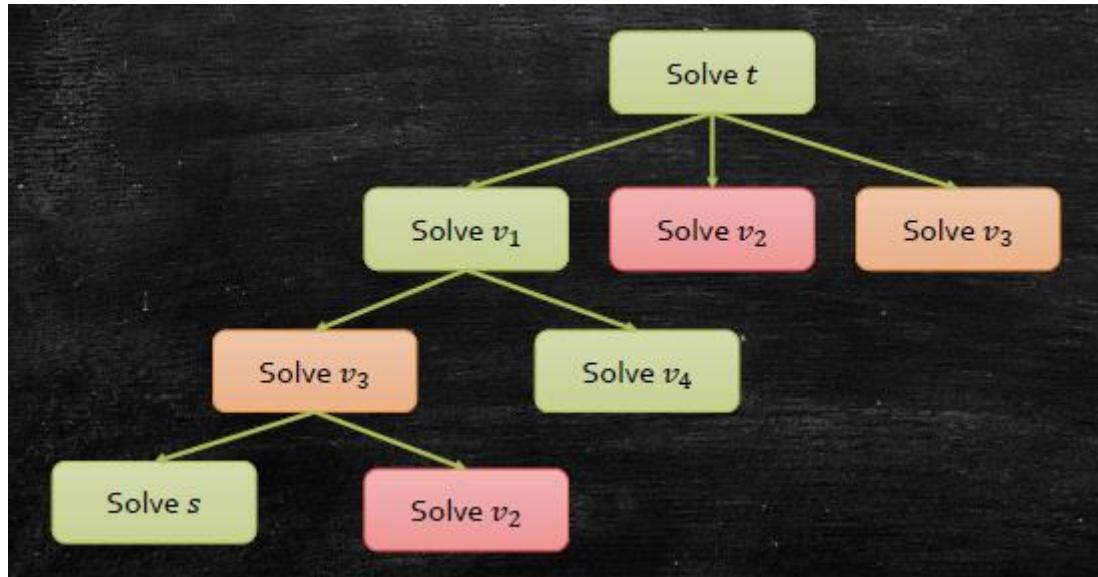
设计递归算法

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

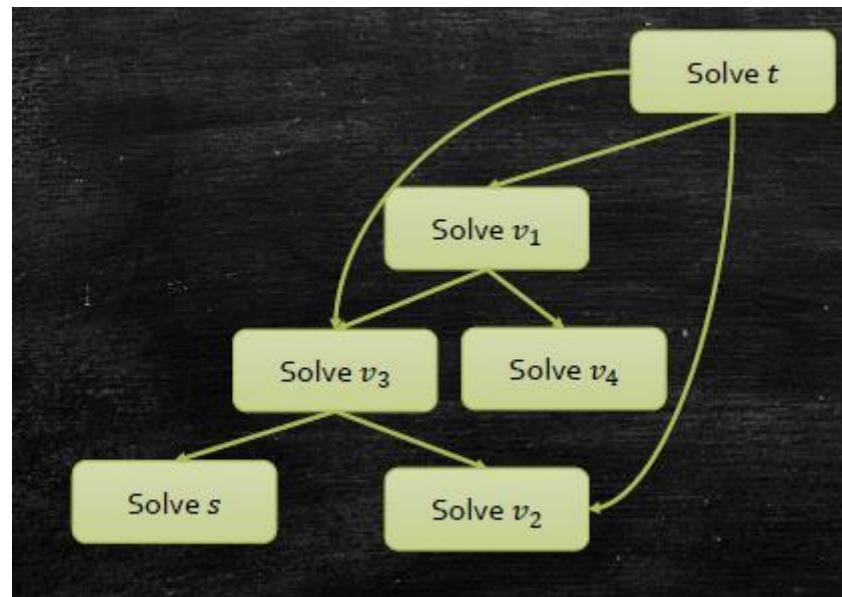
按照拓扑序解决并存储子问题的值

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。

$LIS[k] = \max(LIS[I] + 1 \text{ if } a[i] \leq a[k])$  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  $\log n$  时间就确定到恰好夹中间的位置，那么可以直接找到  $\max(LIS[I])$   
详情看 11.16 笔记的单调队列部分。

## 字符串修改距离 Edit Distance：分辨两个字符串的相似程度

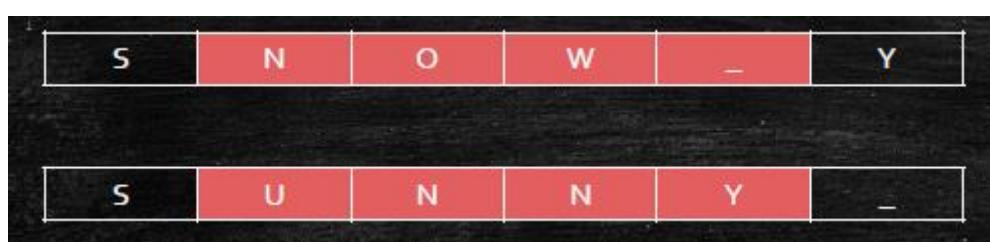
如何用最小的操作次数从一个字符串变换到另一个字符串呢？

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

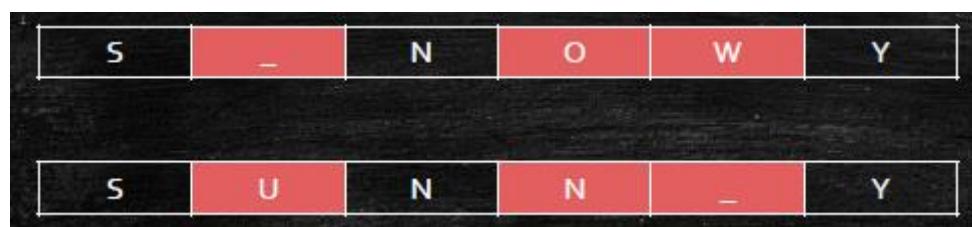
换一个角度看：首先是 0 代价的插入过程去“对齐”，然后插入、删除、替换都变成了重写

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？

- $ED[i, j]$ : The edit distance between  $X[1..i]$  and  $Y[1..j]$ .
- $ED[n, m]$ : The edit distance between  $X$  and  $Y$ .
- How to solve  $ED[i, j]$ : min of three cases
  - $ED[i - 1, j - 1] + \mathbf{1}_{x_i \neq x_j}$
  - $ED[i, j - 1] + 1$
  - $ED[i - 1, j] + 1$

## 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

W = 100000		
	Value	Cost
iPhone	8888	8888
Algorithm Book	10000	500
Laptop	8888	8500
Hermès	90000	100000

But when we become rich...

Problem: items are not divisible!

(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！

$O(nW)$  is not polynomial!

- Input:  $n$  items with cost  $c_i$  and value  $v_i$ , and a capacity  $W$ .
- Input size: the unit of bits to represent the input.
- $W = 2^n$  by using  $n$  bits
  - time complexity becomes  $O(2^n)$ .

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  $F[W]$

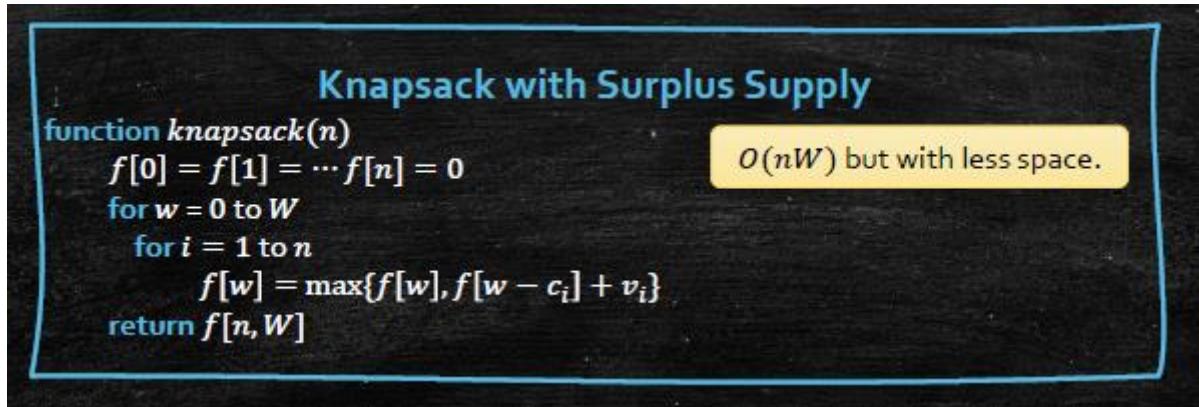
其中  $k$  表示最多可以选取的数目，因此  $k = 1 \rightarrow W/c[i]$

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  $O(2^n)$

## Dynamic Programming 动态规划

【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

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

```

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

- $dist[k, v] = \min\{dist[k - 1, v], dist[k - 1, u] + d(u, v)\}$

$f[k, v]$	$s$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$	$v_7$	...	$v_{ V }$
0	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
1									
2									
3							$f[k, v]$		
...									
$ V $									

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？

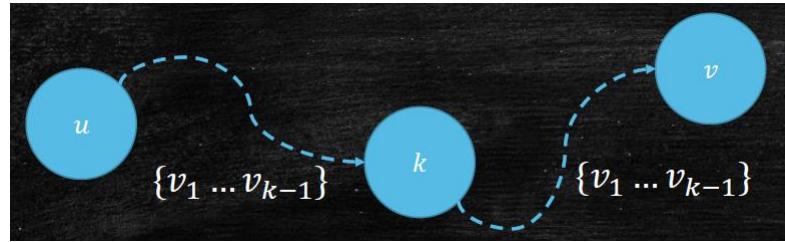
Floyd-Warshall Algorithm：换了一个子问题来定义

$dist[k, u, v]$ ：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  $\infty$

$dist[|V|, u, v]$ : 我们最终想要的结果。

- Solve  $dist[k, u, v]$  (give addition power  $k$  to all pairs)
  - Case 1: the shortest path do not go across  $k$ .
  - Case 2: the shortest path go across  $k$ .
  - $dist[k, u, v] = \min\{dist[k - 1, u, v], dist[k - 1, u, k] + dist[k - 1, k, v]\}$



这种方法也可以用于判断负环（如果更新了不止一次）。

$k - 1$	$v_1$	$v_2$	$v_3$	...	$v_{ V }$	$k$	$v_1$	$v_2$	$v_3$	...	$v_{ V }$
$v_1$							$v_1$				
$v_2$							$v_2$				
$v_3$							$v_3$				
$v_4$							$v_4$				
...							...				
$v_{ V }$							$v_{ V }$				

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  $dist[k - 1, u, v]$ ,  $dist[k - 1, u, k]$ ,  $dist[k - 1, k, v]$

Floyd-Warshall $O( V ^3)$											
<b>function</b> floyd_marshall( $G$ )											
$dist[0, u, v] = d(u, v)$ for all $(u, v) \in E$ , $dist[0, u, v] = \infty$ otherwise.											
<b>for</b> $k = 1$ to $ V $											
<b>for</b> $u = 1$ to $ V $											
<b>for</b> $v = 1$ to $ V $											
$dist[k, u, v] = \min\{dist[k - 1, u, v], dist[k - 1, u, k] + dist[k - 1, k, v]\}$											

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？

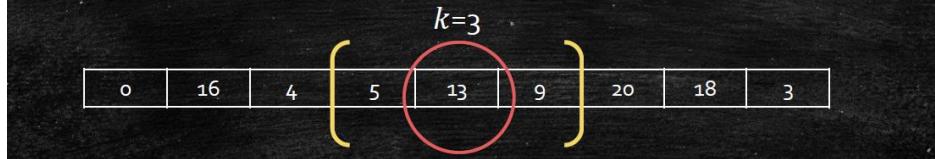
Floyd-Warshall											
<b>function</b> floyd_marshall( $G$ )											
$dist[u, v] = d(u, v)$ for all $(u, v) \in E$ , $dist[u, v] = \infty$ otherwise.											
<b>for</b> $k = 1$ to $ V $											
<b>for</b> $u = 1$ to $ V $											
<b>for</b> $v = 1$ to $ V $											
$dist[u, v] = \min\{dist[u, v], dist[u, k] + dist[k, v]\}$											

More Smarter Subproblem Definitions

## 单调队列

应用 1：求连续  $k$  个数中最大数

- **Input:** A sequence of numbers  $a_1, a_2, \dots, a_n$ , and a number  $k$ .
- **Output:** The largest number in every  $k$  consecutive numbers.



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有必要存在，直接弹出即可。

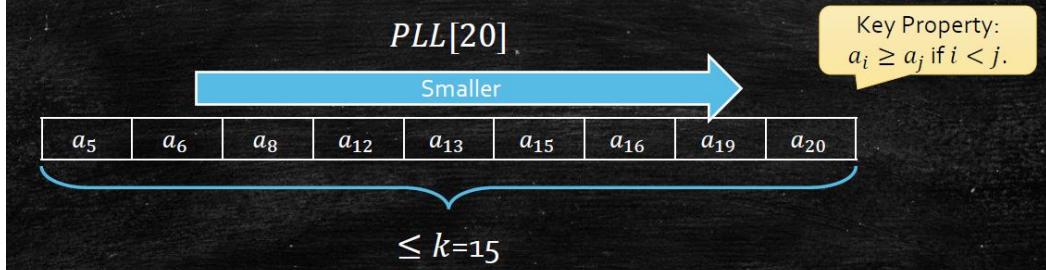
Ref: [11618. 【原 1618】 Interesting Queue](#)

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  $\text{large}[i]$  (从  $i-k+1$  到  $i$  的最大数) 而言  $\text{large}[i-1]$  的影响情况是什么？

关键想法：Potential Largest List(PLL)

- $\text{PLL}[i]$ : the Potential Largest List for  $a_{i-k+1} \sim a_i$ .
- At most  $k$  numbers.
- Sorted by the index.
- $i - k + 1 \leq \text{Index} \leq i$



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。

- Keep Inserting  $a_1 \sim a_k$  & kicking to make  $PLL[k]$ .
- Solve every  $PLL[k < i \leq n]$  by inserting & kicking.
- We can easily get  $large[i]$  by  $PLL[i]$ .
- It is efficient:  $O(n)!$  Each number at most:
  - Inserted once.
  - Kicked once.
  - **Pass once** (because once we pass, we kick it).

## 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)

- **Input:** A sequence  $a_1, a_2, \dots, a_n$ .
- **Output:** the Longest Increasing Subsequence (LIS)
  - $a_{i_1} < a_{i_2} < a_{i_3} \dots < a_{i_k}$
  - $i_1 < i_2 < i_3 \dots < i_k$

1	5	13	2	6	24	15	23	2	16
---	---	----	---	---	----	----	----	---	----

- $lis[i] = \max_{a_j < a_i, j < i} \{lis[j] + 1\}$
- **Definition: Potential Prefix**
  - The set of  $a_j$  that is possible to be the prefix of future numbers.

$a[i]$	1	5	13	2	6	24	15	23	2	16
$lis[i]$	1	2	3	2	3	-	-	-	-	-

It is not because  $a[i] > a[j]$  and  $lis[i] = lis[j]$

Who are the Potential Prefixes?

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）

▪  $Sm[i, len]$ : the **smallest ended number** for an increasing subsequence with **length**  $len$  by using  $a_1 \dots a_i$ .

▪ Remark: it is enough to record all **Potential Prefixes** (length and number).

	$len=0$	$len=1$	$len=2$	$len=3$	$len=4$	$len=5$	$len=6$	$len=7$	$len=8$	$len=9$
$sm[i, len]$	0	1	2	6	-	-	-	-	-	-

Case 1:  $a_i > sm[i - 1, len]$

- it can create a longer LIS.
- it can not update  $sm[i, len]$ .

Case 2:  $a_i \leq sm[i - 1, len]$

- It **must** update  $sm[i, len]$ .
- it can not create a longer LIS.

发现了吗？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  $\log n$  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

- Input:** A sequence of items with cost  $a_1, a_2, \dots, a_n$ .
- Need to Do:**
  - Manufacture these items.
  - Operation  $man(l, r)$ : manufacture the items from  $l$  to  $r$ .
  - $cost(l, r) = C + (\sum_{i=l}^r a_i)^2$ .
- Output:** The **minimum** cost to manufacture all items.

需要尽可能优化  $cost(l, r) = C + (a_1 + \dots + a_r)^2$ , 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：

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

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？

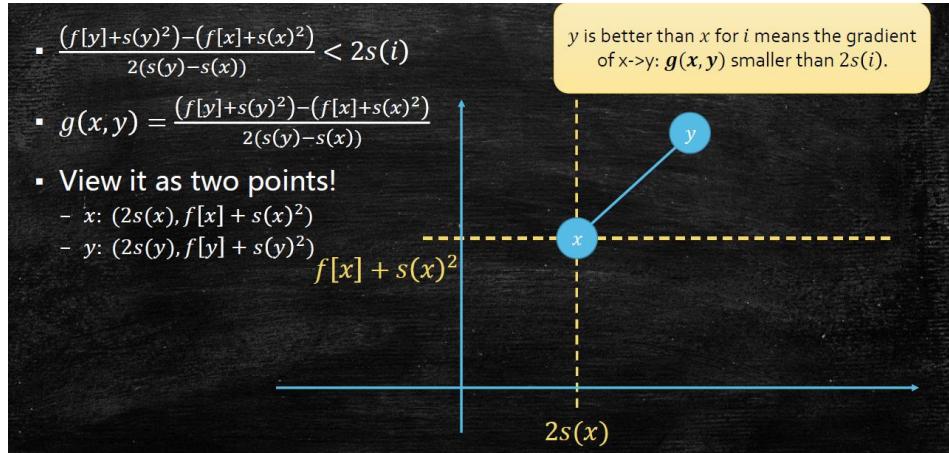
不妨从数学的角度分析一下这几个数据切分点的关系：

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

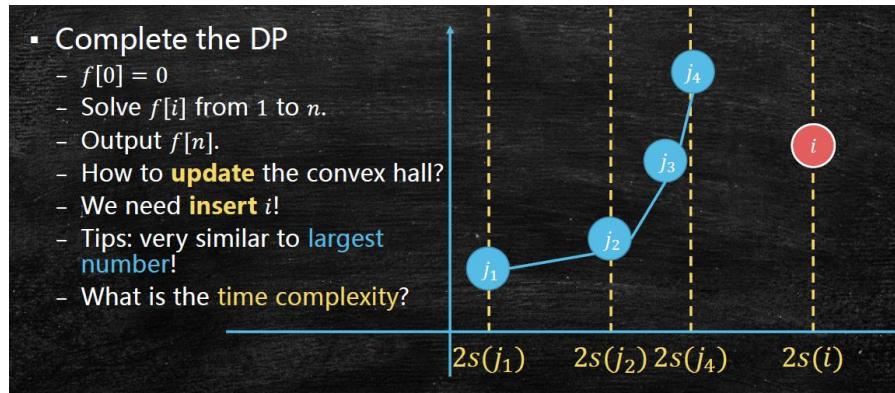
不妨令  $s(i) = \sum_{j=1}^i a_j$  那么移项后可以得到的是：

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

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



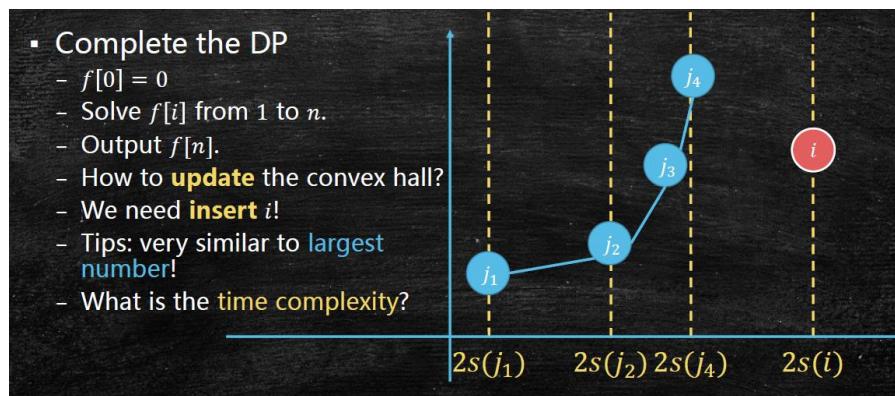
最后，本质上就是构造凸集，则确保这个凸集足够完美就可以了。



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

- Input:** A sequence of items with cost  $a_1, a_2, \dots, a_n$ .
- Need to Do:
  - Manufacture these items.
  - Operation  $\text{man}(l, r)$ : manufacture the items from  $l$  to  $r$ .
  - $\text{cost}(l, r) = C + (\sum_{i=l}^r a_i)^2$ .
- Output:** The minimum cost to manufacture all items.

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。



## Dynamic Programming(But not so efficient) 动态规划

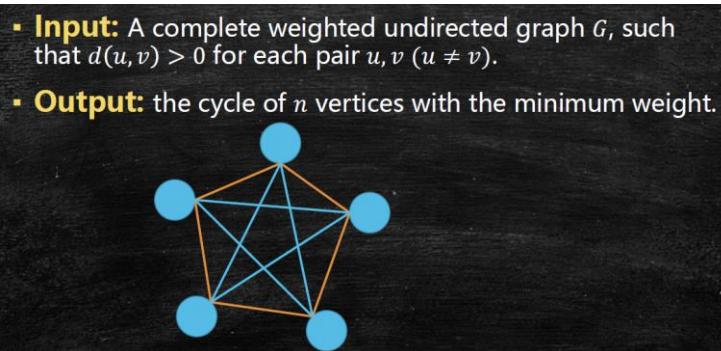
【REVIEW】对于 DP 问题的更简单的设计思路：

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

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。

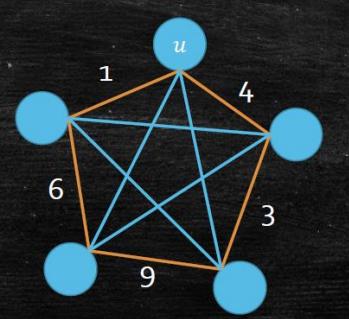
【如果暴力做，怎么办呢？】

- **TSP**
  - **Output:** the cycle of  $n$  vertices with the minimum weight.
- **All Pair Shortest Path**
  - **Output:** the minimum weight path from  $u$  to  $v$ .

- $f[k, u, v]$ 
  - The shortest path from  $u$  to  $v$ , with inter-vertex chosen in  $v_1 \dots v_k$ .
- What we should do now?
- We can directly try this subproblem!

- $f[k, u, v]$ 
  - The shortest path from  $u$  to  $v$  with inter-vertex **exactly**  $v_1 \dots v_k$  except  $u$  and  $v$ .

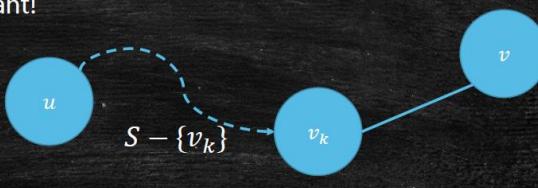
- How to solve TSP?
  - $\min_u f[|V|, u, u]$  is what we want!
- How to solve  $f[k, u, v]$ ?



计划 A 似乎不太行？计划 B：

- $f[\mathbf{S}, u, v]$ 
  - The shortest path from  $u$  to  $v$  with inter-vertex **exactly  $s \subset V$**  except  $u$  and  $v$ .
  - $\min_u f[\mathbf{V}, u, u]$  is what we want!

- How to solve  $f[\mathbf{S}, u, v]?$



- How to solve  $f[\mathbf{S}, u, v]?$
- $f[\mathbf{S}, u, v] = \min_{k \in V} f[S - \{k\}, u, k] + d(k, v)$

这题的正解应考虑状压 DP

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

- **Independent Set:** a set  $S$  of vertices:
  - $\forall u, v \in S$ , we have  $(u, v) \notin E$

- Start from the root
  - Case 1: We choose the root, what happens?
  - Case 2: We do not choose the root, what happens?

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  $O(2^{k^2} \cdot n)$

- $O(f(k) \cdot n^c)$
- $f(k)$  do not need to be polynomial.
- Many Optimization Problem in these graphs can use tree DP to get  $O(2^{O(k)} \cdot \text{poly}(n))$ !
- They are FPT in terms of the treewidth!
- Compare to Approximation Algorithms!

## //////////期中考 tips:

### 1. Divide and Conquer

回忆起来树状模型，大问题变成小问题进行求解。知道递归式满足形如：

$$T(n) = 2T\left(\frac{n}{2}\right) + O(\dots) \rightarrow T(n) = O(\dots) \rightarrow \text{Master Theorem}$$

子问题的数量一般就与最后的 cost 相关（一般拆下来之后最后的子问题规模是 n 个，每个 O(1)）  
合并的 cost 如果过大那么其实就没有优化的意义（如 merge\_sort 没有优化的情况下就是  $n^2$  的）。

- ①  $n^2$       ②  $n\log n$       ③ 更小的情况

怎么证明？要么展开找规律，要么 induction 归纳证明复杂度（Basic, Assumption k, Prove k+1）

**Note:  $T(n)=O(n)$ 最好改写为  $T(n)\leq B(n)$ 这样是比较安全不容易错的方式。**

How to merge? 分治法好不好关键在于 merge 快不快。    关键：额外 information 去合并

### 2. Graph

DFS 遍历图且不会重复走。**关键性质**: finish\_time=>拓扑序、强连通、环

**这三个应用比较重要，需要注意。**

最短路问题：BFS 方法、Dijkstra 算法=>Pick the closest and update.

Why:实际上当前最近的点已经不会再被更新了。

考察方式：shortest path like 问题（如作业中的最短路（乘法方式））

- Check Correctness.

Bellman-Ford: 关键：为什么 $|V|$ 次就能够找到最短路？与负环的关系？

=>与 Floyd 算法联系起来？

### 3. Greedy

单向图（逐层递进的类型）直接由 Big->Small->Smaller（与分治的层级递减不同）

**正确性检测：**

- 为什么这是对的，有无反例？
- 套路证明：检验每一步的贪心选择始终处于 OPT 中

- 假设我们在 OPT 里，那么在选择下一步的 greedy choice 我仍然在 OPT 内
- 或者我并不会把 OPT 排除在外（不会导致 OPT 变差）
- How to prove? Switch and replace.

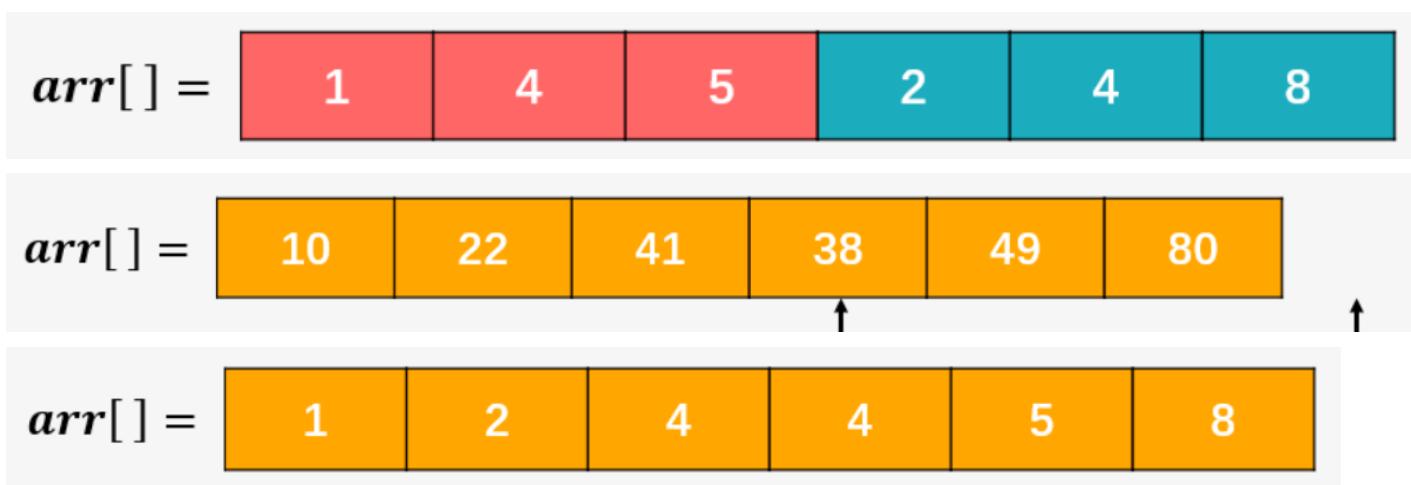
主定理<sup>2</sup> 如果对于常数  $a > 0, b > 1$  以及  $d \geq 0$ , 有  $T(n) = aT(\lceil n/b \rceil) + O(n^d)$  成立,

则

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

O(1)空间复杂度的归并排序:

$\text{arr}[i] = \text{arr}[i] + \text{arr}[j] * \text{maxval}$  来同时表示原始数组中的  $\text{arr}[i]$  和  $\text{arr}[j]$



- |   |   |
|---|---|
| (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) $T(n) = O(\log \log n)$                      |

```

template <typename Type>
void Partition(Type *array, int begin, int end, Type v, int &l, int &r)
{
    l = begin;
    for (int i=begin; i!=end; ++i)
    {
        if (array[i] < v)
            swap(array[i], array[l++]);
    }
    r = l;
    for (int j=l; j!=end; ++j)
    {
        if (array[j] == v)
            swap(array[j], array[r++]);
    }
}

```

### K 小数的 partition (3 buckets)

2.16 给定一个无穷数组  $A[\cdot]$ , 其中前  $n$  个元素都是整数, 且已排好顺序, 剩余元素均为  $\infty$ .  $n$  的值未知。给出一个算法, 以一个整数  $x$  为输入, 以  $O(\log n)$  时间找到数组中的一个位置, 并满足其上的元素为  $x$ (当然前提是如果这样的位置存在)。(如果您被该数组的长度无穷所困扰, 建议您可以假定它的长度为  $n$ , 只是您并不知道这个长度的实际值, 并且在您的算法实现代码中, 一旦数组元素  $A[i]$  满足  $i > n$ , 则您的代码应该返回错误信息  $\infty$ 。)

$$A[2^i] < x < A[2^{i+1}]$$

每次指数递增查找范围, 易知为  $\log n$  至多。

起始与  $A[1]$  比较, 后与  $A[2]$ , 后与  $A[4]$  等等等等。

快速排序的伪代码以及对于快速排序的复杂度证明。

```

template <typename Type>
void QuickSort(Type *array, int begin, int end)
{
    if (begin < end)
    {
        int l, r, v = array[begin];
        Partition(array, begin, end, v, l, r);
        QuickSort(array, begin, l);
        QuickSort(array, r, end);
    }
}

```

$$\begin{aligned} T(n) &= O(n) + \frac{2}{n} \sum_{i=1}^{n-1} T(i) \\ T(n-1) &= O(n-1) + \frac{2}{n-1} \sum_{i=1}^{n-2} T(i) \\ \Rightarrow \frac{n-1}{n} T(n-1) &= \frac{n-1}{n} O(n-1) + \frac{2}{n} \sum_{i=1}^{n-2} T(i) \\ \Rightarrow T(n) - \frac{n-1}{n} T(n-1) &= O(n) - O(n-1) + \frac{1}{n} O(n-1) + \frac{2}{n} T(n-1) \\ \Rightarrow T(n) &= \frac{n+1}{n} T(n-1) + O(1) \Rightarrow T(n) = O(1) \left( 1 + \frac{n+1}{n} + \frac{n+1}{n-1} + \frac{n+1}{n-2} + \dots \right) \\ \Rightarrow T(n) &= O(n \log n) \quad (\text{why? 注意 Ex. 1.5 的结论}) \end{aligned}$$

- 3.19. As in the previous problem, you are given a binary tree  $T = (V, E)$  with designated root node. In addition, there is an array  $x[\cdot]$  with a value for each node in  $V$ . Define a new array  $z[\cdot]$  as follows: for each  $u \in V$ ,

$z[u] =$  the maximum of the  $x$ -values associated with  $u$ 's descendants.

### Ex.3.19

首先将  $z[\cdot]$  初始化为 0, 然后进行 DFS。在遍历过程中, 若有节点出栈 (比如  $v$ ),

将当前栈顶的节点  $u$  (即为  $v$  的父亲) 的  $z$  值更新:  $z[u] = \max(z[u], x[v])$ 。

### Ex.3.20

这里需要 DFS 的栈也可以进行随机访问, 设这个 DFS 栈为  $stack$ , 在遍历过程中, 若有节点进栈 (比如  $v$ ), 则将它的  $l$  值更新为:

$$l(v) = l(stack.at(\max(stack.size() - l(v), 1)))$$

其中  $stack.at(i)$  操作即是取出在栈中位于第  $i$  位的顶点, 设栈底元素为第 1 位。

- 3.21. Give a linear-time algorithm to find an odd-length cycle in a directed graph. (Hint: First solve this problem under the assumption that the graph is strongly connected.)
- 3.22. Give an efficient algorithm which takes as input a directed graph  $G = (V, E)$ , and determines whether or not there is a vertex  $s \in V$  from which all other vertices are reachable.
- 3.23. Give an efficient algorithm that takes as input a directed acyclic graph  $G = (V, E)$ , and two vertices  $s, t \in V$ , and outputs the number of different directed paths from  $s$  to  $t$  in  $G$ .

### 3.21 染色, 3.22 强连通 supernode 合并 3.23 直接 DFS 或者拓扑序 (入度计算后) 后 DP

- 3.24. Give a linear-time algorithm for the following task.

*Input:* A directed acyclic graph  $G$

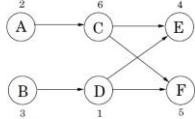
*Question:* Does  $G$  contain a directed path that touches every vertex exactly once?

定义过程  $Search(G)$ : 若  $G$  中只有一个顶点, 返回真。否则首先在图  $G$  中找到入度为 0 的顶点, 如果这样的顶点不只一个, 则返回假, 否则设这个唯一的顶点为  $s$ , 将  $s$  的所有邻接点的入度减 1, 并在  $G$  中除去顶点  $s$  后得到图  $G'$ , 然后在  $G'$  上递归地进行  $Search$  过程, 即返回  $Search(G')$ 。

- 3.25. You are given a directed graph in which each node  $u \in V$  has an associated *price*  $p_u$  which is a positive integer. Define the array  $\text{cost}$  as follows: for each  $u \in V$ ,

$\text{cost}[u] = \text{price of the cheapest node reachable from } u \text{ (including } u \text{ itself).}$

For instance, in the graph below (with prices shown for each vertex), the  $\text{cost}$  values of the nodes  $A, B, C, D, E, F$  are  $2, 1, 4, 1, 4, 5$ , respectively.



Your goal is to design an algorithm that fills in the *entire*  $\text{cost}$  array (i.e., for all vertices).

- (a) Give a linear-time algorithm that works for directed *acyclic* graphs. (*Hint:* Handle the vertices in a particular *order*.)  
 (b) Extend this to a linear-time algorithm that works for all directed graphs. (*Hint:* Recall the “two-tiered” structure of directed graphs.)

- a) 按拓扑顺序的逆序依次计算每个顶点，即设定待计算顶点的  $\text{cost}$  值为它自身的  $p$  值与它的所有邻接点的  $\text{cost}$  值中的最小值。  
 b) 将每个强连通分支作为一个顶点看待，并将它的  $p$  值设定为在该分支子图中所有顶点的  $p$  值中的最小值。所有的强连通分支即构成了一个  $\text{meta-graph}$ ，再将这个  $\text{meta-graph}$  按 a) 中的过程处理。

- 4.3. *Squares.* Design and analyze an algorithm that takes as input an undirected graph  $G = (V, E)$  and determines whether  $G$  contains a simple cycle (that is, a cycle which doesn’t intersect itself) of length four. Its running time should be at most  $O(|V|^3)$ .

You may assume that the input graph is represented either as an adjacency matrix or with adjacency lists, whichever makes your algorithm simpler.

邻接矩阵，找从  $u$  到  $v$  的两条边即可（利用三重循环。）

```

bool hasSimpleCyc4(Matrix edges[numVertices][numVertices])
{
    for (int i=0; i!=numVertices; ++i)
    {
        int flag[numVertices] = {0};
        for (int j=0; j!=numVertices; ++j)
        {
            if (j==i || edges[i][j] == 0) continue;
            for (int k=0; k!=numVertices; ++k)
            {
                if (k == j || k == i || edges[j][k] == 0) continue;
                if (flag[k] == 1)
                    return true;
                else flag[k] = 1;
            }
        }
    }
    return false;
}
  
```

- 4.5. Often there are multiple shortest paths between two nodes of a graph. Give a linear-time algorithm for the following task.

*Input:* Undirected graph  $G = (V, E)$  with unit edge lengths; nodes  $u, v \in V$ .

*Output:* The number of distinct shortest paths from  $u$  to  $v$ .

这题通过多个队列按照 BFS 层级的方式，首次遇到终点时记录下该层终点数目即可。

- 4.10. You are given a directed graph with (possibly negative) weighted edges, in which the shortest path between any two vertices is guaranteed to have at most  $k$  edges. Give an algorithm that finds the shortest path between two vertices  $u$  and  $v$  in  $O(k|E|)$  time.

- 4.11. Give an algorithm that takes as input a directed graph with positive edge lengths, and returns the length of the shortest cycle in the graph (if the graph is acyclic, it should say so). Your algorithm should take time at most  $O(|V|^3)$ .

#### Ex.4.10

类似于 Bellman-Ford 算法，以  $u$  为源点，每次 *update* 所有的边，循环  $k$  次即可。

#### Ex.4.11

依次以每个顶点  $s \in V$  为源点，利用 Dijkstra 算法寻找到其余顶点的最短路径，这样在  $O(|V||V|^2) = O(|V|^3)$  时间内就找到了任意两个顶点之间的最短路径。这时，

对于任意顶点对  $(u, v)$ ，有  $(u, v)$  之间的环的最短长度为  $\text{dist}(u, v) + \text{dist}(v, u)$ 。

于是枚举所有的边  $e(u, v)$ ，依次计算  $\text{dist}(u, v) + \text{dist}(v, u)$ ，取其中的最小值就得到了图  $G$  中最短环的长度，如果这个最小值是  $\infty$ ，则说明图  $G$  中不存在环。

先  $V^3$  再遍历  $E$

- 4.13. You are given a set of cities, along with the pattern of highways between them, in the form of an undirected graph  $G = (V, E)$ . Each stretch of highway  $e \in E$  connects two of the cities, and you know its length in miles,  $l_e$ . You want to get from city  $s$  to city  $t$ . There's one problem: your car can only hold enough gas to cover  $L$  miles. There are gas stations in each city, but not between cities. Therefore, you can only take a route if every one of its edges has length  $l_e \leq L$ .

- (a) Given the limitation on your car's fuel tank capacity, show how to determine in linear time whether there is a feasible route from  $s$  to  $t$ .
- (b) You are now planning to buy a new car, and you want to know the minimum fuel tank capacity that is needed to travel from  $s$  to  $t$ . Give an  $O((|V| + |E|) \log |V|)$  algorithm to determine this.

a) 以  $s$  为起点 DFS 或 BFS 即可，遇到长度大于  $L$  的边则当作断路处理。  
 b) 即是求从  $s$  到  $t$  的所有路径的最长边的最小值，同样可以利用 Dijkstra 算法来完成。只是需要修改 update 过程如下：

```
for all edges  $(u, v) \in E$  :
  if  $\text{fuel}(v) > \max(\text{fuel}(u), l(u, v))$ 
     $\text{fuel}(v) = \max(\text{fuel}(u), l(u, v))$ 
  ...
  ...
```

这个算法的正确性依赖于下面这个事实，如果路径  $s \rightarrow \dots \rightarrow r \rightarrow t$  所需的最小储油量为  $l$ ，则路径  $s \rightarrow \dots \rightarrow r$  所需的最小储油量定然小于等于  $l$ ，这使得

得 Dijkstra 算法总是能按照正确的顺序依次找到从源点  $s$  到其余所有顶点的路径的最长边的最小长度，也即是所需的最小储油量。

- 4.14. You are given a strongly connected directed graph  $G = (V, E)$  with positive edge weights along with a particular node  $v_0 \in V$ . Give an efficient algorithm for finding shortest paths between all pairs of nodes, with the one restriction that these paths must all pass through  $v_0$ .

依次以每个顶点  $s \in V$  为源点，使用 Dijkstra 算法寻找最短路径，可找到所有顶点到  $v_0$  的最短路径，以及  $v_0$  到其余所有顶点的最短路径。则对于顶点对  $(u, v)$ ，它们之间经过点  $v_0$  的最短路径长度为  $\text{dist}(u, v_0) + \text{dist}(v_0, v)$ 。

- 4.20. There is a network of roads  $G = (V, E)$  connecting a set of cities  $V$ . Each road in  $E$  has an associated length  $l_e$ . There is a proposal to add one new road to this network, and there is a list  $E'$  of pairs of cities between which the new road can be built. Each such potential road  $e' \in E'$  has an associated length. As a designer for the public works department you are asked to determine the road  $e' \in E'$  whose addition to the existing network  $G$  would result in the maximum decrease in the driving distance between two fixed cities  $s$  and  $t$  in the network. Give an efficient algorithm for solving this problem.

为方便表示，设 Dijkstra 算法的时间复杂度为  $\text{Dijkstra}(G)$ 。首先与  $s$  为源点，计算到其余所有点的最短路。遍历边集  $E'$ ，对于任意边  $e'(u, v) \in E'$ ，如果  $\text{dist}(v, t)$  未计算过，则以  $v$  为源点寻找到顶点  $t$  的最短路。设  $dec$  为图  $G$  加上边  $e'$  后  $\text{dist}(s, t)$  的缩减值，则如果  $\text{dist}(s, u) + l(u, v) + \text{dist}(v, t) \geq \text{dist}(s, t)$ ， $dec = 0$ ，否则  $dec = \text{dist}(s, t) - (\text{dist}(s, u) + l(u, v) + \text{dist}(v, t))$ ，找出令  $dec$  最大的边  $e'$  即可，时间复杂度为  $O(\text{Dijkstra}(G) \cdot \max(|V|, |E'|))$ 。

## 离散数学：每条边的权重均不相同的带权图有唯一最小生成树

假设存在两个最小生成树  $T, T'$ ，其边按权重升序排列分别为  $\{e1, e2, \dots, en\}$  和  $\{e1', e2', \dots, en'\}$ 。

那么存在一个最小的  $k$  使得  $\text{weight}(ek) \neq \text{weight}(ek')$ 。（也即  $e1=e1', e2=e2', \dots, ek-1=ek-1'$ ）

此时  $T'$  中没有  $ek$ 。不妨设  $w(ek) < w(ek')$ ，则  $T'+ek$  里必然会有环，而且这个环有除了  $\{e1', e2', \dots, en'\}$  之外的边（否则在  $T$  中就会有这样的环）。删去任一这样的边，即可得到一个更小的生成树，这与  $T'$  是最小生成树矛盾。

由上，题设得证。

5.10. Let  $T$  be an MST of graph  $G$ . Given a connected subgraph  $H$  of  $G$ , show that  $T \cap H$  is contained in some MST of  $H$ .

由图  $G$  得到  $H$  的过程可以看成是由一系列对边的删除操作组成（如果被删除边与某个度数为 1 的顶点相连，则同时也将该顶点删除）。于是现在只需要证明，若在  $mst_G - e \in mst_{G'}$ 。若  $e$  属于  $mst_G$ ，且  $e$  所连结的两个顶点的度均大于 1，那么图  $G$  删除一条边  $e$  后得到图  $G'$ ，有  $mst_G - e \in mst_{G'}$  即可。下面分三种情况讨论：

若  $e$  不属于  $mst_G$ ，则最小生成树不变，显然满足  $mst_G - e \in mst_{G'}$ 。若  $e$  属于  $mst_G$ ，且  $e$  与  $G$  中的某个度为 1 的顶点相连，则  $mst_G - e$  即为图  $G'$  的最小生成树，同样在  $e$  所在的环中找出不属于  $mst_G$  的最小权值边  $c$ ，则  $mst_G - e + c$  即为图  $G'$  的最小生成树，这样依然有  $mst_G - e \in mst_{G'}$ ，于是得证。

5.22. You are given a graph  $G = (V, E)$  with positive edge weights, and a minimum spanning tree  $T = (V, E')$  with respect to these weights; you may assume  $G$  and  $T$  are given as adjacency lists. Now suppose the weight of a particular edge  $e \in E$  is modified from  $w(e)$  to a new value  $\bar{w}(e)$ . You wish to quickly update the minimum spanning tree  $T$  to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each case give a linear-time algorithm for updating the tree.

- (a)  $e \notin E'$  and  $\bar{w}(e) > w(e)$ .
- (b)  $e \notin E'$  and  $\bar{w}(e) < w(e)$ .
- (c)  $e \in E'$  and  $\bar{w}(e) < w(e)$ .
- (d)  $e \in E'$  and  $\bar{w}(e) > w(e)$ .

- a) 设这个环的顶点集为  $C$ ，最大权边为  $e(u, v)$ 。设在某一时刻， $C$  被 cut 成两个部分  $s_1, s_2$ ，其中  $s_1$  包含  $u$ ，而  $s_2$  包含  $v$ 。因  $C$  是一个环，所以连结  $s_1, s_2$  的边至少有两条，其中至少有一条不为  $e$  的边  $e'$ ，其权不大于  $e$ ，由 cut property 可知，存在不含有  $e$  的最小生成树。
- b) 因为每次删除的都是图  $G$  中权最大的环边，由题述中的性质可知。
- c) 参考 Ex.3.11。
- d)  $O(|E|^2)$ 。

5.24. A binary counter of unspecified length supports two operations: `increment` (which increases its value by one) and `reset` (which sets its value back to zero). Show that, starting from an initially zero counter, any sequence of  $n$  `increment` and `reset` operations takes time  $O(n)$ ; that is, the amortized time per operation is  $O(1)$ .

5.25. Here's a problem that occurs in automatic program analysis. For a set of variables  $x_1, \dots, x_n$ , you are given some *equality* constraints, of the form " $x_i = x_j$ " and some *disequality* constraints, of the form " $x_i \neq x_j$ ". Is it possible to satisfy all of them?

For instance, the constraints

$$x_1 = x_2, x_2 = x_3, x_3 = x_4, x_1 \neq x_4$$

cannot be satisfied. Give an efficient algorithm that takes as input  $m$  constraints over  $n$  variables and decides whether the constraints can be satisfied.

### Ex.5.23

在图  $G$  中，将  $U$  删掉后，得到子图  $G'$ ，求出  $G'$  的最小生成树  $T'$ （如果  $G'$  并非是连通的，则图  $G$  不存在  $U$  全部是叶子节点的最小生成树）。对于  $U$  中的每个顶点  $v$ ，遍历该顶点的所有邻边，找到与  $T'$  相连结（利用并查集）且权值最小的一条边  $e$ ，将  $e$  加入到  $T'$  中。在按上述过程处理完  $U$  中的所有顶点后， $T'$  即为所求。

### Ex.5.24

5.27. Alice wants to throw a party and is deciding whom to call. She has  $n$  people to choose from, and she has made up a list of which pairs of these people know each other. She wants to pick as many people as possible, subject to two constraints: at the party, each person should have at least five other people whom they know *and* five other people whom they don't know.

Give an efficient algorithm that takes as input the list of  $n$  people and the list of pairs who know each other and outputs the best choice of party invitees. Give the running time in terms of  $n$ .

利用给定的关系建立无向图  $G(V, E)$ ，然后求出每个顶点的度。遍历所有顶点：

若有顶点  $v$  的度满足关系  $d(v) < 5$ ，显然  $v$  就不能来参加 party 了，将  $v$  加入到一个待删除队列中。若  $d(v) > |V| - 1 - 5$ ，那么  $v$  也不可能来参加 party，因为即使将  $v$  的某个邻结点排除在邀请名单之外，这使得  $d(v)$  减小了 1，但是同时  $|V|$  也减小

了 1，所以依然保持  $d(v) > |V| - 1 - 5$ 。所以，也将  $v$  加入到待删除队列中。遍历完成之后，将待删除队列中的所有顶点删除，同时更新这些顶点的邻结点的度。然后在剩余的子图上重复这个遍历过程，直到某一轮遍历完成后，待删除队列为空为止。因为  $|E| = O(|V|^2)$ ，所以整个算法的时间复杂度为  $O(|V|^2 + |E|) = O(n^2)$ 。