伪递归

> 至多一次递归调用，且调用后立刻return

EuclidGCDRec(m, n):

```
if (n==0)
  return m
else
  rem=m%n
  return EuclidGCDRec(n, rem)
```

- 特点：最底层返回值，层层都一样

转换为迭代

> 优点，算到最底层时不用像递归一样层层回溯

```
EuclidGCDIter (m, n):
while (true)
  if (n==0)
    return m
  else
    rem=m%n
    m=n
    n=rem
```

分治法举例

```
Solve(I):
if (I is small enough)
  solution = DirectSolve(I)
else
  <I_1,I_2,…,I_k> = DivideProblem(I)
  for (j=1 to k)
    solution_j = Solve(I_j)
  solution = Combine(solution_1,…,solution_k)
return solution
```

Or, use brute-force if (sub)problem is simple.

Divide the problem into smaller subproblems.

Recursively solve subproblems.

Combine solutions of subproblems to get solution for original problem.

1. 划分成小问题
2. 递归解决小问题
3. 组合

- [ ] 证明正确性

> 归纳法

- 归纳基础：规模足够小时，可以解决

1、归并排序mergesort

> 对已排好序的两组再排序
> 时间复杂度线性

- 证明正确性，对输入规模归纳

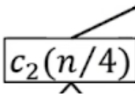
```
Mergesort(A[1,n]):
if (n==1)
    sol=A
else
    left[1,n/2]=Mergesort(A[1,n/2])
    right[1,n/2]=Mergesort(A[n/2+1,n])
    sol[1,n]= Merge(left[1,n/2],right[1,n/2])
return sol[1,n]
```

- 时间复杂度固定

A **recurrence** equation:

$$\begin{cases} T(1) = c_1 \\ T(n) = 2 \cdot T(n/2) + c_2 \cdot n \end{cases}$$

$\lg n + 1$ levels.
Each level incur $\Theta(n)$.
Total cost is $\Theta(n \lg n)$.



merge

```
Merge(L[1,a],R[1,b]):
for (i = 1 to a )do
    for (j=1 to b)do
        while(L[i]<R[j])
            A.append(L[i])
            i+=1
        A.append(R[j])
一旦1个已赋值完，另一个直接加到末尾
```

- 证明正确性：寻找循环不变量

迭代化

```
MergeSortIter(A[1...n]):
Deque Q
for (i=1 to n)
  Q.AddLast(A[i])
while (Q.Size()>1)
  L=Q.RemoveFirst(),R=Q.RemoveFirst()
  Q.AddLast(Merge(L,R))
return Q.RemoveFirst()
```

2、整数乘法

> 几何级数，只需要看最后一项
>
> 并没有更快，还是n^2
>
> 想再简化：额外做加减法抵消乘法

Integer Multiplication

- Assume we want to multiply x and y, each having n bits.
- Split each of x and y into their left and right halves.
  - $x = 2^{n/2} \cdot x_L + x_R$ and $y = 2^{n/2} \cdot y_L + y_R$
- $xy = 2^n \cdot x_L y_L + 2^{n/2} \cdot (x_L y_R + x_R y_L) + x_R y_R$
  - Only need four multiplications, instead of six.
- Apply above strategy recursively until $n = 1$
- Recurrence: $T(n) = 4 \cdot T(n/2) + O(n)$

- Time complexity is $T(n) = O(n^2)$, *we are not doing better*!

- 改进

> 二进制表示代码

```
FastMulti(x, y):
if (x and y are both of 1 bit)
  return x*y
xl, xr = most, least significant |x|/2 bits of x
yl, yr = most, least significant |y|/2 bits of y
z1 = FastMulti(xl,yl)
z2 = FastMulti(xr,yr)
z3 = FastMulti(xl+xr,yl+yr)
return z1*(2^n)+(z3-z1-z2)*(2^(n/2))+z2
```

- We only need **three** multiplications, instead of **four**!
- $T(n) = 3 \cdot T(n/2) + O(n)$
- $T(n) = O(n^{\lg 3}) = O(n^{1.59})$

- 核心改进：加法换乘法

- $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$

3、矩阵乘法

- 普通二分没用

Matrix Multiplication

- Suppose we want to multiply two $n \times n$ matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$.
- The most straightforward method needs $\Theta(n^3)$ time.

- Matrix multiplication can be performed *block-wise*!
- $\boldsymbol{X} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ and $\boldsymbol{Y} = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$
- $\boldsymbol{XY} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$

- Recurrence: $T(n) = 8 \cdot T(n/2) + \Theta(n^2)$
- $T(n) = \Theta(n^3)$, we are not doing better...

- 改进：举一反三

- Multiply two $n \times n$ matrices $\boldsymbol{X} = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$ and $\boldsymbol{Y} = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$
- $XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$

- $P_1 = A(F - H)$, $P_2 = (A + B)H$, $P_3 = (C + D)E$, $P_4 = D(G - E)$
- $P_5 = (A + D)(E + H)$, $P_6 = (B - D)(G + H)$, $P_7 = (A - C)(E + F)$

- Recurrence: $T(n) = 7 \cdot T(n/2) + \Theta(n^2)$

计算分治时间

先猜后归纳（替换法）

- [ ] O可以有多种猜法
- [ ] 猜-小幂次，可以增强归纳假设

递归树

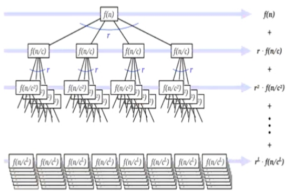
Solving recurrence

The recurrence-tree method

- $T(n) = r \cdot T(n/c) + f(n), T(1) = f(1)$
- Total cost is $\sum_{i=0}^{L} r^i \cdot f\left(n/c^i\right)$, where $L = \log_c n$

Three common cases for the series:

- **Decreasing** (exponentially): $T(n) = O\big(f(n)\big)$
  - Cost dominated by top level, such as $T(n) = T(n/2) + n$
- **Equal**: $T(n) = O(f(n) \cdot L) = O(f(n) \cdot \lg n)$
  - All levels have equal cost, such as $T(n) = 2T(n/2) + n$
- **Increasing** (exponentially): $T(n) = O\left(n^{\log_c r}\right)$
  - Cost dominated by bottom level, such as $T(n) = 4T(n/2) + n$
  - $T(n) = O\left(r^{\log_c n}\right) = O\left(n^{\log_c r}\right)$



- [ ] r和c可互换

主方法：万能公式不万能

Master Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

- The Master Theorem does not cover all cases!
- Be careful what does, e.g., $f(n) = O\left(n^{\log_b a - \epsilon}\right)$, mean.
  - If $a = b = 2$ and $f(n) = n/\lg n$, case one does *not* apply!

$f(n) = \frac{n}{\lg n} \in O(n^{1-\epsilon})$?

- [ ] 对于不平均递推式，建议先用递推树猜，再用替换法证

二分查找

- 已知顺序时，很快

- [ ] 下标越界说明算法没考虑一些特殊情况

1、当x肯定在A中时，正确：

```
BinarySearch(A, x):
left=1, right=n
while (true)
  middle = (left+right)/2
  if (A[middle]==x)
    return middle
  else if (A[middle]<x)
    left = middle+1
  else
    right = middle-1
```

正确性

1. 归纳证明每次迭代前

$$A[left] \le x \le A[right]$$

2. 某次迭代使left=right

3. 而且此时

$$A[left] = A[right] = x$$

2、标准算法

```
Binarysearch(A,x):
left=1,right=n
while(left<=right)
    mid=(l+r)/2
    if(A[mid]==x)
        return mid
    else if(A[mid]<x)
        l=mid+1
```

```
    else
        r=mid-1
```

Why this algorithm works?

If $x \in A$, previous argument still holds.

If $x \notin A$, then:
- After each iteration, we reduce input size by at least half.
- At some iteration, $left = right$.
- After that iteration, $left > right$.

peak finding

- 二分思想，分类讨论

PeakFinding(A):

```
left=1, right=n    Current array is not empty.
while (left<=right)
  middle = (left+right)/2
  if (middle>1 and A[middle-1]>A[middle])
    right = middle-1
  else if (middle<n and A[middle+1]>A[middle])
    left = middle+1
  else
    return A[middle]
```

正确性

1. 一定会返回
2. 返回的一定正确：说细

- 递归不变性

2维peek finding

先在中间列找一个peek，然后横向再来递归

- How fast is this algorithm?
- $\cancel{T(n) \le T(n/2) + \Theta(n) \text{ implying } T(n) = \Theta(n)}$
- $T(n, n') \le T(n/2, n') + O(n')$
- $T(n, n') \le (\lg n) \cdot O(n') = O(n' \lg n) = O(n \lg n)$

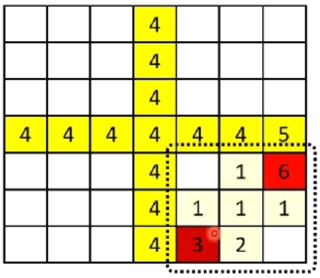
- 改进1：从列最大到十字架最大

正确性

- Does this algorithm work?
- Max in the cross is a peak; or a peak exists in the quadrant containing the large neighbor, and that peak is the max of some cross.
- A peak (found by the algorithm) in the quadrant containing the large neighbor is also a peak in the original matrix.
- The algorithm eventually returns a peak of some (sub)matrix.

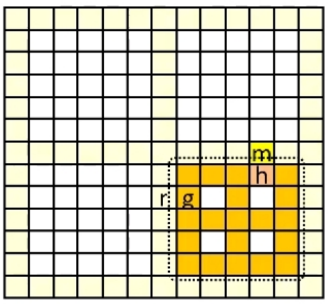
反例：

False Claim: A peak (found by the algorithm) in the quadrant containing the large neighbor is also a peak in the original matrix.



- 改进2：把十字架改成田字框，包含边界

- [ ] 让额外空间拓展，不影响时间复杂度层次



How fast is this algorithm?

$$T(n, n) \le T(n/2, n/2) + \Theta(n)$$
$$T(n, n) = O(n)$$