

## Fall 2021 Pro: SS

Chaoyu Li

Update: 12/03/2021

### 1. Stable Matching (稳定匹配)

Problem:

N个男生和n个女生需要两两配对，其中每个男生只能和一个女生配对，女生亦然。每个男生都对不同的女生有不同的好感度，女生亦然。问如何才能让所有男生都找到女生配对且每个配对都是稳定的。

完美配对：每个男女都有唯一的配对对象。

稳定配对：一个配对中的男生或女生都没有潜在的出轨对象。（此问题中出轨仅发生在一个男生和一个女生对彼此的好感度都大于现任配对对象时。）

正当配偶(valid partner)：一个稳定配对中的女生称为男生的正当配偶。

最佳正当配偶：在一个稳定匹配中，对一个人n来说没有比目前对象m排名更靠前的人可以和n组成正当配偶，则(n,m)就是最佳正当配偶。

Example of 不稳定配对（参考《Algorithm Design》）

	favorite ↓	least favorite ↓	
	1 <sup>st</sup>	2 <sup>nd</sup>	3 <sup>rd</sup>
Xavier	Amy	Bertha	Clare
Yancey	Bertha	Amy	Clare
Zeus	Amy	Bertha	Clare

*Men's Preference Profile*

	favorite ↓	least favorite ↓	
	1 <sup>st</sup>	2 <sup>nd</sup>	3 <sup>rd</sup>
Amy	Yancey	Xavier	Zeus
Bertha	Xavier	Yancey	Zeus
Clare	Xavier	Yancey	Zeus

*Women's Preference Profile*

可以看出在这个配对关系中Xavier和Amy各自的配对就都是不稳定的，因为跟他们配对的对象在他们的喜好顺序中都不如彼此，因为存在出轨可能。

Solution: Gale-Shapley算法

让每个男生按照对不同女生的好感度 **降序** 排序，同时女生也按照对不同男生的好感度 **降序** 排序。然后男生按1-n的顺序向当前自己列表中最喜欢的女生表白。

1. 如果女生在收到男生告白的时候没有对象，则女生必须接受男生成为一个配对。
2. 如果女生在收到男生告白的时候已经**有对象了**，则女生**比较现任和告白者自己更喜欢哪个**。
  - 1) 如果女生**更喜欢现任**，则**拒绝告白者**。
  - 2) 如果女生**更喜欢告白者**，则**甩掉现任，和告白者组成新的配对**。**现任重回自由**要重新**在下一轮选择新的次喜欢的女生告白**。

由此可知，在最差情况下n^2轮以后所有男生和女生必定组成稳定的配对。

Example如下(参考<https://www.cnblogs.com/jielongAI/p/9463029.html>)

	1st	2nd	3rd
男-1	女-1	女-2	女-3
男-2	女-2	女-1	女-3
男-3	女-1	女-2	女-3

	1st	2nd	3rd
女-1	男-3	男-1	男-3
女-2	男-1	男-2	男-3
女-3	男-1	男-2	男-3

图 1

Step - 1: 男-1-->女-1，此时女-1接受男-1，因为此前没有更稳定的匹配关系。

	1st	2nd	3rd
男-1	女-1	女-2	女-3
男-2	女-2	女-1	女-3
男-3	女-1	女-2	女-3

	1st	2nd	3rd
女-1	男-3	男-1	男-2
女-2	男-1	男-2	男-3
女-3	男-3	男-2	男-1

Step - 2: 男-2-->女-2，此时女-2接受男-2，因为此前没有更稳定的匹配关系。

	1st	2nd	3rd
男-1	女-1	女-2	女-3
男-2	女-2	女-1	女-3
男-3	女-1	女-2	女-3

	1st	2nd	3rd
女-1	男-3	男-1	男-2
女-2	男-1	男-2	男-3
女-3	男-3	男-2	男-1

Step - 3: 男-3-->女-1，此时女-1接受男-3，并且抛弃男-1，因为男-3排序更靠前。

	1st	2nd	3rd
男-1	女-1	女-2	女-3
男-2	女-2	女-1	女-3
男-3	女-1	女-2	女-3

	1st	2nd	3rd
女-1	男-3	男-1	男-2
女-2	男-1	男-2	男-3
女-3	男-3	男-2	男-1

Step - 4: 男-1-->女-2，此时女-2接受男-1，并且抛弃男-2，因为男-1排序更靠前。

	1st	2nd	3rd
男-1	女-1	女-2	女-3
男-2	女-2	女-1	女-3
男-3	女-1	女-2	女-3

	1st	2nd	3rd
女-1	男-3	男-1	男-2
女-2	男-1	男-2	男-3
女-3	男-3	男-2	男-1

Step - 5: 男-2 --> 女-1, 此时女-1不接受男-2。

	1st	2nd	3rd
男-1	女-1	女-2	女-3
男-2	女-2	女-1	女-3
男-3	女-1	女-2	女-3

	1st	2nd	3rd
女-1	男-3	男-1	男-2
女-2	男-1	男-2	男-3
女-3	男-3	男-2	男-1

Step - 6: 男-2 --> 女-3, 此时女-3接受男-2, 完成。

	1st	2nd	3rd
男-1	女-1	女-2	女-3
男-2	女-2	女-1	女-3
男-3	女-1	女-2	女-3

	1st	2nd	3rd
女-1	男-3	男-1	男-2
女-2	男-1	男-2	男-3
女-3	男-3	男-2	男-1

**Proof:**

1. 有穷性: 算法最多在  $n^2$  轮之后结束。因为每一轮男生都只会向一个女生求婚，一定至少有一个女生配对成功。

2. 完美性: 所有男生女生都能有人配对。

证明 :

- 1) 假设有一个男生到最后还没配对成功。
- 2) 那么一定有一个女生也没有配对成功。
- 3) 根据Solution, 单身的女生一定会同意向她告白的男生。
- 4) 所以没有配对成功的女生一定还没有被告白过。
- 5) 根据Solution, 男生整个算法过程中如果一直被拒绝，最终会对所有女生告白完才结束。
- 6) 所以该男生一定向所有女生告白过，发现4和5冲突了。

假设不成立，证明结束。

3. 稳定性: 所有的配对都是稳定配对。

证明 :

- 1) 假设有两个匹配  $(ni, mi), (nj, mj)$  是不稳定配对，其中  $ni, mj$  可能出轨。
- 2) 那么有对于  $ni$ , 有  $mj > mi$ , 对于  $mj$ , 有  $ni > nj$ 。
- 3) 根据Solution, 由于  $ni$  最后和  $mi$  配对，因此  $ni$  一定已经和  $mj$  表白过并且被拒绝了。
- 4) 又根据Solution,  $ni$  被拒绝一定是因为他在向  $mj$  表白时  $mj$  已经有比他更好的配对对象了。
- 5) 由2)可知, 如果  $ni$  都会被  $mj$  拒绝, 那  $nj$  向  $mj$  表白时也一定会被拒绝，与1)中假设矛盾。

假设不成立，证明结束。

#### 4. G-S算法是一个男生有利、女生不利的算法。

感性的理解：表面上看上去每一轮男生都有可能被比自己更好的男生取代，女生都可以根据自己的喜好将对象替换成自己更喜欢的。然而，实际上在整个循环结束之后，男生的稳定配对对象永远都是自己能力范围内能匹配到的最优女生，而女生的稳定配对对象虽然一定都是向自己表白的男生中最喜欢的，但男生对自己的满意程度很低。

理性的证明：

##### 1. 证明G-S算法是一个男性最优算法：

1) 假设到达稳定匹配后，仍存在一个男生 $n_i$ 匹配到的对象不是最佳正当配偶。即存在当前稳定匹配集合 $S_1$ 中的 $n_i$ 比起自己的对象 $m_i$ 喜欢 $n_j$ 的对象 $m_j$ ，且他和 $m_j$ 可以组成新的稳定配对 $(n_i, m_j)$ 。

2) 那么根据Solution, 一定有对于 $n_i:m_j > m_i$ ; 对于 $m_j:n_j > n_i$ ， $n_i$ 一定被 $m_j$ 拒绝过或是替换掉了，并且由于 $m_j$ 是 $n_i$ 的最喜欢的对象，因此这一定是 $n_i$ 第一次被拒绝。

3) 假设还有一个新的稳定匹配集合 $S_2$ ，在该集合中 $n_i$ 和 $m_j$ 配对，且此时 $n_j$ 和一个新的对象 $m_k$ 配对了。

4) 由于在 $S_1$ 中 $n_j$ 和 $m_j$ 配对，说明在 $n_i$ 被 $m_j$ 拒绝或替换成 $n_j$ 之前， $n_j$ 都没有被拒绝过，因此对于 $n_j$ 一定有 $m_j > m_k$ 。

5) 综合2), 4)，对于 $m_j$ 有 $n_j > n_i$ ；对于 $n_j$ 有 $m_j > m_k$ 。那么在 $S_2$ 中的 $(n_i, m_j)$ ,  $(n_j, m_k)$ 就是不稳定的，因为存在 $(n_j, m_j)$ 这个不稳定对。与3)的假设矛盾，证明完毕。

##### 2. 证明G-S算法是一个女性最劣算法：

1) 假设达到稳定匹配 $S_1$ 后仍有一个女生 $m_i$ 匹配到的对象不是最差正当配偶。即存在一个男生 $n_j$ 比 $m_i$ 现在的对象 $n_i$ 还糟糕，且他们可以组成新的稳定配对 $(n_j, m_i)$ 。

2) 假设这个新的稳定匹配集合为 $S_2$ ，在该集合中 $n_j$ 和 $m_i$ 配对了，且此时 $n_i$ 和一个新的对象 $m_k$ 配对了。

3) 根据男性最优原则，对于 $n_i$ 一定有 $m_i > m_k$ 。

4) 此时，在 $S_2$ 中存在 $n_j:m_i > m_k$ ,  $n_i:m_k > m_i$ ; 且 $n_i:m_i > m_k$ ,  $m_i:n_i > n_j$ 。因此 $n_i$ 和 $m_i$ 可能出轨， $S_2$ 是不稳定的，与2)矛盾，证明完毕。

#### 5. G-S算法的解具有唯一性(一定是男生最佳女生最差)

**Conclusion:**

**Stable Matching**问题中谁主动谁有利。

## 2. 二分图基本定理

1. 定义：一张无向图如果可以分成两个点集 **A, B** 满足所有边都是一端连着**A**中的点另一端连着**B**中的点，就是二分图。
2. 定理1：二分图中不可能出现奇数环（可以出现偶数环！）
3. 定理2：判定一个无向图是二分图：点黑白染色，邻点不同色。从任意一点开始 BFS 或 DFS，起点不妨染色为白。当前在  $u$  点时，尝试将所有邻点染为不同的颜色，复杂度  $O(m+n)$ 。接着每条边遍历一遍，如果邻点已经染色且颜色不符，则不是二分图，复杂度  $O(m)$ 。总复杂度  $O(m+n)+O(m)=O(m+n)$

## 3. 渐进阶

四种记号：**O**、 **$\Omega$** 、 **$\Theta$** 和 **$o$** ：

- $f(n) = O(g(n))$ :  $\exists c > 0, n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \leq cg(n)$ ;  $f$  的阶不高于  $g$  的阶。
- $f(n) = \Omega(g(n))$ :  $\exists c > 0, n_0 \in \mathbb{N}, \forall n \geq n_0, f(n) \geq cg(n)$ ;  $f$  的阶不低于  $g$  的阶。
- $f(n) = \Theta(g(n))$ :  $\iff f(n) = O(g(n)) \& f(n) = \Omega(g(n))$ ;  $f$  的阶等于  $g$  的阶。
- $f(n) = o(g(n))$ :  $\forall \varepsilon > 0, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, f(n)/g(n) < \varepsilon$ ;  $f$  的阶低于  $g$  的阶。

可见，记号 **O** 给出了函数  $f(n)$  在渐进意义下的上界（但不一定是最小的）。相反，记号  **$\Omega$**  给出的是下界（不一定是最大的）。如果上界与下界相同，表示  $f(n)$  和  $g(n)$  在渐进意义下是同阶的( **$\Theta$** )，亦即复杂度一样。

列举一些常见的函数之间的渐进阶的关系：

- $\log n! = \Theta(n \log n)$
- $\log n^2 = \Theta(\log n)$
- $\log n^2 = O(\sqrt{n})$
- $n = \Omega(\log^2 n)$
- $\log^2 n = \Omega(\log n)$
- $2^n = \Omega(n^2)$
- $2^n = O(3^n)$
- $n! = o(n^n)$
- $2^n = o(n!)$

即使问题的规模相同，随着输入数据本身属性的不同，算法的处理时间也可能会不同。于是就有了最坏情况、最好情况和平均情况下算法复杂度的区别。它们从不同的角度反映了算法的效率，各有用处，也各有局限。

有时候也可以利用最坏情况、最好情况下算法复杂度来粗略地估计算法的性能。  
比如某个算法在最坏情况下时间复杂度为 $\theta(n^2)$ , 最好情况下为 $\theta(n)$  , 那这个算法的复杂度一定是 $O(n^2)$ 、 $\Omega(n)$ 的。也就是说 $n^2$ 是该算法复杂度的上界 ,  $n$ 是其下界。

#### 4. BFS, DFS和拓扑排序

1. **DFS**: 寻找一张图中的一条路径 ( 深度优先 )
2. **BFS**: 寻找一张图中距离为x的所有点 ( 广度优先 )
3. 判断一张n个点m条边的有向图是强连通图 :
  - 1) 暴力算法: 每个点跑一次BFS/DFS判断能不能走到所有点。复杂度 $O(n(m+n))$ 。
  - 2) 反转法 :
    1. **Step1**: 选定一个初始点s , 从s跑一遍BFS/DFS判断能不能走到所有点。复杂度 $O(m+n)$ 。
    2. **Step2**: 构建一张新的图G' , G'和原图G的关系是所有的边反向。复杂度 $O(m+n)$
    3. **Step3**: 重新从s出发跑一遍BFS/DFS判断能不能走到所有点 , 如果可以则原图G是一张强连通图。复杂度 $O(m+n)$ 。
  4. **正确性证明** 对于有向图来说 , 如果从任意一个顶点v开始DFS/BFS可以到达所有其他顶点 , 然后把有向图的边反向 , 在从顶点v开始DFS/BFS可以再次到达所有其他顶点 , **那么就意味着在原图中所有其他顶点都可以到达顶点v**。而如果v可以到达所有其他顶点 , 所有其他顶点又可以到达v,那么其他的顶点至少可以通过一条包含v的路径来到达除自己外的顶点。

对于上文中红色粗体的解释 ( 下文中**highlight**部分 ) **如果顶点u可以到达v , 那么如果反转u和v之间的路径的边缘 , 则现在v可以等效地到达u , 并且如果在原始点中没有从u到v的路径 , 则在转置v和u'之间没有路径。**

From the definition of strongly connected components : if every vertex is reachable from every other vertex.

The first DFS is to find all the vertices that are reachable from root vertex v. The second DFS is to check the reverse , i.e to find the subset(of all the above vertices) that can reach v.

Now this may sound a bit confusing, but the question 'whether any vertex u has a path to v', can be converted to an equivalent question on the transpose graph, 'whether v has a path to u'. Further detailing,'if a vertex u can reach v, then if you reverse the edges in the path between u and v, now equivalently v can reach u, and if there is no path from u to v in the original, then in the transpose there is no path between v and u'.

So instead of testing each vertex u ( which are reachable from v) and can reach v back, the second DFS on the transpose equivalently tests, if v can reach all u. The algorithm then returns a component, which has vertices that are both in the first DFS and the second one.

#### 4. 判断一个图中是否存在环

##### 1) 无向图：

- 方法1：选定一个点  $s$ ，从  $s$  开始跑DFS，一开始所有的点都是0，被访问过的点就设为1，这个点的所有邻接点都访问过就设为2。如果在DFS的过程中访问到一个是1且不是自己父亲的节点，图中就存在环。否则整个DFS完成以后再选一个还没被访问过的点重新进行上述过程（有可能存在森林）。每个点最多只被访问一次，每条边也最多只被访问一次，所以复杂度是  $O(m+n)$ 。用邻接矩阵的话建立矩阵需要  $O(n^2)$ ，所以总复杂度是  $O(n^2)$ 。
- 方法2：通过度数判断是否存在环（类似拓扑排序）。找出所有度数  $\leq 1$  的节点，删掉它们以及和它们连接的边，边的另一端的点的度数减1。重复上述操作直到无法选点。如果此时图中还有点说明图中有环。复杂度也是  $O(m+n)$ 。

##### 2) 有向图：

- 方法1：跟无向图的方法1完全一样。
- 方法2：拓扑排序。和无向图的方法2几乎一样，只是每次要找的是度数为0的点。

#### 5. 拓扑排序的实际应用。

##### 1) 拓扑排序求最长路径。

- 问题简介：求一个带权DAG中从  $s$  出发能到达的最长路径（权值之和最大）
- 方法：类似 Dij，首先求出整个图的一个拓扑排序（拓扑排序不是唯一的）。然后按照拓扑排序的顺序对点  $v$  进行逆松弛操作： $\text{PATH}(s,w) < \text{PATH}(s,v) + \text{PATH}(v,w)$ ，则更新  $\text{PATH}(s,w) = \text{PATH}(s,v) + \text{PATH}(v,w)$ 。复杂度  $O(m+n)$ 。

##### 2) 优先级限制下的并行调度问题。

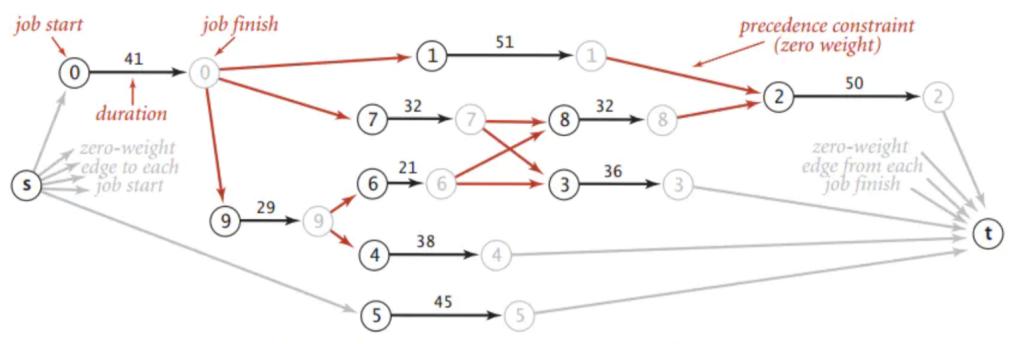
- 问题简介：给定一组需要完成的任务和每个任务所需要的时间，以及一组关于任务完成的先后次序的优先级限制。在满足限制条件的前提下，该如何在若干相同处理器上安排任务并在最短的时间内完成所有任务？
- 方法：首先，将问题转化为一幅加权有向无环图，然后利用基于拓扑排序的最长路径算法求解。

- 1) 对于有  $V$  个任务的优先级调度问题，创建  $2*V+2$  个顶点（1个起点  $s$ ，1个终点  $t$ ，每个任务2个顶点  $v$  和  $v'$ ）；
- 2) 每个任务添加一条从  $v \rightarrow v'$  的边，权重为任务所需时间；
- 3) 对于每条优先级限制  $v \rightarrow w$ ，添加一条从  $v$  的结束顶点  $v'$  到  $w$  的起始顶点的权重为0的边，即  $v' \rightarrow w$ ；
- 4) 每个任务  $v$  还要添加： $s \rightarrow v$  的权重为0的边、 $v' \rightarrow t$  的权重为0的边。

经过上述处理后，每个任务  $v$  的开始时间就是从起点  $s$  到  $v$  的起始顶点的最长路径。

job	duration	must complete before		
0	41.0	1	7	9
1	51.0	2		
2	50.0			
3	36.0			
4	38.0			
5	45.0			
6	21.0	3	8	
7	32.0	3	8	
8	32.0	2		
9	29.0	4	6	

A job-scheduling problem



## 5. 贪心证明正确性三部曲

1. 设一个最优算法O，证明贪心算法的第一步一定不比最优算法差。
2. 数学归纳法。先证明n=1的情况是正确的，推到n=k的情况正确时，n=k+1也正确。即贪心算法和O在前k步都不差的情况下第k+1步一定不会更差。
3. 反证法（交换论证）。

步骤：

Step0: 给出贪心算法A的描述

Step1: 假设O是和A最相似(假设O和A的前k个步骤都相同，第k+1个开始不同，通常这个临界的元素最重要)的最优算法

Step2: [Key] 修改算法O(用Exchange Argument, 交换A和O中的一个元素)，得到新的算法O'

Step3: 证明O'是feasible的，也就是O'是对的

Step4: 证明O'至少和O一样，即O'也是最优的

Step5: 得到矛盾，因为O'比O更和A相似。

证毕。

当然上面的步骤还有一个变种，如下：

Step0: 给出贪心算法A的描述

Step1: 假设O是一个最优算法（随便选，arbitrary）

Step2: 找出O和A中的一个不同。（当然这里面的不同可以是一个元素在O不再A，或者是一个pair的顺序在A的和在O的不一样。这个要根据具体题目）

Step3: Exchange这个不同的东西，然后argue现在得到的算法O'不必O差。

Step4: Argue 这样的不同一共有Polynomial个，然后我exchange Polynomial次就可以消除所有的不同，同时保证了算法的质量不比O差。这也就是说A是as good as一个O的。因为O是arbitrary选的，所以A是optimal的。

具体证明样例可以参考下面最小延迟调度问题。

## 6. 最小延迟调度问题 ( 贪心 )

### 一、最小延迟调度问题

给定等待服务的客户集合  $A = \{1, 2, \dots, n\}$ , 预计对客户  $i$  的服务时间是  $t_i$ , 该客户希望的完成时间是  $d_i$ , 即  $T = \{t_1, t_2, \dots, t_n\}, D = \{d_1, d_2, \dots, d_n\}$ . 如果对客户  $i$  的服务在  $d_i$  之前结束, 那么对客户  $i$  的服务没有延迟; 如果在  $d_i$  之后结束, 那么这个服务就被延迟了, 延迟的时间等于该服务结束的时间减去  $d_i$ . 假设  $t_i, d_i$  都是正整数, 一个调度用函数  $f: A \rightarrow N$  表示, 其中  $f(i)$  是对客户  $i$  的服务开始的时间, 要求所有区间  $(f(i), f(i) + t_i)$  互不重叠。一个调度  $f$  的最大延迟是所有客户延迟时间的最大值。那么给定  $A, T, D$ , 求最大延迟达到最小的调度, 即求  $f$  使得

$$\min_f \left\{ \max_{i \in A} \{f(i) + t_i - d_i\} \right\}$$

### 二、问题举例

给定  $(A, T, D)$  如下:

$$A = \{1, 2, 3, 4, 5\}$$

$$T = \langle 5, 8, 4, 10, 3 \rangle$$

$$D = \langle 10, 12, 15, 11, 20 \rangle$$

那么对于调度

$$f_1: \{1, 2, 3, 4, 5\} \rightarrow N$$

$$f_1(1) = 0, f_1(2) = 5, f_1(3) = 23, f_1(4) = 5, f_1(5) = 27$$

那么客户  $1, 2, 3, 4, 5$  的延迟分别是:  $0, 1, 2, 16, 10$ ; 最大延迟是  $16$ .



### 三、列举贪心策略

策略 1: 按照  $t_i$  从小到大安排;

策略 2: 按照  $d_i - t_i$  从小到大安排;

策略 3: 按照  $d_i$  从小到大安排;

...

### 四、通过举反例排除策略

- 如果某个客户服务时间  $t_i$  很长, 而且其还很着急想早些结束, 即  $d_i$  很小; 而其他客户服务时间  $t_i$  很短, 但是不着急结束, 即  $d_i$  很大。如果此时使用策略 1, 则会先让那些不着急的客户先调度, 而着急的客户后调度, 必然造成长延迟。因此, 策略 1 不正确。

- 如果某个客户不是很着急, 即  $d_i$  很大; 同时其服务的时间也很长, 即  $t_i$  很大; 但两者相减却很小, 得到提取的调度。那么该客户会占有大量的时间, 其他的客户得不到服务。

...

### 五、贪心策略

按完成时间从早到晚安排任务, 而且没有空闲的时间。



## 六、正确性证明

交换论证(将任意解不断进行交换得到最优解，这个最优解满足贪心策略的选择)

1. 引理 1：所有没有逆序、没有空闲时间的调度具有相同的最大延迟（即相等的活动安排的先后不影响最大延迟）

设  $f$  是一个没有逆序、没有空闲时间的调度，在  $f$  中具有相同期望完成时间的客户  $i_1, i_2, \dots, i_k$  必定被连续安排。那么在这  $k$  个客户中，最后一个客户的延迟最大，他被延迟的时间为  $t_0 + \sum_{j=1}^k t_{i_j} - d$ ；其中  $t_0$  表示这  $k$  个任务的开始时间， $\sum_{j=1}^k t_{i_j}$  表示这

$k$  个客户的服务时间，那么  $t_0 + \sum_{j=1}^k t_{i_j}$  就表示这  $k$  个任务的最后结束时间；可见，这个时间与  $i_1, i_2, \dots, i_k$  的顺序无关。<https://blog.csdn.net/bqw18744018044>

2. 交换相邻逆序( $i, j$ )不影响最优性

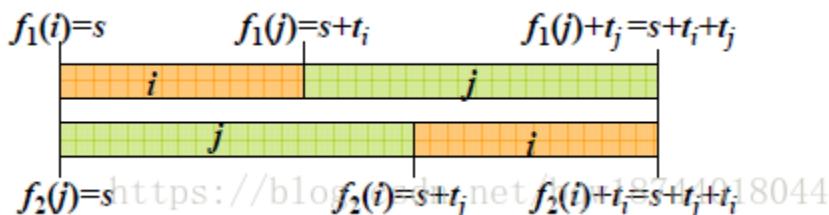
如果存在一个相邻的逆序，即  $i$  先于  $j$  调度，但是  $d_i > d_j$ 。那么

(1) 交换这两个逆序任务对其他的客户没有影响；

(2) 原来  $d_j < d_i$ ，但是  $j$  任务却后调度。现在将  $i$  和  $j$  互换，那么任务  $j$  的延迟会减小（至少不会增加）。

(3) 现在考虑任务  $i$ ，

<https://blog.csdn.net/bqw18744018044>



假设交换前，任务  $i$  的开始时间为  $s$ 。那么交换前， $i$  和  $j$  中， $j$  的延迟是最大的，其延迟时间为  $\max_b = s + t_i + t_j - d_j$ 。

交换后， $i$  和  $j$  中， $i$  的延迟是最大的，其延迟时间为  $\max_a = s + t_j + t_i - d_i$ 。

由于  $i$  和  $j$  是逆序，那么  $d_i > d_j$ ，得出  $\max_a < \max_b$ ，即交换后  $i$  和  $j$  的最大延迟较低了。

- 4) 在不断的交换逆序对（类似冒泡），整个任务的最大延迟有可能不变，也有可能不断的减小，直到没有逆序对，此时便是最优解。这个最优解也符合我们的贪心策略

<https://blog.csdn.net/bqw18744018044>

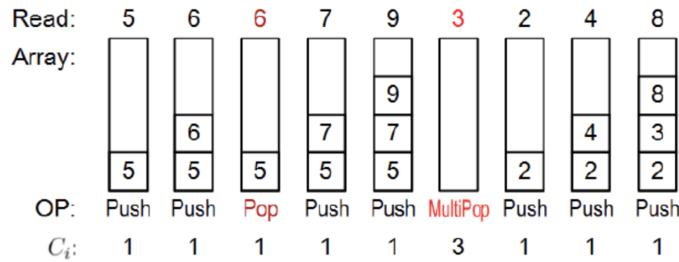
整个流程参考<https://blog.csdn.net/bqw18744018044/article/details/80285414>

主要关注如何证明解法的**最优性**以及给其他解法举反例。

## 7. 均摊分析 (Amortized Analysis)

- 概念: 给定一连串操作, 大部分的操作是非常廉价的, 有极少的操作可能非常昂贵, 因此一个标准的最坏分析可能过于消极了。因此, 其基本理念在于, **当昂贵的操作特别少的时候, 他们的成本可能会均摊到所有的操作上**。如果人工均摊的花销仍然便宜的话, 对于整个序列的操作我们将有一个更加严格的约束。本质上, 均摊分析就是在最坏的场景下, 对于一连串操作给出一个更加严格约束的一种策略。
- 聚类分析(Aggregate Analysis): 证明对所有的n, 由n个操作所构成的序列的总时间在最坏情况下为T(n), 每一个操作的平均成本为T(n)/n; 比如栈的操作, 对于一个空栈的入栈和出栈的操作。

**Example:** 有MULTIPOP操作的栈。有两种基本的栈操作都分别花费O(1)的时间: PUSH(S,x)和POP(S)分别是将对象x压入栈中, 从栈S的顶部弹出并返回弹出的对象。将每一个操作的花销都赋为1。MULTIPOP(S,k)是弹出栈S的前k个对象。例子如下:



按正常复杂度分析MULTIPOP的复杂度是O(n)因为最坏情况下MULTIPOP需要弹出栈里的所有元素。总复杂度最坏也就是n个MULTIPOP, 为O( $n^2$ )。聚类分析则旨在给出

一个均摊成本 $\hat{C}$ 满足  $T(n) = \sum_{i=1}^n C_i \leq \sum_{i=1}^n \hat{C}_i$ 。在这个问题中, 我们可以发现**POP的次数一定不大于PUSH的次数**。所以实际上包含n个PUSH, POP, MULTIPOP操作的序列的总时间为O(n)因为POP最多就把栈清空。所以每个操作的平均代价为O(n)/n = O(1)。聚集分析中, 将每个操作的平摊代价指派为平均代价。所以三个栈操作的平摊代价都是O(1)。T(n)表示最多操作数。

$$\begin{aligned}
 T(n) &= \sum_{i=1}^n C_i \\
 &= \#Push + \#Pop \\
 &\leq 2 \times \#Push \\
 &\leq 2n
 \end{aligned}$$

- 银行家分析(Accounting Analysis): 我们对一个操作的收费的数量称为平摊代价。当一个操作的平摊代价超过了它的实际代价时, 两者的差值就被当作存款(credit), 并赋予数据结构中的一些特定对象, 可以用来补偿那些平摊代价低于其实际代价的操作。这种方法与聚

集分析不同的是，对后者，所有操作都具有相同的平摊代价。数据结构中存储的总存款等于总的平摊代价和总的实际代价之差。注意：总存款不能是负的。

**Example:** 还是MULTIPOP的例子，我们可以把PUSH的花费定为2因为最坏情况就是PUSH以后的数被POP掉了，总花费为2。则POP和MULTIPOP的花费都是0。

Operation	Real Cost $C_{op}$	Amortized Cost $\widehat{C}_{op}$
PUSH	1	2
POP	1	0
MULTIPOP	$\min\{ S , k\}$	0

存款(credit)就是预计花费-实际花费的值，注意存款不能<0。例子如下：

Read:	5	6	6	7	9	3	2	4	8
Array:									
OP:	Push	Push	Pop	Push	Push	MultPop	Push	Push	Push
$C_i$ :	1	1	1	1	1	3	1	1	1
$\widehat{C}_i$ :	2	2	0	2	2	0	2	2	2
Credit:	1	2	1	2	3	0	1	2	3

因此，从一个空栈开始， $n_1$ 个PUSH,  $n_2$ 个POP和 $n_3$ 个MULTIPOP操作的任意序列

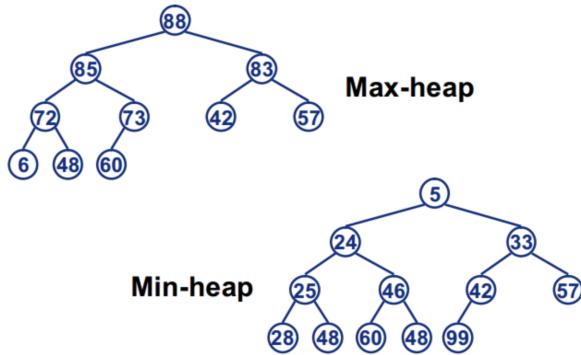
$$T(n) = \sum_{i=1}^n C_i \leq \sum_{i=1}^n \widehat{C}_i = 2n_1$$

最多的花销是  $n_1$ ，这里  $n=n_1+n_2+n_3$ 。

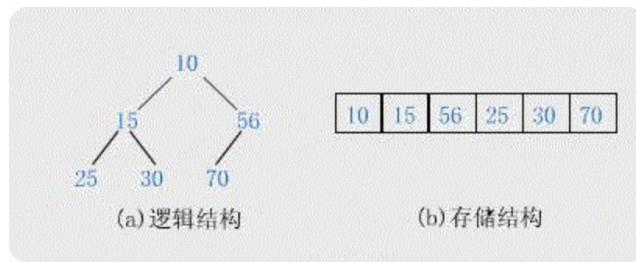
## 8. 优先队列

### 1. 大根堆小跟堆的概念：

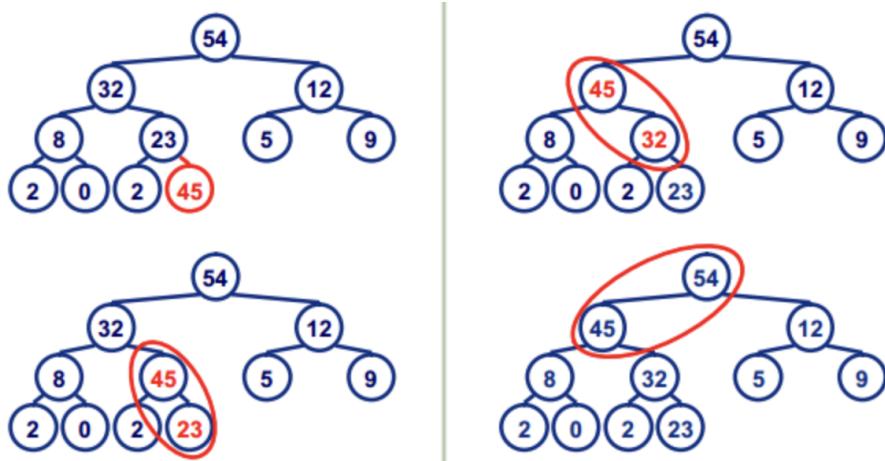
- 堆一定是完全二叉树，大根堆满足父节点一定大于自己的子节点；小根堆满足父节点一定小于自己的子节点。如下图所示：



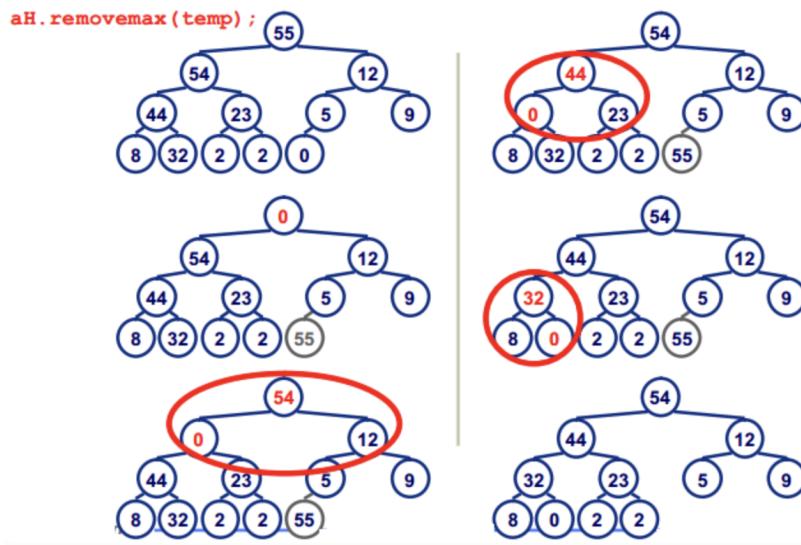
- 堆的存储：一般都用数组来表示堆， $i$ 结点的父结点下标就为 $(i-1)/2$ 。它的左右子结点下标分别为 $2 * i + 1$ 和 $2 * i + 2$ 。如第0个结点左右子结点下标分别为1和2。



- 堆中插入元素：新元素被加入到heap的末尾，然后更新树以恢复堆的次序。每次插入都是将新数据放在数组最后。可以发现从这个新数据的父结点到根结点必然为一个有序的数列，现在的任务是将这个新数据插入到这个有序数据中——这类似于直接插入排序中将一个数据并入到有序区间中。

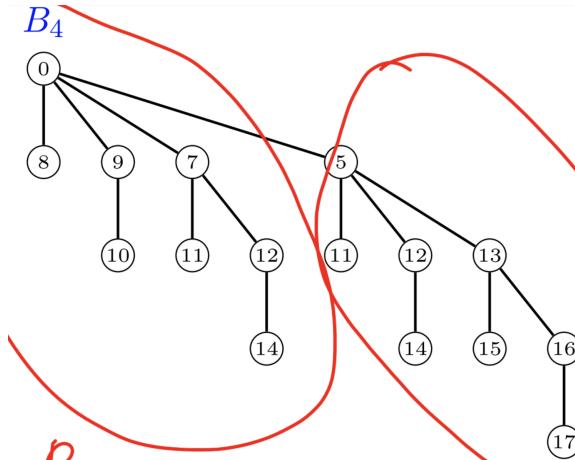


- 堆中删除堆顶元素：堆中每次都删除第0个数据。为了便于重建堆，**实际的操作是将最后一个数据的值赋给根结点，然后再从根结点开始进行一次从上向下的调整**。调整时先在左右儿子结点中找最大的，如果父结点比这个最小的子结点还大说明不需要调整了，反之将父结点和它交换后再考虑后面的结点。相当于从根结点将一个数据的“下沉”过程。



- 堆中删除任意元素 :将最后一个元素的值赋给当前要删除的节点，删除掉最后一个元素，再从根节点开始进行一次从上向下的调整。
2. 基于二叉堆的优先队列实现 :优先队列实际上就是一个以 **关键字的大小作为值的大根堆或者小根堆**。建堆、插入、删除顶部元素的操作都和上面完全一样。只是在 **优先队列中可以增加或减小任意节点的值**，其操作为通过数组下标定位到需要更改的元素，修改之后整个堆可能违反性质，所以需要重新更新一遍堆以满足大根堆或者小根堆的性质。
  3. 基于**二项堆**的优先队列：

1)二项树的概念 : $B_k$ 树就是二项树，他一共有 $2^k$ 个节点。一棵 $B_0$ 树就是一棵只有一个节点的树， $B_1$ 树就是一棵 $B_0$ 树接在另一棵 $B_0$ 树上，以此类推。**二项树并不需要是一棵二叉树**。可以看下面这个 $B_4$ 树的例子，我们可以认为它是由两棵 $B_3$ 树接在一起组成的。



2)二项树的性质 :对于一棵 $B_k$ 树来说，最重要的性质就是它**每一层的节点个数就是杨辉三角形的第 $k+1$ 层的数**，也就是 $C_k^i$ 。例如上面这棵 $B_4$ 树就满足1 4 6 4 1，就是杨辉三角形第5层。也就是二项展开式的系数。所以叫做二项树。

## 证明：数学归纳法

即假设 $B_k$ 树的第*i*层有 $C_k^i$ 个节点。

当 $k = 0$ 时, $B_0$ 树只有一个0层，而且只有一个节点，所以 $k=0$ 时成立。

当 $k = 1$ 时, $B_1$ 树有两层，第0层有一个节点，0层节点数可以写成 $C_1^0$ 。第1层也只有一个节点，1层节点数可以写成 $C_1^1$ 。依然成立。

假设 $k = n$ 的时候， $B_n$ 树的第*i*层的节点数为 $C_n^i$ 。

那么当 $k = n + 1$ 的时候，我们知道 $B_{n+1}$ 树是由两颗 $B_n$ 合并的，而且是其中一颗 $B_n$ 树的顶作为新的顶，我们把这颗树记做 $L_n$ ；而另外一个树作为一个子树挂在 $B_{n+1}$ 上面的，我们把颗树记做 $R_n$ 。

对于 $B_{n+1}$ 的第*i*的节点来说，它有两种来源：来源于树 $L_n$ 和来源于树 $R_n$ 。现在，我们来确定 $B_{n+1}$ 的第*i*层是如何构成的。

我们可以看到， $B_3$ 树的第1层，是由左边 $L_n$ 树的第1层，和右边 $R_n$ 树的第0层一同构成的。

对于任意的 $B_k$ 树的第*i*层，也可以看做是 $L_n$ 的第*i*层配上 $R_n$ 的第*i*-1层（为什么是*i*-1？因为 $R_n$ 是作为子树挂在 $L_n$ 的根上的）构成的。

所以有 $B_{n+1}$ 树的第*i*层的节点数为 $C_n^i + C_n^{i-1}$ 。这是一个组合数公式，我们来化简一下。

考虑这么一个摸球模型：一个袋子里面有 $n+1$ 个不同的球，现从这个袋子里面摸出*i*个球，求摸出球有多少种组合。很明显，这里一共有 $C_{n+1}^i$ 种球的组合的方式。

从另外一个角度来看，假如这 $n+1$ 个球有个很特别的球，我们把它叫做VIP球吧。那么这个摸球模型也可以这么考虑，我们考虑着*i*个球到底有没有摸到这个VIP球。这就分两种情况了，第一种情况摸到了这个VIP球。那么剩下的*i*-1个球的就有 $C_n^{i-1}$ 种组合。第二种情况没有摸到这个VIP球，那么这*i*个球就得全部从袋子里除VIP球外的*n*个球中摸，于是有 $C_n^i$ 中组合方式。

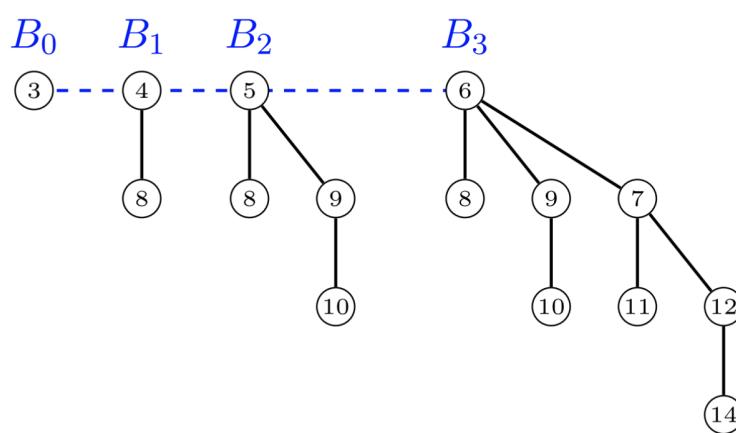
这是同一个实验的不同理解角度，所以对应的总组合数是相等的，于是我们有： $C_{n+1}^i = C_n^i + C_n^{i-1}$ 。

即 $B_{n+1}$ 树的第*i*层有 $C_{n+1}^i$ 个节点。

于是对于 $B_{n+1}$ 树，这个性质也是成立的。

因此，得证。

3) 二项堆的概念 : 二项堆就是由若干个二项树连在一起组成的堆。每种二项树在二项堆里有且只能有一棵( $B_0, B_1, B_2, \dots$ 都只能有一棵)。每颗不同 $B_k$ 树的树顶由一个双向列表链接在一起。这个定义要求如果这个堆里面存在两个 $B_k$ 则把它们合并为一个 $B_{k+1}$ 树。在合并的时候要注意根结点较大的树直接成为根结点较小的树的根结点的子节点(小跟二项堆)。大根堆的话就反过来。合并两棵树所需要的时间只有 $O(1)$ 。



## 4. 基于二项堆的优先队列实现：

1) 插入元素 : 新建一个 $B_0$ 树，这棵树的值就是要插入的数据然后将 $B_0$ 树直接插到二项堆里，如果目前二项堆没有 $B_0$ 树就直接完成了插入过程；如果有就要合并直到满足二项堆的条件。最差情况要合并 $\log n$ 棵树，复杂度 $O(\log n)$ ，但是均摊复杂度 $O(1)$ 。

2) 找到最小元素/最大元素 :由二项堆的定义可知最值元素一定都在二项树的根顶 , 遍历根顶元素 维护最小值就能知道最小元素是多少 , 最大值也一样。 **一共最多需要遍历 $\log n$ 个节点 , 因为一个二项堆里最多只会有 $\log n$ 棵二项树** 。复杂度 $O(\log n)$ 。

3) 提取最值 ( 以最小值为例 )**找到最小值** , 它肯定是某一棵树的根结点。 **直接把这个点去掉 , 然后把它的子树全部都直接挂在堆最后面 , 再跑一次合并** 。复杂度 $O(\log n)$ 。

4) 修改某一个点的值 这个操作实际上只会影响到堆中的某一棵树。 **直接把这个数修改掉然后判断是改大了还是改小了 , 从而判断是向上交换还是向下交换** 。最多只可能交换 $\log n$ 次因为一棵 $n$ 个节点的二项堆上的树最多有 $\log n$ 层。复杂度 $O(\log n)$ 。

5) 合并两个优先队列 :两个堆的根链进行合并。**相同形状的树进行合并** 。最多有 $\log n$ 棵树 , 最多进行 $\log n$ 次合并。复杂度 $O(\log n)$ 。

## 5. 基于斐波那契堆的优先队列 :

我们这里只考虑时间复杂度 , 并且斐波那契堆的时间复杂度是均摊复杂度。

Priority Queues Performance Cost Summary

Operation	Linked List	Binary Heap	Binomial Heap	Fibonacci Heap <sup>†</sup>	Relaxed Heap
<i>make-heap</i>	1	1	1	1	1
<i>is-empty</i>	1	1	1	1	1
<i>insert</i>	1	$\log n$	$\log n$	1	1
<i>delete-min</i>	$n$	$\log n$	$\log n$	$\log n$	$\log n$
<i>decrease-key</i>	$n$	$\log n$	$\log n$	1	1
<i>delete</i>	$n$	$\log n$	$\log n$	$\log n$	$\log n$
<i>union</i>	1	$n$	$\log n$	1	1
<i>find-min</i>	$n$	1	$\log n$	1	1

$n$  = number of elements in priority queue

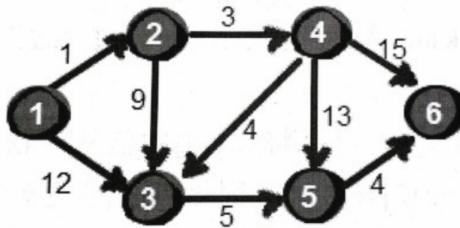
<sup>†</sup> amortized

基于斐波那契堆的优先队列主要用于 :**计算最小生成树** 和**寻找单源最短路径**。

## 9. 最短路(Dijkstra)

1. 单源最短路问题 :从一个点 $s$ 出发求它到所有其他点的最短路径 , **Dij要求不能存在负权边 !**

2. 算法过程 :以下图为例 , 假设1号点为源点 $s$ 。



起始时我们定义一个 $dis$ 数组表示从源点到其他所有点的距离。初始时 $dis[1]=0$ ，其余全为正无穷。

	1	2	3	4	5	6
dis	0	1	12	$\infty$	$\infty$	$\infty$

首先比较 $1 \rightarrow 2$ 和 $1 \rightarrow 3$ 的距离， $dis[2]=1$ ， $dis[3]=12$ 。此时 $dis[2]=1$ 已经被确定因为目前到2是最远的并且所有边都是正数，意味着不可能通过其他点的转移到达2使得距离更近。

所以我们转移到2看2的邻接点：有 $2 \rightarrow 3$ 和 $2 \rightarrow 4$ 。这时候出现了Dij算法的重点**松弛操作(Relax)**：我们先比较 $1 \rightarrow 2 \rightarrow 3$ 是否可能比 $1 \rightarrow 3$ 更短，比较发现 $1+9 < 12$ 确实更短，于是我们更改 $dis[3]=dis[2]+length[2 \rightarrow 3]$ 。然后再看 $2 \rightarrow 4$ ， $1 \rightarrow 2 \rightarrow 4=1+3 < \infty$ ，于是更改 $dis[4]=dis[2]+length[2 \rightarrow 4]=4$ 。松弛完毕以后 $dis$ 数组更新为如下：

	1	2	3	4	5	6
dis	0	1	10	4	$\infty$	$\infty$

接下来，再从 $3, 4, 5, 6$ 中选择离1最近的点，发现是4，所以 $dis[4]$ 被确定，再从4开始再进行上边的松弛操作。以此类推直到所有点都被选择完毕。

### 3. 算法步骤：

- 1) 所有结点分为两部分：已确定最短路的结点集合P、未知最短路的结点集合Q。最开始，P中只有源点这一个结点。(可用一个book数组来维护是否在P中)
- 2) 在Q中选取一个离源点最近的结点u( $dis[u]$ 最小)加入集合P。然后考察u的所有出边，做松弛操作。
- 3) 重复第二步，直到集合Q为空。最终 $dis$ 数组的值就是源点到所有顶点的最短路。

### 4. 算法正确性证明

可以看出Dij实际上是一个贪心算法，我们每次一定选择当前认为的离源点最近的点进行更新，松弛这个点的所有邻接点，如何证明它一定能得到最短路径呢？

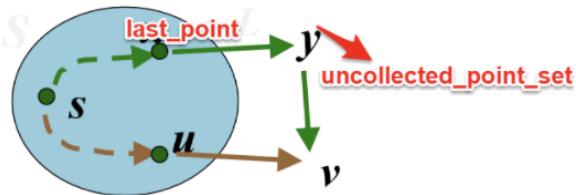
**Proof:** 数学归纳法。

- 1) 设n为Dij中的集合P的点的个数。 $n=1$ 的时候只加入了源点，最短距离为0一定是最优的
- 2) 假设 $n=k$ 的时候一个最优算法O得到的最短路和Dij得到的最短路都相同， $n=k+1$ 的时候Dij现在要加入的点为v， $e=(u,v)$ 是Dij得到的最短路中的最后一段。这里用反证法。假设O

得到了一个更优的解，由于 $n=k$ 的时候O和Dij的解相同，因此O一定选择了一条其他路径 $V'$ 到达 $v$ 。

3) 这条O选择的路径一定经过了集合P中的至少一个点，假设最后一个在P中的点为 $x$ ，那么 $e'=(x,v)$ 一定不会比 $e$ 更短，不然在Dij松弛的时候就会考虑 $e'$ 而不是 $e$ 。因此， $V'$ 一定还经过了至少一个不在当前P中的点 $y$ 然后才到 $v$ ，即 $x \rightarrow y \rightarrow v$ 。

4) 然而，我们可以证明 $s \rightarrow x \rightarrow y$ 这条路径一定不比 $s \rightarrow u \rightarrow v$ 这条路径短。因为P集合的所有点已经经过了松弛操作。因此既然Dij在第 $k+1$ 步选择将 $v$ 加入P集合，说明Dij已经考虑过 $s \rightarrow x \rightarrow y$ 这条路径，但是由于它的长度不比 $s \rightarrow u \rightarrow v$ 短因此最后决定先加入点 $v$ 。既然有 $\text{length}(s \rightarrow x \rightarrow y) \geq \text{length}(s \rightarrow u \rightarrow v)$ ，那么一定有 $\text{length}(s \rightarrow x \rightarrow y \rightarrow v) \geq \text{length}(s \rightarrow u \rightarrow v)$ ，**因为图中所有的边权都为非负数**。



5) 因此，我们反证完了当 $n=k+1$ 的时候不存在一个比Dij选择的路径更优的路径。所以 $n=k+1$ 的时候Dij得到的一定是最短路，归纳证明完毕。

5. 算法优化及复杂度：我们可以用**优先队列**来优化Dij算法。常规的Dij总复杂度是 $O(n^2)$ (每次找点+松弛 $O(n)$ ，总共对 $n$ 个点操作)。我们发现每一轮中有一个查找当前dis最小的点，这个复杂度是 $O(n)$ 。但维护最小值明显可以通过基于小跟堆的优先队列实现：

- 1) 我们以 $\text{dis}[i]$ 作为优先级建立一个优先队列，初始化时除了源点的所有点进队。每次找点实际上就在取优先队列的top元素，复杂度为 $O(\log n)$ ，一共要找 $n$ 次，复杂度为 $O(n \log n)$
- 2) 每次取出点后我们要对其进行松弛，然后需要对优先队列中的dis值进行修改，复杂度为 $O(\log n)$ 。每次松弛只会修改和当前节点相邻的节点，所以总共修改的次数就是边的数量 $m$ ，复杂度为 $O(m \log n)$ 。整个算法的总复杂度就为 $O((m+n)\log n)$ 。
- 3) 当图为稀疏图的时候，我们可以认为 $m=n-1$ ，则总复杂度是 $O(n \log n)$ 的。但是当图为稠密图的时候， $m=n*(n-1)/2$ ，这个优化就退化到了 $O(n^2 \log n)$ 。
- 4) 因此，我们考虑用更优的结构**斐波那契堆**来构建优先队列。斐波那契堆的修改操作为 $O(1)$ ，因此在松弛修改节点的值的时候复杂度只有 $O(m)$ 。总复杂度就是 $O(m+n \log n)$ 。即使在稠密图复杂度也有 $O(n^2)$ 。

	<u>Binary Heap</u>	<u>Binomial Heap</u>	<u>Fibonacci Heap</u>
$n \times \text{Extract Min's}$	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$
$m \times \text{Decrease key's}$	$O(m \lg n)$	$O(m \lg n)$	$O(m)$
Total	$O(m \lg n)$	$O(m \lg n)$	$O(m + n \lg n)$
Sparse Graphs	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$
Dense Graphs	$O(n^2 \lg n)$	$O(n^2 \lg n)$	$O(n^2)$

## 10. 最小生成树 ( MST )

1. 定义 给定一个无向图，如果它任意两个顶点都联通并且是一棵树，那么我们就称之为生成树(Spanning Tree)。如果是带权值的无向图，那么**权值之和最小的生成树**，我们就称之为最小生成树(MST, Minimum Spanning Tree)。

Kruskal算法的时间复杂度

Kruskal算法每次要从都要从剩余的边中选取一个最小的边。通常我们要先对边按权值从小到大排序，这一步的时间复杂度为为 $O(|E| \log |E|)$ 。Kruskal算法的实现通常使用并查集，来快速判断两个顶点是否属于同一个集合。最坏的情况可能要枚举完所有的边，此时要循环 $|E|$ 次，所以这一步的时间复杂度为 $O(|E| \alpha(V))$ ，其中 $\alpha$ 为Ackermann函数，其增长非常慢，我们可以视为常数。所以Kruskal算法的时间复杂度为 $O(|E| \log |E|)$ 。

## 11. 分治算法

**Overview:** 分治包括三个阶段：Divide, Conquer, Combine

1. 归并排序(递归)：

```
Merge_Sort(A,l,r):
    If l < r:
        mid = (l+r)/2
        Merge_Sort(A,l,mid)
        Merge_Sort(A,mid+1,r)
        Merge(A,l,mid,r)
```

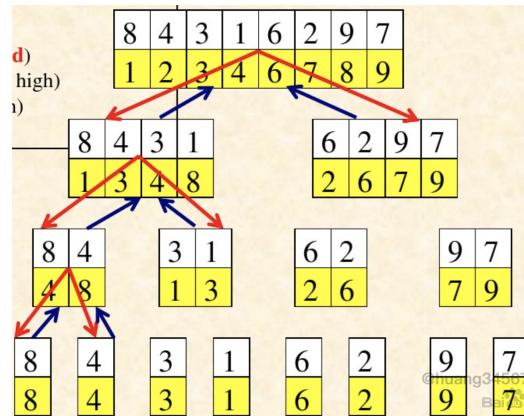
```
Merge_Sort(A,mid+1,r)
```

```
    Merge(A,l,mid,r)
```

```
Endif
```

- 复杂度分析：

- Divide阶段只需要找到中间点，复杂度O(1)
- Conquer阶段是一个两次递归的操作，假设n个数原始问题花费时间T(n),则每次递归花费时间T(n/2).
- Combine阶段是一个结合操作，每个数会被拿来比较递归的层数次，复杂度是O(n)



## 2. 主定理(Master Theorem)

- **Background:** 根据上面归并操作里递归阶段的假设，可以一般化出所有递推公式  $T(n)=aT(n/b)+f(n)$ , 其中a表示每次分治将会分成多少个子问题; b表示每次分治以后的子问题的数据量会减小多少 ; $f(n)$ 表示Divide和Combine阶段复杂度之和是多少。

- **Master Theorem:**

前提条件 : $a \geq 1, b \geq 1$  且  $ab$ 都为常数 ,  $T(n)$ 有一定的初始值(即 $n=1$ 时 $T(n)$ 是一个常数)  
那么, 当 $n$ 充分大的时候, 一定有 :

- 1) 若有  $\varepsilon > 0$ , 使  $f(n)=O(n^{(\log_b a)-\varepsilon})$   
 (即  $f(n)$  的量级多项式地小于  $n^{\log_b a}$  的量级), 则  $T(n)=\Theta(n^{\log_b a})$ 。
- 2) 若  $f(n)=\Theta(n^{\log_b a})$   
 (即  $f(n)$  的量级等于  $n^{\log_b a}$  的量级), 则  $T(n)=\Theta(n^{\log_b a} \log n)$ 。
- 3) 若有  $\varepsilon > 0$ , 使  $f(n)=\Omega(n^{(\log_b a)+\varepsilon})$   
 (即  $f(n)$  的量级多项式地大于  $n^{\log_b a}$  的量级), 且满足正规性条件:  
 存在常数  $c < 1$ , 使得对所有足够大的  $n$ , 有  $a^*f(n/b) \leq c^*f(n)$ ,  
 则  $T(n)=\Theta(f(n))$ 。

这里的正规性条件通俗讲就是  $a$  个子问题的 Divide 和 Combine 的时间之和小于原问题的 Divide 和 Combine 所需要的时间。

- 说人话: 其实就是在分析一个分治算法的时候将递归部分和递归外处理的部分分开计算复杂度。实际上我们不好计算复杂度的只有递归部分因为它无法用多项式表示。所以实际上  $n^{\log_b a}$  代表的就是递归部分的复杂度,  $f(n)$  代表的就是处理的复杂度, 选择更大的一个作为算法的总复杂度。需要注意的是: 1) 在  $f(n)=n^{\log_b a}$  的时候, 总复杂度需要 ~~×~~ 一个  $\log n$ . 2) 在  $f(n) > n^{\log_b a}$  的时候, 还需要验证正规性条件, 即  $a^*f(n/b) \leq c^*f(n)$ ,  $c < 1$ , 如果  $f(n)=n \log n$ ,  $n^{\log_b a}=n$ , 这个时候是不满足 case3 的条件的。
- Example: 以上的归并排序为例,  $f(n)=\text{Divide}+\text{Combine}=O(1)+O(n)=O(n)$   
 $a=2, b=2$ (每次变成两倍的子问题, 每个子问题数据量变为一半)。所以  
 $n^{\log_b a}=n^{\log_2 2}=n$ 。有  $f(n)=n^{\log_b a}$ , 属于上面的第二种情况, 所以归并排序的总复杂度为  $O(n \log n)$ 。

参考 <https://zhuanlan.zhihu.com/p/100531135>

### 3. 股票买卖问题(Leetcode 121)

- 题目简介: 给定一个数组, 它的第  $i$  个元素是一支给定股票第  $i$  天的价格。如果你最多只允许完成一笔交易(即买入和卖出一支股票), 设计一个算法来计算你所能获取的最大利润。注意你不能在买入股票前卖出股票。
- 分析: 这题实际上有三种做法, 最简单的是左右两次循环, 记录某个数之前的最小值和之后的最大值, 最后相减即可, 复杂度  $O(n)$ 。第二种做法是 DP, 这个问题在“代码随想录”的动态规划专题里有更详细的介绍, 复杂度也是  $O(n)$ 。这里专注的算法是分治实现这道题目。

- 思路1：将整个数列由中间分成两段，分别在左边一半和右边一半里找最大差值，最后比较差值取最大值。

上面思路的问题在于最大差值可能出现在跨越中点的区间，这是思路1的分治没考虑到的。

- 思路2：还是由中间分成两段，但考虑三个区间。

- 买入的股票和卖出的股票都在区间 $[l, mid-1]$ 完成。
- 买入的股票和卖出的股票都在区间 $[mid+1, r]$ 完成。
- 区间穿过 $mid$ ，在左区间买入股票，在右区间卖出股票。

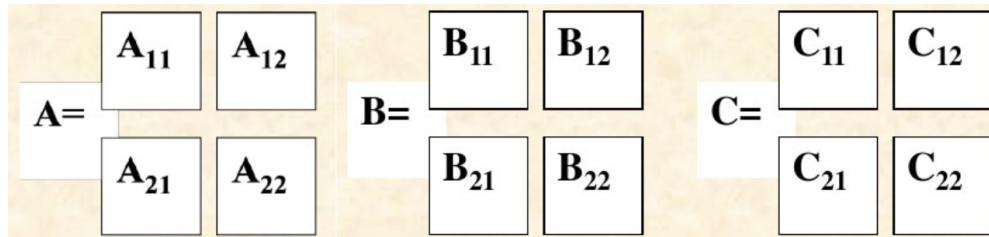
上面两个区间仍然通过递归完成，最下面的区间则需要在每次递归的时候进行处理，过程就是每次循环当前区间的左半边找到最小值，循环右半边找到最大值求差。那么三个区间的最大值就是问题的解。

- 接下来分析复杂度：

- 对于递归部分，明显的一个二分， $a=2, b=2, \log_b a = n$ 。
- 对于Divide和Combine部分，Divide是 $O(1)$ ，Combine实际上只是一个 $\max(x_1, x_2, x_3)$ 也是 $O(1)$ 。**注意这里Combine不要算计算 $x_3$ 的部分即循环部分，因为它实际上是Conquer的一部分！！**
- 所以有 $n^{\log_b a} > f(n)$ ，满足主定理的第一种情况，总复杂度为 $O(n^{\log_b a}) = O(n)$ 。

#### 4. 矩阵乘法问题

- 题目简介：设 $A, B$ 为两个 $n \times n$ 的矩阵，求 $C = AB$ 。
- 分析：最简单的思路就是直接相乘，考虑到对于每一个行向量 $r$ ，总共有 $n$ 行；对于每一个列向量 $c$ ，总共有 $n$ 列；计算它们的内积，总共有 $n$ 次乘法计算。总复杂度为 $O(n^3)$ 。考虑用分治算法进行优化。
- 思路1：分块矩阵法：将 $A, B$ 都平均分成四块，如下图所示：



$$\text{那么有 } C_{11} = A_{11}B_{11} + A_{12}B_{21}, C_{12} = A_{11}B_{12} + A_{12}B_{22},$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

- 复杂度分析：

- 对于非递归部分，分解矩阵复杂度为 $O(1)$ ，合并矩阵复杂度为矩阵加法的复杂度，所以是 $O(n^2)$ ，因此总复杂度 $f(n) = O(n^2)$ 。

- 对于递归部分， $a=8, b=2$ （每次递归A变为四个，B也变为四个所以总共有8个子问题；而对于每个小矩阵，只是变成了 $n/2 \times n/2$ 的规模）。因此有  
 $n^{\log_b a} = n^{\log_2 8} = n^3$ 。
- 所以有 $n^{\log_b a} > f(n)$ ，满足主定理的第一种情况，总复杂度为 $O(n^{\log_b a}) = O(n^3)$ 。  
完全没有任何优化。
- 思路2: Strassen算法。前面分解矩阵的部分和思路1完全一样，但是在得到8个小矩阵之后，创建10个 $n/2 \times n/2$ 的矩阵 $S_1, S_2, \dots, S_{10}$ 。复杂度为 $O(n^2)$ ：

$$\begin{aligned}S_1 &= B_{12} - B_{22} \\S_2 &= A_{11} + A_{12} \\S_3 &= A_{21} + A_{22} \\S_4 &= B_{21} - B_{11} \\S_5 &= A_{11} + A_{22} \\S_6 &= B_{11} + B_{22} \\S_7 &= A_{12} - A_{22} \\S_8 &= B_{21} + B_{22} \\S_9 &= A_{11} - A_{21} \\S_{10} &= B_{11} + B_{12}\end{aligned}$$

接着用递归计算7个矩阵的积 $P_1, P_2, P_3, \dots, P_7$ ，每个矩阵 $P_i$ 都是 $n/2 \times n/2$ 大小。

$$\begin{aligned}P_1 &= A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22} \\P_2 &= S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22} \\P_3 &= S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11} \\P_4 &= A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11} \\P_5 &= S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22} \\P_6 &= S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22} \\P_7 &= S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}\end{aligned}$$

再通过7个 $P$ 矩阵来计算 $C_1, C_2, C_3, C_4$ 。复杂度为 $O(n^2)$ ：

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

- 复杂度分析：
  - 对于非递归部分，分解矩阵复杂度为 $O(1)$ ，合并矩阵复杂度为创建矩阵的复杂度，所以是 $O(n^2)$ ，因此总复杂度 $f(n) = O(n^2)$ 。
  - 对于递归部分， $a=7, b=2$ （子问题被分成了求7个矩阵 $P_i$ ，而不是8个子问题了；而对于每个小矩阵，只是变成了 $n/2 \times n/2$ 的规模）。因此有  
 $n^{\log_b a} = n^{\log_2 7} = n^{2.81}$ 。

- 所以有  $n^{\log_b a} > f(n)$ , 满足主定理的第一种情况 , 总复杂度为  $O(n^{\log_b a})=O(n^{2.81})$

参考<https://zhuanlan.zhihu.com/p/78657463>

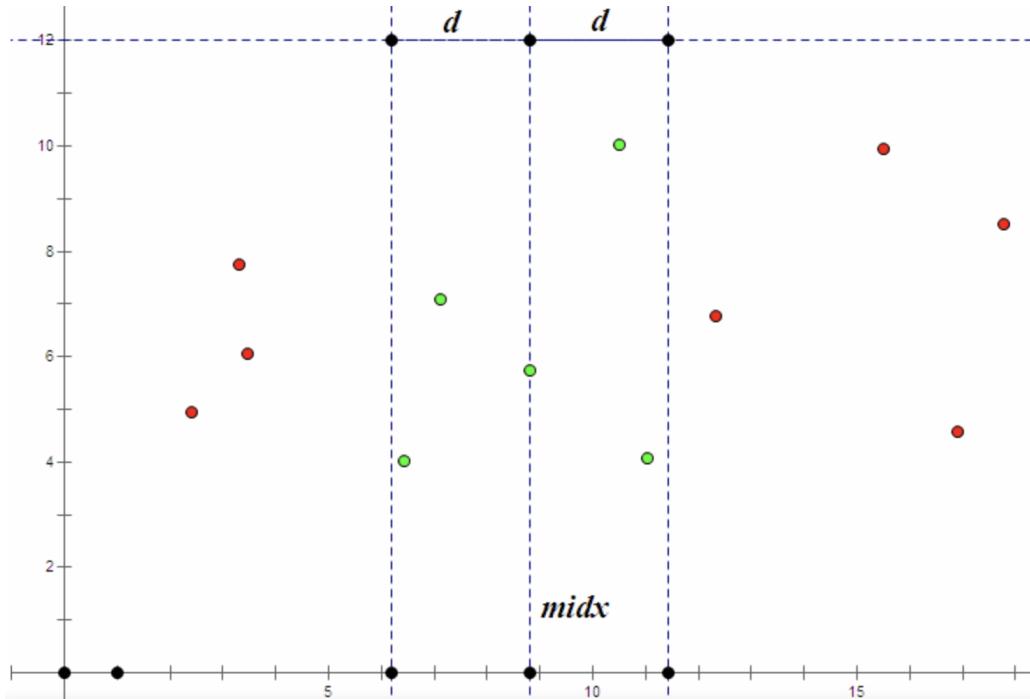
<https://wenku.baidu.com/view/8fdadcf2f61fb7360b4c6521.html>

## 5. 在未排序的数列中求最大最小值

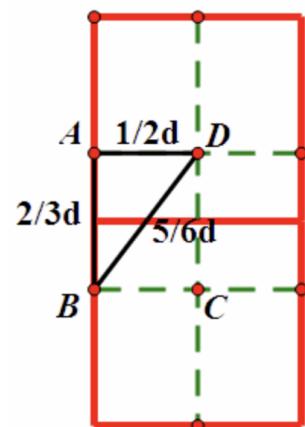
- 思路1:很简单 , 两个for , 一次找最小值一次找最大值 , 总复杂度  $O(n)$ 。
- 思路2:分治 , 每次分成两半 , 递归部分找出左右区间的最大最小值  
 $X_{lmin}, X_{lmax}, X_{rmax}, X_{rmin}$ , 一直到最后要么只剩一个元素 , 它自己就是最大最小值 , 要么剩下两个元素 , 一个最大值一个最小值。非递归操作都是  $O(1)$  的。
- 复杂度分析 : $a=2, b=2, n^{\log_b a}=n$ 。总复杂度也就是  $O(n)$  的。看上去并没有多项式级别的提升 , 但实际上  $T(n)=2T(n/2)+2$  ( 这里最后的2指的是比较次数 )。那么  $T(n)=2^2T(n/2^2)+2^2+2....=2^{k-1}T(2)+2^{k-1}+2^{k-2}+....+2(n=2^k)=2^{k-1}\times 1+2^k-1=3n/2-1$ 。而思路1的复杂度是  $2n-2$ 。

## 6. 最近相邻点对问题

- 问题简介 :给定平面上  $n$  个点 , 找出其中的一对点的距离 , 使得在这  $n$  个点的所有点对中 , 该距离为所有点对中最小的。
- 思路1:分别计算每两个点之间的距离维护一个最小值 , 复杂度是  $O(n^2)$ 。
- 思路2:分治。将整个区间按照横坐标分成两半 , 保证左右两边各有  $n/2$  个点。左右递归维护子区间的距离最小值 , 最后比较大小即可。**但是, 这里有一个和股票问题里一样的trick , 即有可能最近点对是左边区间一个点右边区间一个点。**如果直接搜索左右区间算距离 , 又退化成思路1的算法了。这里就需要优化分治算法。
  - 优化1: 其实我们在递归的时候会得到很多有用的信息。例如 : 在分治处理左右两边的时候 , 已经求出的最近点对距离为  $d$  ( $d$  为左半集合答案与右半集合答案的较小值) 那么如果我们能确定左侧的一个点到右侧的任何一个点的距离一定大于  $d$  的话 , 那么这个点在枚举点的过程中是可以直接被忽略的。这样 , 我们可以把左侧集合最靠右的点的横坐标记为  $midx$  , 如果一个点的横坐标  $x$  满足  $|x-midx| \geq d$  , 就不需要再考虑这个点了 , 其实需要考虑的点只有下图中间一个“竖条”中的点。



- 优化2：如果在中间这个竖条中暴力枚举点对的话时间复杂度还是 $O(n^2)$ 的，因为我们不知道这中间会有多少个点。但实际上我们还可以进一步缩小更优解可能存在的空间。对于每一个左半边的绿色点，我们只需要以它的纵坐标排序，只要某个点的纵坐标和它自己纵坐标的差 $>d$ 那一定也不考虑。
- 优化3：经过优化2我们进一步缩小了搜索范围，但是仍然不能确定有多少点是可能存在的，复杂度还是没有变化。所以我们需要进一步减小搜索的范围，事实上我们要求的范围只在一个“日”字形内。我们画出一个长为 $2d$ 宽为 $d$ 的矩形，在长上我们将它三等分，宽上我们将它二等分，得到右图。
- 根据勾股定理我们可以得到对于任意一个小矩形ABCD，对角线是它们能得到的最长距离为 $5/6d < d$ 。以上图中中线上的点为例，它向右画一个这样的矩形，可知每一个点对的距离都一定 $>d$ （都在右半边）。所以在这个矩形中最多只会存在**6个点！**这样实际上对于 $[midx-d, midx+d]$ 区间内的每个点只需要比较6个点的距离，是常数级别的复杂度，因此整个区间的复杂度为 $O(n)$ ，加上我们之前需要排序，复杂度为 $O(n\log n)$ ，总复杂度就是 $O(n\log n)$ 。

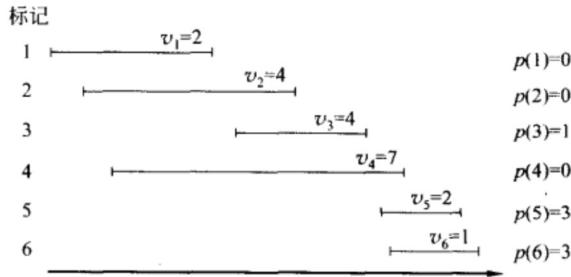


参考[https://blog.csdn.net/GGN\\_2015/article/details/80785621](https://blog.csdn.net/GGN_2015/article/details/80785621)

## 12. 动态规划

### 1. 带权区间调度问题

- 问题简介 :有n个请求，每个请求占用时间 $[s_i, f_i]$ , 并且能获得权值 $v_i$ , 求能获得的最大价值。
- 思路:按照请求的结束时间 $f_i$ 以从小到大排序, 设 $p_j$ 为在请求j之前的最近的可以相容的请求是谁。



设 $dp[i]$ 表示按顺序考虑到第i个请求的时候能获得的最大价值。

转移方程 : $dp[i] = \max(dp[p_j] + v[i], dp[i-1])$ 。转移方程第一部分表示考虑取当前的区间，那么这个区间之前的相交的区间肯定不能取，所以是 $dp[p_j] + v[i]$ ; 第二部分表示不考虑当前区间，那么答案就是 $dp[i-1]$ 。两者之间取最大就是当前的答案。

初始化 : $dp[0]=0, dp[1]=v[1]$ 。

- 复杂度分析 :排序复杂度 $O(n\log n)$ ， $dp$ 主部分 $O(n)$ ，每次需要找可以用二分查找复杂度 $O(\log n)$ ，总复杂度 $O(n\log n)$ 。

### 2. 游戏问题 (类似走楼梯)

- 问题简介 :游戏主人公要从0走到家n，格子i上有代价 $w_i$ ，当主人公走到i时会花费 $w_i$ 的体力。同时，每一次主人公可以选择走一步消耗50体力，走两步消耗150体力，走三步消耗350体力。求主人公走到家消耗的最小体力。
- 思路:设 $dp[i]$ 表示走到坐标i时消耗的最小体力。

转移方程 : $dp[i] = \min(dp[i-1]+50, dp[i-2]+150, dp[i-3]+350) + w_i$ 。 $\min()$ 里的三部分分别表示走一步，走两步，走三步到达i的情况。不管是哪一步走来的，最后都会消耗 $w_i$ 的体力。

初始化 : $dp[0]=0, dp[1]=50+w_1, dp[2]=150+w_2$ 。

- 复杂度分析 :只需要一个 $3 \sim n$ 的循环，复杂度 $O(n)$ 。

### 3. 硬币问题

- 问题简介 :现在有面值为1, 5, 10, 20, 25的硬币，问凑成n需要最少多少枚硬币。
- 思路:设 $dp[i]$ 表示凑成i需要的最少硬币数。

转移方程 : $dp[i]=\min(dp[i-1], dp[i-5], dp[i-10], dp[i-20], dp[i-25])+1$ 。min()里的五部分表示最后一次取了哪个面值的硬币凑到了i，不管用哪种面值都会增加一枚硬币  
初始化 : $dp[0]=0, dp[1]=1, dp[2]=2 \dots dp[25]=1$

- 复杂度分析 :只需要一个25~n的循环，复杂度O(n)。

#### 4. 背包问题（重点为状态压缩）

- 问题简介 :01背包，背包容量W，一共n个物品。
- 思路 : $dp[i][j]$ 表示考虑到第i个物品，背包大小为j的时候的最大价值。考虑每一个物品取不取。

转移方程 : $dp[i][j]=\max(dp[i-1][j], dp[i-1][j-w[i]]+v[i])$ 。前面一部分表示当前物品不取，答案就是i-1个物品装到容量为j的背包的答案；后一部分表示当前物品取，答案就是i-1个物品装到容量为j-w[i]的背包的答案加上i的价值v[i]。

初始化 : $dp[0][j]=0, dp[i][0]=0$

- 复杂度分析 :外部循环1~n，内部循环W~w[i]，总时间复杂度O(nW)。空间复杂度O(nW)。
- 优化 :

- 状态压缩 :如果一个二维dp的状态只由它的相邻状态转移而来，它可以被压缩成一维的以减小空间复杂度。在这个问题里， $dp[i][j]$ 只会由 $dp[i-1]$ 这个状态转移过来，那么每次只需要把 $dp[i-1]$ 拷贝到 $dp[i]$ ， $dp[i-1]$ 这一行就完全没用了。与其这样，我们不如只维护一个一维数组 $dp[j]$ 来滚动更新。
- 转移方程 : $dp[j]=\max(dp[j], dp[j-w[i]]+v[i])$ 。
- 初始化 : $dp[0]=0$
- 复杂度分析 :此时外部循环1~n，内部循环W~w[i]使用滚动数组的01背包只能从大到小循环，完全背包只能从小到大循环)，时间复杂度O(nW)，空间复杂度O(W)。

- 状态压缩扩展 :<https://blog.csdn.net/longlongqin/article/details/118682105>

- 由相邻两个状态转移来的情况 : $dp[i][j]=\max(dp[i-1][j], dp[i-1][j-1])$ 的压缩为 $dp[i]=\max(dp[i-1], dp[i])$ 。假设我们初始化了 $dp[0][j]$ ， $dp[i][0]$ 。 $dp[1][1]$ 应该来自 $dp[0][1]$ 或 $dp[1][0]$ ，压缩成一维以后 $dp[1]=\max(dp[0], dp[1])$ 。其中外部的 $dp[1]$ 表示的是 $dp[1][1]$ ， $dp[0]$ 表示的是 $dp[0][1]$ ， $\max$ 内部的 $dp[1]$ 表示的是 $dp[1][0]$ (先比较才赋值)。
- 由相邻三个状态转移来的情况 : $dp[i][j]=\max(dp[i-1][j], dp[i-1][j-1], dp[i-1][j-2])$ 的压缩为 $dp[i]=\max(dp[i-1], pre, dp[i])$ 。三个状态的转移无法完全用一维状态表示，因此要引入一个临时变量pre来存 $dp[i-1][j-1]$ 的数据。具体使用方法为先定义一个temp在外层循环， $temp=dp[0]$ 表示存上一行的第一个数据

$dp[i-1][0]$ 。内层循环里每次 $pre=temp$ ,  $temp=dp[j]$ , 此时 $pre$ 表示的就是 $dp[i-1][j-1]$ 的数据, 而 $temp$ 表示的是 $dp[i-1][j]$ 的数据, 即下一次循环时 $dp[i-1][j-1]$ 的数据。通过这样的方法实现多保存一个数据的功能。

- 求背包的取物品路径:

同样, 怎么求路径?

利用前面讲到的Path[][], 需空间消耗O(NV)。这里不能用 $F[j]==F[j-C[i]]+W[i]$ 来判断是因为一维数组并不能提供足够的信息来寻找二维路径。

加入路径信息的伪代码如下:

```
1   F[] ← {0}
2
3   Path[][]←0
4
5   for i←1 to N
6
7       do for k←V to C[i]
8
9           if(F[k] < F[k-C[i]]+W[i])
10          then F[k] ← F[k-C[i]]+W[i]
11
12          Path[i][k] ← 1
13
14
15 return F[V] and Path[][]
16
```

打印背包内物品的伪代码如下:

```
1   i←N
2
3   j←V
4
5   while(i>0 && j>0)
6
7       do if(Path[i][j] = 1)
8
9           then Print W[i]
10
11           j←j-C[i]
12
13       i←i-1
14
15
```

## 5. 序列比对 ( 重点仍然是状态压缩 ( 课上讲的是分治但其实是一样的 ) )

<https://zhuanlan.zhihu.com/p/65362522>

Align two sequences: x and y

- $F(i,j)$  is the score of the best alignment between  $x_{1..i}$  and  $y_{1..j}$
- $s(A,B)$  is the score for substituting A with B; d is the (linear) gap penalty

$$F(0,0) = 0$$

$$F(i,j) = \max \begin{cases} F(i-1, j-1) + s(x_i, y_j) & x_i \text{ aligned to } y_j \\ F(i-1, j) + d & x_i \text{ aligned to a gap} \\ F(i, j-1) + d & y_j \text{ aligned to a gap} \end{cases}$$

- 复杂度分析 :时间复杂度O(mn), 空间复杂度O(nm)。
- 优化 ( 分治 )看上面的图就可以发现实际上这就是一个由相邻 三个 状态转移来的 dp, 当然是可以把空间优化到O(m+n)的。

## 6. 矩阵链乘法 ( 类似于区间dp(例如最长xx序列) ) :

[https://blog.csdn.net/c18219227162/article/details/50412333?spm=1001.2101.3001.6650.4&utm\\_medium=distribute.pc\\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-4.essearch\\_pc\\_relevant&depth\\_1-utm\\_source=distribute.pc\\_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-4.essearch\\_pc\\_relevant](https://blog.csdn.net/c18219227162/article/details/50412333?spm=1001.2101.3001.6650.4&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-4.essearch_pc_relevant&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-4.essearch_pc_relevant)

- 问题简介 :给定n个矩阵的链 $\langle A_1, A_2, \dots, A_n \rangle$ , 矩阵 $A_i$ 的规模为 $p_{i-1} \times p_i$  ( $1 \leq i \leq n$ ), 求完全括号化方案, 使得计算乘积 $A_1 A_2 \dots A_n$ 所需标量乘法次数最少。
- 思路 :设 $dp[i][j]$ 表示 $[A_i, A_j]$ 所需标量乘法次数的最少值。  
转移方程 : $dp[i][j] = \min(dp[i][k], dp[k+1][j]) + p_{i-1} p_k p_j$  ( $i \leq j$ )。前面一部分表示当 $i=j$ 的时候, 只有一个矩阵, 不存在乘法; 后一部分当 $i < j$ 时, 我们举一个*i≤k≤j*, 表示以k为分界将 $[i,j]$ 分成两个部分最后获得的乘法次数最少。这样 $[i,j]$ 就分成了 $[i,k]$ ,  $[k+1,j]$ 和最后一次乘法这三个部分。由于 $A_i$ 的规模为 $p_{i-1} \times p_i$ ,  $[i,k]$ 得到的矩阵就是 $p_{i-1} * p_k$ 的; 同理,  $[k+1,j]$ 得到的矩阵是 $p_k * p_j$ 的。因此最后一次乘法就是 $p_{i-1} p_k p_j$ 。  
初始化 : $dp[i][i] = 0$
- 复杂度分析 :外部循环 $2 \sim n$ , 内部循环 $i-1 \sim n$ , 每次枚举k的复杂度是 $O(n)$ 。所以总时间复杂度 $O(n^3)$ 。空间复杂度 $O(n^2)$ 。

## 7. 最短路(Bellman-Ford & SPFA):

- 思路 :设 $dp[i]$ 到i这个点的最短路径长度。

转移方程 : $dp[v[i]] = \min(dp[v[i]], dp[u[i]] + w[i])$ 。这个方程转移的条件就是 $u[i] \sim v[i]$ 这条路是否能让 $s \sim v[i]$ 这条路的距离更短。所以要枚举每个点。

初始化 : $dp[s] = 0$ ,  $dp[\text{其余点}] = \infty$ 。

- 复杂度分析 :两重循环，外部循环 $1 \sim n$ 枚举每个点，内部循环 $1 \sim m$ 枚举每条边，所以总时间复杂度 $O(nm)$ 。相比于dij的 $O((m+n)\log n)$ 要差一些。
- 优点 :**Bellman-Ford**可以判断负环，也可以容忍图里有负权边。如果松弛操作的次数超过了 $n-1$ 次，那么一定存在负环。
- 优化 :通过优先队列优化**Bellman-Ford->SPFA**。同样可以将正常情况下的复杂度优化到 $O((m+n)\log n)$ ，但是最差情况下(图非常稠密)还是 $O(nm)$ 。同样可以判断负环，如果一个点进队的次数超过 $n$ 次，一定存在负环。

	Floyd	Dijkstra	Bellman-Ford	队列优化的Bellman-Ford
空间复杂度	$O(N^2)$	$O(M)$	$O(M)$	$O(M)$
时间复杂度	$O(N^3)$	$O((M+N)\log N)$	$O(NM)$	最坏也是 $O(NM)$
适用情况	稠密图 和顶点关系密切	稠密图 和顶点关系密切	稀疏图 和边关系密切	稀疏图 和边关系密切
负权	可以解决负权	不能解决负权	可以解决负权	可以解决负权
有负权边	可以处理	不能处理	可以处理	可以处理
判定是否存在负权回路	不能	不能	可以判定	可以判定

### 判断题 :

- 1) **False.** Any Dynamic Programming algorithm with  $n$  unique subproblems will run in  $O(n)$  time.
- 2) **False.** The running time of a pseudo polynomial time algorithm depends polynomially on the size of the input.
- 3) **False.** In dynamic programming you must calculate the optimal value of a subproblem twice, once during the bottom up pass and once during the top down pass. x2
- 4) **True.** Bellman-Ford algorithm solves the shortest path problem in graphs with negative cost edges in polynomial time.
- 5) **False.** An optimal solution to a 0/1 knapsack problem will always contain the object  $i$  with the greatest value-to-cost ratio  $V_i/C_i$ .
- 6) **False.** Complexity of a dynamic programming algorithm is equal to the number of unique sub-problems in the solution space.
- 7) **False.** 0-1 knapsack problem can be solved using dynamic programming in polynomial time
- 8) **False.** The Bellman-Ford algorithm always fails to find the shortest path between two nodes in a graph if there is a negative cycle present in the graph.

- 9) **True.** The sequence alignment algorithm can be used to find the longest common subsequence between two given sequences. x2
- 10) **True.** It is possible for a dynamic programming algorithm to have an exponential running time.
- 11) **False.** In a connected, directed graph with positive edge weights, the Bellman-Ford algorithm runs asymptotically faster than the Dijkstra algorithm.
- 12) **True.** There exist some problems that can be solved by dynamic programming, but cannot be solved by greedy algorithm.
- 13) **True.** The Floyd-Warshall algorithm is asymptotically faster than running the Bellman-Ford algorithm from each vertex.
- 14) **True.** If we have a dynamic programming algorithm with  $n^2$  subproblems, it is possible that the space usage could be  $O(n)$ .
- 15) **False.** In a dynamic programming solution, the space requirement is always at least as big as the number of unique sub problems.
- 16) **True.** Any problem that can be solved with a greedy algorithm can also be solved with dynamic programming.
- 17) **True.** If we have a dynamic programming algorithm with  $n^2$  subproblems, it is possible that the running time could be asymptotically strictly greater than  $\Theta(n^2)$ .
- 18) **False.** The dynamic programming solution (presented in class) to the 0/1 knapsack problem has a polynomial run time.
- 19) **True.** Bellman-Ford algorithm runs in strongly polynomial time.
- 20) **True.** In a graph with negative weight cycles, one such cycle can be found in  $O(nm)$  time where  $n$  is the number of vertices and  $m$  is the number of edges in the graph.
- 21) **True.** An algorithm runs in weakly polynomial time if the number of operations is bounded by a polynomial in the number of bits in the input, but not in the number of integers in the input.

### 13. 网络流

Ford-Fulkerson  $O(Cm)$  pseudo-polynomial

Scaled version of FF  $O(m^2 \lg C)$  weakly polynomial

Edmonds-Karp  $O(nm^2)$  strongly polynomial

Orlin + KTR  $O(nm)$  ~

Recently developed methods solve max flow in  
close to linear time WRT  $m$ .

approximation

C是所有capacity的和。

判断题：

- 1) **True.** The problem of deciding whether a given flow  $f$  of a given flow network  $G$  is maximum flow can be solved in linear time. X2
- 2) **True.** If you are given a maximum  $s - t$  flow in a graph then you can find a minimum  $s - t$  cut in time  $O(m)$ .
- 3) **True.** An edge that goes straight from  $s$  to  $t$  is always saturated when maximum  $s - t$  flow is reached.
- 4) **False.** In any maximum flow there are no cycles that carry positive flow. (A cycle  $\langle e_1, \dots, e_k \rangle$  carries positive flow if  $f(e_1) > 0, \dots, f(e_k) > 0$ .)
- 5) **True.** There always exists a maximum flow without cycles carrying positive flow.
- 6) **False.** In a directed graph with at most one edge between each pair of vertices, if we replace each directed edge by an undirected edge, the maximum flow value remains unchanged.
- 7) **True.** The Ford-Fulkerson algorithm finds a maximum flow of a unit-capacity flow network (all edges have unit capacity) with  $n$  vertices and  $m$  edges in  $O(mn)$  time.

- 8) **True.** The Ford-Fulkerson Algorithm finds a maximum flow of a unit-capacity flow network with  $n$  vertices and  $m$  edges in time  $O(mn)$ .
- 9) **False.** In a flow network, if maximum flow is unique then min cut must also be unique.
- 10) **False.** In a flow network, if min cut is unique then maximum flow must also be unique.
- 11) **True.** The Ford-Fulkerson algorithm is based on greedy.
- 12) **True.** A flow network with unique edge capacities may have several min cuts.
- 13) **True.** Given the min-cut, we can find the value of max flow in  $O(|E|)$ .
- 14) **True.** Maximum value of an  $s-t$  flow could be less than the capacity of a given  $s-t$  cut in a flow network.
- 15) **False.** Suppose the maximum  $(s, t)$ -flow of some graph has value  $f$ . Now we increase the capacity of every edge by 1. Then the maximum  $(s, t)$ -flow in this modified graph will have value at most  $f + 1$ .
- 16) **False.** There are no known polynomial-time algorithms to solve maximum flow.
- 17) **False.** If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.
- 18) **False.** In the scaled version of the Ford Fulkerson algorithm, choice of augmenting paths cannot affect the number of iterations.
- 19) **True.** The Ford-Fulkerson algorithm solves the maximum bipartite matching problem in polynomial time.
- 20) **True.** Given a solution to a max-flow problem, that includes the final residual graph  $G_f$ . We can verify in a linear time that the solution does indeed give a maximum flow.
- 21) **True.** In a flow network, a flow value is upper-bounded by a cut capacity.
- 22) **False.** In a flow network, a min-cut is always unique.
- 23) **False.** A maximum flow in an integer capacity graph must have an integer flow on each edge.
- 24) **False.** The residual graph of a maximum flow  $f$  can be strongly connected.
- 25) **True.** The Ford-Fulkerson algorithm can be used to efficiently compute a maximum matching of a given bipartite graph.
- 26) **False.** In successive iterations of the Ford-Fulkerson algorithm, the total flow passing through a vertex in the graph never decreases.
- 27) **True.** In a flow network  $G$ , if we increase the capacity of one edge by a positive amount  $x$  and we observe that the value of max flow also increases by  $x$ , then that edge must belong to every min-cut in  $G$ .

- 28) **True.** If we modify the Ford-Fulkerson algorithm by the following heuristic, then the time complexity of the resulting algorithm will be strongly polynomial: At each step, choose the augmenting path with fewest edges.
- 29) **False.** For any edge  $e$  that is part of a minimum cut in a flow network  $G$ , if we increase the capacity of that edge by any integer  $k > 1$ , then that edge will no longer be part of a minimum cut.
- 30) **True.** The scaled version of the Ford-Fulkerson algorithm can compute the maximum flow in a flow network in polynomial time.
- 31) **False.** Given a set of demands  $D = \{dv\}$  on a circulation network  $G(V,E)$ , if the total demand over  $V$  is zero, then  $G$  has a feasible circulation with respect to  $D$ .
- 32) **True.** In a flow network, the maximum value of an  $s-t$  flow could be less than the capacity of a given  $s-t$  cut in that network.
- 33) **True.** If  $f$  is a max  $s-t$  flow of a flow network  $G$  with source  $s$  and sink  $t$ , then the capacity of the min  $s-t$  cut in the residual graph  $G_f$  is 0.
- 34) **True.** Let  $G(V,E)$  be an arbitrary flow network, with a source  $s$ , a sink  $t$ . Given a flow  $f$  of maximum value in  $G$ , we can compute an  $s-t$  cut of minimum capacity in  $O(|E|)$  time.
- 35) **True.** The basic Ford-Fulkerson algorithm can be used to compute a maximum matching in a given bipartite graph in strongly polynomial time.
- 36) **False.** Let  $G$  be a weighted directed graph with exactly one source  $s$  and exactly one sink  $t$ . Let  $(A,B)$  be a maximum cut in  $G$ , that is,  $A$  and  $B$  are disjoint sets whose union is  $V$ ,  $s \in A$ ,  $t \in B$ , and the sum of the weights of all edges from  $A$  to  $B$  is the maximum for any two such sets. Now let  $H$  be the weighted directed graph obtained by adding 1 to the weight of each edge in  $G$ . Then  $(A,B)$  must still be a maximum cut in  $H$ .

## 14. P, NP, NPC问题

**Theorem 1:** If  $A \leq_p B$  and  $B \in \mathbf{P}$ , then  $A \in \mathbf{P}$ .

**Theorem 2:** If  $A \leq_p B$  and  $B \in \mathbf{NP}$ , then  $A \in \mathbf{NP}$ .

Intuitively, if  $A \leq_p B$ , then  $A$  is “not harder than”  $B$  and  $B$  is “at least as hard as”  $A$ .

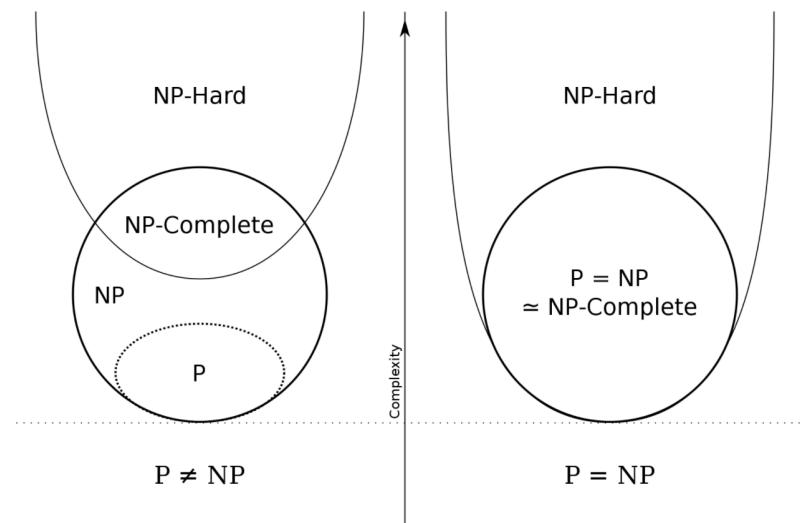
**Theorem:** If any  $\mathbf{NP}$ -complete language is in  $\mathbf{P}$ , then  $\mathbf{P} = \mathbf{NP}$ .

**Theorem:** If any  $\mathbf{NP}$ -complete language is not in  $\mathbf{P}$ , then  $\mathbf{P} \neq \mathbf{NP}$ .

**Theorem:** Let  $A$  and  $B$  be languages. If  $A \leq_P B$  and  $A$  is **NP-hard**, then  $B$  is **NP-hard**.

**Theorem:** Let  $A$  and  $B$  be languages where  $A \in \text{NPC}$  and  $B \in \text{NP}$ . If  $A \leq_P B$ , then  $B \in \text{NPC}$ .

- (1) If  $A \leq_P B$  and  $B \leq_P C$ , then  $A \leq_P C$ .
- (2) If  $A \leq_P B$  then  $\overline{A} \leq_P \overline{B}$ .
- (3) If  $A \leq_P B$  and  $B \in \text{NP}$ , then  $A \in \text{NP}$ .
- (4) If  $A \leq_P B$  and  $A \notin \text{NP}$ , then  $B \notin \text{NP}$ .
- (5) If  $A \leq_P B$  and  $B \in \text{P}$ , then  $A \in \text{P}$ .
- (6) If  $A \leq_P B$  and  $A \notin \text{P}$ , then  $B \notin \text{P}$ .



判断题：

- 1) **True.** If  $\text{NP} = \text{P}$ , then all problems in  $\text{NP}$  are  $\text{NP}$  hard. (the image above)
- 2) **True (but the answer is false?).** L1 can be reduced to L2 in Polynomial time and L2 is in  $\text{NP}$ , then L1 is in  $\text{NP}$ . (Theorem 2)
- 3) **False.** The simplex method solves Linear Programming in polynomial time.
- 4) **False.** Integer Programming is in  $\text{P}$ .
- 5) **False.** If a linear time algorithm is found for the traveling salesman problem, then every problem in  $\text{NP}$  can be solved in **linear** time.
- 6) **True.** If there exists a polynomial time 5-approximation algorithm for the general traveling salesman problem then 3-SAT can be solved in polynomial time

If we drop the assumption that the cost function  $c$  satisfies the triangle inequality, good approximate tours cannot be found in polynomial time unless  $P = NP$ .

Theorem 37.3

If  $P \neq NP$  and  $\rho \geq 1$ , there is no polynomial-time approximation algorithm with ratio bound  $\rho$  for the general traveling-salesman problem.

- 7) **Unknown?** A problem can be both NP-complete and undecidable.
- 8) **False.** If problem A is in P and  $A \leq_p B$  for some other problem B, then B is in P as well.
- 9) **True.** If there is a polynomial time algorithm for some problem in NP, then all problems in NP can be solved in polynomial time.
- 10) **True.** If a problem A is NP-hard and  $A \leq_p B$  for some other problem B, then B is NP-hard as well.
- 11) **Unknown.** It is proven that there is a problem that belongs in P but it is not NP-complete.  
**(P $\neq$ NP的时候是对的)**
- 12) **True.** If  $A \leq_p B \leq_p C$  and  $C \in NP$ , then  $A \in NP$ .
- 13) **False.** If  $A \leq_p B$  and  $B \in NP$  - Complete, then  $A \in NP$  - Complete
- 14) **True.** Let A and B be decision problems. If A is polynomial time reducible to B and B is in NP-Complete, then A is in NP.
- 15) **True.** Let ODD denote the problem of deciding if a given integer is odd. Then ODD is polynomial time reducible to 3-SAT.
- 16) **False.** Not every decision problem in P has a polynomial time certifier.
- 17) **True.** The set of all vertices in a graph is a vertex cover.
- 18) **False.** Given a graph  $G=(V, E)$  and an approximation algorithm that solves the vertex cover problem in G with an approximation ratio r, then this algorithm can also provide a solution to the independent set of G with the same approximation ratio r.
- 19) **False.** All the NP-hard problems are in NP.
- 20) **True.** If a problem can be reduced to linear programming in polynomial time then that problem is in P.
- 21) **False.** If we can prove that  $P \neq NP$ , then a problem  $A \in P$  does not belong to NP.
- 22) **True.** Linear programming problems can be solved in polynomial time
- 23) **True.** Consider problem A: given a flow network, find the maximum flow from a node s to a node t. problem A is in NP.
- 24) **False.** If a problem is not in P, then it must be in NP.
- 25) **False.** L1 can be reduced to L2 in Polynomial time and L1 is in NP, then L2 is in NP
- 26) **False.** If  $L_1 \leq_p L_2$ ,  $L_2 \leq_p L_1$ , and  $L_1, L_2 \in NP$ , then  $L_1$  and  $L_2$  are both NP-complete.
- 27) **False.** If  $L_1$  is NP-complete and  $L_1 \leq_p L_2$ , then  $L_2$  is NP-complete.

- 28) **False** If  $P \neq NP$  and if two decision problems A and B both in NP are polynomial time reducible to each other, then A and B are both in NP-Complete.
- 29) **False** If any two problems in NP are polynomial time reducible to each other then  $P = NP$ -Complete.

## 26, 28 29 都是一样的

- 30) **False**. If a problem is not solvable in polynomial time, it is in the NP-Complete class.
- 31) **True**. Linear programming is at least as hard as the Max Flow problem in a flow network.
- 32) **True**. If  $SAT \leq_p A$ , then A is NP-hard.
- 33) **False**. If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.
- 34) **True**. If  $P$  equals  $NP$ , then  $NP$  equals  $NP$ -complete
- 35) **False**. Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to Hamiltonian Cycle, we can conclude that X is NP-complete.
- 36) **False**. If  $P = NP$ , then all NP-Hard problems can be solved in Polynomial time.
- 37) **False**. The linear programming solution to the shortest path problem discussed in class can fail in the presence of negative cost edges.
- 38) **True**. Halting Problem is an NP-Hard problem.
- 39) **True**. Every decision problem in P has a polynomial time certifier.
- 40) **True**. Every problem in NP has a polynomial time certifier.
- 41) **False**. Every decision problem is in NP-complete.
- 42) **True**. An NP problem is NP-complete if 3-SAT reduces to it in polynomial time.
- 43) **False**. All Integer linear programming problems can be solved in polynomial time.
- 44) **False**. If the linear program is feasible and bounded, then there exists an optimal solution.
- 45) **True**. If  $X \leq_p Y$ , and X is NP-complete, then Y is NP-hard.
- 46) **False**. If  $X \leq_p Y$ , and X is NP-complete, then Y is NP-complete.
- 47) **False** If  $X \leq_p$  Integer Programming, then X is NP-hard.
- 48) **True** If  $X \leq_p$  Linear Programming, then X is in P.
- 49) **Unknown** 3-SAT cannot be solved in polynomial time.
- 50) **False**. Although the general Travelling Salesman Problem is NP-complete, in class, we presented a 2-approximation algorithm for it that runs in polynomial time.
- 51) **True**. If a problem is in P, it must also be in NP.
- 52) **Unknown**. If a problem is in NP, it must also be in P.
- 53) **True**. If a problem is NP-complete, it must also be in NP.

- 54) **True.** If a problem is NP-complete, it must also be NP-hard.
- 55) **False.** If a problem is not in P, it must be NP-complete.
- 56) **False.** If  $Y \leq_p X$  and we have a 2-approximation algorithm to solve X, we can use that to find a 2-approximation to Y.
- 57) **False.** Any NP-hard problem can be solved in time  $O(2^{\text{poly}(n)})$ , where n is the input size and  $\text{poly}(n)$  is a polynomial.
- 58) **True.** Any NP problem can be solved in time  $O(2^{\text{poly}(n)})$ , where n is the input size and  $\text{poly}(n)$  is a polynomial.
- 59) **True.** If 3-SAT  $\leq_p$  2-SAT, then P = NP.
- 60) **False.** Assuming  $P \neq NP$ , there can exist a polynomial-time approximation algorithm for the general Traveling Salesman Problem.
- 61) **False.** If problem X can be solved using dynamic programming, then X belongs to P.
- 62) **False.** All instances of linear programming have exactly one optimal solution.
- 63) **False.** Let  $Y \leq_p X$  and there exists a 2-approximation for X, then there must exist a 2-approximation for Y.
- 64) **True.** There is no known polynomial-time algorithm to solve an integer linear programming.
- 65) **True.** Every problem in P can be reduced to 3-SAT in polynomial time.
- 66) **False.** If there is a polynomial-time algorithm for 2-SAT, then every problem in NP has a polynomial-time algorithm (2-SAT本身就是P问题)
- 67) **False.** If A is in NP, and B is NP-complete, and  $A \leq_p B$  then A is NP-complete.
- 68) **False.** Given a problem B, if there exists an NP-complete problem that can be reduced to B in polynomial time, then B is NP-complete.
- 69) **False.** Let X be a decision problem. If we prove that X is in the class NP and give a poly-time reduction from X to 3-SAT, we can conclude that X is NP-complete.
- 70) **True.** If there is a polynomial time algorithm to solve problem A then A is in NP.
- 71) **True.** To prove that a problem X is NP-hard, it is sufficient to prove that SAT is polynomial time reducible to X.
- 72) **False.** If a problem Y is polynomial time reducible to X, then a problem X is polynomial time reducible to Y.
- 73) **True.** Every problem in NP can be solved in **polynomial** time by a **nondeterministic** Turing machine.
- 74) **True.** Every problem in NP can be solved in **exponential** time by a **deterministic** Turing machine. (注意和上一题的区别 )

- 75) **False.** A linear program with all integer coefficients and constants must have an integer optimum solution.
- 76) **True.** A linear program can have an infinite number of optimal solutions.
- 77) **False.** If problem A is polynomial-time reducible to a problem B (i.e.,  $A \leq_p B$ ), and B is NP-complete, then A must be NP-complete.
- 78) **False.** If problem A is polynomial-time reducible to a problem B (i.e.,  $A \leq_p B$ ), and A is NP-complete, then B must be NP-complete.
- 79) **False.** If  $P \neq NP$ , then the general optimization problem TRAVELING-SALESPERSON has a poly-time approximation algorithm with approximation factor 1.5.
- 80)

## 15. 近似算法 (Approximation Algorithms)

## 16. 线性规划 (Linear Programming)

### Other

- 1) **False.** Consider an undirected graph  $G = (V, E)$ . Suppose all edge weights are different. Then the longest edge cannot be in the minimum spanning tree.
- 2) **False.** Given a set of demands  $D = \{d_v\}$  on a directed graph  $G(V, E)$ , if the total demand over  $V$  is zero, then  $G$  has a feasible circulation with respect to  $D$ . (反过来是对的)
- 3) **True.** For a connected graph  $G$ , the BFS tree, DFS tree, and MST all have the same number of edges.
- 4) **False.** Dynamic programming sub-problems can overlap but divide and conquer sub-problems do not overlap, therefore these techniques cannot be combined in a single algorithm.
- 5) **True.** In a flow network, there always exists a maximum flow without cycles carrying positive flow.

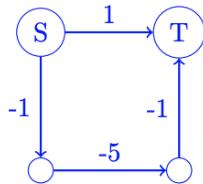
There always exists a maximum flow without cycles carrying positive flow.

#### *Solution*

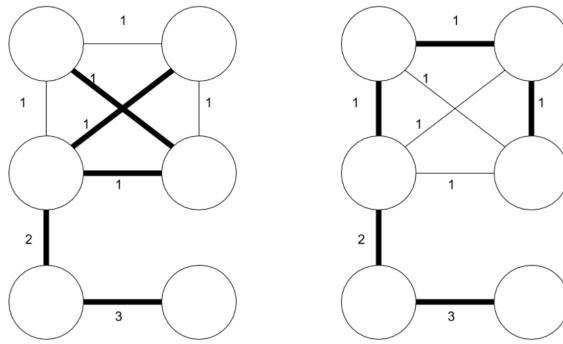
True. Let  $f$  be a maximum flow and let  $C$  be a cycle with positive flow. Let  $\delta = \min_{e \in C} f(e)$ . Reducing the flow of each edge in  $C$  by  $\delta$  maintains the value of the flow and sets the flow  $f(e)$  of at least one of the edges  $e \in C$  to zero.

- 6) **False.** In a directed graph with at most one edge between each pair of vertices, if we replace each directed edge by an undirected edge, the maximum flow value remains unchanged.

- 7) **False.** In the worst case, merge sort runs in  $O(n^2)$  time.  $O(n\log n)$
- 8) **True.** Fibonacci heaps can be used to make Dijkstra's algorithm run in  $O(|E| + |V| \log |V|)$  time on a graph  $G=(V,E)$ .
- 9) **False.** If some of the edge weights in a graph are negative, the shortest path from  $s$  to  $t$  can be obtained using Dijkstra's algorithm by first adding a large constant  $C$  to each edge weight, where  $C$  is chosen large enough that every resulting edge weight will be nonnegative.



- 10) **False.** A minimum weight edge in a graph  $G$  must be in all minimum spanning trees of  $G$ .  
(有可能有一个环里全是最小权值)
- 11) **True.** For a given problem with input size  $n$ , there must be some  $N$  that when  $n>N$ , a  $\Theta(n\log n)$  algorithm runs faster than a  $\Theta(n^2)$  algorithm
- 12) **False.** If two minimum spanning trees on the same graph only have 2 edges in common, then those two edges must be the lowest costs edges in the graph.



(如果边权是唯一的就是True)

- 13) **True.** In a network with source  $s$  and sink  $t$  where each edge capacity is a positive integer, there is always a max  $s-t$  flow where the flow assigned to each edge is an integer.
- 14) **True.** A minimum spanning tree of a connected undirected graph remains being a minimum spanning tree even if each edge weight is doubled.
- 15) **False.** A minimum spanning tree of a bipartite graph is not necessarily a bipartite graph.
- 16) **False.** Dijkstra's algorithm can always find the shortest path between two nodes in a graph as long as there is no negative cost cycle in the graph.

- 17) **False.** Given a binary max heap of size  $n$ , the complexity of finding the smallest number in the heap is  $O(\log n)$ . **O(n)**.
- 18) **False.** Given a weighted graph and two nodes, it is possible to list all shortest paths between these two nodes in polynomial time.
- 19) **True.** In the memory efficient implementation of Bellman-Ford, the number of iterations it takes to converge can vary depending on the order of nodes updated within an iteration
- 20) **False.** There is a feasible circulation with demands  $\{d_v\}$  if  $\sum_v d_v = 0$ .
- 21) **False.** If all capacities in a flow network are integers, then every maximum flow in the network is such that flow value on each edge is an integer.
- 22) **False.** In a dynamic programming formulation, the sub-problems must be mutually independent.
- 23) **True.** In the final residual graph constructed during the execution of the Ford–Fulkerson Algorithm, there's no path from sink to source.
- 24) **False.** In a flow network, if all edge capacities are distinct, then the max flow of this network is unique.
- 25) **True.** To find the minimum element in a max heap of  $n$  elements, it takes  $O(n)$  time. **17)**
- 26) **True.** Let  $T$  be a spanning tree of graph  $G(V, E)$ , let  $k$  be the number of edges in  $T$ , then  $k=O(V)$
- 27) **True.** Given  $n$  numbers, it takes  $O(n)$  time to construct a binary min heap.
- 28) **True.** Kruskal's algorithm for finding the MST works with positive and negative edge weights.
- 29) **False.** Breadth first search is an example of a divide-and-conquer algorithm.
- 30) **False.** The maximum weight edge cannot be part of any minimum spanning tree for a graph that has cycles.
- 31) **False.???** In a flow network whose edges have capacity 1, the maximum flow always corresponds to the maximum degree of a vertex in the network.
- 32) **True.** Bellman-Ford algorithm is more suitable for distributed processing than Dijkstra's algorithm is.
- 33) **True.** Suppose  $f(n)=8f(n/2)+56$ , then  $f(n)=\theta(n^3)$ .
- 34) **False.** Suppose  $f(n)=f(n/2)+56$ , then  $f(n)=\theta(n)$ .
- 35) **True.** The recurrence  $T(n)=2T(n/2)+3n$ , has solution  $T(n)=\theta(n\log(n^2))$ .
- 36) **False.** On a connected, directed graph with only positive edge weights, Bellman-Ford runs asymptotically as fast as Dijkstra.

- 37) **True.** If you are given a maximum s-t flow in a graph then you can find a minimum s-t cut in time  $O(m)$  where  $m$  is the number of the edges in the graph.
- 38) **False.** A graph with non-unique edge weights will have at least two minimum spanning trees
- 39) **True.** If all edge capacities in a flow network are integer multiples of 7, then the maximum value of flow is a multiple of 7.
- 40) **False.** Let  $T$  be a complete binary tree with  $n$  nodes. Finding a path from the root of  $T$  to a given vertex  $v \in T$  using breadth-first search takes  $O(\log n)$  time.
- 41) **False.** In a flow network, if we increase the capacity of an edge that happens to be on a minimum cut, this will increase the max flow in the network.
- 42) **False.** If the capacity of every arc is odd, then there is a maximum flow in which the flow on each arc is odd.
- 43) **True.** If the capacity of every arc is even, then the value of the maximum flow must be even.
- 44) **True.** If the capacity of every arc is even, then there is a maximum flow in which the flow on each arc is even.
- 45) **False.** If the capacity of every arc is odd, then the value of the maximum flow must be odd.
- 46) **True.** If the edge weights of a weighted graph are doubled, then the number of minimum spanning trees of the graph remains unchanged
- 47) **False.** In a divide and conquer solution, the sub-problems are disjoint and are of the same size.
- 48) **False.** If all edges in a graph have capacity 1, then Ford-Fulkerson runs in linear time.  
 $O(m^2 + mn)$
- 49) **False.** Let  $T$  be a minimum spanning tree of  $G$ . Then, for any pair of vertices  $s$  and  $t$ , the shortest s-t path in  $G$  is the path in  $T$ .
- 50) **True.** If the running time of a divide-and-conquer algorithm satisfies the recurrence  $T(n) = 3 T(n/2) + \Theta(n^2)$ , then  $T(n) = \Theta(n^2)$ .
- 51) **False.** Suppose we have a data structure where the amortized running time of Insert and Delete is  $O(\lg n)$ . Then in any sequence of  $2n$  calls to Insert and Delete, the worst-case running time for the  $n$ th call is  $O(\lg n)$ .
- 52) **True.** If graph  $G$  has no cycles, then the independent set problem in  $G$  can be solved in polynomial time.
- 53) **False.** Breadth first search is an example of a divide-and-conquer algorithm.

- 54) **True.** Memoization requires memory space which is linear in size with respect to the number of unique sub-problems.
- 55) **False.** The smallest element in a binary max-heap of size  $n$  can be found with at most  $n/2$  comparisons.
- 56) **True.** If the edge weights of a weighted graph are doubled, then the number of minimum spanning trees of the graph remains unchanged.
- 57) **True.** A greedy algorithm follows the heuristic of making a locally optimum choice at each stage with the hope of finding the global optimum.
- 58) **True.** For a given problem with input size  $n$ , there must be some  $N$  that when  $n > N$ , a  $\Theta(n \log n)$  algorithm runs faster than a  $\Theta(n^2)$  algorithm.
- 59) **False.** If in a flow network all edge capacities are distinct, then there exists a unique min-cut.
- 60) **True.** Given a minimum cut, we could find the maximum flow value in  $O(E)$  time.
- 61) **False.** Let  $(S, V-S)$  be a minimum  $(s, t)$ -cut in the network flow graph  $G$ . Let  $(u, v)$  be an edge that crosses the cut in the forward direction, i.e.,  $u \in S$  and  $v \in V-S$ . Then increasing the capacity of the edge  $(u, v)$  necessarily increases the maximum flow of  $G$ .
- 62) **True.** If all edge weights are 1, 2, or 3, the shortest path problem can be solved in linear time.
- 63) **True.** Suppose  $G$  is a graph with  $n$  vertices and  $n^{1.5}$  edges, represented in adjacency list representation. Then depth-first search in  $G$  runs in  $O(n^{1.5})$  time.
- 64) **False.** The weight of a minimum spanning tree in a positively weighted undirected graph is always less than the total weight of a minimum spanning path (Hamiltonian Path with lowest weight) of the graph.
- 65) **True.** If an undirected connected graph has the property that between any two nodes  $u$  and  $v$ , there is exactly one path between  $u$  and  $v$ , then that graph is a tree.
- 66) **True.** Suppose that a divide and conquer algorithm reduces an instance of size  $n$  to four instances of size  $n/5$  and spends  $\Theta(n)$  time in the divide and combine steps. The algorithm runs in  $\Theta(n)$  time.
- 67) **False.** An integer  $N$  is given in binary. An algorithm that runs in time  $O(\sqrt{N})$  to find the largest prime factor of  $N$  is considered to be a polynomial-time algorithm. Ex: Prime factorization of 84:  $2 \times 2 \times 3 \times 7$ , so the largest prime factor of 84 is 7
- 68) **False.** Let  $A$  be an algorithm that operates on a list of  $n$  objects, where  $n$  is a power of two.  $A$  spends  $\Theta(n^2)$  time dividing its input list into two equal pieces and selecting one of

the two pieces. It then calls itself recursively on that list of  $n/2$  elements. Then A's running time on a list of  $n$  elements is  $O(n)$ .

- 69) **False.** A pseudo-polynomial time algorithm is always slower than a polynomial time algorithm.
- 70) **False.** In a dynamic programming formulation, the sub-problems must be non-overlapping.
- 71) **True.** A spanning tree of a given undirected, connected graph  $G = (V, E)$  can be found in  $O(E)$  time.
- 72) **True.** Ford-Fulkerson can return a zero maximum flow for flow networks with non-zero capacities.
- 73) **True.** If  $f(n) = \Theta(g(n))$  and  $g(n) = \Theta(h(n))$ , then  $h(n) = \Theta(f(n))$
- 74) **True.** There is a polynomial-time solution for the 0/1 Knapsack problem **if all items have the same weight but different values.**
- 75) **False.** If there are negative cost edges in a graph but no negative cost cycles, Dijkstra's algorithm still runs correctly.
- 76) **True.** Suppose that a divide and conquer algorithm reduces an instance of size  $n$  into 4 instances of size  $n/5$  and spends  $\Theta(n)$  time in the conquer steps. The algorithm runs in  $\Theta(n)$  time.
- 77) **False.** Let  $M$  be a spanning tree of a weighted graph  $G=(V, E)$ . The path in  $M$  between any two vertices must be a shortest path in  $G$ .
- 78) **True.** Suppose that a Las Vegas algorithm has expected running time  $\Theta(n)$  on inputs of size  $n$ . Then there may still be an input on which it runs in time  $\Omega(n^2)$ .
- 79) **False.** The total amortized cost of a sequence of  $n$  operations gives a lower bound on the total actual cost of the sequence.
- 80) **False.** The maximum flow problem can be efficiently solved by dynamic programming.
- 81) **False.** An undirected graph is said to be Hamiltonian if it has a cycle containing all the vertices. Based on this definition, is the following true or false? Any DFS tree on a Hamiltonian graph must have depth  $V-1$ .
- 82) **True.** At the termination of the Bellman-Ford algorithm, even if the graph has a negative length cycle, a correct shortest path is found for a vertex for which shortest path is well-defined.
- 83) **False.** The main difference between divide and conquer and dynamic programming is that divide and conquer solves problems in a top-down manner whereas dynamic programming does this bottom-up.

- 84) **True.** Flow  $f$  is maximum flow if and only if there are no augmenting paths.
- 85) **False.** We currently only have a weakly polynomial time solution for the 0/1 knapsack problem, but not a strongly polynomial time solution.
- 86) **True.** Suppose that all edge capacities  $c(e)$  is a multiple of  $\alpha$ . Then the value of the maximum flow is also a multiple of  $\alpha$
- 87) **False.** Suppose that all edge capacities  $c(e)$  is a multiple of  $\alpha$ . In a maximum flow  $f$ , the flow  $f(e)$  on edge  $e$  is always a multiple of  $\alpha$

## Fall 2019

[ TRUE/FALSE/UNKNOWN ]

Let ODD denote the problem of deciding if a given integer is odd. Then ODD is polynomial time reducible to Vertex Cover.

[ TRUE/FALSE ]

If we find a quadratic time solution to an NP-Complete problem, then all NP- Complete problems can be solved in quadratic time. 只能保证poly-time

[ TRUE/FALSE/UNKNOWN ]

Independent Set problem has a polynomial run time on certain types of graphs.

[ TRUE/FALSE ]

Consider two decision problems Q1, Q2 such that Q1 reduces in polynomial time to Hamiltonian Cycle and Hamiltonian Cycle reduces in polynomial time to Q2. Then it is possible for both Q1 and Q2 to be in NP.

[ TRUE/FALSE ]

The problem of determining whether there exists a cycle in an undirected graph is in NP. 所有p问题都是np问题

[ TRUE/FALSE ]

If the edge weights in a connected undirected graph  $G$  are NOT all distinct, then there will be more than one minimum spanning tree in  $G$ .

[ TRUE/FALSE ]

Dijkstra's algorithm works correctly on graphs with negative-weight edges, as long as there are no negative-weight cycles.

[ TRUE/FALSE/UNKNOWN ]

If Integer Linear Programming  $\leq_p A$ , then  $A$  cannot be solved in polynomial time.

[ TRUE/FALSE ]

Suppose that the capacities of the edges of an s-t flow network are integers that are all evenly divisible by 3. (E.g., 3, 6, 9, 12, etc.) Then there exists a maximum flow, such that the flow on each edge is also evenly divisible by 3.

[ TRUE/FALSE ]

Suppose that G is a directed, acyclic graph and has a topological ordering with v1 the first node in the ordering and vn the last node in the ordering. Then there is a path in G from v1 to vn . (拓扑顺序不代表一定有路径 )

## Fall 2015

For the next four questions consider a flow network G, increase the capacity of an edge e by an amount  $x > 0$  to obtain Network G'. Let  $f, f'$  denote the value of max flow in G, G'. Then,

- a) [ TRUE/FALSE]  $f' - f$  is either 0 or  $x$ .
- b) [ TRUE/FALSE] If there is a min-cut in G containing e and  $f' > f$ , then there's a min-cut in G' containing e.
- c) [ TRUE/FALSE] If  $f' = f$ , there is no min-cut in G' containing e
- d) [ TRUE/FALSE] If  $f' = f$ , there is no min-cut in G containing e.

[ TRUE/FALSE/UNKNOWN ]

Let S denote the set of problems reducible to problem X, where X is NP-hard. Then every problem in S must be in NP.

[ TRUE/FALSE/UNKNOWN ]

A general 3-SAT problem cannot be solved in polynomial time.

[ TRUE/FALSE/UNKNOWN ]

If a general integer programming optimization problem can be reduced in polynomial time to a linear programming problem, then P = NP.

[ TRUE/FALSE/UNKNOWN ]

If a problem is both in NP and NP-hard, then this problem is NP-complete.

[ TRUE/FALSE ]

A matching in a bipartite graph  $G=(V, E)$  can be tested in  $O(|V|+|E|)$  time to determine if it is a maximum size matching.

[ TRUE/FALSE ]

For a graph with all edges having distinct weights there is a unique MST