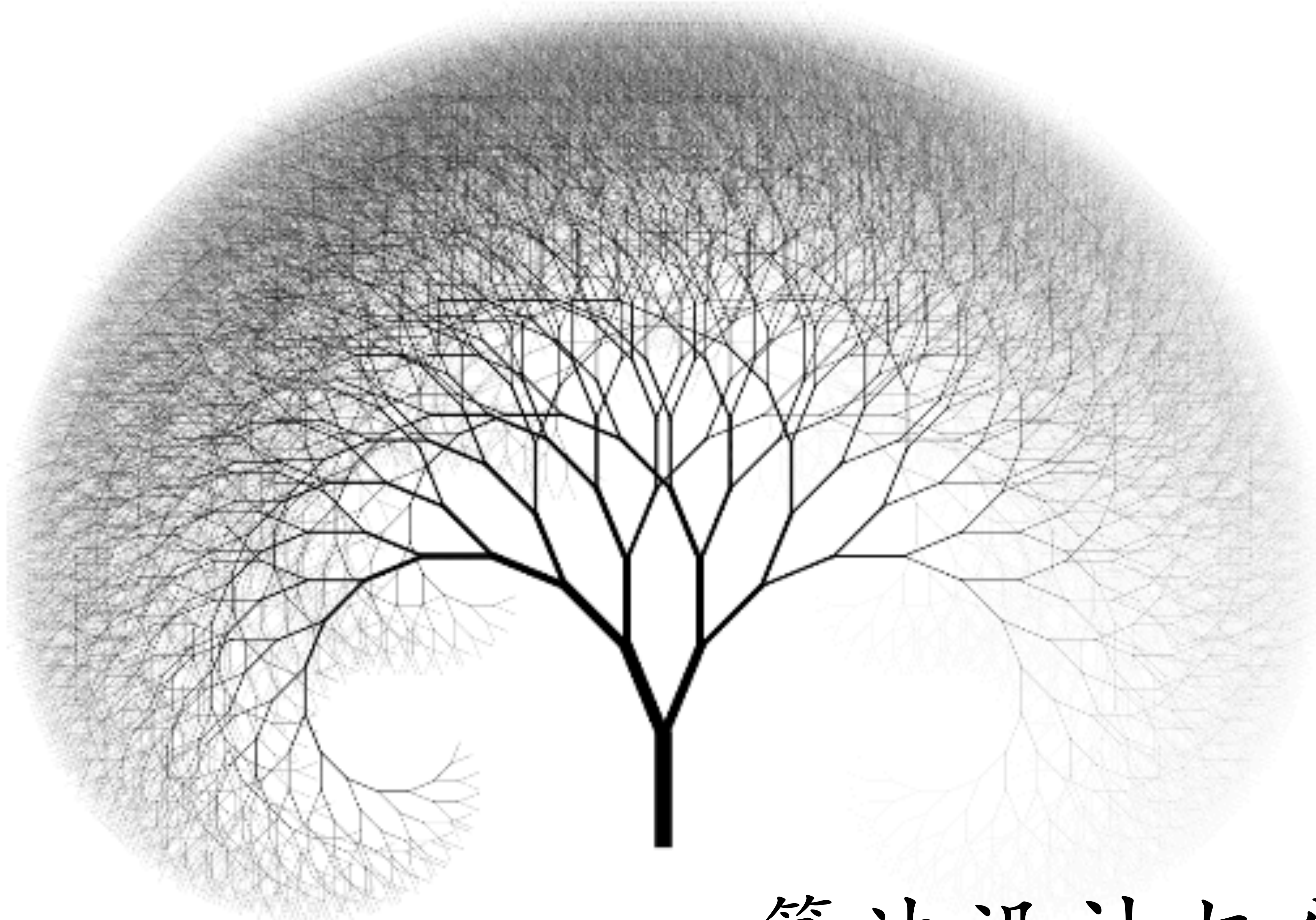


算法设计与分析

刘 安
苏州大学 计算机科学与技术学院
http://web.suda.edu.cn/anliu/

课程说明

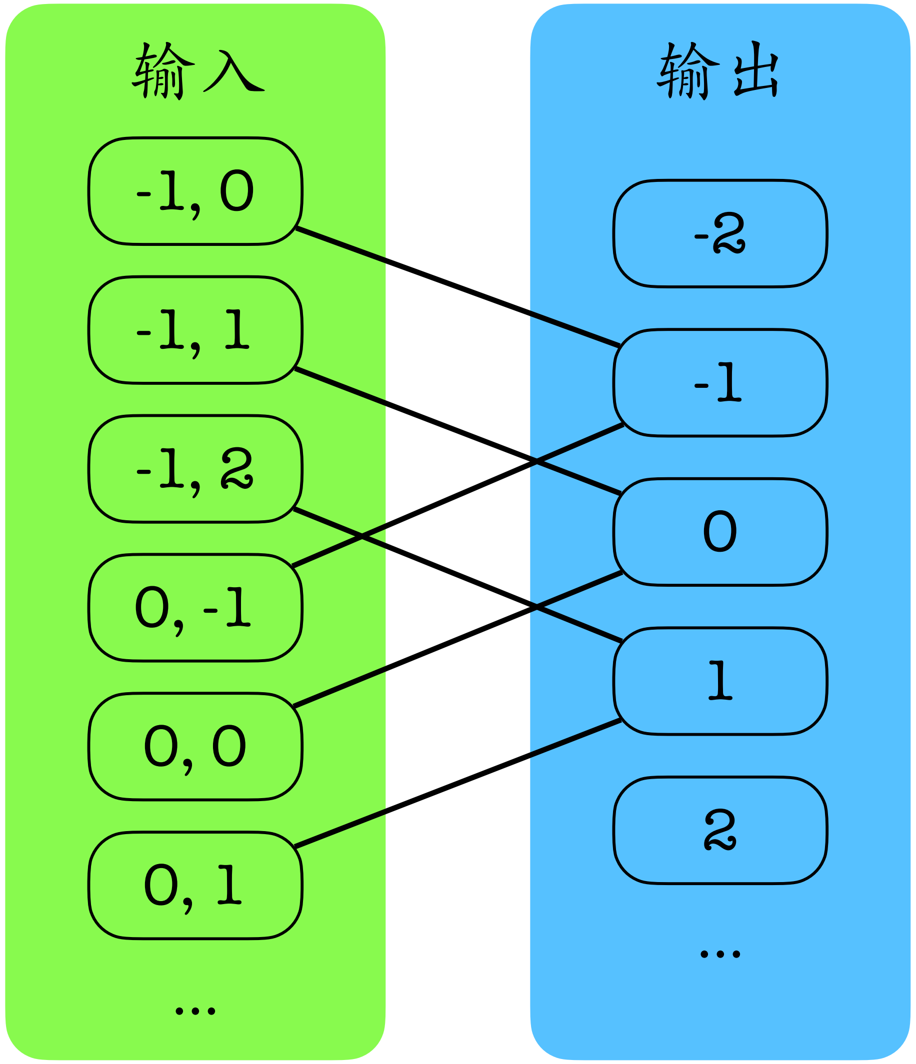
课程说明

- 教材：算法导论（原书第三版）
- 教学内容
  - 基础知识
  - 算法设计策略：分治、动态规划、贪心
  - 问题复杂性：NP问题及其对策（回溯、近似算法）
- 成绩评定
  - 平时成绩20%：出勤 + 平时作业
  - 期中考试20%：闭卷
  - 期末考试60%：闭卷
- 作业
  - 在规定时间前提交至相关FTP服务器
  - 理论作业提交pdf文件，编程作业提交C++源码

基本概念

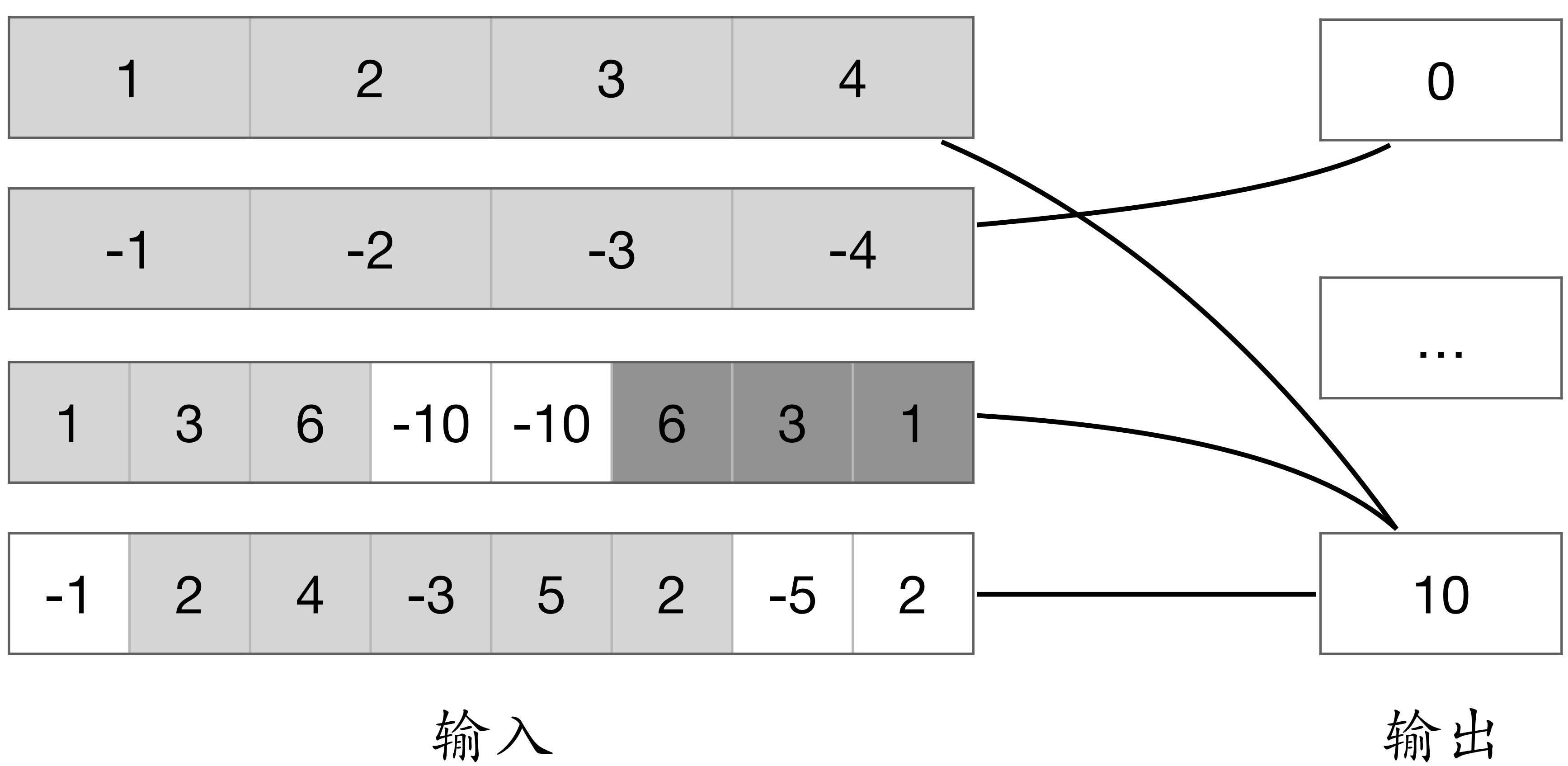
计算问题和算法

- 计算问题：输入和输出之间的二元关系
  - 给定两个整数，计算它们的和
- 每一个输入可能对应零个、一个或者多个输出
- 算法：找到以上二元关系的方法
  - 对于每一个输入，能在有限时间内正确找到其对应的输出
- 算法的效率
- 算法的正确性



最大子数组

- 给定n个整数组成的数组$a[1..n]$，求一个子数组，其所有元素的和最大
  - 子数组和：$S(a[i..j]) = \sum_{k=i}^{j} a[k]$
  - 数组元素可能是负整数
  - 如果所有元素都为负，那么定义最大和为0
    - 此时，最大子数组为空



穷举法求最大子数组

- 穷举：对于任意的子数组$a[i..j]$，计算其和，找到最大者
  - $1 \leq i \leq j \leq n$
- 优化：$S(a[i..j]) = \sum_{k=i}^{j} a[k] = S(a[i..j-1]) + a[j]$

```
int brute_force(const vector<int>& a)
{
    int best = 0;
    for (int i = 0; i < a.size(); i++)
        for (int j = i; j < a.size(); j++) {
            int sum = accumulate(a.begin() + i, a.begin() + j + 1, 0);
            best = max(best, sum);
        }
    return best;
}
```

穷举法求最大子数组

- 穷举：对于任意的子数组$a[i..j]$，计算其和，找到最大者
  - $1 \le i \le j \le n$
- 优化：$\boxed{S(a[i..j])} = \sum_{k=i}^{j} a[k] = \boxed{S(a[i..j-1])} + a[j]$

大问题 小问题

```
int brute_force_opt(const vector<int>& a)
{
    int best = 0;
    for (int i = 0; i < a.size(); i++) {
        int sum = 0;
        for (int j = i; j < a.size(); j++) {
            sum += a[j];
            best = max(best, sum);
        }
    }
    return best;
}
```

归纳法求最大子数组

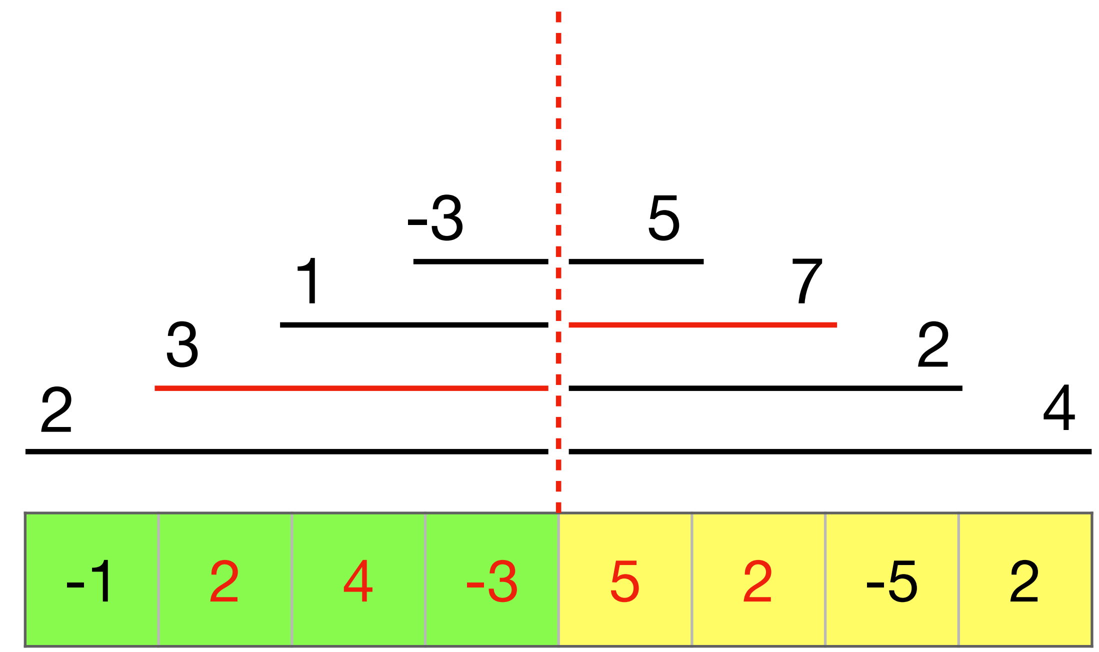
- 归纳：如何基于较小规模的问题来求解较大规模的问题
  - 基本情况：足够小的问题可以直接求解
  - 归纳步骤：建立小问题和大问题之间的关系（递归关系）
- 子问题：求数组$a[i..j]$的最大子数组，其中$1 \le i \le j \le n$
  - 基本情况：如果$i = j$，那么解是$\max\{a[i], 0\}$
  - 归纳步骤：建立递归关系
    - 一种想法：$OPT(a[1..n]) = \max\limits_{1 \le i \le j \le n} \{a[i..j]\}$
    - 另一种想法：$OPT(a[1..n]) = \max\{OPT(Left), OPT(Right), S(Cross)\}$
      - 求$Cross$不是一个子问题

-1	2	4	-3	5	2	-5	2

-1	2	4	-3	5	2	-5	2

均匀划分

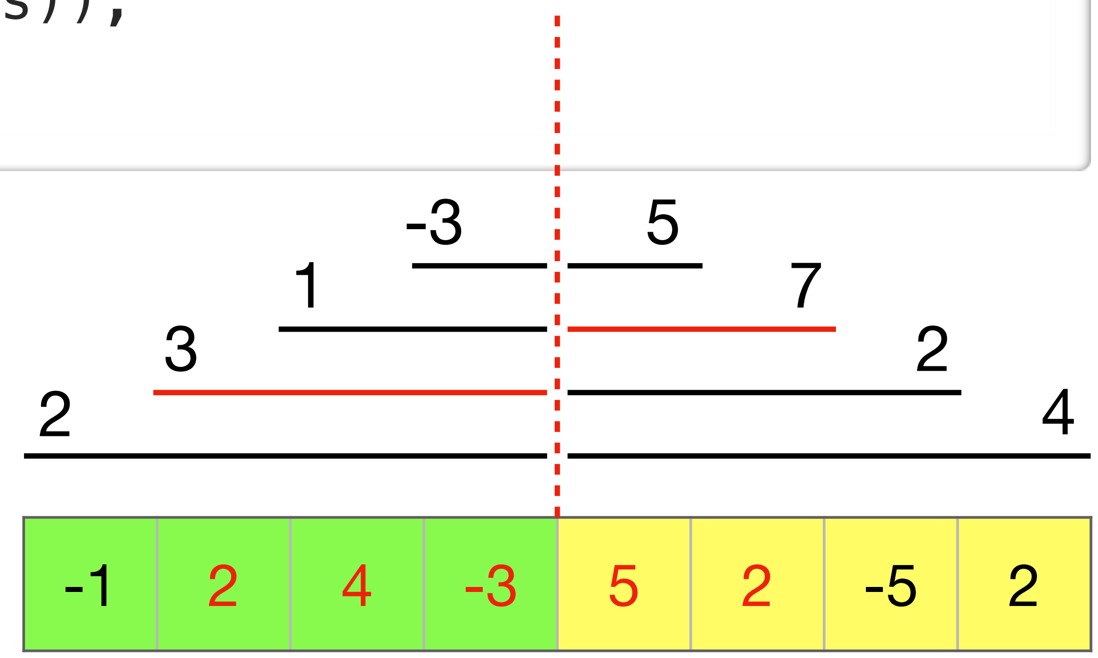
- $OPT(a[1..n]) = \max\{OPT(Left), OPT(Right), S(Cross)\}$
- 将数组分成长度尽量相等的两部分：$a[1..m]$和$a[m+1..n]$，其中 $m = \lfloor (1+n)/2 \rfloor$
- 注意$Cross$的最优性，令$Cross = a[i..j]$，那么
  - $a[i..m]$必然是以$a[m]$结束的最大子数组
  - $a[m+1..j]$必然是以$a[m+1]$开始的最大子数组
- 如何计算$a[i..m]$以及$a[m+1..j]$
  - 从$a[m]$开始，向前扫描数组
  - 从$a[m+1]$开始，向后扫描数组



均匀划分

- $OPT(a[1..n]) = \max\{OPT(Left), OPT(Right), S(Cross)\}$

```
int divide_and_conquer(const vector<int>& a, int i, int j)
{
    if (i == j) return max(a[j], 0);
    int m = (i + j) / 2;
    int left = divide_and_conquer(a, i, m);
    int right = divide_and_conquer(a, m + 1, j);
    int cross = left_mid_max(a, i, m) + right_mid_max(a, m + 1, j);
    return max(left, max(right, cross));
}
```



不均匀划分

- $OPT(a[1..n]) = \max\{OPT(Left), OPT(Right), S(Cross)\}$
- 将数组分成长度尽量不等的两部分：$a[1..n-1]$和$a[n..n]$

```cpp
int divide_and_conquer_imba(const vector<int>& a, int i, int j)
{
    if (i == j) return max(a[j], 0);
    int left = divide_and_conquer_imba(a, i, j - 1);
    int right = max(a[j], 0);
    int cross = left_mid_max(a, i, j - 1) + a[j];
    return max(left, max(right, cross));
}
```

divide_and_conquer_imba(a, 1, 8)

divide_and_conquer_imba(a, 1, 7)

left_mid_max(1, 6)

left_mid_max(1, 7)

$L(1, n) = \max\{a[n], L(1, n-1) + a[n]\}$

优化：不需从头计算$L(1, n)$

-1	2	4	-3	5	2	-5	2

不均匀划分

- $OPT(a[1..n]) = \max\{OPT(Left), OPT(Right), S(Cross)\}$
- 将数组分成长度尽量不等的两部分：$a[1..n-1]$和$a[n\,..\,n]$

```
int divide_and_conquer_imba_opt(const vector<int>& a, int i, int j)
{
    if (i == j) return max(a[j], 0);
    int left = divide_and_conquer_imba_opt(a, i, j - 1);
    int right = max(a[j], 0);
    int cross = left_mid_max_recur(a, j - 1) + a[j];
    return max(left, max(right, cross));
}
```

```
int left_mid_max_recur(const vector<int>& a, int n)
{
    L[n] = (n == 0) ? a[0] : max(L[n - 1] + a[n], a[n]);
    return L[n]; // global variable L = vector<int>(a.size());
}
```

关键的递归关系

- 令$L(k)$表示以$a[k]$结尾的子数组的最大和，那么
  - $k = 1$： $L(k) = a[1]$
  - $k > 1$： $L(k) = \max\{a[k], L(k-1) + a[k]\}$
- $OPT(a[1..n]) = \max\{\max_{1 \le k \le n} \{L(k)\}, 0\}$

```
int dynamic_programming(const vector<int>& a)
{
    int best = 0, sum = 0;
    for (int k = 0; k < a.size(); k++) {
        sum = max(a[k], sum + a[k]);
        best = max(best, sum);
    }
    return best;
}
```

算法正确性的证明

- 对于基于归纳的算法，可以使用数学归纳法来证明其正确性
- 归纳假设：对于某个整数k，算法能够正确地计算：
  - $L[k]$：以$a[k]$结尾的子数组的最大和
  - best：数组$a[1..k]$的子数组的最大和
- 基本情况：$k = 1$，数组只有一个元素$a[1]$，算法执行完毕后，$L[k] = a[1]$，$best = \max\{a[1], 0\}$，显然正确

```
int best = 0, sum = 0;
for (int k = 0; k < a.size(); k++) {
    sum = max(a[k], sum + a[k]);
    best = max(best, sum);
}
return best;
```

算法正确性的证明

- 归纳步骤：证明对于整数$k+1$，算法也能正确地计算$L[k+1]$和best
  - 令以$a[k+1]$结尾的最大子数组是$a[i..k+1]$
    - 那么，要不$i=k+1$，要不$i<k+1$
    - 对于后一种情况，$a[i..k]$必然是以$a[k]$结尾的最大子数组，根据归纳假设，其解为$L[k]$
  - 所以$\max\{a[k+1], L(k)+a[k+1]\}$能够正确地计算$L[k+1]$
  - 显然，算法也能够正确地计算数组$a[1..k+1]$的子数组的最大和best

```
int best = 0, sum = 0;
for (int k = 0; k < a.size(); k++) {
    sum = max(a[k], sum + a[k]);
    best = max(best, sum);
}
return best;
```

算法的时间效率

- 算法的运行时间：对于输入x需要y秒
- 受多种因素影响：机器硬件配置、工作负载、编程语言、编译器、…
- 好的衡量标准：与机器无关，方便使用
- 更关注运行时间的增长率：如果输入是原来的x倍，时间要增加多少？

求解最大子数组的算法的运行时间（单位：微秒）

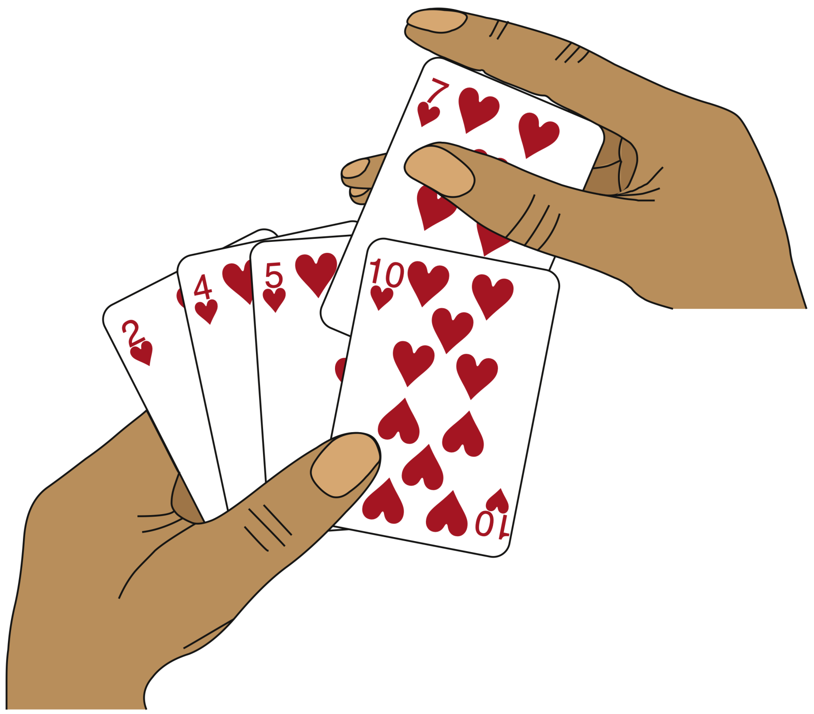
算法（默认C++实现）	数组长度				
	100	1000	10000	100000	1000000
brute_force	1209	865695	-	-	-
brute_force_opt	28	2608	243257	-	-
divide_and_conquer	7	93	1025	8993	96328
divde_and_conquer_imba	19	1468	122222	-	-
divde_and_conquer_imba_opt	4	35	367	?	?
dynamic_programming	1	11	102	1037	9232
dynamic_programming (Python)	18	162	1630	16255	153118

计算模型：Word-RAM

- 对现实计算机的抽象，简化算法分析
- 内存：由一系列连续的内存单元组成，每个内存单元可以存放w个比特，且具有一个地址，地址范围：$0 \sim 2^w - 1$
  - 内存单元也称为word
- 处理器
  - 可以在常数时间内读写一个内存单元
  - 可以在常数时间内对两个内存单元中的内容进行基本的二元运算
    - 加、减、乘、除、模、位运算、逻辑运算
- 输入输出：按内存单元逐一读入或写出
- 算法性能
  - 时间：基本运算的数量
  - 空间：使用内存单元的数量

最坏情况分析

- 某些算法的运行时间和输入有关
  - 插入排序（将数组元素从小到大排列）
    - 最好情况：输入数组元素升序排列
    - 最坏情况：输入数组元素降序排列
- 为什么使用最坏情况分析
  - 给出了任何输入的运行时间的上界
  - 最坏情况经常出现
  - 平均情况往往与最坏情况一样差
  - 仅仅考虑问题规模，而不关注具体输入，具有通用性

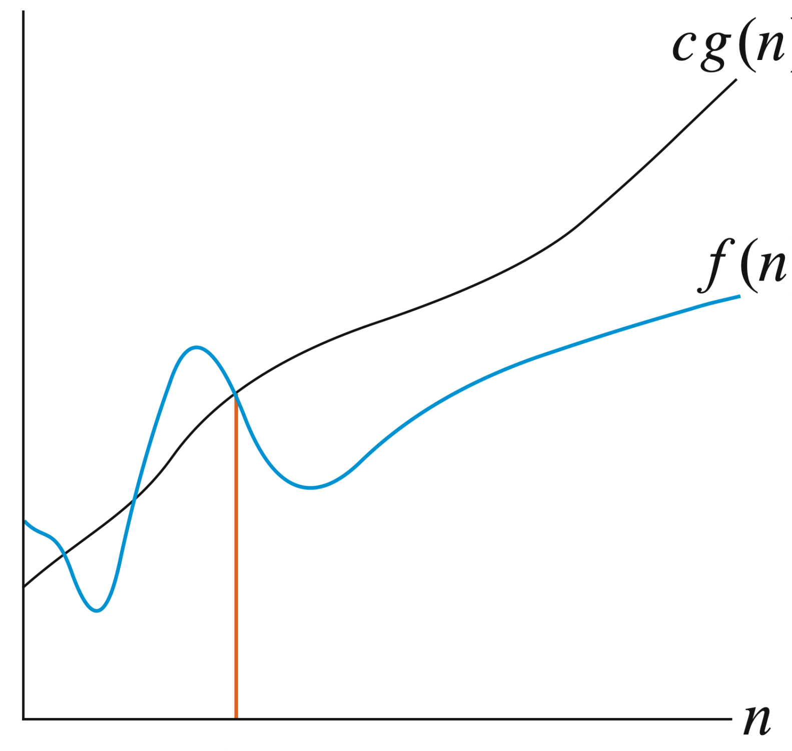


渐近记号

- 最坏情况下算法使用的基本操作的数量
  - $5n^2 + 4n + 3$
- 在分析基本操作数量时，渐近记号忽略
  - 常数因子：仍然和机器、编程语言等因素相关
  - 低阶项：输入规模很大时无关紧要
- 忽略低阶项：$5n^2 + 4n + 3 \rightarrow 5n^2$
- 忽略常数因子：$5n^2 \rightarrow n^2$

O记号

- $O(g(n)) = \{f(n)$: 存在正常量c和n_0，使得对所有$n \geq n_0$，有$0 \leq f(n) \leq cg(n)\}$
- $f(n) = O(g(n))$表示函数$f(n)$属于函数集合$O(g(n))$



- 如果$f(n) = 32n^2 + 17n + 1$，那么$f(n) = O(n^2)$
  - 令$c = 50$，$n_0 = 1$，对于所有的$n \geq n_0 = 1$，$f(n) = 32n^2 + 17n + 1 \leq 50n^2 = cn^2$
- 如何知道$c = 50$，$n_0 = 1$
  - $g(n) = n^2 \Rightarrow c \cdot g(n) = c \cdot n^2$
  - $f(n) = 32n^2 + 17n + 1 \leq 32n^2 + 17n^2 + n^2 = 50n^2$（当$n \geq 1$时）
  - 所以令$c = 50$，$n_0 = 1$

O记号

- $O(g(n)) = \{f(n)$: 存在正常量c和n_0，使得对所有$n \geq n_0$，有$0 \leq f(n) \leq cg(n)\}$
- $f(n) = O(g(n))$表示函数$f(n)$属于函数集合$O(g(n))$
- 如果$f(n) = 32n^2 + 17n + 1$，那么$f(n) = O(n^2)$
- $f(n)$不是$O(n)$，也不是$O(\log n)$
  - 下面通过反证法来证明$f(n)$不是$O(n)$
  - 假设存在c和n_0，使得对所有$n \geq n_0$，有$32n^2 + 17n + 1 \leq c \cdot n$
  - 两边同除以n，有$32n + 17 + \dfrac{1}{n} \leq c$，即$32n \leq c - 17 - \dfrac{1}{n} \leq c$
  - 当$32n > c$时，上面的不等式不成立，所以$f(n)$不是$O(n)$

O记号的性质

- 自反：$f = O(f)$
- 常量：如果$f = O(g)$并且$c > 0$，那么$cf = O(g)$
- 乘积：如果$f_1 = O(g_1)$并且$f_2 = O(g_2)$，那么$f_1 f_2 = O(g_1 g_2)$
- 和：如果$f_1 = O(g_1)$并且$f_2 = O(g_2)$，那么$f_1 + f_2 = O(\max\{g_1, g_2\})$
- 传递：如果$f = O(g)$并且$g = O(h)$，那么$f = O(h)$

- 存在正常量c_1和n_1，使得对于所有$n \geq n_1$，有$0 \leq f_1(n) \leq c_1 g_1(n)$
- 存在正常量c_2和n_2，使得对于所有$n \geq n_2$，有$0 \leq f_2(n) \leq c_2 g_2(n)$
- 所以，对于所有的$n \geq \max\{n_1, n_2\}$
  - 乘积：$0 \leq f_1(n) f_2(n) \leq c_1 c_2 g_1(n) g_2(n)$
  - 和：$0 \leq f_1 + f_2 \leq c_1 g_1 + c_2 g_2 \leq c_3(g_1 + g_2) \leq 2c_3 \max\{g_1, g_2\}$　　$c_3 = \max\{c_1, c_2\}$

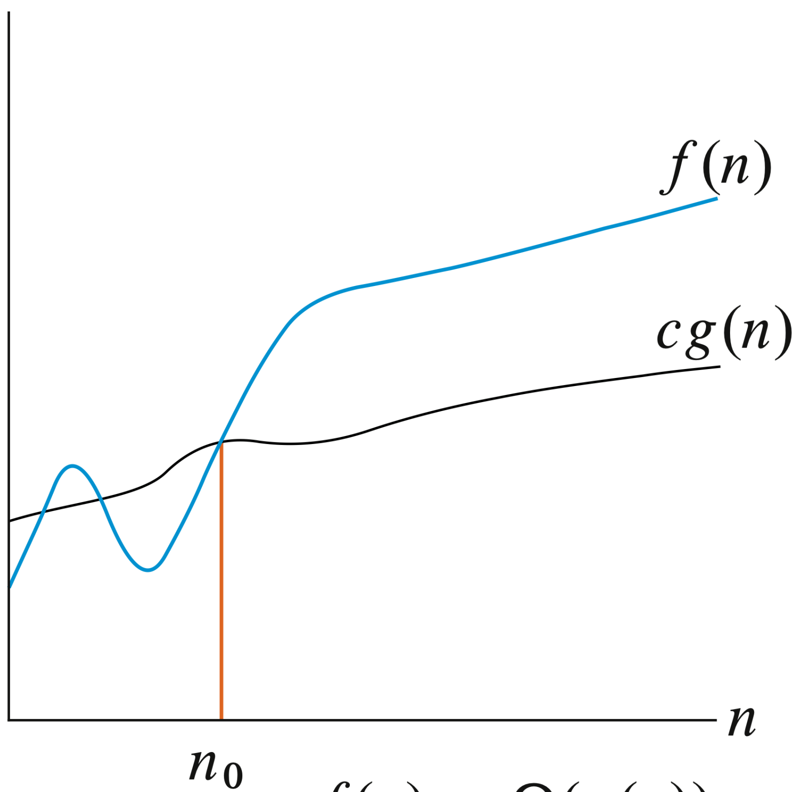
Ω记号

- $\Omega(g(n)) = \{f(n)$: 存在正常量c和n_0，使得对所有$n \geq n_0$，有$0 \leq cg(n) \leq f(n)\}$
- $f(n) = \Omega(g(n))$表示函数$f(n)$属于函数集合$\Omega(g(n))$
- 如果$f(n) = 2n^3 - 7n + 1$，那么$f(n) = \Omega(n^3)$

  令$c = 1$，$n_0 = 3$，对于所有的$n \geq n_0 = 3$

  $f(n) = 2n^3 - 7n + 1$

  $= n^3 + (n^3 - 7n) + 1 \geq n^3 + 1 \geq n^3$



$f(n) = \Omega(g(n))$

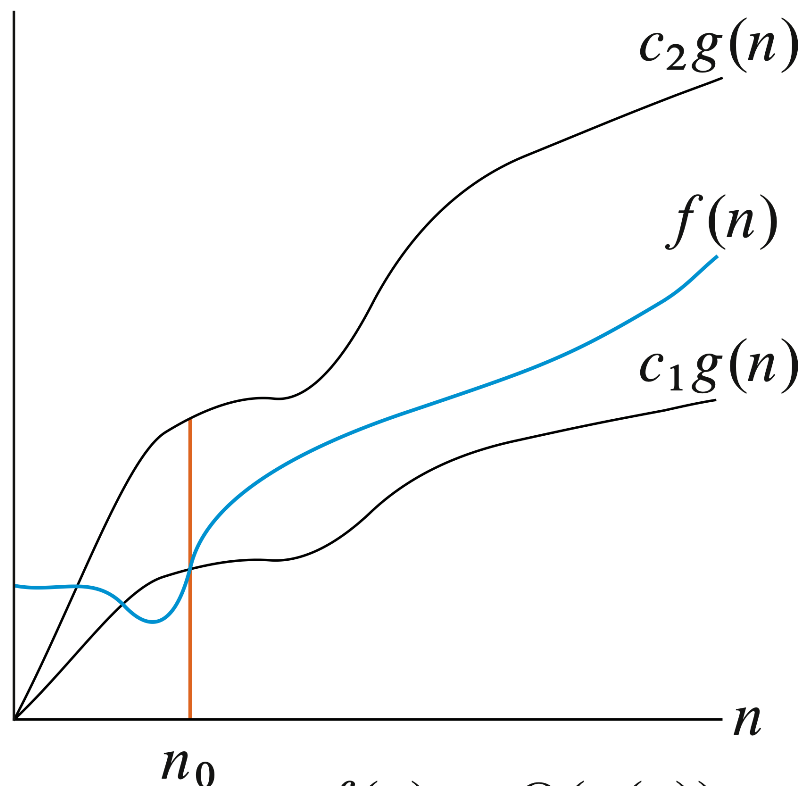
- 如何知道$c = 1$，$n_0 = 3$
  - $g(n) = n^3 \Rightarrow cg(n) = cn^3$
  - $f(n) = 2n^3 - 7n + 1 = n^3 + (n^3 - 7n) + 1$
  - 当$n^3 - 7n > 0$时，即$n > 3$时，可以将$f(n)$进一步缩小为n^3
  - 所以令$c = 1$，$n_0 = 3$

Ω记号

- $\Omega(g(n)) = \{f(n)$: 存在正常量c和n_0，使得对所有$n \geq n_0$，有$0 \leq cg(n) \leq f(n)\}$
- $f(n) = \Omega(g(n))$表示函数$f(n)$属于函数集合$\Omega(g(n))$
- 如果$f(n) = 2n^3 - 7n + 1$，那么$f(n) \neq \Omega(n^4)$
  - 假设存在正常量c和n_0，使得对所有$n \geq n_0$，有$2n^3 - 7n + 1 \geq c \cdot n^4$
    - 看能否推出矛盾，即在某些情况下$2n^3 - 7n + 1 < c \cdot n^4$
  - 显然，$2n^3 - 7n + 1 \leq 2n^3 + 1 \leq 2n^3 + n^3 = 3n^3$（当$n \geq 1$时）
  - 如果$3n^3 \leq cn^4$，那么就能推出矛盾
  - 而$3n^3 \leq cn^4 \Rightarrow n \geq \dfrac{3}{c}$
  - 所以令$n > \max\{1, \dfrac{3}{c}\}$，可以得到$2n^3 - 7n + 1 < c \cdot n^4$，矛盾！

Θ记号

- $\Theta(g(n)) = \{f(n)$: 存在正常量c_1,c_2和n_0，使得对所有$n \geq n_0$，有 $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$



$f(n) = \Theta(g(n))$

- $f(n) = \Theta(g(n))$表示$f(n)$是集合$\Theta(g(n))$的成员
- $f(n) = 32n^2 + 17n + 1$
  - $f(n) = \Theta(n^2)$
  - $f(n) \neq \Theta(n)$，$f(n) \neq \Theta(n^3)$
- $f(n) = \Theta(g(n))$当且仅当$f(n) = O(g(n))$并且$f(n) = \Omega(g(n))$
- 如果$f(n) = \Theta(g(n))$，$g(n)$称为$f(n)$的渐近紧确界
  - 如果$f(n) = O(g(n))$，$g(n)$称为$f(n)$的渐近上界
  - 如果$f(n) = \Omega(g(n))$，$g(n)$称为$f(n)$的渐近下界

渐近记号与极限

- 如果 $\lim_{n\to\infty}\dfrac{f(n)}{g(n)}=c$ ，其中c是一个正常量， 那么 $f(n)=\Theta(g(n))$
  - 根据极限定义，对任意 $\epsilon>0$, 存在 n_0 使得对于所有$n\ge n_0$，有$c-\epsilon\le\dfrac{f(n)}{g(n)}\le c+\epsilon$
  - 令 $\epsilon=\dfrac{1}{2}c>0$，两边同乘以 $g(n)$ ，有 $\dfrac{1}{2}c\cdot g(n)\le f(n)\le\dfrac{3}{2}c\cdot g(n)$
  - 令$c_1=1/2\cdot c$，$c_2=3/2\cdot c$，根据Θ定义，有$f(n)=\Theta(g(n))$
- 如果 $\lim_{n\to\infty}\dfrac{f(n)}{g(n)}=0$，那么$f(n)=O(g(n))$，$f(n)\ne\Omega(g(n))$
  - 此时，也称$f(n)=o(g(n))$，其中o记号表示一个非渐近紧确的上界
  - 比如，$2n=o(n^2)$，但$2n^2\ne o(n^2)$
- 如果 $\lim_{n\to\infty}\dfrac{f(n)}{g(n)}=\infty$，那么$f(n)=\Omega(g(n))$，$f(n)\ne O(g(n))$
  - 此时，也称$f(n)=\omega(g(n))$，其中ω记号表示一个非渐近紧确的下界
  - 比如，$n^2/2=\omega(n)$，但$n^2/2\ne\omega(n^2)$

常见函数的渐近记号

- 多项式函数： Let $f(n) = a_0 + a_1 n + \cdots + a_d n^d$ with $a_d > 0$. Then $f(n)$ is $\Theta(n^d)$.
  - $\lim_{n\to\infty} \frac{a_0 + a_1 n + \cdots + a_d n^d}{n^d} = a_d > 0$
- 对数函数： $\log_a n$ is $\Theta(\log_b n)$ for every $a > 1$ and every $b > 1$.
  - $\frac{log_a n}{\log_b n} = \frac{1}{\log_b a}$
- 对数函数与多项式函数. $\log_a n$ is $O(n^d)$ for every $a > 1$ and every $d > 0$.
  - $\lim_{n\to\infty} \frac{\log_a n}{n^d} = 0$
- 指数函数与多项式函数. n^d is $O(r^n)$ for every $r > 1$ and every $d > 0$.
  - $\lim_{n\to\infty} \frac{n^d}{r^n} = 0$

等式中的渐近记号

- 等式的右边只有渐近记号，比如$n = O(n^2)$
  - 等号实际上是集合的成员关系，即$n \in O(n^2)$
- 等式的右边包含渐近记号，比如$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$
  - $\Theta(n)$代表一个匿名函数$f(n)$，其中$f(n) \in \Theta(n)$
  - 用于隐藏无关紧要的细节
- 等式的左边包含渐近记号，比如$2n^2 + \Theta(n) = \Theta(n^2)$
  - 无论怎样选择等号左边的匿名函数$f(n) \in \Theta(n)$，总有一种办法来选择等号右边的匿名函数$g(n) \in \Theta(n^2)$，使得等式成立，即$2n^2 + f(n) = g(n)$

求解最大子数组算法的效率

- brute_force： $O(n^3)$

```
int brute_force(const vector<int>& a)
{
    int best = 0;
    for (int i = 0; i < a.size(); i++)
        for (int j = i; j < a.size(); j++) {
            int sum = accumulate(a.begin() + i, a.begin() + j + 1, 0);
            best = max(best, sum);
        }
    return best;
}
```

求解最大子数组算法的效率

- brute_force： $O(n^3)$
- brute_force_opt： $O(n^2)$

```cpp
int brute_force_opt(const vector<int>& a)
{
    int best = 0;
    for (int i = 0; i < a.size(); i++) {
        int sum = 0;
        for (int j = i; j < a.size(); j++) {
            sum += a[j];
            best = max(best, sum);
        }
    }
    return best;
}
```

求解最大子数组算法的效率

- brute_force： $O(n^3)$
- brute_force_opt： $O(n^2)$
- divide_and_conquer： $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$

```
int divide_and_conquer(const vector<int>& a, int i, int j)
{
    if (i == j) return max(a[j], 0);
    int m = (i + j) / 2;
    int left = divide_and_conquer(a, i, m);
    int right = divide_and_conquer(a, m + 1, j);
    int cross = left_mid_max(a, i, m) + right_mid_max(a, m + 1, j);
    return max(left, max(right, cross));
}
```

求解最大子数组算法的效率

- brute_force： $O(n^3)$
- brute_force_opt： $O(n^2)$
- divide_and_conquer： $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$
- divide_and_conquer_imba： $T(n) = T(n-1) + O(n) \Rightarrow T(n) = O(n^2)$

```
int divide_and_conquer_imba(const vector<int>& a, int i, int j)
{
    if (i == j) return max(a[j], 0);
    int left = divide_and_conquer_imba(a, i, j - 1);
    int right = max(a[j], 0);
    int cross = left_mid_max(a, i, j - 1) + a[j];
    return max(left, max(right, cross));
}
```

求解最大子数组算法的效率

- brute_force： $O(n^3)$
- brute_force_opt： $O(n^2)$
- divide_and_conquer： $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$
- divide_and_conquer_imba： $T(n) = T(n-1) + O(n) \Rightarrow T(n) = O(n^2)$
- divide_and_conquer_imba_opt： $T(n) = T(n-1) + O(1) \Rightarrow T(n) = O(n)$

```cpp
int divide_and_conquer_imba_opt(const vector<int>& a, int i, int j)
{
    if (i == j) return max(a[j], 0);
    int left = divide_and_conquer_imba_opt(a, i, j - 1);
    int right = max(a[j], 0);
    int cross = left_mid_max_recur(a, j - 1) + a[j];
    return max(left, max(right, cross));
}
```

求解最大子数组算法的效率

- brute_force： $O(n^3)$
- brute_force_opt： $O(n^2)$
- divide_and_conquer： $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$
- divide_and_conquer_imba： $T(n) = T(n-1) + O(n) \Rightarrow T(n) = O(n^2)$
- divide_and_conquer_imba_opt： $T(n) = T(n-1) + O(1) \Rightarrow T(n) = O(n)$
- dynamic_programming： $O(n)$

```
int dynamic_programming(const vector<int>& a)
{
    int best = 0, sum = 0;
    for (int k = 0; k < a.size(); k++) {
        sum = max(a[k], sum + a[k]);
        best = max(best, sum);
    }
    return best;
}
```

求解最大子数组算法的效率

求解最大子数组的算法的运行时间（单位：微秒）

算法 （默认C++实现）	时间 复杂度	数组长度				
		100	1000	10000	100000	1000000
brute_force	$O(n^3)$	1209	865695	-	-	-
brute_force_opt	$O(n^2)$	28	2608	243257	-	-
divide_and_conquer	$O(n \log n)$	7	93	1025	8993	96328
divde_and_conquer_imba	$O(n^2)$	19	1468	122222	-	-
divde_and_conquer_imba_opt	$O(n)$	4	35	367	?	?
dynamic_programming	$O(n)$	1	11	102	1037	9232
dynamic_programming (Python)	$O(n)$	18	162	1630	16255	153118