Chapter 1 - 1.1稳定匹配

定义(一对一匹配)

输入：两个集合 $A = a1, \cdots, an, B = b1, \cdots, bn$，以及每个元素对另一集合所有元素的偏好表。

完美匹配：$f : A \to B$是双射

不稳定对：未出现在匹配中，且双方对对方的偏好程度均高于当前匹配的偏好程度。例如匹配中存在$(a1, b1), (a2, b2)$，但$a1$相比$b1$更偏好$b2$，且$b2$相比$a2$更偏好$a1$，那么$(a1, b2)$被称为不稳定对。

稳定匹配：不存在不稳定对的完美匹配

Gale-Shapley 算法

保证可以找到一个稳定匹配的算法，此处以婚配为例。

```
Initialize each person to be free.
while (some man is free and hasn't proposed to every woman)
{
    Choose such a man m
    w = 1st woman on m's list to whom m has not yet proposed
    if (w is free)
        assign m and w to be engaged
    else if (w prefers m to her fiancé m')
        assign m and w to be engaged, and m' to be free
    else
        w rejects m
    return the set S of engaged pairs
}
```

GS算法 - 正确性证明

Observation.

1. 主动方按照偏好表降序提出匹配请求
2. 被动方一旦匹配，就不会回到未匹配状态，只会更换到更偏好的匹配

算法一定终止

Claim. 算法在至多n^2轮while循环后终止

Proof. 每次while循环，主动方会向一个新的被动方提出匹配请求，两个大小为n的集合至多有n^2种匹配，因此算法复杂度$O(n^2)$。

匹配的完美性

Claim. 所有元素在算法终止后都有匹配

Proof. 反证法：假如某个主动方未被匹配，那么一定存在一个未被匹配的被动方。由Observation 2，该被动方一定从未收到匹配请求。但算法终止时主动方已对所有被动方进行了请求且仍未被匹配，引出矛盾。

匹配的稳定性

Claim. 结果中不存在不稳定对

Proof. 反证法：

记结果匹配为$S^\star$，其中存在配对$(a,y),(x,b)$。

假设存在不稳定对(a,b)，即a相比y更偏好b，而b相比x更偏好a

1. a未向b发起匹配请求

   说明a相比b更偏好y，引出矛盾

2. a向b发起过匹配请求

   b拒绝了a，当即 或是 过了一会儿

   说明b相比a更偏好x，引出矛盾

两种情形均矛盾，因此不存在不稳定对。

算法实现

全局维护一个未匹配的主动方集合，每次选取其中的主动方进行匹配。

对于主动方，维护一个偏好值降序的队列，依次进行匹配请求。

对于被动方，维护一个从主动方到偏好值的映射，用于比对当前匹配是否更优，如有替换则将前任放回全局未匹配主动方集合。

主被动方利益分析

有效对象：如果某个稳定匹配中存在(a,b)，那么a,b互为有效对象。

主动方最优性

Claim. GS算法得到的解$S^\star$对于主动方是最优的(每个主动方得到最优有效对象)

Proof. 反证法：

假如不是主动方最优的，那么一定有某个主动方未与其最优有效对象匹配。

由于主动方是降序请求的，因此一定是被其最优有效对象拒绝了。

令首个这样的主动方为a，其最优有效对象为b。在b拒绝a时，一定更偏好于某个主动方x。由于a是首个被最优有效对象拒绝的主动方，所以x此时没有被任何有效对象拒绝过，即b是x的最优有效对象。

随后，由于a的最优有效对象是b，我们找到包含(a,b)的稳定匹配S，S包含(x,y)。根据上述分析，b相比a更偏好x，而x相比任何其他有效对象更偏好b，因此(x,b)是匹配S的不稳定对，引出矛盾。

被动方最劣性

Claim. GS算法得到的解$S^\star$对于被动方是最劣的(每个主动方得到最劣有效对象)

Proof. 反证法：

假如不是被动方最劣的，那么一定有某个被动方未与其最劣有效对象匹配。

假如$S^\star$包含(a,b)，但a并不是b的最劣有效对象，假设其最劣有效对象为x。

我们找到包含(x,b)的稳定匹配S，S包含(a,y)。b相比x更偏好a，由主动方最优性，b一定是a的最优有效对象，即a相比y更偏好b，因此(a,b)是匹配S的不稳定对，引出矛盾。

一对多匹配

一对多匹配中，集合A中的一个元素可以与集合B中的多个元素匹配，此处将A中元素作为主动方进行讨论。

不稳定对(a,b)满足以下所有条件：

- a,b 相互是可接受的
- 主动方a仍然有匹配容量，或者a相比其已经匹配的所有被动方元素更偏好b
- 被动方b未被匹配，或者b相比其已经匹配的主动方更偏好a

扩展GS算法

将具有容量的一方作为被动方，全局维护未匹配主动方的集合。

对于每个主动方，维护一个偏好值降序列表，依次进行匹配请求。

对于每个被动方，维护一个长度为容量的优先队列，存放当前被匹配的所有被动方。如果有更偏好的主动方请求匹配，将队首(偏好值最低)的主动方放回未匹配集合。

扩展GS算法的正确性和利益分析与上文所述类似，证明可参考Assignment 1。

1.2五个典型问题

区间调度(interval scheduling)　$O(nlogn)$ greedy algorithm

输入：包含开始时间和结束时间的任务集合。
目标： 找到互不冲突任务的最大子集，任务之间不重叠。

加权区间调度　$O(nlogn)$ dynamic programming algorithm

输入: 包含开始时间、结束时间和权重的任务集合。
目标: 找到相互兼容任务的最大权重子集。

二分匹配（Bipartite Matching$O(n^k)$ max-flow based algorithm

输入：二分图

目标：寻找最大基数匹配（cardinality matching）

独立集（independent set）　NP-complete

输入：图　目标：寻找最大基数独立集（节点子集，使得其中任意两个节点之间不存在边连接）

竞争性设施选址（competitive facility location）　PSPACE-complete

输入：带权重的图，每个节点都有权重。
游戏：两名竞争者轮流选择节点，如果某个节点的任何邻居已被选择，则不允许选择该节点。
目标：选择权重最大的节点子集。（独立集）

Chapter2 算法分析

从四个方面衡量一个算法：完整性，最优性，时间复杂度，空间复杂度

运行时间分析

Polynomial-Time（多项式时间）e.g. BruteForce

最坏运行时间：给定输入数据规模，在所有不同的输入中，算法最大运行时间

平均运行时间：给定输入数据规模，对于随机的一个输入，算法的运行时间

最坏运行时间为多项式复杂度的算法被认为是高效的

存在一些例外情况

- 有些多项式时间算法由于常数或指数过大，在实际中可能无用。
- 有些指数时间算法在实际中表现良好，因为最坏情况很少发生。

Asymptotic Order of Growth渐进增长阶

Upper bounds. T(n) is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

Lower bounds. T(n) is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

Tight bounds. T(n) is $\Theta(f(n))$ if T(n) is both $O(f(n))$ and $\Omega(f(n))$.

Ex: $T(n) = 32n^2 + 17n + 32$.

- T(n) is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- T(n) is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

一些性质：传递性，可加性，可乘性

---

1. 多项式（Polynomials）

- **定义**：一个多项式表达式为：

  $$a_0 + a_1 n + \cdots + a_d n^d$$

  其中 $a_d > 0$。
- **渐进表示**：如果 $a_d > 0$，则该多项式的时间复杂度为：

$$\Theta(n^d)$$

这意味着最高次项 n^d 决定了多项式的增长速度。

---

2. 多项式时间（Polynomial Time）

- **定义**：算法的运行时间为 $O(n^d)$，其中 d 是一个与输入大小 n 无关的常数。
- **解释**：多项式时间算法的运行时间随着输入规模 n 的增加而呈多项式增长，例如 $O(n)$、$O(n^2)$、$O(n^3)$ 等。

---

3. 对数（Logarithms）

(1) 对数的基无关性

- **性质**：对于任意常数 $a, b > 0$，有：

$$O(\log_a n) = O(\log_b n)$$

- **原因**：对数的换底公式为：

$$\log_a n = \frac{\log_b n}{\log_b a}$$

因此，不同基的对数在大O表示法中是等价的，可以忽略基的差异。

(2) 对数的增长速度

- **性质**：对于任意 $x > 0$，有：

$$\log n = O(n^x)$$

- **解释**：对数函数 $\log n$ 的增长速度比任何正幂次多项式 n^x 都慢。换句话说，对数函数的增长非常缓慢。

---

4. 指数（Exponentials）

(1) 指数与多项式的比较

- **性质**：对于任意 $r > 1$ 和任意 $d > 0$，有：

$$n^d = O(r^n)$$

- **解释**：指数函数 r^n 的增长速度比任何多项式 n^d 都快。换句话说，指数函数的增长非常迅速。

(2) 总结

- **对数 vs. 多项式**：对数函数的增长速度比任何多项式都慢。
- **指数 vs. 多项式**：指数函数的增长速度比任何多项式都快。

常见算法时间复杂度

类型	时间复杂度	应用
常数(Constant)	$O(1)$	打表
对数(Logarithmic)	$O(logn)$	二分查找
线性(Linear)	$O(n)$	求极值
线性算数(Linearithmic)	$O(nlogn)$	归并排序，堆排
平方(Quadratic)	$O(n^2)$	暴力最近点对
立方(Cubic)	$O(n^3)$	集合不相容
指数(Exponential)	$O(c^n)$	暴力最大独立集

Chapter3 图

一些基本的概念：

无向图 G（V,E）

邻接矩阵Adjacency matrix. $n \times n$ matrix with $A_{uv} = 1$ if(u, v) is an edge.

邻接表 Adjacency list. Node indexed array of lists：

- 空间与$m + n$成正比
- Checking if (u, v) is an edge takes $O(deg(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time

邻接表与邻接矩阵的相对优势

比较内容	胜者
测试 (x, y) 是否在图中更快?	邻接矩阵
查找顶点度数更快?	邻接表
在稀疏图中占用更少内存?	邻接表（$m + n$） vs. （n^2）
在稠密图中占用更少内存?	邻接矩阵（略胜一筹）
边的插入或删除?	邻接矩阵 $O(1)$ vs. 邻接表 $O(d)$
遍历图更快?	邻接表 $\Theta(m + n)$ vs. 邻接矩阵 $\Theta(n^2)$
对大多数问题更好?	邻接表

路径的定义，环的定义，树的定义，有根树的定义，性质（n个节点，n-1条边，没有环）。

图的遍历：

- s-t connectivity problem： Given two node s and t, is there a path between s and t?
- s-t shortest path problem： what is the length of the shortest path between s and t?
- review of BFS (上面这两个问题都要用BFS解答) 广度优先搜索

```
BFS(Graph, start):
    // 初始化
    for each vertex u in Graph:
        u.color = WHITE   // 白色表示未访问
        u.distance = ∞    // 到起点的距离初始化为无穷大
        u.parent = NIL    // 父节点初始化为空

    start.color = GRAY    // 灰色表示即将被探索
    start.distance = 0
    start.parent = NIL

    create a queue Q
    enqueue(Q, start)     // 将起点加入队列

    while Q is not empty:
        u = dequeue(Q)    // 取出队首元素
        for each v in adjacency list of u:
            if v.color == WHITE:  // 如果该邻居未访问过
                v.color = GRAY
                v.distance = u.distance + 1
                v.parent = u
                enqueue(Q, v)     // 加入队列等待访问
        u.color = BLACK     // 黑色表示完全访问完毕

    return all visited nodes with their distances and parents
```

定理（Theorem）

- 当图以邻接表的形式给出时，BFS 实现的时间复杂度为 $O(m+n)$。
- n 表示图中的顶点数,m 表示图中的边数。

证明（Proof）

(1) 容易证明 $O(n^2)$ 时间复杂度

证明步骤：

- **最多 n 个列表 $L[i]$：**
  - 在 BFS 的实现中，通常会使用一个队列来存储待访问的节点。每个节点最多会被加入队列一次，因此最多会有 n 个节点被处理。
- **每个节点最多出现在一个列表中；for 循环运行次数 $\leq n$：**
  - 每个节点最多被访问一次，因此 for 循环的总运行次数不会超过 n 次。

- **当我们考虑节点 u 时，有 $\leq n$ 条关联边 (u, v)，并且处理每条边的时间为 $O(1)$：**
  - 对于每个节点 u，我们需要检查其所有相邻节点。在最坏情况下，每个节点最多有 n 条边（即完全图的情况），因此处理每条边的时间为 $O(1)$。

---

(2) 实际上运行时间为 $O(m+n)$

证明步骤：

- **当我们考虑节点 u 时，有 $\deg(u)$ 条关联边 (u, v)：**
  - 对于每个节点 u，我们只需要检查与其直接相连的边，即 $\deg(u)$ 条边。这比前面提到的 n 条边要少得多，特别是在稀疏图中。
- **处理边的总时间为 $\sum_{u \in V} \deg(u) = 2m$：**
  - 图中所有节点度数之和等于边数的两倍（因为每条边连接两个顶点）。因此，处理所有边的时间为：

$$\sum_{u \in V} \deg(u) = 2m$$

  - 这里的关键点是：每条边 (u, v) 被计算了两次——一次在 $\deg(u)$ 中，一次在 $\deg(v)$ 中。

- connected component

3.4检测二分性

二分图（每条边的两端可以涂成两种不同颜色）有点类似二叉树

一些引理（Lemma）：

1. 若图 G 为二分图，则其不包含奇数长度的环。
2. 设 G 是一个连通图，且 L0, ..., Lk 是由 BFS 从节点 s 开始生成的图的层。以下情况中恰好有一种成立。
   (i) 图 G 中没有边连接同一层的两个节点，且 G 是二分图。
   (ii) 图 G 中有边连接同一层的两个节点，且 G 包含一个奇数长度的环（因此不是二分图）。

   （可以进行证明）

推论 Corollary： A graph G is bipartite iff it contain no odd length cycle

3.5有向图中的连通性

有向图 Edge (u, v) goes from node u to node v

强连通：

- Def：Node u and v are mutually reachable if there is a path from u to v and also a path from v to u.
- Def： A graph is strongly connected if every pair of nodes is mutually reachable
- Lemma： Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node

判断强连通的算法：选一个节点s,从节点s开始跑BFS,再将图反向跑一边BFS，如果都跑到了——>returen true

3.6 DAGs和拓扑排序

一些引理（Lemma）：

1. ### If G has a topological order, then G is a DAG.

   proof反证法：

   1. 假设图 *G* 有一个拓扑排序 $v_1, \ldots, v_n$，并且 *G* 中也存在一个有向环 *C* 。让我们看看会发生什么。
   2. 设v_i是有向环 *C* 中索引最小的节点，设 v_j 是 v_i 的前驱节点，因此 (v_j, v_i) 是一条边。
   3. 根据我们对 *i* 的选择，我们有 *i<j* 。
   4. 另一方面，由于 (v_j, v_i) 是一条边，并且 $v_1, \ldots, v_n$ 是一个拓扑排序，我们必须有 *j<i* ，与假设矛盾。

2. ### If G is a DAG, then G has a node with no incoming edges

   proof反证法：

   1. 假设 *G* 是一个有向无环图（DAG），并且每个节点至少有一条入边。让我们看看会发生什么。
   2. 选择任意一个节点 *v* ，并从 *v* 开始沿着边向后遍历。由于 *v* 至少有一条入边 (*u*,*v*) ，我们能向后走到 *u* 。
   3. 然后，由于 *u* 至少有一条入边 (*x*,*u*) ，我们可以继续向后走到 *x* 。
   4. 不断重复这个过程，直到我们访问到某个节点 *w* 两次。
   5. 让 *C* 表示在连续两次访问 *w* 之间遇到的节点序列。*C* 形成了一个环。

3. ### If G is a DAG, then G has a topological ordering

   proof数归

   1. **基本情况**： 当 *n*=1 时，命题为真。
   2. 归纳步骤：
      - 给定一个有 *n*>1 个节点的 DAG，找到一个没有入边的节点 *v* 。
      - *G*−{*v*} 仍然是一个 DAG，因为删除 *v* 不会创建环。
      - 根据归纳假设，*G*−{*v*} 有一个拓扑排序。
      - 将 *v* 放在拓扑排序的最前面，然后按拓扑顺序追加 *G*−{*v*} 中的节点。这是有效的，因为 *v* 没有入边。

Theorem: Algorithm finds a topological order in O(m + n) time

计算 G的拓扑排序的方法

1. 找到一个没有入边的节点 *v* ，并将其排在第一位。
2. 从 *G* 中删除 *v* 。
3. 递归地计算 *G*−{*v*} 的拓扑排序，并将该排序追加到 *v* 之后。

Chapter4 贪心

区间调度（Interval scheduling）

问题：工作用一系列区间表示，工作不可并行，每个工作等权，最大化截止时间前完成的工作数。

贪心模板：按某种自然顺序考虑任务。只要每个任务与已分配的任务兼容，就分配它。

贪心算法：按照DDL升序排序，贪心地选择当前尚未错过开始时间且DDL最近的一个完成。

Theorem. 贪心算法是最优的

Proof. 反证法：假如贪心算法结果不是最优，令贪心结果为i_n，最优结果为j_m，且两种方案的前r项相同。我们在所有最优解中，找到一个j_m使得其和贪心方案的重叠部分最多，即r最大。

由贪心策略，i_r+1的结束时间一定不会晚于j_r+1，那么我们将i_r+1替换j_m方案中的j_r+1，依然会得到一个最优的可行解，但是此时的重叠部分长度为$r+1$，引出矛盾。

区间划分(Interval partitioning)

问题：课程用一系列区间表示，课程不可在同一教室并行，求安排下所有课程所需的最小教室数量。

贪心算法：按照开课时间升序排列，申请的所有教室按照最后一堂课的结束时间放进优先队列，对于每节课，看上节下课最早的教室能不能放下，能的话就放，如果放不下那就新开一个教室加入队列。（复杂度：$O(nlogn)$）

Theorem. 贪心算法是最优的

记贪心算法申请的教室数为d

第d个教室是因为贪心算法在规划第j节课时，前d−1个教室都放不下了，即该d−1个教室的最后一堂课的下课时间都比j的上课时间晚。

由于贪心按上课时间升序排列课程，所以这d−1个教室中的课程开始时间都比j的开始时间要晚，因此在第j节课开课后有d节课在同时上课。

根据题意，要能支持d节课同时进行至少需要d个教室，即不存在比贪心解更优的解。

最小化迟到(scheduling to minimize lateness)

问题：给定工作耗时和DDL，同时只能做一件事，迟到值设定为完成时间与DDL之间的差值，最小化最大的迟到值。

贪心算法：DDL越早越先做

Observation. 存在没有任何空闲间隔(idle time)的最优解，贪心解也没有任何空闲间隔

Observation. 贪心调度没有逆序对

- **贪心调度**：
  - 贪心算法通常按照某种局部最优策略进行调度，例如按截止时间升序安排任务。
  - 在最小化延迟问题中，贪心算法通常会优先安排截止时间最早的任务，以确保尽可能少的任务延迟。
- **结论**：
  - 贪心调度方案不会产生逆序对。这是因为贪心算法总是按照任务的截止时间升序安排任务，从而避免了任务编号较小但被安排在后面的情况。

Observation. 对于没有空闲间隔的解，如果存在两个任务，DDL早的后完成，DDL晚的先完成，则称为一个逆序对(inversion)。如果一个解有逆序对，那一定有一对连续规划的逆序对。

- **解释**：
  - 假设调度方案中有逆序对 (*i*,*j*) ，即 *i*<*j* 但任务 *j* 被安排在任务 *i* 之前。
  - 如果任务 *i* 和任务 *j* 不是连续安排的，那么中间可能存在其他任务。在这种情况下，可以通过调整任务 *i* 和任务 *j* 的顺序，消除逆序对，同时保持调度的完整性。
  - 因此，如果有逆序对，必然存在一对逆序的任务被连续安排。

Claim. 交换逆序对不会使最大迟到值增加。

Proof. 记交换前的迟到值为l，交换后为$l\prime$，工作完成时间为f，截止时间为d。

- 对于$k \neq i, j,\ lk\prime = lk$
- $li\prime \leq li$
- $lj\prime = max\{0, fj - dj\} = max\{0, fi - dj\} \leq max\{0, fi - di\} = li$

Theorem. 贪心解S是最优的

Proof. 令$S^\star$为一个没有空闲间隔且逆序对数量最少的最优解。

如果$S^\star$没有逆序对，那么$S = S^\star$;

如果有逆序对，记相邻逆序对为(i, j)。交换i, j不会得到更劣解，但是将逆序对数量减少了1，引出矛盾。

4.3最优高速缓存

问题：缓存大小 k，请求序列 $R = (r1, r2, r3 \cdots,\ r_n)$,要求最小化缺页(Page Fault)次数。

Bellady算法：当发生缺页时，如果缓存已满，选择将来最久不会被访问的页面(Farthest-in-furture) 进行替换。
(算法和定理是直观的，证明subtle,所以应该不考证明感觉)

claim: 任何未优化的调度 S 都可以被转换为一个优化后的调度 S'，并且在转换过程中，缓存缺失的数量不会增加。

Proof: 通过分析未优化调度 S 中无请求引入缓存的元素 d ，判断其是否有必要存在。如果 d 在被替换前没有被请求（Case 1），则可以直接跳过其引入操作；如果 d 在被替换前被请求（Case 2），则其引入是合理的，无需优化。(归纳法)

Theorem: FF算法是最优的。

Proof:
1 使用数学归纳法，基于请求次数j
2 假设前j次请求中存在一个最优调度S，与 FF 一致
3 分析第$j+1$次请求的所有情况：

- case1: 请求命中 → 无需替换，继承原调度
- case2: 请求未命中，且替换相同元素 → 继承原调度
- case3: 请求未命中，替换不同元素 → 构造新调度$S\prime$ ，替换 FF 所选元素，并在后续适当时候补回差异，确保缓存缺失不增加

4 最终得出：始终可以构造一个与 FF 行为一致的最优调度
5 结论：FF 是最优缓存替换算法

缓存算法分析：
离线算法(offline)假设请求序列已知，而在线(online)算法必须实时处理未知请求。
LIFO(last in first out)、LRU(least recently used) 和 FF(farthest in furture) 是常见的缓存替换策略，其中：

- LIFO 性能较差。
- LRU 是 k-competitive，在线性能接近最优。
- FF 是离线最优算法，提供理论基准。

FF 的性能优于 LRU 和 LIFO，但它是离线算法，无法直接应用于现实场景

Dijkstra（最短路径问题）

每次贪心地挑dis最小的一个点去更新其他所有点。

Invariant. 对于每个已探索集合S中的点u，d(u)即为s−u最短路的长度。

Proof. 归纳法induction：

基态：|S|=1时，显然。

归纳假设：假设对于|S|=1,⋯,k的时候结论均成立

记下一个探索的点为v，选中的边是(u,v)，此时s−v路径的长度为$\pi(v) = d(u) + l(u,v)$。考虑任一s−v路径P，假设(x,y)是路径上离开S的第一条边，记到x的子路径为P'，那么有如下不等式：

$$l(P) \geq l(P\prime) + l(x,y) \geq d(x) + l(x,y) \geq \pi(y) \geq \pi(v)$$

从左到右四个不等号的原因分别为：边权非负，归纳假设，$\pi(y)$和$\pi(v)$的定义。由此证明了$d(v) = \pi(v)$即为s−v最短路的长度，$|S| = k+1$的时候也成立。

$A^\star$ algorithm

(ppt里没有提及)
给定起点和终点，引入启发式函数，用$h(v,t) + \pi(v)$作为优先级。

$h(v,t) < d(v,t)$时保证结果正确，有可能比Dijkstra更快

$h(v,t) = d(v,t)$时保证结果正确，最快，但需要完全正确的知识

$h(v,t) > d(v,t)$时，并不总能得到正确结果，不过可能还是快一些

最小生成树（Minimum Spanning Tree MST）

Prim

维护一个已选点的集合，迭代n−1次，每次将连接生成树和树外一点的最短边加入树，并且把树外断点加入集合。$O(mlogn)$

Kruskal

将边按边权升序排列依次加入生成树，如果成环了就不加（并查集）。$O(mlogm)$

对比总结

算法	初始化	边的处理顺序	关键操作	数据结构
Kruskal	空集$T = \emptyset$	按边权重升序	插入边（避免形成环）	并查集（Union-Find）
Reverse-Delete	所有边 T = E	按边权重降序	删除边（保持连通性）	并查集（Union-Find）
Prim	单个顶点 s	按边权重升序	扩展树（选择最便宜的边）	优先队列（Min-Heap）

切割性质： 对于任意切割，权重最小的跨切割边一定属于 MST。
Proof: (交换论证)证明的关键是假设 $e \in T^*$，然后通过构造一个新的生成树 $T\prime$ 并比较权重，得出矛盾，从而证明 $e \in T^*$

循环性质： 对于任意循环，权重最大的边一定不属于 MST。
Proof： 如果 MST 包含了某条环路中的最大边 f ，那么我们可以把这条边从 MST 中删掉。
因为它是环路的一部分，删除它后图仍然是连通的（因为环路中还有其他路径连接两端点）。
删除 f 后得到的新树总权值比原来的 MST 更小，说明原来的 MST 并不是最优的，矛盾！
所以：MST 一定不包含环路中的最大边。

claim: 一个环和一个割集的交点数为偶数

字典序破局（lexicographic tiebreaking）

场景： 在实际应用中，边权重可能相等，这会导致算法在选择边时出现“平局”(tie-breaking）问题。
实现： 字典序破局的规则是：当两条边的权重相等时，根据边的索引（index）来决定优先级。
边的索引可以是边在图中的编号或其他唯一标识符。
边的索引越小，优先级越高。
影响： 如果扰动足够小，扰动后的 MST 与原始权重下的 MST 相同。

聚类分析（clustering）

聚类定义：
聚类（Clustering） 是一种无监督学习方法，目标是将一组对象（数据点）划分为若干个“相干”(coherent）组。
每个组称为一个 簇（cluster） ，簇内的对象彼此相似，而不同簇的对象之间差异较大。

单链路 K-聚类

问题：给定集合U，初始包含n个对象，将其分入k个非空集合，最大化不同集合之间距离的最小值。

贪心算法：用Kruskal算法，但是只加$n - k$条边。等价于找最小生成树，然后删掉最贵的$k - 1$条边。

Theorem. 从一个图的最小生成树（MST）中删除权重最大的 k−1 条边后，剩下的连通分量构成的聚类C^*是一个具有最大间距的 k -聚类。
Proof. 利用 MST 的性质（Kruskal 算法按边权从小到大添加边）证明C^*的间距 d^*是最优的。
通过比较 C^* 和其他任意 k -聚类 C 的间距，证明 C^*的间距不小于任何其他聚类的间距

哈夫曼编码

算法流程

对于一个前缀编码格式，$ABL(averagebitsperletter)$是每个字符的出现频率与其编码位数的乘积的和：

$$\mathrm{ABL}(\gamma)=\sum_{x\in S} f_x|\gamma(x)|$$

哈夫曼编码：每次贪心选取出现频率最低的两个结点，将其连接到同一个父结点上，定义父节点的出现频率为两个子节点之和。

```
Huffman(S) {
    if |S|=2 {
        return tree with root and 2 leaves
    } else {
        let y and z be lowest-frequency letters in S
        S′ = S
        remove y and z from S′
        insert new letter ω in S′ with f_ω=f_y+f_z
        T′ = Huffman(S′)
        T = add two children y and z to leaf ω from T′
        return T
    }
}
```

时间复杂度：$T(n)=T(n-1)+O(logn)$，因此$T(n)=O(nlogn)$。

Claim. $ABL(T')=ABL(T)-f_\omega$

Pf.

$$\begin{aligned}
\mathrm{ABL}(T) &= \sum_{x\in S} f_x\cdot \mathrm{depth}_T(x)\\
&= f_y\cdot \mathrm{depth}_T(y)+f_z\cdot \mathrm{depth}_T(z)+\sum_{x\in S,x\neq y,z} f_x\cdot \mathrm{depth}_T(x)\\
&= (f_y+f_z)\cdot(1+\mathrm{depth}_T(\omega))+\sum_{x\in S,x\neq y,z} f_x\cdot \mathrm{depth}_T(x)\\
&= f_\omega\cdot(1+\mathrm{depth}_T(\omega))+\sum_{x\in S,x\neq y,z} f_x\cdot \mathrm{depth}_T(x)\\
&= f_\omega+\sum_{x\in S'} f_x\cdot \mathrm{depth}_{T'}(x)\\
&= f_\omega+\mathrm{ABL}(T')
\end{aligned}$$

Observations.

- 在最优的前缀编码树中，出现频率最低的字符对应的结点应该出现在最底层
- (n>1)时，最低层总是有至少两个叶子节点
- 同一层中的顺序并不影响编码最优性

Claim. 存在最优的前缀编码树，使得出现频率最低的两个字符作为树的两个相邻的叶子节点。

Claim. 对于给定字符集S，在所有前缀编码中，哈夫曼编码有最小的ABL。

Proof. 归纳法：
基态：对于$|S| = 2$，没有比一根两叶更短的编码了

归纳假设：假设对于字符集S'，哈夫曼树T'给出的编码是最优的。其中S'没有加入最小频率的x, y而将w作为叶节点。、

归纳步骤：反证法

令 T 是哈夫曼树，假设存在树 (Z) 使得$ABL(Z) < ABL(T)$，不失一般性地假设 Z 中 x, y 是相邻的叶子节点。将 x, y 从 Z 中删除得到 Z'，记其父节点为 w'。

由于 $ABL(Z) = ABL(Z') + f_{w'}$，$ABL(T) = ABL(T') + f_w$，且 $ABL(Z) < ABL(T)$，$f_w = f_{w'}$，得到 $ABL(Z') < ABL(T')$，与假设矛盾。

Chapter5 分治

介绍mergesort

复杂度$O(nlogn)$

逆序对计数

归并排序求逆序对：$O(nlogn)$

```
Sort-and-Count(L)
{
    if list L has one element
        return (0, L)
    Divide the list into two halves A and B
        (rA, A) = Sort-and-Count(A)
        (rB, B) = Sort-and-Count(B)
        (rAB, L) = Merge-and-Count(A, B)
    return (rA + rB + rAB, L)
}
```

最邻近点对（closest pair of point）

问题：平面上n个点，找到点对间最小欧氏距离。

分治算法：按x坐标排序，分为数量相等的左右两部分。拆分：递归地分成两部分求子区域的最近点对距离；合并：左半边，右半边，以及左右两部分之间的点对距离。

记左右两部分内部的最近点对距离为δ，合并时，只需要考虑横坐标在中点$\pm\delta$的点就足够了。对于中心区域的每个点i，我们也只需要考虑$y_i - \delta < y_j \leq y_i$的点$j$，因为每个点都要考虑所以只保证半边就行。为了方便比较y坐标，按y坐标再进行排序，这里排序可以用类似归并的方法，将低层递归排序后的点集归并即可，复杂度是$O(n)$的。

对于点i，记它需要比较的点集为C_i，C_i可能的分布范围是一个长为2δ、宽为δ的矩形，且i位于上长边的中点。将该矩形切分为8个$\frac{\delta}{2} * \frac{\delta}{2}$的小正方形，那么每个小正方形内至多只能有一个点，除去i之后最多只有7个点。反证：每个小正方形内如果存在两个点，那么它们的距离最大值是$\frac{\delta}{\sqrt{2}}$，比之前设定的单边最小距离δ还要小，引出矛盾。所以对于每个点i，我们只需要比对至多7个点。

```
Closest-Pair(p1, ..., pn)
{
    Compute vertical line L such that half the points
    are on one side and half on the other side.

    δ1 = Closest-Pair(left half)
    δ2 = Closest-Pair(right half)
    δ = min(δ1, δ2)

    Delete all points further than δ from line L.

    Sort remaining points by y-coordinate.

    Scan points in y-order and compare distance between
    each point and next 7 neighbors. If any of these
    distances is less than δ, update δ.

    return δ
}
```

1. **初始时间复杂度分析：**

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n \log n) \quad \Rightarrow \quad T(n) = O(n \log^2 n)$$

2. **优化目标：** 是否可以将时间复杂度优化到 $O(n \log n)$?
3. **优化方法：**
   - 不要每次重新排序带状区域中的点。
   - 每次递归返回按 (y)-坐标和 (x)-坐标排序的列表，并通过合并操作实现高效排序。
4. **优化后的时间复杂度：**

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)$$

通过避免重新排序和利用预排序列表，可以将最近点对问题的算法时间复杂度优化到 $O(n \log n)$。

整数乘法（Integer multiplication）

引入：

Divide into more than 2 subproblems

What happens if the divide-and-conquer algorithms that create recursive calls on q sub-problems of size $n/2$ each with $q>2$?

If $T(n)$ obeys the following recurrence relation

$$T(n) \leq qT(n/2) + cn$$

when $n>2$ and $T(2) \leq c$.

$T(\cdot)$ satisfying the above with $q > 2$ is bounded by $O(n^{\log_2 q})$.

When $q=3$, $O(n^{\log_2 q}) = O(n^{1.585})$

When $q=4$, $O(n^{\log_2 q}) = O(n^2)$

For details, please read the Section 5.2 of the Textbook

1. 普通整数乘法（Brute-force）

- 两个 n 位二进制数相乘：
  - 逐位相乘再加，共 $O(n^2)$ 次操作。
- 这是传统的乘法方式，效率较低。

2. 分治整数乘法（Divide-and-Conquer Multiplication）

- 我们有两个 n 位整数 X 和 Y。
- 将它们分成两半：

$$X = A \cdot 2^{n/2} + B, \quad Y = C \cdot 2^{n/2} + D$$

- 那么：

$$X \cdot Y = AC \cdot 2^n + (AD + BC) \cdot 2^{n/2} + BD$$

需要计算 4 个子乘积：AC, AD, BC, BD

时间复杂度递归式：

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n^{\log_2 4}) = O(n^2)$$

结果和暴力法一样，并没有改进！

3. Karatsuba 算法（关键突破）

Karatsuba 提出了一种聪明的方法，只需要 **3 个子乘积** 来代替原来的 4 个：

$$X \cdot Y = AC \cdot 2^n + [(A+B)(C+D) - AC - BD] \cdot 2^{n/2} + BD$$

只需计算：

- AC
- BD
- $(A+B)(C+D)$

于是变成了：

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) \Rightarrow T(n) = O(n^{\log_2 3}) \approx O(n^{1.585})$$

比 $O(n^2)$ 快得多，尤其适用于大整数乘法。

卷积与FFT

傅里叶定理： 任何周期函数都可以表示为一系列正弦波的和。

傅里叶级数 (Fourier Series, FS)
连续时间傅里叶变换 (Continuous Time Fourier Transform, CTFT)
离散时间傅里叶变换 (Discrete Time Fourier Transform, DTFT)
离散傅里叶变换 (Discrete Fourier Transform, DFT)
快速傅里叶变换 (Fast Fourier Transform, FFT)。

名称	时间域	频率域	适用信号类型
傅里叶级数 (FS)	连续	离散	周期性信号
连续时间傅里叶变换 (CTFT)	连续	连续	非周期性连续信号
离散时间傅里叶变换 (DTFT)	离散	连续	离散时间信号
离散傅里叶变换 (DFT)	离散	离散	离散且有限长的信号
快速傅里叶变换 (FFT)	离散	离散	离散且有限长的信号（高效实现）

多项式表示

系数表示

- 多项式 (A(x)) 和 (B(x)) 的系数表示为：

$$A(x) = \sum_{i=0}^{n-1} a_i x^i, \quad B(x) = \sum_{i=0}^{n-1} b_i x^i$$

多项式乘法（线性卷积）

- 多项式乘法的结果为：

$$A(x) \times B(x) = \sum_{i=0}^{2n-2} c_i x^i, \quad \text{其中 } c_i = \sum_{j=0}^{i} a_j b_{i-j}.$$

- 暴力计算复杂度为 $O(n^2)$。

点值表示

- 点值表示形式：

$$A(x) = \{(x_i, y_i) \mid i \in [0, n-1]\}, \quad B(x) = \{(x_i, z_i) \mid i \in [0, n-1]\}$$

- 多项式乘法的点值表示为：

$$A(x) \times B(x) = \{(x_i, y_i \times z_i) \mid i \in [0, 2n-1]\}, \quad O(n)$$

  但需要 $2n$ 个点来表示 $A(x)$ 和 $B(x)$。

系数转点值

对于一个 $n-1$ 次多项式 $A(x)$，给定系数集合 a_i，输出多项式在 n 个点处的值。（多项式快速求值）

单位根

- 选取 n 阶单位根$w^0, w^1, \ldots, w^{n-1}$，其中 $w = e^{2\pi i/n}$。
- 单位根的性质：

$$(w^k)^n = (e^{2\pi i k/n})^n = (e^{2\pi i})^k = 1.$$

- $n/2$ 阶单位根为 $v^0, v^1, \ldots, v^{n/2-1}$，其中 $v = w^2 = e^{4\pi i/n}$。

多项式拆分

- 将多项式 $A(x)$ 拆分为奇数次幂和偶数次幂两部分：

$$A_{\text{even}}(x) = a_0 + a_2 x^2 + a_4 x^4 + \cdots + a_{n/2-2} x^{(n-1)/2},$$
$$A_{\text{odd}}(x) = a_1 + a_3 x + a_5 x^2 + \cdots + a_{n/2-1} x^{(n-1)/2}.$$

关系式

- 则有：

$$A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2),$$
$$A(-x) = A_{\text{even}}(x^2) - x A_{\text{odd}}(x^2).$$

代入 n 阶单位根

- 代入 n 阶单位根 w^k和 $w^{k+n/2}$，有：

$$A(w^k) = A_{\text{even}}(v^k) + w^k A_{\text{odd}}(v^k), \quad 0 \le k < n/2,$$
$$A(w^{k+n/2}) = A_{\text{even}}(v^k) - w^k A_{\text{odd}}(v^k), \quad 0 \le k < n/2.$$

- 这样，我们就把求长度为 n 的多项式 A 在 w^k 和 $w^{k+n/2}$ 两处的点值问题转化为求两个长度为 $n/2$ 的多项式的点值问题。
- 对于求 n 个单位根的问题，我们在 $O(n)$ 时间内将其拆分成了两个规模减半的问题：
  $T(n) = 2T(n/2) + O(n)$, 根据主定理，时间复杂度为：$T(n) = O(n \log n)$.

```
fft(n, a0,a1,…,an-1) {
    if (n == 1) return a0

    (e0,e1,…,en/2-1) ← FFT(n/2, a0,a2,a4,…,an-2)
    (d0,d1,…,dn/2-1) ← FFT(n/2, a1,a3,a5,…,an-1)

    for k = 0 to n/2 - 1 {
        ωk ← e2πik/n
        yk     ← ek + ωk dk
        yk+n/2 ← ek - ωk dk
    }

    return (y0,y1,…,yn-1)
}
```

点值转系数 （inverse FFT）

只需要将w^{-1}作为单位根的底数即可，同样是$O(nlogn)$的。
（高效的插值法）

```
ifft(n, a0,a1,…,an-1) {
    if (n == 1) return a0

    (e0,e1,…,en/2-1) ← FFT(n/2, a0,a2,a4,…,an-2)
    (d0,d1,…,dn/2-1) ← FFT(n/2, a1,a3,a5,…,an-1)

    for k = 0 to n/2 - 1 {
        ωk ← e-2πik/n
        yk+n/2 ← (ek + ωk dk) / n
        yk+n/2 ← (ek - ωk dk) / n
    }

    return (y0,y1,…,yn-1)
}
```

Chapter6 动态规划

动态规划（Dynamic Programming, DP）通过将原问题分解为若干个子问题，并利用子问题的解来构建原问题的解。
最优子结构（Optimal Substructure） 意味着原问题的最优解可以由其子问题的最优解构成。

贪心算法： 每一步选择当前最优解，但不保证全局最优。
分治算法： 将问题分解为互不重叠的子问题，独立求解后再合并。
动态规划： 将问题分解为重叠的子问题，记录每个子问题的解，避免重复计算。

Dynamic Programming: Binary Choice

Notation. OPT(j) = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- Case 1: OPT selects job j.
  - collect profit v_j
  - can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

- Case 2: OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\{ v_j + OPT(p(j)),\ OPT(j-1) \} & \text{otherwise} \end{cases}$$

算法实现思路：
使用自底向上的方法（迭代）或自顶向下的方法（递归 + 记忆化搜索）来计算 OPT(j) 。
需要预处理每个任务的结束时间 p(j) ，以便快速查找兼容任务。

带权区间调度（Weighted Interval Scheduling）

- **问题描述：** 给定一组带权重的任务（开始时间、结束时间、价值），从中选出一组互不冲突的任务，使得总价值最大。
- **关键点：**
  - 使用 `OPT(j)` 表示前 j 个任务的最大收益。
  - 引入 `p(j)` 表示与任务 j 不冲突的最大索引。
  - 状态转移方程：

$$\mathrm{OPT}(j) = \max(v_j + \mathrm{OPT}(p(j)), \mathrm{OPT}(j-1))$$

- **优化手段：**
  - 使用记忆化搜索（Memorization）或自底向上递推（Bottom-up DP）来避免重复计算。

分段最小二乘拟合（Segmented Least Squares）

- **问题描述：** 给定平面上的一组点，用若干条线段去拟合这些点，使误差最小。
- **目标函数：**

$$E + cL$$

  其中 E 是所有线段的平方误差和，L 是线段数量，c 是惩罚系数。

- **状态转移：**

$$\mathrm{OPT}(j) = \min_{1 \le i \le j} (e(i,j) + c + \mathrm{OPT}(i-1))$$

背包问题（Knapsack Problem）

- **问题描述：** 给定一组物品（重量 + 价值），在不超过容量 W 的前提下，最大化总价值。
- **状态定义：**

$$\mathrm{OPT}(i, w) = \text{前 i 个物品，在容量 w 下的最大价值}$$

- **状态转移方程：**

$$\mathrm{OPT}(i,w) = \max(\mathrm{OPT}(i-1,w), v_i + \mathrm{OPT}(i-1, w-w_i))$$

- **空间优化：**
  - 可以使用一维数组进行空间压缩。

类型	特点	转移方程	DP 方式	时间复杂度	应用场景
0-1 背包	每个物品只能选一次	$dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w_i] + v_i)$	逆序遍历	$O(nW)$	投资选择、物流
完全背包 (unbounded)	每个物品可无限选	$dp[w] = \max(dp[w], dp[w-w_i] + v_i), \quad \text{for all } i, w \ge w_i$	顺序遍历	$O(nW)$	零钱兑换
多重背包 (bounded)	每个物品最多选有限次	❌	二进制拆分	$O(nW \log k)$	库存管理
二维背包	有两个容量限制	$dp[x][y] = \max(dp[x][y], dp[x-a_i][y-b_i] + v_i)$	二维 DP	$O(nWX)$	多维资源分配
分组背包(packet)	每组只能选一个	❌	分组处理	$O(nW)$	品牌选择
依赖背包（group）	存在依赖关系	❌	树形 DP 或图模型	$O(nW)$	技能解锁

RNA 二级结构预测（RNA Secondary Structure）

- **问题描述：** 给定一个 RNA 字符串，找出其中形成 Watson-Crick 配对最多的结构（A-U, C-G）。
- **限制条件：**
  - 每个配对之间至少隔 4 个字符。
  - 所有配对不能交叉。
- **动态规划状态定义：**

$$\mathrm{OPT}(i,j) = \text{子串 } b_i \ldots b_j \text{ 的最大配对数}$$

- **状态转移：**
  - 如果 b_j 不参与配对，则 $\mathrm{OPT}(i, j) = \mathrm{OPT}(i, j-1)$
  - 如果 b_t 和 b_j 配对，则 $\mathrm{OPT}(i, j) = 1 + \mathrm{OPT}(i, t-1) + \mathrm{OPT}(t+1, j-1)$
- **时间复杂度：**

$$O(n^3)$$

第 6.6 节：字符串编辑距离（Sequence Alignment / Edit Distance）

- **问题描述：** 如何将两个字符串对齐，使得插入、删除、替换操作的代价最小。
- **应用场景：**
  - 文件比较（Unix diff）
  - 语音识别
  - 生物信息学中的 DNA 序列比对
- **状态定义：**

$$\mathrm{OPT}(i, j) = \text{前 i 个字符与前 j 个字符的最小编辑距离}$$

- **状态转移方程：**

$$\mathrm{OPT}(i, j) = \min \begin{cases} \alpha_{x_i y_j} + \mathrm{OPT}(i-1, j-1) \\ \delta + \mathrm{OPT}(i-1, j) \\ \delta + \mathrm{OPT}(i, j-1) \end{cases}$$

- **空间优化：**
  - 可以使用 Hirschberg 算法实现 **O(m+n) 的空间复杂度**，同时保持 $O(mn)$ 的时间复杂度。

最短路径（Bellman-Ford算法）

```
Push-Based-Shortest-Path(G, s, t) {
    foreach node v ∈ V {
        M[v] ← ∞
        successor[v] ← ϕ
    }

    M[t] = 0
    for i = 1 to n-1 {
        foreach node w ∈ V {
        if (M[w] has been updated in previous iteration) {
            foreach node v such that (v, w) ∈ E {
                if (M[v] > M[w] + c_vw) {
                    M[v] ← M[w] + c_vw
                    successor[v] ← w
                }
            }
        }
        If no M[w] value changed in iteration i, stop.
    }
}
```

矩阵链乘法

- 兼容性（Compatibility）
  - 两个矩阵 *A* 和 *B* 可以相乘的前提是：*A* 的列数必须等于 *B* 的行数。
  - 如果 *A* 是一个$p\times q$ 的矩阵，*B* 是一个 $q\times r$ 的矩阵，则结果矩阵 *C* 是一个 $p\times r$ 的矩阵。
- 时间复杂度
  - 矩阵乘法的主要计算量来自于标量乘法（scalar multiplications），其数量为 pqr 。

$$\mathrm{m}[i,j]=\begin{cases}0 & \text{if } i=j\\ \min\limits_{i\le k\le j}\{m[i,k]+m[k+1,j]+p_{i-1}p_kp_j\} & \text{if } i<j\end{cases}$$

考试的时候最重要要写好初始条件和递推公式！！！

Chapter 7 网络流

流量网络$G=(V,E,s,t,c)$

在网络流（**Network Flow**）问题中，**流（flow）** 和 **割（cut）** 是两个核心概念。它们分别描述了网络中流量的传输情况和网络的“瓶颈”结构。

一个**流（flow）** 是一个函数：

$$f:V\times V\to\mathbb{R}$$

满足以下四个条件：

1. 容量限制（Capacity Constraint）

对于所有 $u, v \in V$，有：

$$f(u, v) \leq c(u, v)$$

> 流过的量不能超过边的容量。弱双对性（weak duality）

2. 反向流（Skew Symmetry）

$$f(u, v) = -f(v, u)$$

> 如果从 u 到 v 有正的流，那么从 v 到 u 就是负的流。

3. 流守恒（Flow Conservation）

对所有中间节点 $u \in V \setminus \{s, t\}$：

$$\sum_{v \in V} f(u, v) = 0$$

> 中间节点流入等于流出（没有"存储"或"产生"流的能力）。

4. 总流量定义

从源点出发的总净流出为：

$$|f| = \sum_{v \in V} f(s, v)$$

> 这就是整个流的**值（value of the flow）**。

割（cut） 是将图中的顶点集合 V 分成两个不相交子集 S 和 T，使得：

- $s \in S$
- $t \in T$

记作：(S, T)，即把图"切"成两部分，源在一边，汇在另一边。

割的容量（Capacity of a Cut）

$$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$$

> 所有从 S 到 T 的边的容量之和（注意只算正方向，不考虑反向边）。

割的净流（Net Flow）

$$f(S, T) = \sum_{u \in S, v \in T} f(u, v)$$

> 所有从 S 到 T 的边上的流之和（同样只计算正方向）。

名称	数学定义	含义
流（flow）	函数 $f(u, v)$，满足容量、反对称性、守恒性	描述从源到汇的流量分布

名称	数学定义	含义
割（cut）	顶点划分 (S,T)，$s \in S, t \in T$	表示图的一个“瓶颈”，限制最大流
最大流最小割定理	最大流 = 最小割容量	网络流的核心理论依据

Ford-Fulkerson算法

正向边容量减去流量，反向边全权值设为流量

残量网络：边集为 有剩余容量的边 和 流量非零的反向边

增广路：残量网络中s-t的一条简单路径，处理后每条边容量都减去瓶颈容量，反向边都加上瓶颈容量。

```
Augment(f, c, P) {
    b ← bottleneck(P)
    foreach e ∈ P {
        if (e ∈ E)  f(e) ← f(e) + b
        else        f(e^R) ← f(e^R) - b
    }
    return f
}
```

forward edge

reverse edge

```
Ford-Fulkerson(G, s, t, c) {
    foreach e ∈ E   f(e) ← 0
    G_f ← residual graph

    while (there exists augmenting path P) {
        f ← Augment(f, c, P)
        update G_f
    }
    return f
}
```

时间复杂度$O(mnC)$，因为每次增广至少提升一个流量，找到一个增广路需要$O(m)$时间。由于和容量相关，FF算法对于输入规模而言并不是多项式的。

输入规模：$m, n, logC$，算法可能需要进行$2C$次迭代。

最大流与最小割

增广路径定理（Augmenting Path Theorem）

- **内容**：流 f 是最大流当且仅当不存在相对于 f 的增广路径。

通过证明以下三个命题相互等价来同时证明增广路径定理和最大流最小割定理：

（i）存在一个割 (A, B)，使得 $v(f) = \text{cap}(A, B)$

（ii）流 f 是最大流

（iii）不存在相对于 f 的增广路径

通过证明这三个命题相互等价（TFAE，The Following Are Equivalent）来完成证明：

（i） ⇒ （ii）

- **含义**：如果存在一个割 (A, B)，使得 $v(f) = \text{cap}(A, B)$，那么流 f 是最大流。
- **解释**：由于割的容量是网络中流的上限，如果流的值已经等于割的容量，说明流 f 已经达到最大值。

（ii） ⇒ （iii）

- **含义**：如果流 f 是最大流，那么不存在相对于 f 的增广路径。
- **证明方法**：通过反证法（Contrapositive）。
  - 假设流 f 是最大流，但存在增广路径。
  - 如果存在增广路径，可以增加f, 与 f 是最大流的假设矛盾。

（iii） ⇒ （i）

- **含义**：如果不存在相对于 f 的增广路径，那么存在一个割 (A, B)，使得 $v(f) = \text{cap}(A, B)$。
- **解释**：如果没有增广路径，说明流 f 已经达到网络瓶颈，此时可以构造一个割，其容量恰好等于流的值。

Capacity scaling算法

设置一个阈值Δ，初始值为$2^{\lfloor logC \rfloor}$，每次只考虑瓶颈容量在Δ以上的增广路，随后将Δ折半直到为0。

```
Capacity-Scaling(G, s, t, c)
{
    foreach e ∈ E: f(e) = 0
    Δ = largest power of 2 ≤ C
    Gf = residual network with respect to flow f

    while (Δ ≥ 1)
    {
        Gf(Δ) = Δ-residual network of G with respect to flow f
        while (there exists an augmenting path P in Gf(Δ))
        {
            f = Augment(f, c, P)
            update Gf(Δ)
        }
```

```
        Δ = Δ / 2
    }
    return f
}
```

Lemma. 外层循环$1+\lfloor log2C \rfloor$次终止

Lemma. f是每次Δ缩减循环后的流，最大流的值不会超过$v(f)+m\Delta$

Lemma. 每次缩减循环中，至多有$2m$条增广路

因此该算法时间复杂度$O(m^2logC)$

Edmonds Karp算法*

利用BFS找边数最少的增广路

```
Edmonds-Karp(G, s, t, c)
{
    foreach e ∈ E: f(e) = 0
    Gf = residual network of G with respect to flow f

    while (there exists an augmenting path P in Gf)
    {
        P = Breath-First-Search(Gf)
        f = Augment(f, c, P)
        update Gf
    }
    return f
}
```

Lemma. 最短增广路的长度不会减少

Lemma. 至多m次最短路增广后，最短增广路的长度严格上升

找最短增广路$O(m)$，至多有$n-1$种不同的长度，对每种长度至多增广m次。因此该算法时间复杂度$O(m^2n)$

Dinic 算法

- 建立分层图L_G
- 从s开始在L_G中搜索，直到卡住或者到达t
- 如果到达t，增广，更新分层图，从s再开始
- 如果卡住，那就从分层图中删除该结点，跳回到上个结点重来

```
Dinic-Normal-Phase(Gf, s, t)
{
    LG = level graph of Gf
    P = empty path
    Advance(s)
}

Advance(v)
```

```
{
    if (v = t)
        f = Augment(f, c, P)
        remove bottleneck edges from LG
        P = empty path
        Advance(s)
    if (there exists (v, w) ∈ LG)
        add edge (v, w) to P
        Advance(w)
        Retreat(v)
}

Retreat(v)
{
    if (v = s) return
    else
        delete v and incident edges from LG
        remove last edge (u, v) from P
        Advance(u)
}

Dinitz(G, s, t, c)
{
    foreach e ∈ E: f(e) = 0
    Gf = residual network of G with respect to flow f

    while (there exists an augmenting path P in Gf)
    {
        Dinitz-Normal-Phase(Gf, s, t)
    }
    return f
}
```

每个阶段初始化需要$O(m)$的BFS；增广至多m次，耗时$O(mn)$；回溯至多n次，耗时$O(m+n)$；至多mn次前进，耗时$O(mn)$。

每个阶段耗时$O(mn)$，至多有$n-1$个阶段，因此该算法时间复杂度$O(mn^2)$

二分图匹配

原图所有边容量设为∞，将二分图的一方用容量为1的边全部连接到源点，另一方用容量为1的连到汇点得到图$G\prime$，$G\prime$的最大流就是最大匹配数。

赫尔婚姻定理：二分图G两点集L, R大小相同。G有完美匹配当且仅当对于所有L的子集S都有$|N(S)| \geq |S|$。

不相交路径

def: 两条不共用任何边的路径是不相交路径。

设置每条边容量为1，也就是最多只能被一条增广路通过，**最大流就是不相交的路径数量**，但并不一定是简单路径(可能有环，不共享边也算进去了)。

简单路径：在一条路径中，**除了起点和终点外，所有的顶点（节点）都不重复** 的路径。

$s-t$割：每条$s-t$路径都用到了其中的至少一条边

Menger' s theorem：最大的$s-t$不相交路径数等于最小$s-t$割所包含的边数。

如果是求无向图上的不相交路径数，那就每条无向边建两条有向边代替即可。

对最大流问题的推广

一、主要扩展模型及技术

1. 多源点多汇点（Multiple Sources and Sinks）

问题描述：

在标准最大流问题中，只有一个源点 s 和一个汇点 t。但在实际中，可能有多个源点和多个汇点。

解法：

- 引入一个新的超级源点 s'
- 引入一个新的超级汇点 t'
- 将所有原来的源点连接到 s'，容量设为 ∞
- 将所有原来的汇点连接到 t'，容量也设为 ∞
- 在这个新图上求从 s' 到 t' 的最大流即可

这样就将多源点多汇点的问题转化为标准的最大流问题。

---

2. 带供需约束的循环流（Circulation with Supplies and Demands）

问题描述：

每个节点都有一个供需值 $d(v)$：

- 如果 $d(v)>0$：表示该节点是一个**需求节点**（需要接收这么多流量）
- 如果 $d(v)<0$：表示该节点是一个**供应节点**（可以提供这么多流量）
- 如果 $d(v)=0$：表示是普通中间节点

目标是找到一个满足以下条件的循环流 f：

- 满足边容量限制
- 满足节点的供需平衡（即流入 - 流出 = $d(v)$）

解法：

- 添加一个超级源点 s 和一个超级汇点 t
- 对于每个供应节点 v（$d(v)<0$），从 s 向 v 连一条容量为 $-d(v)$ 的边
- 对于每个需求节点 v（$d(v)>0$），从 v 向 t 连一条容量为 $d(v)$ 的边
- 然后在这个新图中求最大流
- 如果最大流等于所有需求节点的需求总和，则原问题存在可行解

---

3. 整数性定理（Integrality Theorem）

- 如果所有的容量和供需值都是整数，并且存在可行解，那么一定存在一个**整数值的可行解**。
- 证明思路：通过上述建模方式，结合标准最大流问题的整数性定理（Integral Flow Theorem）直接推出。

---

4. 可行性判定与最小割分析（Feasibility Check via Min-Cut）

- 判断是否存在可行的循环流，可以通过检查构造后的图中是否能实现足够大的最大流。
- 进一步地，如果不存在可行解，说明存在某个节点划分 (A, B)，使得：
  - 节点集合 B 中的总需求大于：
    - B 中节点的总供应量
    - 加上从 A 到 B 的所有边的总容量

这实际上是一个最小割的应用。

二、核心思想总结

扩展类型	技术手段	应用场景
多源点多汇点	添加超级源/汇点	实际物流、交通网络
带供需约束的循环流	添加虚拟源汇 + 容量匹配供需	电力调度、资源分配
整数解的存在性	利用最大流整数性定理	需要整数解的实际问题
可行性判断	最小割分析	网络可靠性、资源瓶颈分析