数据结构与算法分析

华中科技大学软件学院

2014年秋

大纲

1. 时间复杂度的表示
2. 一些算法的复杂度分析
3. 最大子序列和问题的求解

课程计划

- 已经了解
  - 算法的概念
  - 算法的评价
  - 用clock ()来度量执行时间

课程计划

- 已经了解
  - 算法的概念
  - 算法的评价
  - 用clock()来度量执行时间
- 即将学习
  - 时间复杂度的O表示
  - 一些常用的算法和分析
  - 递归与迭代的转换
  - 最大子序列和问题的求解

Roadmap

1. 时间复杂度的表示
2. 一些算法的复杂度分析
3. 最大子序列和问题的求解

Big O

- 需要定义算法的时间复杂度
  - 不必非常精确
  - 通常只需要了解其上界，相对简单

Definition

1. $f(n) = O(g(n))$, if $\exists c > 0 : c * g(n) \geq f(n)$
2. $f(n) = \Omega(g(n))$, if $\exists c > 0 : c * g(n) \leq f(n)$
3. $f(n) = \Theta(g(n))$, if $\exists c1 > 0, c2 > 0 : c1 * g(n) \leq f(n) \leq c2 * g(n)$

- $f(n) = \Theta(g(n))$, if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

特殊情况

- 考虑下列情况：$f(x) = x, g(x) = 1/x$
- $\lim_{x \to 0^+} g(x) = \infty$
- 但是，当$x > 1$，$g(x) > f(x)$
- 对于$f(x)$和$g(x)$，哪一种关系成立？

松弛要求

- 根据定义，$f(x)$和$g(x)$无法用O，Ω，Θ表达：需要把条件松弛一些
- 松弛：开始时并不重要，因为只有一小段与后面的规则不一致（从0到1）
- O，Ω，Θ只要求不等式“最终”成立
- 例如，$f(x) < c * g(x)$，$\forall x > x_0$，$x_0 = 1$
- 即使如此，再某些情况下也无法找到合适的不等式

对数换底

- $\log_b X = \Theta(\log_c X), \forall b > 1, c > 1$
- 证明： $\log_b X = \frac{\log_c X}{\log_c b}$
- 而$\log_c b$只是一个常系数
- 同样，对数内部的表达式通常影响不大
- 例如， $\log(n^{10000}) = O(?)$

一些常见的算法复杂度

复杂度	名称	例子
O(1)	常数	A[0], A[10000], 2+3, 2*3
O(log n)	对数	折半查找
O(n)	线性	文件下载、无序查找
$O(n^2)$	二次型	插入排序
$O(n^k)$	多项式	矩阵乘法
$O(2^n)$	指数	三色图、蛋白质折叠
O(n!)	阶乘	TSP

复杂度函数的运算规则

- $O(T_1(n) + T_2(n)) = \max(O(T_1(n)), O(T_2(n)))$
- 如果T(n) 是阶数为k的任意多项式，
  则$O(T(n)) = O(n^k)$
- $O(T_1(n) * T_2(n)) = O(T_1(n)) * O(T_2(n))$
- $O(dominant\ terms + others) = O(dominant\ terms)$
- $O(T_1(n) - T_2(n)) = unknown$

0和=

- 对多项式$f_1(n) = 3n^2 - 1000n + 25$，有 $f_1(n) = 0(n^2)$
- 同样，对$f_2(n) = 2n^2 + 5$，有 $f_2(n) = 0(n^2)$
- 是否意味着$f_1(n) = f_2(n)$?
- 显然，若果 x = y 且 y = z，则 x = z
- 对于用0表示的复杂度，结合律不成立，=等价于$\in$

0表示中的常数

- 常系数无关紧要，可以丢弃
- 低阶项无关紧要，可以不要
- 以常数为底的对数函数中的常数指数也可以省略
- 能否去掉所有的指数？$0(n) \equiv 0(n^2)$?

硬件与算法

- 需要更快的硬件还是更好的算法？有些情况下，硬件无法弥补算法的差异
- 假设算法A_2是$O(n^2)$，A_1是$O(n)$，能否找两个计算机M_2，M_1 使得两个算法在相同时间内运行？
- 如果M_2 比M_1快10,000 倍，但输入规模为10,000时，$M_1(A_1, 10,000) = 10,000, M_2(A_2, 10,000) = 10,000^2/10000 = 10000$
- 但是，如果规模到达20,000， $M_1(A_1, 20,000) = 20,000 M_2(A_2, 20,000|) = 20,000^2/10000 = 40000$

Roadmap

1. 时间复杂度的表示
2. 一些算法的复杂度分析
3. 最大子序列和问题的求解

插入排序代码

```
void insSort(int *a, int n)
{
    int i, j, key;

    for (i = 1; i < n; i++)
    {
        key = a[i];
        for (j = i-1; j >= 0 && a[j] > key; j--)
        {
            a[j+1] = a[j];
            a[j] = key;
        }
    }
}
```

插入排序的复杂度

- 对于给定长度的数组，第一重循环于数组中数的排列无关
- 第二重循环依赖于数的排列情况
- 例如，对于数组$\{1, 2, 3, 4, 5, 6\}$和$\{6, 5, 4, 3, 2, 1\}$，插入排序分别需要多少次比较和交换？

插入排序的复杂度

- 对于给定长度的数组，第一重循环于数组中数的排列无关
- 第二重循环依赖于数的排列情况
- 例如，对于数组$\{1,2,3,4,5,6\}$和$\{6,5,4,3,2,1\}$，插入排序分别需要多少次比较和交换？
- 插入排序的时间复杂度
  - 最好情况，顺序排列：$O(n)$
  - 最坏情况，倒序排列：$O(n^2)$
  - 平均情况：$O(n^2)$，为什么？

最大公约数的欧几里德算法：递归代码

```
int gcd (int m, int n)
{
    if (0 == n)
        return (m);

    return (gcd (n, m%n));
}
```

很容易改写为迭代形式：递归发生在return内，只需修改输入参数

欧几里德迭代形式

```
int gcd (int m, int n)
{
    int tmp;

    while (n != 0)
    {
        tmp = m;
        m = n;
        n = tmp % n;
    }
    return (m);
}
```

Euclid算法的步骤

gcd（2014，1113）

- $r = m\%n = 2014\%1113 = 901, m = 1113, n = 901$

Euclid算法的步骤

gcd（2014，1113）

- $r = m\%n = 2014\%1113 = 901, m = 1113, n = 901$
- $r = 1113\%901 = 212, m = 901, n = 212$

Euclid算法的步骤

gcd（2014，1113）

- $r = m\%n = 2014\%1113 = 901, m = 1113, n = 901$
- $r = 1113\%901 = 212, m = 901, n = 212$
- $r = 901\%212 = 53, m = 212, n = 53$

Euclid算法的步骤

gcd（2014，1113）

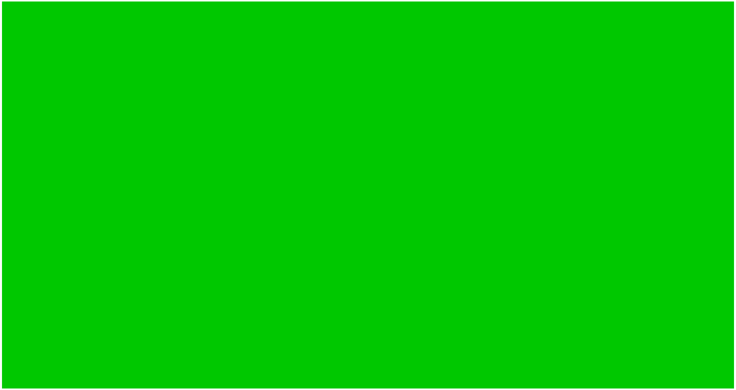
- $r = m\%n = 2014\%1113 = 901, m = 1113, n = 901$
- $r = 1113\%901 = 212, m = 901, n = 212$
- $r = 901\%212 = 53, m = 212, n = 53$
- $r = 212\%53 = 0, m = 53, n = 0$

Euclid算法的步骤

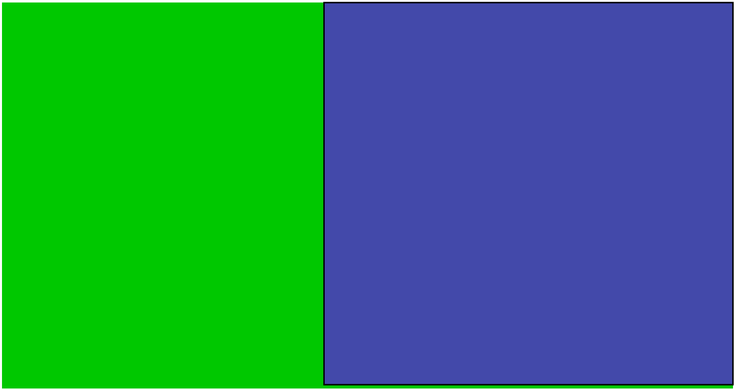
gcd（2014，1113）

- $r = m\%n = 2014\%1113 = 901, m = 1113, n = 901$
- $r = 1113\%901 = 212, m = 901, n = 212$
- $r = 901\%212 = 53, m = 212, n = 53$
- $r = 212\%53 = 0, m = 53, n = 0$
- gcd（2014，1113） = 53

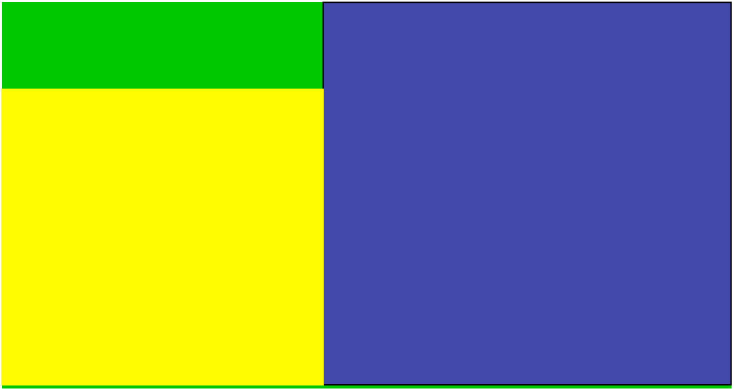
Euclid算法的步骤



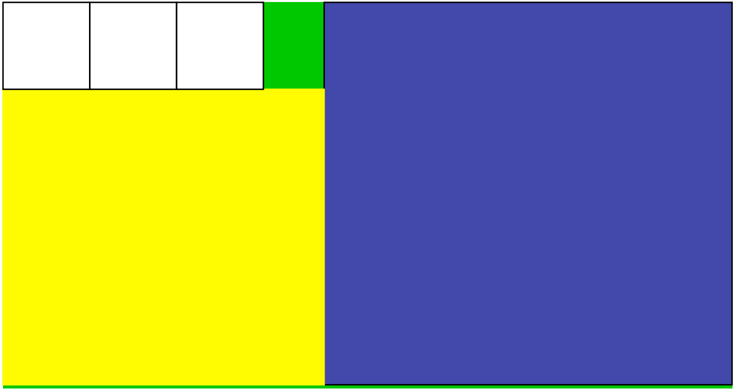
Euclid算法的步骤



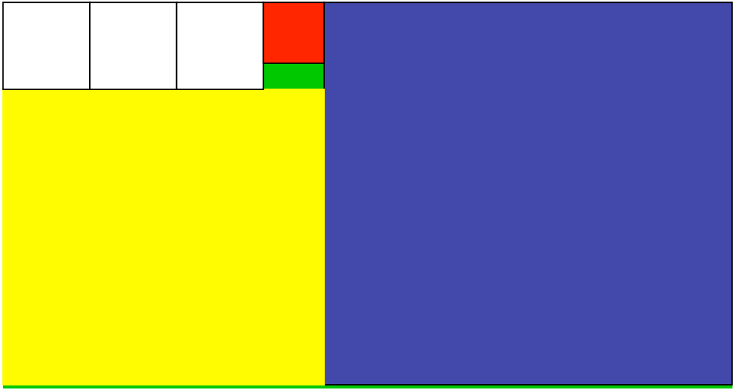
Euclid算法的步骤



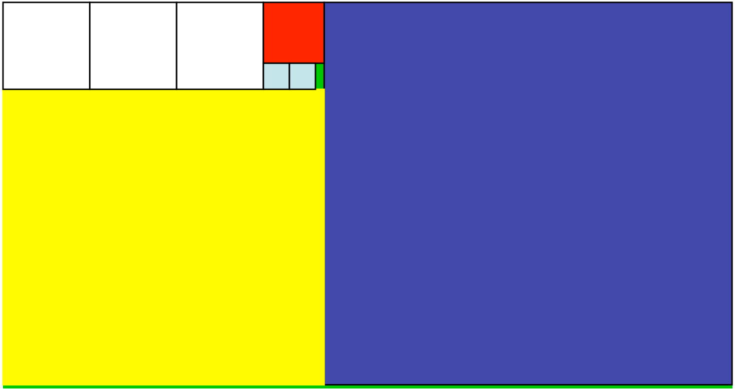
Euclid算法的步骤



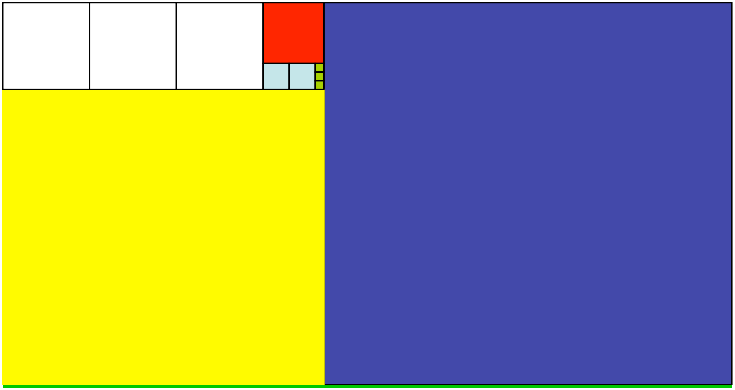
Euclid算法的步骤



Euclid算法的步骤



Euclid算法的步骤



Euclid算法的正确性

- 令m_k, n_k为第k次迭代时的参数，根据参数的计算过程

$$\begin{bmatrix} m_k \\ n_k \end{bmatrix} = \begin{bmatrix} q_k & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} n_k \\ m_k \% n_k \end{bmatrix} = \begin{bmatrix} q_k & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} m_{k+1} \\ n_{k+1} \end{bmatrix}$$

- 如果N次迭代之后，参数变为$r, 0$，那么

$$\begin{bmatrix} m \\ n \end{bmatrix} = \left(\prod_{k=1}^{N} \begin{bmatrix} q_k & 1 \\ 1 & 0 \end{bmatrix} \right) \begin{bmatrix} r \\ 0 \end{bmatrix}$$

- 根据矩阵运算，可以得出r是m和n的一个公约数。假设r不是最大公约数，记最大公约数为$g > r$，则g是n和m%n的约数。重复这一过程，可知g是r和0的约数。因此，r一定是最大公约数

Euclid算法的复杂度

- 什么样的参数能够使得算法快速得出r和0？当n是m的约数的时候，算法立刻退出
- 什么情况下算法进展缓慢？当n一直不是m的约数时，即m和n始终互质
- 考虑Fibonacci数列，$f_n = f_{n-1} + f_{n-2}$，$f_0 = 1, f_1 = 1$
- 如果$m = f_k$，而$n = f_{k-1}$时，Euclid算法的执行过程时什么样的？计算gcd（21，13）试一试
- 可以知道，Euclid算法在最坏情况下的时间复杂度是$\Omega(\log_\phi n)$，其中ϕ大约是1.618

Euclid算法的复杂度

- 如果 $m > n$，则$m\%n < m/2$。分为两种情况
  - $n \leq m/2 : m\%n < n \leq m/2$
  - $n > m/2 : m/2 < n < m, m\%n = m - n < m/2$
- 每经过两次迭代，m为m%n替代：首先m, n变为n, m%n，然后n, m%n变为m%n, n%(m%n)
- 因此，经过不超过$2\log m$次迭代，要么参数变为(1，0)，要么找到不为1的最大公约数
- Euclid算法是$O(\log n)$

有序数组中的查找

折半查找：如果找到x，返回其索引；否则返回−1

```
int bin_search (int a[], int x,
                int first, int last)
{
    int mid = (first + last)/2;

    if (first > last)
        return (-1);
    if (x > a[mid])
        return (bin_search (a, x, mid+1, last));
    if (x < a[mid])
        return (bin_search (a, x, first, mid - 1));

    return (mid);
}
```

改写折半查找为迭代形式

```
int bin_search (int a[], int n, int x)
{
    int first = 0, last = n - 1;
    int mid = (first + last)/2;

    for (; first <= last; mid = (first + last)/2)
    {
        if (x > a[mid])
            first = mid + 1;
        else if (x < a[mid])
            last = mid - 1;
        else
            return (mid);
    }
    return (-1);
}
```

折半查找算法的时间复杂度

- 循环执行的次数取决于是否$x = a[mid]$条件成立
- 最好情况：第一次循环即有$x = a[mid]$， $O(1)$
- 最坏情况：始终无法找到x，循环结束时$first > last$， $O(\log n)$
- 每一次循环之后，搜索空间小于原来的一半，因此叫做折半查找
- x不在数组中，当$first > last$时表明找不到x，退出
- 折半$k+1$次后，数组长度为0，即$\frac{n}{2^k} = 1, k = \log n$

折半查找算法的时间复杂度

- 平均情况？如果每个元素各不相同，在找得到时，一次循环可以找到1个数，2次循环可以找到2个数，3次循环可以找到4个数，...，log n次循环之后可以找到n/2个数
- 因此平均时间复杂度为?

折半查找算法的时间复杂度

- 平均情况？如果每个元素各不相同，在找得到时，一次循环可以找到1个数，2次循环可以找到2个数，3次循环可以找到4个数，...，$\log n$次循环之后可以找到n/2个数
- 因此平均时间复杂度为?
- $S = \sum_{i=1}^{\log n} i * 2^{i-1}$，等比差数列
- $2S - S = n\log n - n$
- 总循环次数$S \approx n(\log n - 1)$，平均复杂度为$O(\log n)$

分治的思想

- For complex problems, we may
  - Divide the original problem into smaller sub-problems
  - Solve the sub-problems
- Use the solutions to sub-problems to construct a solution to the original one This can be implemented with recursions, if the same methodology can be used to solve sub-prolems

求整数幂

- 如何使用较少的乘法次数求x^{27}?
- 缓存中间结果，避免重复计算

求整数幂

- 如何使用较少的乘法次数求x^{27}?
- 缓存中间结果，避免重复计算
- $x^3 = x * x * x$， $x^9 = x^3 * x^3 * x^3$， $x^{27} = x^9 * x^9 * x^9$
- $x^2 = x * x$， $x^4 = x^2 * x^2$， $x^8 = x^4 * x^4$，
  $x^{16} = x^8 * x^8$， $x^{27} = x^{16} * x^8 * x^2 * x$

分治

- 问题的规模是n，把n分解
- 如果n是偶数，$n = 2 * \frac{n}{2}$；否则$n = 2 * \frac{n}{2} + 1$
- 因此，$x^0 = 1$

$$x^n = \begin{cases} (x^2)^{\frac{n}{2}} & n \text{ is even} \\ (x^2)^{\frac{n}{2}} * x & \text{otherwise} \end{cases}$$

- 最坏情况，n始终为奇数

求幂的递归代码

```
int power (int x, int n)
{
    if (0 == n)
        return (1);

    if (0 == n % 2)
        return (power (x*x, n/2));

    return (power (x*x, n/2)*x);
}
```

改写递归为迭代

- Tail recursion, recursive calls happen inside return statement Always easy to convert
- May use temps to store recursive arguments and/or intermediate results
- Recursive algorithms usually go top-down towards one of the trivial cases
- Iterations often go bottom up to build the solution

求幂的迭代代码

```
int pow (int x, int n)
{
    int res = 1;

    if (n == 0)
        return (1);

    for (; n > 0; n = n >> 1)
    {
        if (n % 2 == 1)
            res *= x;
        x *= x;
    }
    return (res);
}
```

复杂度

- 循环次数为log n次
- 最好情况下的乘法次数为log n次：n%2始终为0
- 最坏情况下的乘法次数为2 log n次：n%2始终为1
- 算法复杂度O(log n)

其他分解方式

- 问题的规模是n，仍然按照n分解
- 如果n是偶数，$n = n/2 + n/2$；否则$n = n/2 + n/2 + 1$
- 因此，$x^0 = 1$

$$x^n = \begin{cases} x^{\frac{n}{2}} * x^{\frac{n}{2}} & n \text{ is even} \\ x^{\frac{n}{2}} * x^{\frac{n}{2}} * x & \text{otherwise} \end{cases}$$

- 最坏情况，还是n始终为奇数的情况

递归代码

```
int power (int x, int n)
{
    if (0 == n)
        return (1);

    if (0 == n % 2)
        return (power (x, n/2)*
                power (x, n/2));

    return (power (x, n/2)*
            power (x, n/2)*x);
}
```

复杂度

- Denote the complexity of power (x, n) T(n)
- It follows that T(n) = 2T(n/2) + O(1), because power (x, n/2) is recursively called twice in the code (repeated thus redundant)
- For simplicity, let

$$\begin{cases} T(0) = 1 \\ T(n) = 2T(n/2) + 1 \end{cases}$$

- It is obvious that
  $T(n) + 1 = 2(T(n/2) + 1) = 2^{\log n}(T(1) + 1)$, so
  $T = O(n)$

解决办法

```
int power (int x, int n)
{
    int tmp;

    if (0 == n)
        return (1);

    tmp = power (x, n/2);
    if (0 == n % 2)
        return (tmp*tmp);

    return (tmp*tmp*x);
}
```

以3为底

- 基础情况

$$x^n = \begin{cases} 1 & n = 0 \\ x & n = 1 \\ x * x & n = 2 \end{cases}$$

- 递归过程

$$x^n = \begin{cases} (x^3)^{\frac{n}{3}} & n\%3 = 0 \\ (x^3)^{\frac{n}{3}} * x & n\%3 = 1 \\ (x^3)^{\frac{n}{3}} * x * x & n\%3 = 2 \end{cases}$$

- 最坏情况，n%3始终为2

以3为底的递归代码

```
int power (int x, int n)
{
    if (n == 0)
        return (1);
    if (n == 1)
        return (x);
    if (n == 2)
        return (x*x);

    if (n % 3 == 0)
        return (power (x*x*x, n/3));
    if (n %3 == 1)
        return (power (x*x*x, n/3)*x);
    return (power (x*x*x, n/3)*x*x);
}
```

改写为迭代

- Remember all tail recursions can be rewritten into iterations by simply writing arguments transformation and inductive construction explicitly
- In the previous recursion, arguments $n / = 3, x = x * x * x$
- Construction power (x, n) = power (x*x*x, n/3) or multiplied by 1 or 2 x's when needed

以3为底的迭代代码

```
int power (int x, int n)
{
    int res = 1;
    if (n == 0)
        return (1);

    for (; n > 0; n /=3, x = x*x*x)
    {
        if (n % 3 == 1) res *= x;
        if (n % 3 == 2) res *= x*x;
    }

    return (res);
}
```

幂的计算过程

- Think of representing any integer as binary, say, 27 = 11011B
- What divide and conquer did was
    1. Set the result to 1
    2. if no more digit to check, return result
    3. check if a digit is 1, If it is, multiply the result with x
    4. Replace x with x*x, go to 2
- Then, using tertiary, $27 = (1000)_3$, and divide n by 3, 6 multiplications
- What about more aggressive algorithms? Say, divide n by 10?

最坏情况的时间复杂度

- When will the algorithm go to the worst case? Always take the most time-consuming path, denote the time needed for input n as T(n)
- For 3-based power calculation, n%3 is always 2, then we need 4 more multiplications in each iteration or recursion, $T(n) = T(n/3) + 4$, $T(0) = 0$
- Remember in C, $n/3$ = round down
- For 2-based power calculation, $T(n) = T(n/2) + 2$, $T(0) = 0$

寻找质数

- Problem: find all prime numbers between 3 and a given n
- May check all numbers sequentially, if it can be divided by any integer in [2.. Sqrt(N)], skip it, complexity? (recall the convexity of sqrt(x))
- Or use Erastothenes method: start with an array from 2 to n, , find the minimum I in the array, delete all ki’s; if I reaches sqrt(n), stop
- What's left is a prime
- Estimate the time complexity

质数代码

```
for (i = 2; i*i <= n; i++)
{
    if (a[i] == -1)
        continue;
    for (k = i; k*i < n; k++)
    {
        if (a[k] == -1)
            continue;
        a[k*i] = -1;
    }
}
```

筛选过程

- $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, \cdots$
- First delete 2i, $i > 1$
  $1, 2, 3, -1, 5, -1, 7, -1, 9, -1, 11, -1, 13, -1, 15, -1, \cdots$
- Delete 3i, $i > 1$
  $1, 2, 3 -1, 5, -1, 7, -1, -1, -1, 11, -1, 13, -1, -1, -1, \cdots$
- Delete 4i, $i > 1$
  $1, 2, 3 -1, 5, -1, 7, -1, -1, -1, 11, -1, 13, -1, -1, -1, \cdots$
- For harmonic series

$$\sum^{n} \frac{1}{k} = \ln n + \lambda + \epsilon_n$$

Roadmap

问题描述

- Problem: given list of integers (positives and negatives): a[1], ⋯, a[n], find max value of $\sum_{i=j}^{k} a[i]$, $\forall i, j, 1 \leq i \leq j \leq n$
- e.g. −2, 11, −4, 13, −5, −2
- Answer: 20
- If all numbers are negative, just return 0
- How to solve?

三重循环

```
int maxsub1 (int a[], int n)
{
    int maxsum = 0, currsum = 0;
    int i, j, k;

    for (i = 0; i < n; i++)
        for (j = i; j < n; j++)
        {
            for (currsum = 0, k = i; k <=j; k++)
                currsum += a[k];
            if (currsum > maxsum)
                maxsum = currsum;
        }
    return (maxsum);
}
```

复杂度

- 对每一对有效的(i, j)，用j-i次循环求和求出一个子序列的和，选出最大值
- 三重循环：第一重针对i，循环n次；第二重针对j，循环n-i次；第三重求和，循环j-i次
- 复杂度$\sum_{i=1}^{n}(\sum_{j=i}^{n}(j-i))$
- $\sum_{j=i}^{n} j-i = \frac{(n-i)(n-i+1)}{2} = \frac{(n-i)^2}{2} + \frac{n-i}{2}$
- $\sum_{i=1}^{n}(\frac{(n-i)^2}{2} + \frac{n-i}{2}) = 0(n^3)$
- 能不能做得更好？去除重复计算

两重循环

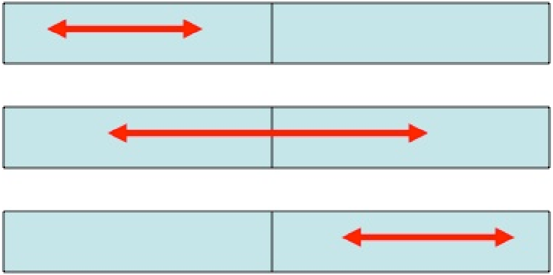
```
int maxsub2 (int a[], int n)
{
    int maxsum = 0, currsum = 0;
    int i, j;

    for (i = 0; i < n; i++)
    {
        for (currsum = 0, j = i; j < n; j++)
        {
            currsum += a[j];
            if (currsum > maxsum)
                maxsum = currsum;
        }
    }
    return (maxsum);
}
```

复杂度

- 对每一对有效的(i, j)，用n-i次循环求和求出n-i个子序列的和，选出最大值
- 复杂度$\sum_{i=1}^{n}((n-i))$，$O(n^2)$
- 能不能做得更好？

分治

- Divide the sequence to reduce the search space
- Where can the max subsequence be in the original sequence?
- Look at [1, 2, 3, −4, −5, 6, 7, 8]



分治

- Recursive algorithm
- Idea: divide sequence in half
  - Find max subsequence in first half,
  - Max subsequence in second half
  - Max subsequence overlapping
- To find max overlapping,
  - Find max at end of first half
  - Find max at start of second half
  - Combine

分治代码

```
int max_subseq (int a[], int first, int last)
{
    int mid = (first + last)/2;
    int max_left, max_right, max_mid;

    if (first > last)
        return (0);
    if (first == last)
        return (a[first]);

    max_left = max_subseq (a, first, mid);
    max_right = max_subseq (a, mid + 1, last);
    max_mid = find_overlap (a, first, mid, last);
    if (max_left > max_right)
        if (max_mid > max_left)
            return (max_mid);
        else
           return (max_left);
    else if (max_mid > max_right)
       return (max_mid);
    return (max_right);
}
```

分治代码

```
int find_overlap (int a[], int first, int mid,
                  int last)
{
    int i, sum1 = 0, sum2 = 0, sum;
    for (i = mid, sum = 0; i >= first; i--)
    {
        sum += a[i]
        if (sum > sum1)  sum1 = sum;
    }
    for (i = mid + 1, sum = 0; i <= last; i++)
    {
        sum += a[i]
        if (sum > sum2) sum2 = sum;
    }
    return (sum1 + sum2);
}
```

分治的复杂度

- Repeat the common procedures:
  1. Divide arguments into groups, until trivial cases found and solve the trivial cases
  2. Use recursive steps to get solutions to small groups
  3. Use non-recursive steps to combine small group solutions to form the final solution
- Use $T(n)$ to get recursive relations from steps 2 and 3, and get the trivial case complexity from step 1 (often $O(1)$)
- If always divide into halves, it takes $\log n$ steps to get to the trivial case $n = 1$
- Non-recursive step may take $O(1)$, $O(n)$ or bigger

复杂度

- How long does recursive algorithm take?
- Denote the time needed for n numbers T(n)
- Base case: length 1, T(1) = 1
- For larger n, apply to both halves, and compare the results of both halves, as well as the combined (may have to look at the whole array)

$$T(n) = 2T(n/2) + O(n)$$

- T(n) = O(nlogn), only dominant item matters

线性算法

- Observation 1: no subsequence starting with a negative can be the best (minimal) one
- Observation 2: More generally, a negative subsequence can’t start the best subsequence
- Observation 3: if find that sequence i..j is negative, can jump from i to j+1
- Fear: miss good subsequence in the middle
- Not the case

一重循环

```
int maxsub (int *a, int n)
{
    int maxsum = 0;
    int currsum = 0, i;
    for (i = 0; i < n; i++)
    {
        currsum += a[i];
        if (currsum > maxsum)
            maxsum = currsum;
       else if (currsum < 0)
            currsum = 0;
    }
    return (maxsum);
}
```

小结

- 算法的时间复杂度可以表示成在RAM上执行的简单操作的次数之和，用输入规模的函数表示
- O, Ω, Θ分别表示函数的上界、下界和等价关系，函数中的主导项起到关键作用
- 分析了插入排序、欧几里德算法、折半查找、求幂、寻找质数等问题的时间复杂度
- 分治算法：将问题分解，用小问题的解构建出原问题的解，由递归步骤和非递归步骤构成
- 尾递归算法可以直接转换为非递归实现，参数演化和非递归步骤的构建
- 分析问题、理解问题、设计算法以减少不必要的计算

实验2

- 用基于2和3的方式分别写出算法来求power(n, x)。分析两种算法的复杂程度，设计试验来验证你的想法
- 教材中的2.19，设计并实现使用分治求数组的主元的算法。如果不用分治，通过比较和计数，复杂程度是多少？
- 实验报告要求由以下部分构成：问题描述、问题分析与算法设计、实验方案/步骤、算法实现、测试结果与分析