



第二章 算法分析基础

李 翔

<https://implus.github.io/>



算法分析

- 什么叫一个算法好，运行得有效率？
- 概念设计：效率



定义效率

- 提出效率定义(1) 当实现一个算法时，如果它在真实的输入实例上运行的快，那么这个算法是有效的。
- 寻找效率的定义：与平台无关，实例无关，并且随着输入规模的增长是可以预测的



定义效率

- 提出效率定义(2) 在分析的层次上，如果一个算法与蛮力搜索(**Brute Force**)比较，最坏情况下达到质量上更好的性能，就说这个算法是有效的。

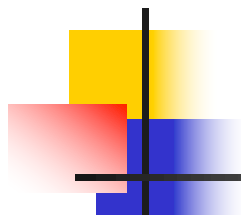


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long



定义效率

一个算法被称为是多项式时间的如果满足如下的性质： 当算法输入的规模增长时，算法的运行时间是多项式有界的。也就是：存在常数 $c > 0$, $d > 0$, 使得对于每一个问题输入的规模 N , 算法的运行都能够在 cN^d 步骤内完成。



定义效率

- 提出效率定义(3) 如果一个算法有多项式运行时间，称为这个算法是有效的。
- 来自于数学形式和经验证据
- 但并不绝对反映真实的运行时间
- 符合实际情况的合理近似



定义效率

注记:

- 这种定义有利于一般性的研究
- 这种定义能够清楚的表达：对某个问题不存在有效的算法
- ✓ $6.02 \times 10^{23} \times N^{20}$ 也是多项式时间,但是在实际上是不可行的。
- ✓ 实际中，大多数开发出的多项式时间的算法中幂次指数，系数，常数项都比较小。
- ✓ 一些最坏情形是指数阶复杂度的算法也可能广泛使用，这种情形极少出现。
- ✓ 能够改进指数阶穷举算法的要点在于发现问题内在的一些特殊结构。



增长的渐进阶

- 为什么需要这样一个概念？
- ✓ 得到一个准确的界是很困难的；
- ✓ 我们的目标是识别类似行为算法的大类，按照**粗粒度**进行分析；
- ✓ 讨论算法执行的步数可能依赖于所使用的语言，因此复杂度的多项式系数会不一样。



复杂度符号

- 因为上述原因，希望以不受常数因子，低项影响的方式表达运行时间的增长率。
- O, Ω, Θ



复杂度符号

- **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 $\Omega(f(n))$.



复杂度符号

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



复杂度符号

- 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 $\Theta(n^3)$.




复杂度性质

- 设 f 和 g 是两个函数, $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$ 存在, 并且等于某个常数 $c > 0$, 那么 $f(n) = \Theta(g(n))$.
- 证明: 根据极限的定义, 存在 n_0 , 对所有的 $n \geq n_0$, $f(n) \leq 2cg(n)$ 从而推出 $f(n) = O(g(n))$. 类似的 $f(n) = \Omega(g(n))$. 根据定义可知命题成立。



极限的定义

 [zh.wikipedia.org/wiki/极限_\(数学\)](https://zh.wikipedia.org/wiki/极限_(数学))

形式上，我们可以定义：

$$\lim_{x \rightarrow \infty} f(x) = L$$

为

$$(\forall \epsilon > 0)(\exists \delta > 0)(\forall x \in D_f)[(\delta < x) \Rightarrow (|f(x) - L| < \epsilon)]$$



复杂度性质

■ 传递性

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.



复杂度性质

■ 传递性

- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

本质是：不等号的传递性！



复杂度性质

■ 可加性

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = \Theta(h)$ then $f + g = \Theta(h)$.



复杂度性质

■ 可加性

- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = \Theta(h)$ then $f + g = \Theta(h)$.

本质是：不等号的可加性！



复杂度性质

- 命题 令 f 是一个 d 阶多项式, $a_0 + a_1n + \dots + a_d n^d$, $a_d > 0$, 那么 $f = \Theta(n^d)$.
- 命题 对于任何正整数, $O(\log_a n) = O(\log_b n)$.



复杂度性质

- 命题 对于 $x > 0$, $\log n = O(n^x)$.

对数函数增长比多项式增长慢！

- 命题 对每个 $r > 1$ 和每个 $d > 0$, $n^d = O(r^n)$.

多项式增长比指数函数慢！



一般运行时间的描述

- 对日常中我们经常碰到的算法问题，按照阶数的大小进行分类
- $O(\log n)$
- $O(n), O(n \log n)$
- $O(n^2), O(n^3), O(n^k)$
- $O(a^n), O(n!)$



亚线性时间($O(\log n)$)

- 例子：给定一个排好序的含有 n 个数的数组 A ，我们想确定一个给定的数 p 是否属于这个数组。
 - 算法要点：每次比较 p 与剩余集合中间元素的值
 - k 次试探以后，剩余的区间大小为： $\left(\frac{1}{2}\right)^k n$
- 所以选择 $k=\log_2(n)$ ，剩余集合的大小减少到一个常数，最后可在常数时间内直接搜索。



线性时间

- 线性时间 $O(n)$

问题：计算 n 个数 a_1, \dots, a_n 中的最大数。

```
max  $\leftarrow$   $a_1$ 
for i = 2 to n {
    if ( $a_i >$  max)
        max  $\leftarrow$   $a_i$ 
}
```



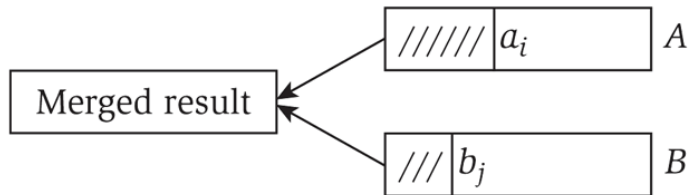

线性时间

归并两个排好序的数表

- 问题：给定两个数组，每个包含有 n 个数，并且每个按照上升的顺序排列， $\mathbf{A} = \mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n$ ； $\mathbf{B} = \mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_n$ ，把它们归并成按照上升顺序排列的数组。



线性时间



```
i = 1, j = 1
while (both lists are nonempty) {
    if ( $a_i \leq b_j$ ) append  $a_i$  to output list and increment i
    else( $a_i \leq b_j$ ) append  $b_j$  to output list and increment j
}
append remainder of nonempty list to output list
```



$O(n \log n)$ 阶时间

- 快速排序(Quicksort)
- FFT(快速傅立叶变换)
- 归并排序，堆排序



$O(n \log n)$ 阶时间

- 实际中碰到的问题：
- **最大时间间隔问题**：给定一组 n 个时间标签 x_1, \dots, x_n ，一个文件的副本在这些时间到达一个服务器，我们想找出在第一个和最后一个时间标签之间的**最大的**没有文件副本到达的**时间区间**。
- 本质上是一个排序问题：寻找最大的时间间隔



平方时间

- 问题：假设平面上给定 n 个点，每个点由 (x,y) 坐标指定。需要找最邻近(距离最小)的点对。
- $O(n^2)$ 解决方法：一个个点对尝试



平方时间

```
min ←  $(x_1 - x_2)^2 + (y_1 - y_2)^2$ 
for i = 1 to n {
  for j = i+1 to n {
    d ←  $(x_i - x_j)^2 + (y_i - y_j)^2$ 
    if (d < min)
      min ← d
  }
}
```

$O(n^2)$ 的解决办法看起来是最好的
，但还有更好的方法！



立方时间

- 问题：给定集合 S_1, \dots, S_n ，它们都是 $\{1, 2, \dots, n\}$ 的子集，我们想知道这些子集中是否有某对子集是不相交的，也就是没有共同的元素。
- 解决思路：对每对集合 S_i, S_j ，确定是否 S_i, S_j 有一个共同的元素。



立方时间

```
foreach set  $S_i$  {  
    foreach other set  $S_j$  {  
        foreach element  $p$  of  $S_i$  {  
            determine whether  $p$  also belongs to  $S_j$   
        }  
        if (no element of  $S_i$  belongs to  $S_j$ )  
            report that  $S_i$  and  $S_j$  are disjoint  
    }  
}
```

$O(n^3)$ solution



$O(n^k)$ 时间

- 问题： 对某个固定常数 k , 我们想知道给定的 n 个结点的输入图是否有一个大小为 k 的独立集。
- 如果用蛮力搜索的方法： 枚举所有 k 个结点的子集，并且对每个子集 S 检查是否存在一条边与 S 中的任意两个元素相交



$O(n^k)$ 时间

```
foreach subset S of k nodes {  
    check whether S is an independent set  
    if (S is an independent set)  
        report S is an independent set  
    }  
}
```

枚举所有 k 个结点的子集： $\binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!}$

判断 k 个结点的子集是否独立： $O(k^2)$

所以蛮力搜索的代价为： $O(k^2 n^k / k!) = O(n^k)$



指数时间(Exponential time)

- 问题：假设给定一个图，需要找一个**最大规模的独立集**。
- 解决思路：蛮力搜索的方法，检查每一个子集是否是独立集，是否最大



指数时间(Exponential time)

```
S* ← ∅  
foreach subset S of nodes {  
    check whether S is an independent set  
    if (S is largest independent set seen so far)  
        update S* ← S  
}  
}
```

估算计算时间的代价

$$O(n^2 2^n)$$



$n!$ 时间

- 对稳定匹配问题的穷举搜索: $n!$
- 二分匹配(二部图每边存在 n 个结点)问题
如果穷举搜索, 代价是 $n!$
- 巡回售货员(TSP: Travelling Salesman Problem)问题: 给定 n 个城市的集合, 每对城市之间都有距离, 什么是访问所有城市的最短旅行? 固定第一个(结束)城市
穷举代价是 $(n-1)!$



n!时间

- Stirling公式 (1730)

$$\lim_{n \rightarrow \infty} \frac{n!}{\sqrt{2\pi n} n^n e^{-n}} = 1.$$

记成当 $n \rightarrow \infty, n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$.



稳定匹配上机实验

■ 2位同学一组 (A_B)

第一次课 (C++)

第二次课 (Python)

第三次课

A C++实现G-S算法并优化速度

写Python验证程序判断B同学算法在X数据上是否正确

继续优化

B 1) 写程序构造1个测试数据X ($n=10^3$)
2) 写C++验证程序判断A同学算法在X数据上是否正确

Python实现G-S算法并优化速度

提交物 **1) 构造数据**
2) C++代码, (保证正确)
当堂验证速度排名

1) C++/Python代码, (保证正确)
当堂验证速度排名

1) C++/Python代码, (保证正确)
2) 当堂验证速度排名
3) 总结交流

```
class Solution {
public:
    vector<vector<int>> stableMatching(vector<vector<int>> man_list, vector<vec
        int man_num = man_list.size();
        int woman_num = woman_list.size();
        // 男女数量不同, 无法完美匹配
        if (man_num != woman_num) return {};
        int n = man_num;

        // 将女性喜好列表的索引和数据置反, 方便处理
        vector<vector<int>> inverse_womanlist;
        for (int i = 0; i < n; ++i) {
            vector<int> prefer(n, -1);
            for (int j = 0; j < n; ++j) {
                prefer[woman_list[i][j]] = j;
            }
            inverse_womanlist.push_back(prefer);
        }
        // 记录最终的匹配结果
        vector<vector<int>> S = {};
        // 记录单身男性
        queue<int> free_man;
        for (int i = 0; i < n; ++i) {
            free_man.push(i);
        }
        // husband[w] = m: 女性 w 的伴侣是 m
        vector<int> husband(n, -1);
        // 为每个男性维护一个指针 (索引), 记录喜爱列表中下一个求婚的女性索引
        vector<int> man_pointer(n, 0);

        while (!free_man.empty()) {
            int m = free_man.front();
            int w = man_list[m][man_pointer[m]];
            // 男方求婚指针向前
            man_pointer[m]++;
            // 女方无配偶
            if (husband[w] == -1) {
                S.push_back({m,w});
                free_man.pop();
                husband[w] = m;
            }
            // 男人m比当前配偶更有吸引力
            else if (inverse_womanlist[w][husband[w]] > inverse_womanlist[w][m]) {
                // 找到 S 中原来关于 w 的一对
                int index;
                for (index = 0; index < S.size(); index++) {
                    if (S[index][1] == w) break;
                }
                int old_m = S[index][0];

                // 从 S 中删除 m'-w
                S.erase(S.begin() + index);
                // 在 S 中添加 m-w
                S.push_back({m, w});
                free_man.pop();
                // w 原来的配偶变为单身汉
                free_man.push(old_m);
                husband[w] = m;
            }
            else {
                // w 拒绝 m
            }
        }
        return S;
    }
};
```


输入输出统一格式

out.txt按照男方0,1,...,n-1
的顺序输出

```
int main() {
    int n, tmp;
    while(scanf("%d", &n) != EOF) {
        vector<vector<int>> manlist(n), womanlist(n);
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                scanf("%d", &tmp);
                manlist[i].push_back(tmp);
            }
        }
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                scanf("%d", &tmp);
                womanlist[i].push_back(tmp);
            }
        }

        Solution test;
        vector<vector<int>> result;

        result = test.stableMatching(manlist, womanlist);
        printf("%d\n", n);
        for (int i = 0; i < manlist.size(); ++i) {
            printf("%d %d\n", result[i][0], result[i][1]);
        }
    }
    return 0;
}
```

in.txt

```
3
0 1 2
1 0 2
0 1 2
1 2 0
0 1 2
0 1 2
3
0 1 2
1 0 2
0 1 2
1 2 0
0 1 2
0 1 2
```

out.txt

```
3
0 1
1 0
2 2
3
0 1
1 0
2 2
```



提交A_B.cpp

- `g++ A_B.cpp -o A_B -std=c++11`
- `time ./A_B < in.txt > out.txt` 测时间
测正确