Divide and Conquer 第二次课*

You me

2022 年 12 月 3 日

目录

*可能是第二次课

今天我们面对的是 Divide and Conquer 的算法思想, 基本就是递归调用嘛. 一般都会有递归方程: $T(n) = aT(n/b) + f(n)$ 这样的东西.

前面我们已经学习了很多分析上面递归方程的复杂度的方法, 以及学习了两个经典案例: 归并排序以及快速排序. 今天讲的依旧是这些东西.

1 what is divide and conquer?

divide and conquer is a method to solve a question, to design an algorithm. It has two parts as whose name have suggested: Divide and Conquer. The basic idea is to divide the problem into some subproblems and try to conquer those subproblems.

The precedure consists of three parts in fact. Let's make a list.

1. Divide: to divide the problem.
2. Conquer: try to solve the subproblems
3. Combine: given the solution of some subproblems, to derive the the solution to the original one from those solutions.

So the outline of the analysis of a divide-and-conquer algorithm is that to find the recurrence function of the algorithm, and by using some previous knowledge of recurrence function, to figure the time complexity of the algorithm, that is to know: given the scale of input, let's say n, the order of $T(n)$, which stands for the time complexity.

It is clear that the operations of division and combination take time. Say that they take $D(n)$ and $C(n)$ separately. And we assume that the operation of division divides a n scale problem into a subproblems with n/b. Then we can write out the recurrence function:

$$T(n) = \begin{cases} \Theta(1) & \text{if} n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Comment 1. *There is actually a much more complex and annoying situation, that is what if the division divide a problem into one subproblem with scale $n \times \frac{1}{3}$ and a subproblem with scale $n \times \frac{2}{3}$. So you can not use master theorem.*

How are you going to work this out?

2 排序法

A sort is an operation to or, let's say, a process of some 'unordered sequence', to make it 'ordered'.

Example 1. *Give a sequence '*$52, 49, 80, 36, 14, 58, 61, 23, 97, 75$*', after the ordering, the output is*

$$14, 23, 36, 49, 52, 58$$

and so on and so on.

the defintion of ordering We can describe the above intuitive expression with more acuracy. Let's put it straight. We are given a sequence $\{R_1, \cdots, R_n\}$. (You know how to precisely define a sequence. Check some mathematical analysis textbooks.)

And the sequence has a corresponding sequence of key

$$\{K_1, \cdots, K_n\}$$

which act as an identificator, to determine the position of the order of some R_i, that is there exists a relation on those keys. (and the relation is partial order relation, and moreover is linearly ordered.):

$$K_{p_1} \leq K_{p_2} \leq \cdots \leq K_{p_n}$$

So what sort do is to reordering the sequence $\{R_1 \cdots\}$ into $\{R_{p_1}, \cdots\}$

The interior ordering and external ordering When scale of the sequence is so big that the memory can not hold it viz. the ordering can not complete within the memory. The sort algorithms that work within the memory are called interior sort.

2.1 the classification of ordering algorithms

We call the collection of R which are in order as 'ordered area', and respectively call the rest the 'unordered area'.

Definition 1 (round)**.** *a round within the algorithm is the operation that enlarge the number of* R_i *that is in order.*

There are 4 basic catergory of ordering algorithm. Let's make a list:

1. insert: to insert some member of unordered area to ordered area
2. select: to select the highest or lowest in unordered area, and put it right next to order area.
3. swap: to swap the target in unordered area, and swap it with some other R in unordered area, making it ordered.
4. merge: given two or more ordered sequence, to merge them into a bigger ordered sequence.

Comment 2. *note that the definitions of* ***select*** *and* ***swap*** *are wrong*

Base on the idea of divide and conquer, we have some prototype of the ordering algorithm.

One is to select a position basing on which we divide the problem (the description here might be not very clear). Another is to choose a member of the sequence, base on which we make a partition where members that are bigger than it are placed behind while the members that are smaller than it are placed in the front.

And the combination operations are indeed different accordingly.

```
Merge-sort(A,i,j):
Input:  A[i,⋯,j]
Output: 排序后的数组A
1.  k← (i+j)/2;
2.  Merge-sort(A,i,k);     % 递归调用
3.  Merge-sort(A,k+1,j);
%将已排序的子序列A[i,⋯,k], A[k+1,⋯,j]%
4.  l← i; h← k+1; t← i //设置指针
5.  While l ≤k & h < j Do
6.     IF A[l] < A[h] THEN B[t] ← A[l]; l←l + 1; t←t + 1;
7.     ELSE B[t] ← A[h]; h←h + 1; t←t + 1;
8.  IF l < k THEN //第一个子问题有剩余元素
9.     For v ←l To k Do
10.      B[t] ← A[v]; t←t + 1;
11. IF h < j THEN //第二个子问题有剩余元素
12.    For v ←h To j Do
13.      B[t] ← A[v]; t←t + 1;
14. For v← i To j Do
15.    A[v]← B[v] //将归并后的数据复制到A中
```

图 1: code of merge sort

2.2 merge sort

As we have stated, we choose a position can divide the sequence accordingly, by which we get two subproblems.

Definition 2. *given the sequence, let describe in the form of array* $A[i,\cdots,j]$*. We divide it with the position* k*.*

Then we have $A[i,\cdots,k]A\left[k+1,\cdots,j\right]$ *,where* $k=\dfrac{i+j}{2}$*. Then we solve the subproblems recurrently and combine them afterwards. That is merge sort.*

The key is that we assume the $A\left[i,\cdots,k\right], A[k+1,\cdots,j]$ are ordered. And we combine them to make to a larger sequence that is ordered.

2.2.1 code

Figure 1 shows the pseudo-code of merge sort. The main idea is merge, which use the idea of pointers.

2.2.2 analysis

We can easily write out the recurrence function of the algorithm:

$$T\left(n\right)=\begin{cases}2T(n/2)+n\\ O\left(1\right) & n<c\end{cases}$$

Because, operation that combine the two sequences need $\Theta\left(n\right)$ costs. And then we use master theorem:

$$T\left(n\right)=\Theta\left(n\log n\right)$$

2.3 快速排序

the sketch of the algorithm is that we choose a x and parition the sequence basing on it. Then we have two subproblems and deal with them in the same way. Then we will have the answer.

2.3.1 the code

```
QuickSort(A,i,j):
Input:  A[i,⋯,j],x
Output: 排序后的数组A[i,⋯,j]
1.  x ← A[i];                    // 以确定的策略选择x
2.  k = Partition(A,i,j,x);      // 用x完成划分
3.  QuickSort(A,i,k);            // 递归求解子问题
4.  QuickSort(A,k+1,j);
```

```
Partition(A,i,j,x):
1.  low ← i; high ← j;
2.  While (low < high) Do
3.     While (A[low] < x) Do
4.        low ← low + 1; //从左向右扫描，找第1个关键字大于x的A[low]
5.     While (A[high] >= x) Do
6.        high ← high - 1; //从右向左扫描，找第1个关键字小于x的A[high]
7.     swap(A[low],A[high]);
8.  return(high)
```

图 2: the pseudo code of quick sort

Figure 2 shows the pseudo code of the quick sort.

2.3.2 the time complexity analysis

worst case The time it takes in the worst case is :

$$\Theta\left(n^2\right)$$

let's say we are given an ordered sequence but in a reverse order. So every partition takes a long time, like $\Theta(n)$ where it have to compare to every other member of the sequence. More precisely, for the first chosen x, the partition takes $n-1$ operations. For the i th x, the partition takes $n-i$ operations.

The time algorithm cost is consequently $\Theta\left(n^2\right)$

best case every time the scale of the subproblems are equal in a best case, which result to that the recurrence function is that

$$T(n) = 2T(n/2) + \Theta(n)$$

We can use master theorem to figure that $T(n) = \Theta(n \log n)$

average case the average case is actually $\Theta(n \log n)$ the same to the best case. It is not saying that they have the same costs.

If we consider the scale of subproblems, the 'average' can be viewed as follows:

$$\frac{1}{n}\sum_{s=1}^{n}\left(T(s)+T(n-s)\right)$$

Because the case where scales differ have actually equal chances to happen.

Consequently, we have

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{s=1}^{n}\Big(T(s)+T(n-s)\Big)+cn \\ &= \frac{1}{n}\left(2T(1)+\cdots+2T(n-1)+T(n)\right)+cn \end{aligned}$$

Here is some technique we need to use :

$$(n-1)T(n) = 2T(1)+2T(2)+\cdots+2T(n-1)+cn^2 \tag{1}$$

$$(n-2)T(n-1) = 2T(1)+\cdots+2T(n-2)+c(n-1)^2 \tag{2}$$

substract equation (2) from equation (1) . Anyway, we have

$$(n-1)T(n)-(n-2)T(n-1) = 2T(n-1)+c(2n-1)$$

which leads to

$$\begin{aligned} &(n-1)T(n)-nT(n-1) = c(2n-1) \\ \Longrightarrow\ &\frac{T(n)}{n} = T(n-1)/(n-1)+c\left(\frac{1}{n}+\frac{1}{n-1}\right) \end{aligned}$$

which allows us to calculate the value of $T(n)$ recurrently, viz.

$$\begin{aligned} T(n)/n &= c\left(\frac{1}{n}+\frac{1}{n-1}\right)+c\left(\frac{1}{n-1}+\frac{1}{n-2}\right)+\cdots+c\left(\frac{1}{2}+1\right)+T(1) \\ &= c\left(\frac{1}{n}+\frac{1}{n-1}+\cdots+\frac{1}{2}\right)+c\left(\frac{1}{n-1}+\frac{1}{n-2}+\cdots+1\right)+T(1) \end{aligned}$$

These two parts are same. There is no need to separate them like what we have done here.

Next, we use a asymptotical analysis of Harmony series, viz.

$$\begin{aligned} H_n &= 1+\frac{1}{2}+\cdots+\frac{1}{n} \\ &= \log n+\gamma+\frac{1}{2n}-\frac{1}{12n^2}+\frac{1}{120n^4}-\varepsilon \end{aligned}$$

where $0<\varepsilon<\frac{1}{252n^6}$, and gamma is euler const. Just forget about that. This proposition is too strong. Anyway, it is crystal clear that $H_n = O(\log n)$. So apparently,

$$T(n)/n = c(H_n-1)+cH_{n-1} = O(\log n)$$

then we have

$$T(n) = O(n \log n)$$

2.3.3 random quick sort

In the algorithm above, we can notice that the worst case happens when the sequence is in reverse order. That is because the chosen x is the first member of the sequence. Consequently, a natural way to improve the algorithm, is to randomly choose the x, in the partition operation.

Thus for different sequences, they have the same expectation of the time it consumes.

That was great.

the code the code is left to the readers as exercise, because it is very easy. Just a slight change.

analysis Here is some notation:

Definition 3. *$S_{(i)}$ stands for the i^{th} smallest element. For example $S_{(1)}, S_{(n)}$, the former is smallest member, and latter is the biggest member.*

X_{ij} is a random variable, defined as follows: The value of X_{ij} is equal to the number of the comparison between i, j in the algorithm.

Consequently, the comparison made during the algorithm is

$$\sum_{i=1}^{n} \sum_{j>i} X_{ij}$$

Note that it is still a random variable.

Consequently, what we want to know is the expectation of the series above, viz.

$$E\left[\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i} E\left[X_{ij}\right]$$

Actually the X_{ij} can only have the value of 0 and 1, which leads to that $E\left[X_{ij}\right] = p_{ij} \times 1 + (1 - p_{ij}) \times 0$. So the question is to find p_{ij}

In a partition, the chosen x has to be compared to every other members in the sequence. Moreover, once the partition is over, x will not be compared again. Now you should know why the value of X_{ij} can only be 1 or 0.

Every operation of partition, actually, will form a partition of the sequence. May you can draw some pictures to figure this out. Anyway, two members, let's say i and j, can be compared only when they are in 'a' same class of the parition (the class is short for equivalent class).

The analysis can be carried on now. Please take a notice to the next procedure of analysis:

1. Only when $S_i, S_{i+1}, \cdots, S_j$ is in a same class, $S_{(i)}$ and $S_{(j)}$ can be compared.
2. Only when $S_{(i)}$ or $S_{(j)}$ is chosen as the x, these two can be compared.
3. Moreover, if we choose some $S_{(k)}$ where $k < i$ or $k > j$, $S_{(i)}, S_{(j)}$ are still possible to be compared.
4. The probability of $S_{(i+k)}$ to be chosen as x are equals. So the probability of $S_{(i)}, S_{(j)}$ to be chosen as x is

   $$\frac{2}{j - i + 1}$$

   viz.

   $$p_{ij} = \frac{2}{j - i + 1}$$

Remark 1. *One might notice that the number of members in the class might not be precisely the $j-i+1$. We just ignore this situation. Why?*

It is available to calculate the expection now.

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j>i} E\left[X_{ij}\right] &= \sum_{i=1}^{n}\sum_{j>i}\frac{2}{j-i+1}\\ &\leq \sum_{i=1}^{n}\sum_{k=1}^{n-i+1}\frac{2}{k} \leq 2\sum_{i=1}^{n}\sum_{k=1}^{n-i+1}\frac{1}{k}\\ &= 2nH_n = O\left(n\log n\right)\end{aligned}$$

Over !

2.4 the lower bound of sort problem

It describe how fast the sort can be under certain cases. For example, under the worst case or under the average case.

Definition 4. *If a problem has a low bound, for example, $\Omega\left(n\log n\right)$, and if the time complexity of an algorithm is $O\left(n\log n\right)$, then we say that algorithm is optimal.*

Definition 5 (decision trees)**.** *A procedure and the all the cases of a sort can be expressed as tree, where, the leaf nodes stands for the resulted sequences. and the non-leaf nodes stands for the operations of comparsion.*

And moreover, the number of the non-leaf node in a path stand of the comparison it shall take under this case. That is the level of the leaf stands for the number of operations.

Thus, the low bound of a sort, is the lowest level of a leaf.

let's say that the decision tree has height h and have l leaves. All the possible resulted sequences are in number of $n!$.

Then we can easily get an inequality :

$$n! \leq l \leq 2^h$$

Then

$$h \geq \log\left(n!\right) = \Omega\left(n\log n\right)$$

We can see that [1] , in the worst case, the time complexity has a low bound $\Omega\left(n\log n\right)$

the average case we consider the average length of paths, which is that

$$\frac{\text{total length}}{n!}$$

when the decision tree is **balanced**, the average length above takes minimum.

Definition 6 (balanced decision tree)**.** *In a balanced decision tree, every leaf has level either d or $d-1$, where d is the height of the tree.*

[1] the approximation can be deduced from Stirling formula: $n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

Because what we want is a lower bound. So only the balanced decision tree is concerned, which is the minimum, which is, what we want.

1. it is clear that $d = \lceil \log c \rceil$, where $c = x_1 + x_2$
2. x_1 in level $d-1$ and x_2 in level d
3. $x_1 + \dfrac{x_2}{2} = 2^{d-1}$. 2^{d-1} is the number of nodes in level $d-1$
4. total length:

$$\begin{aligned} M &= x_1 (d-1) + x_2 d \\ &= \left(2^d - c\right)(d-1) + 2\left(c - 2^{d-1}\right) d \\ &= c(d-1) + 2\left(c - 2^{d-1}\right) && (d - 1 = \lfloor \log c \rfloor) \\ &= c \lfloor \log c \rfloor + 2\left(c - 2^{\lfloor \log c \rfloor}\right) \end{aligned}$$

5. $c = n!$ since this is the case when the tree has lowest height.
6. We claim that $c - 2^{\lfloor \log c \rfloor} \geq \dfrac{c}{2}$ so

$$\begin{aligned} M/n! &> \lfloor \log n! \rfloor + 1 \\ &= \Omega(n \log n) \end{aligned}$$

7. so the low bound in average case is $\Omega(n \log n)$

In general, the low bound of sort is $\Omega(n \log n)$ in all possible cases. The quick sort is optimal when the const is ignored.

Oh my god! I mean, this is bad. The deduction is like shit.

3 MAX and MIN 问题

Input: n 个互异的数组成的数组 A , 以及一个正整数 i
Output: $B = \{x \mid |A \leq x| = i\}$ for this proposed problem, if it is sovled, then the Max と Min can be solved.

我们想到的最简单的方法很明显就是直接排序然后进行一个选取, 但是这样未免有点简陋了, 并且效率也是一点也不理想. So we may tackle the simplified problem here: find the Max と Min.

3.1 算法 1, 不使用递归方程

一个明显的想法就是直接从头到尾扫描两边, 就都求出来了, 时间复杂度是 $2(n-1)$, 其中 n 是问题的规模, 在这里是数组的大小. 我们这里能够给出一个算法, 其效率是 $\left\lfloor \dfrac{n}{2} \right\rfloor + 2\left\lceil \dfrac{n}{2} \right\rceil$:

1. 从数组的两端开始, 两两比较, 将较大的一方放在前面, 较小的一方放在后面
2. 我们能够知道, 最大值一定在前面, 最小值一定在后面
3. 我们分别扫描前面和后面, 就能够得出最大值和最小值了

我们对这个算法进行一下分析

1. 中执行次数其实只有 $\lfloor \frac{n}{2} \rfloor$, 这是因为当 n 是奇数的时候, 中间的数字不需要变动, 因为算要变动也是自己和自己交换, 于是向下取整;

3. 中分别扫描 $A\left[1, \cdots, \lceil \frac{n}{2} \rceil\right]$, 以及 $A\left[\lceil \frac{n}{2} \rceil, \cdots, n\right]$, 就能分别求出最大值和最小值了, 比较的次数是 $2\lceil \frac{n}{2} \rceil$. 于是时间复杂度就是

$$\left\lfloor \frac{n}{2} \right\rfloor + 2\left\lceil \frac{n}{2} \right\rceil$$

这个证明并不是很会捏, 这里给出伪代碼:

算法 1

```
Max-min(A)
Input:  数组A[1,⋯,n]
Output: 数组A[1,⋯,n] 中的max和min
{1.  For i ← 1 To n/2 Do
 2.     IF A[i] > A[n-i+1] THEN swap(A[i],A[n-i+1]);
 3.  max ← A[n]; min ← A[1];
 4.  For i ← 2 To ⌈n/2⌉ Do
 5.      IF A[i] < min THEN min ← A[i];
 6.      IF A[n-i+1] > max THEN max ← A[n-i+1];
 7.  print max, min;
}
```

比较次数: $\left\lceil \frac{3n}{2} - 2 \right\rceil$次比较操作

图 3: 不使用遞歸方程的算法

3.2 算法 2, 使用递归方程

1. line 1 と 2 are initial value of recurrence function.
2. line 6 & 7 are division.
3. line 8 & 9 are combination.

算法 2

```
Max-min(A)
Input:  数组A[1,⋯,n]
Output: (x,y), A[1,⋯,n]中的max和min
Max-min(A,1,n)
过程: Max-min(A,low,high)
{1. IF high - low = 1
2.    IF A[low] < A[high] THEN return(A[low],A[high]);
3.    ELSE return(A[high],A[low]);
4. ELSE
5.    mid ← (low+high)/2;
6.    (x1,y1) ← Max-min(A,low,mid)
7.    (x2,y2) ← Max-min(A,mid+1,high)
8.    x ← min{x1,x2}
9.    y ← max{y1,y2}
10.return (x,y);
}
```

图 4: 使用遞歸方程的算法

图中的算法是一个递归求解的过程, 将规模为 n 的问题转化为两个 $n/2$ 的问题, 即求出前半的最值, 后半的最值, 而后进行比较, 得出总的最值, 能够得出:

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

下面我们来看看这个的时间复杂度, 设 $n=2^k$ (因为这样方便一点) ,

$$\begin{aligned} T(n) &= 2T(n/2)+2 \\ &= 4T(n/4)+2^2+2 \\ &= 2^3T(n/8)+2^3+2^2+2 \\ &= \cdots \\ &= 2^{k-1}T(2)+\sum_{i=1}^{k-1}2^i \end{aligned}$$

Remarks.

1. $T(n)=2T(n/2)+2$ 后面常数是 2 , 是因为最大值和最小值各比较一次
2. 如何计算的? $2T(n/2)=2(2T(n/4)+2)+2=4T(n/4)+2^{2+2}\cdots$
3. 不再研究最大最小值选择问题的下界和平均情况下选择的下界 (为什么?[2])

4 选择中位数

4.1 问题描述

Input：一个长度为 n 的数组
Output：这个数组的中位数[3]

4.2 算法设计

直接给出算法:Select (A, i) 的定义

参数 A,i, 其中 A 是数组, i 定义为 $\lfloor (n+1)/2 \rfloor$, 意为选择数组中第 i 大的数字

step1. 将我们的数组分为五个部分

step2. 使用插入排序将这个五个部分排序, 以此选出五个中位数

step3. 在这五个数字中递归调用算法 Select[4], 得出中位数, 记为 x

step4. 使用函数划分 Partition: 将所有小于 x 的放在前面, 大于 x 的放在后面. 并且得到 x 的下标 k

step5. 将下标 k 和 $i=\left\lfloor \dfrac{n+1}{2} \right\rfloor$ 比较

如果说 $i=k$ 则 OK , 如果说 $k>i$, 则在前面寻找第 i 大的数,

[2]这并不是留给读者思考的, 我是真不知道为什么

[3]选取第 $\left\lfloor \dfrac{n+1}{2} \right\rfloor$ 大的数字

[4]Select 之中使用 Par

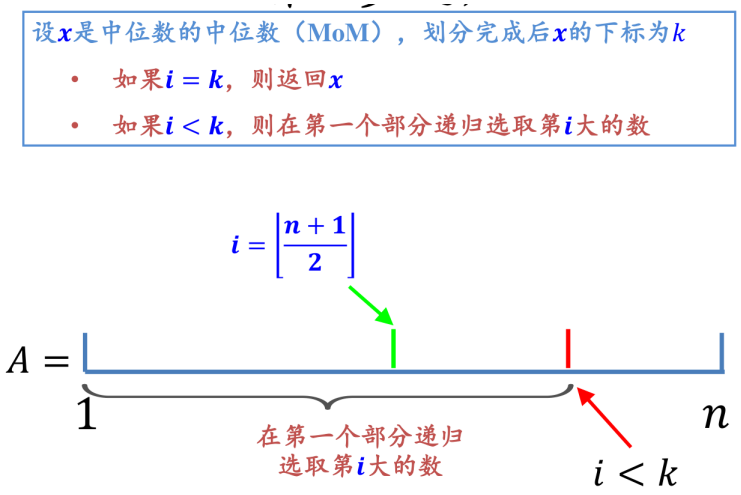


图 5:

如果说 $k < i$ 则在后面寻找一个 $i - k$ 大的数.

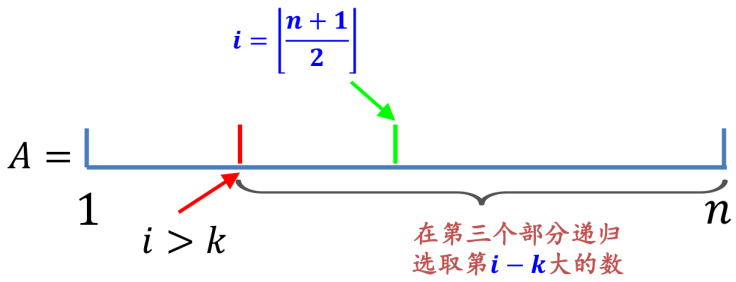
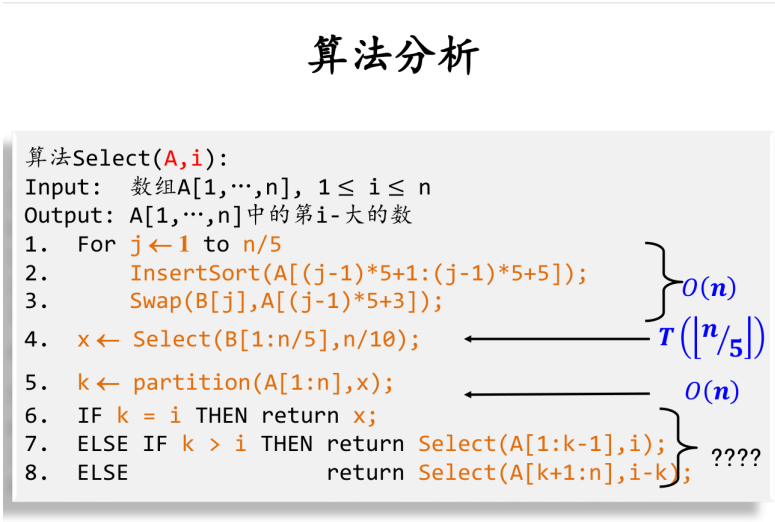


图 6:

4.3 伪代码



5

图 7:

这个东西是不是没有设置初始条件阿? 比如说我有一个长度为 5 的数组, 然后输进去之后还要调用这个 select 吗? 搞不懂啊.

总之, 这个算法的流程就是, 我重复一下上面的这个东西. 这个算法实际上是将, 这个数组, 预处理一下, 选择出一个怀疑对象, 这个预处理使用的是这个 insertion sort; 而后是 divide & Conquer 的部分, 重复使用 partition 得出答案.

但是这里有一个问题, 我们的算法是不是没有处理初始情况?

4.4 select 的一个估计

就是说, 我们将这个 members, 什么 members 呢? 就是那些, 可能被上面 line 6 7 之类删掉的那些数字. 排成一个矩阵, 如果说 a_{ij} 来表示矩阵元素. 并且, 总共有 5 列. 每一列都有 $n/5$ 个元素. 大概是这个数字, 当这个 n 很大的时候, 就能够忽略这个余数带来的差别.

总之这个时候我们要看那个 ppt, 那个 ppt 上的红色标识的元素, 实际上就是, 得出的那个下标的元素, 面对这些东西, 我们能够看出. 这里面的, 虚线框的东西, 就是我们可能剔除掉的东西, 我们对这个数字进行一个估计, 我们估计, 删掉的元素实际上最少有

$$\left\lfloor \frac{3n}{10} \right\rfloor$$

这是因为, 这个时候, 对于一个矩阵成员, $a_{\mu\nu}$, 如果说 $\mu \le i, \nu \le j$ 我们断言, 这个 $a_{\mu\nu} \le a_{ij}$. 这里我们进行了一个非常粗糙的估计: 这里, 减少的成员, 大概有

$$\left\lfloor \frac{3n}{10} \right\rfloor$$

这是一个比较缺少考虑的估计, 我们能够明显地举出反例. 但总之, 这些技术细节, 并不是很重要. 我们知道一个大概就行:

$$n - \left\lfloor \frac{3n}{10} \right\rfloor \le \frac{7n}{10} + 6$$

因此递归调用的操作的复杂度至多是 $T\left(\frac{7n}{10}+6\right)$. 于是我们能够就此得到递归方程:

$$T(n)=\begin{cases}\Theta(1) & n\le c\\ T\left(\left\lfloor\frac{n}{5}\right\rfloor\right)+T\left(\frac{7n}{10}+6\right)+\Theta(n) & \text{otherwise}\end{cases}$$

于是说, 时间复杂度就是

$$T(n)=O(n)$$

线性时间内就能够完成. 非常好算法.

5 大数乘法

废话不多说
我们要做的就是两个大数相乘, 使用类似的方法 (二分) , 转化为几个子问题
为了提高效率, 这两个大数是以二进制表达的[6]

5.1 问题描述

Input：n 位二进制整数 X,Y
Output：X,Y 的乘积
我们正常一位一位计算需要的时间复杂度是 $O\left(n^2\right)$[7] (一位数乘以 n 位整数是 $O(n)$ 的时间, n 位数乘以 n 位数就是 $O(n\times n)$ 的时间), 但是我们用 Divide and Conquer 可以得到复杂度为 $O\left(n^{\log_2 3}\right)\approx O\left(n^{1.59}\right)$ 的算法.

5.2 简单的 Divide and Conquer 算法

将 X 记为 $A\mid B$, 其中 A,B 是两个 $n/2$ 位的数字, X 实际上等于 A 左移加上 B:

$$X=A\times 2^{n/2}+B$$

类似地, Y 记为 $C\mid D$. 于是就有:

$$\begin{aligned}XY&=\left(A\times 2^{n/2}+B\right)\times\left(C\times 2^{n/2}+D\right)\\&=AC\times 2^n+(AD+BC)\times 2^{n/2}+BD\end{aligned}$$

根据这个等式我们有第一个算法:
1. 计算 A,B,C,D
2. 计算 AC,BD
3. 计算 $AD+BC$
4. AC 左移, $(AD+BC)$ 左移 $n/2$ 位
5. 计算 XY

[6]下面用到的左移操作, 在计算机中只需要常数时间
[7]为什么是 O 符號呢, 你看如果说这个大数里有很多 0, 就不需要算那么多次了

显然, 1. 只需要常数时间; 2. 递归调用函数需要 $2T(n/2)$ 的时间; 3. 也是递归调用, 同时加法需要 $\Theta(n)$ 的时间; 4. 左移, 常数时间; 5. 计算 XY 使用加法, $\Theta(n)$ 时间

故

$$T(n) = 4T(n/2) + \Theta(n)$$

运用 Master 定理: $T(n) = \Theta\left(n^2\right)$

5.3 改进的 Divide and Conquer 算法

我们可以将这个等式简化一下:

$$\begin{aligned} XY &= \left(A \times 2^{n/2} + B\right) \times \left(C \times 2^{n/2} + D\right) \\ &= AC \times 2^n + (AD + BC) \times 2^{n/2} + BD \\ &= AC \times 2^n + \left((A - B)(D - C) + AC + BD\right) \times 2^{n/2} + BD \end{aligned}$$

利用上了已经计算了的 AC, BD 来计算中间这块, 明显

$$T(n) = 3T(n/2) + \Theta(n)$$

特别地, 当 $n = 1$ 时, $T(n) = \Theta(1)$. 一算就有: $T(n) = n^{\log_2 3} \approx n^{1.59}$

5.4 思考

能否利用这个想法实现矩阵乘法问题?

感觉可以, 但是我不会.

6 简单的 ChessBoard 问题

6.1 问题描述

在一个 $2^k \times 2^k$ 个方格组成的棋盘中，有一个方格与其它的方格不同，称之为奇异块。要求：若使用以下四种 L 型骨牌覆盖除这个奇异块的其它方格，覆盖过程中 L 型骨牌间不能有互相覆盖，设计算法求出覆盖方案。四个 L 型骨牌如下图[8]

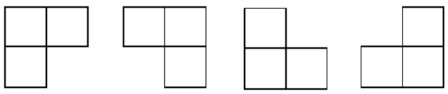


图 8:

6.2 算法设计

1. 当 >0 时，将 $2^k \times 2^k$ 的棋盘分成四块 $2^{k-1} \times 2^{k-1}$ 的子棋盘
2. 使用一个合适的 L 型骨牌，覆盖三个没有特殊方格的子棋盘的相邻方格
3. 继续递归处理 4 子棋盘，直到子棋盘中只有一个特殊方格为止

我们将自己得到的算法记为 ChessBoard(tr, tc,dr,dc, k)
其中 tr, tc 棋盘起点 (最左上) 的坐标, dr,dc 是特殊点的坐标, k 是棋盘之大小.
总体思路是这样
1. 划分棋盘为四个象限
2. 确定特殊点在那个象限
3. 将牌放在中心, 使得牌和特殊点所在的象限无交点

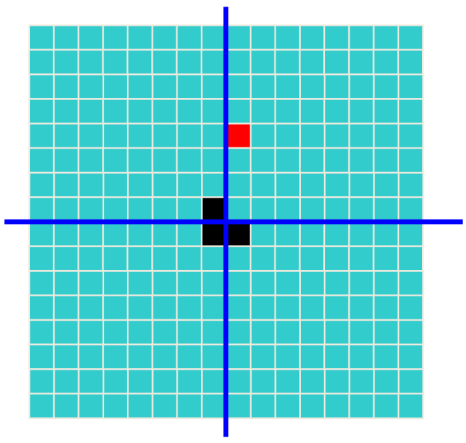


图 9:

4. 将骨牌的各个点视为特殊点
5. 得到了四个子问题, 将参数传进递归调用的函数里

[8]你可以看出这个问题一点也不一般化

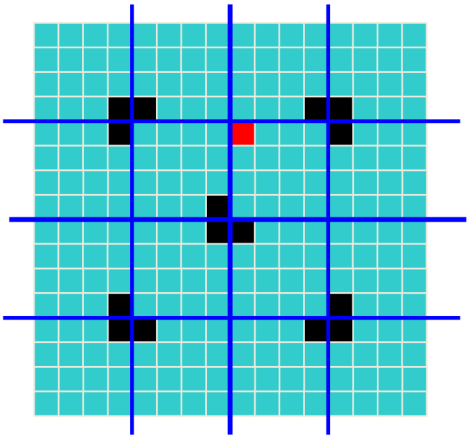


图 10:

6. OK

6.3 算法设计的伪算法

过程： ChessBoard(tr, tc, dr, dc, k)

```
IF k = 0
        return
t ← tile++              //被覆盖的方格记号
k ← k-1                 //分割棋盘
//求解左上角子棋盘
IF dr<tr+2^k AND dc<tc+2^k                      //特殊方格在此棋盘中
        ChessBoard(tr,tc,dr,dc,k)
ELSE
        Board[tr+2^k-1][tc+2^k-1]←t             //标记覆盖右下角
        ChessBoard(tr,tc,tr+2^k-1,tc+2^k-1,k)
//求解右上角子棋盘
IF dr<tr+2^k AND dc>=tc+2^k                     //特殊方格在此棋盘中
        ChessBoard(tr,tc+2^k,dr,dc,k)
ELSE
        Board[tr+2^k-1][tc+2^k] ←t              //标记覆盖左下角
        ChessBoard(tr,tc+2^k,tr+2^k-1,tc+2^k,k)
//求解左下角子棋盘
IF dr>=tr+2^k AND dc<tc+2^k                     //特殊方格在此棋盘中
        ChessBoard(tr+2^k,tc,dr,dc,k)
ELSE
        Board[tr+2^k][tc+2^k-1] ← t             //标记覆盖右上角
        ChessBoard(tr+2^k,tc,tr+2^k,tc+2^k-1,k)
//求解右下角子棋盘
IF dr>=tr+2^k AND dc>=tc+2^k                    //特殊方格在此棋盘中
        ChessBoard(tr+2^k,tc+2^k,dr,dc,k)
ELSE
        Board[tr+2^k][tc+2^k] ←t                //标记覆盖左上角
        ChessBoard(tr+2^k,tc+2^k,tr+2^k,tc+2^k,k)
```

9

图 11:

6.4 复杂度分析

我们根据上面的伪代码找出 recurrence function:

$$T(k) = \begin{cases} \Theta(1) & k = 0 \\ 4T(k-1) + \Theta(1) & k > 0 \end{cases}$$

then we use some simple techniques to work this out

$$\begin{aligned} T(k) &= 4T(k-1) + \Theta(1) \\ &= 4\left(4T(k-2) + \Theta(1)\right) + \Theta(1) \\ &\vdots \\ &= 4^k T(0) + \Theta(1) \sum_{i=0}^{k-1} 4^i \\ &= \Theta\left(4^k\right) \end{aligned}$$