Research Project 4
Red-Black Tree

Author Names

2024-11-15

Chapter 1 : Introduction

A red-black tree is a special type of binary search tree whose nodes are either red or black. Certain constraints on its colors and the identical black height guarantee its symmetry. However, given the number of a red-black tree's internal nodes, the black height of the red-black tree cannot be uniquely determined, nevertheless the tree itself. In this project we are going to count the number of black-rooted red-black trees with n internal nodes. Two methods, dynamic programming and generating function, are discussed. Interestingly, these recurrence relations have a high level of similarity.

Chapter 2 : Algorithm Specification

2.1 Dynamic Programming

Fundamentally speaking, every non-empty red-black tree is composed of three components - its root and two subtrees, which gives us a hint that we can solve two sub-problems first. That leads us to **Dynamic Programming**. Let $black_{ij}$ be the number of black-rooted red-black trees with i internal nodes and of black-height j. The child of the root of a black-rooted red-black tree can either be black or red. Simply using $black_{ij}$ seems not working here.
Then we introduce red_{ij} for help, even though it doesn't necessarily need to appear directly in the final answer. Similarly, red_{ij} represents the number of *red*-rooted red-black trees with i internal nodes and of black-height j. Notice that the subtree of a red node must be black-rooted, by enumerating the size of the left subtree we have

$$red_{ij} = \sum_{k=0}^{i-1} black_{kj} \times red_{(i-k-1)j}$$

Then for a black-rooted red-black tree, the black-height of its subtree must be $j-1$. Therefore,

$$black_{ij} = \sum_{k=0}^{i-1} (black_{k(j-1)} + red_{k(j-1)}) \times (black_{(i-k-1)(j-1)} + red_{(i-k-1)(j-1)})$$

```
/* Compute */
for(int i=1;i<=n;++i)
    for(int j=0;j<=i;++j)
        for(int k=0;k<i;++k)
        {
            if(j>0)
                black[i][j]=(black[i][j]+(black[k][j-1]+red[k][j-1])
                            *(black[i-k-1][j-1]+red[i-k-1][j-1])%p)%p;
                // the son of a black root can be either black or red
            red[i][j]=(red[i][j]+black[k][j]*black[i-k-1][j]%p)%p;
                // but the son of a red root must be black
        }
```

The boundary is a the empty tree:

```
/* Initialize */
black[0][0]=1;
```

To obtain the final answer, we need to enumerate the black-height of the tree and to sum up the numbers.

> ***Theorem:*** A red-black tree of black height i has at least $2^i - 1$ nodes.

The proof can be easily found in the slides from the class.
With this theorem we only need to enumerate black height from 1 to $[\log_2(n+1)]$.

$$ans = \sum_{j=1}^{[\log_2(n+1)]} black_{nj}$$

```
/* Sum up */
for(int i=1;i<=log2(n+1);++i)
    ans=(ans+black[n][i])%p;
```

2.2 Generating Function

Let $T_h(x)$ be the generating function for the number of red-black trees of black-height h.

Firstly, we have the boundary condition:

$$T_1(x) = x + 2x^2 + x^3$$

It's quite obvious that there could be at most three nodes in a black-rooted red-black tree with only one black node. The x term above means a single black root node; $2x^2$ means two-node red-black tree has two possibilities: the red node being the left child or the right; And x^3 means three-node red-black tree with only one black node is unique - two red nodes being its two children.
Then for greater h, we'd like to get the recurrence relation. However, it's hard to directly write out $T_h(x)$. So we'd like to invite $R_h(x)$ to help, which means the number of red-rooted red-black trees. For $R_h(x)$ we have:

$$R_h(x) = x(T_h^2(x))$$

x means the red root. And its two subtrees must be black-rooted, which gives the square of $T_h(x)$.
Then it's time for $T_h(x)$! If a child of it is red, then $R_h(x)$. Otherwise, a black child gives $T_{h-1}(x)$. Therefore we have

$$\begin{aligned} T_{h+1}(x) &= x(R_{h+1}(x) + T_h(x))^2 \\ &= x(xT_h^2(x) + T_h(x))^2 \\ &= xT_h^2(x)(xT_h(x) + 1)^2 \end{aligned}$$

Let

$$T_h(x) = a_0^h + a_1^h x + a_2^h x^2 + \cdots = \sum_{i=0} a_i^h x^i$$

To obtain the final answer, we sum up the coeffecient of the x^n term, which means the number of n-sized red-black trees.

$$ans = \sum_{h=1}^{[\log_2(n+1)]} a_n^h$$

The program goes like follows:

```
for(long long h=1;h<=log2(n+1);++h)
{
    polymul(X,cnt,tmp,N);       //tmp=xT_h(x)
    polymul(tmp,cnt,ttt,N);     //ttt=xT_h(x)T_h(x)
    tmp[0]=1;                   //tmp=xT_h(x)+1
    polymul(tmp,tmp,tmp,N);     //tmp=(xT_h(x)+1)^2
    polymul(ttt,tmp,cnt,N);     //T_{h+1}(x)=xT_h(x)T_h(x)(xT_h(x)+1)^2
    ans=(ans+cnt[n])%p;
}
```

In `polymul()`, we implemented the product of two polynomials:

```
void polymul(long long *a,long long *b,long long *c,long long n)
{
    long long *ans=(long long*)calloc(n+1,sizeof(long long));
    for(long long i=0;i<=n;++i)
        for(long long j=0;j<=i;++j)
            ans[i]=(ans[i]+a[j]*b[i-j]%p)%p;//ans[i]=a[0]*b[i]+a[1]*b[i-1]+...+a[i]*b[0]
    for(long long i=0;i<=n;++i)
        c[i]=ans[i];
}
```

2.3 Fast Fourier Transform (FFT)

We will prove in 4.2 that the time complexity for calculating the recurrence equation above is $\Theta(n^2 \log n)$, the same as dynamic programming. That's not fast enough. Fortunately, to calculate the product of two polynomials, **Fast Fourier Transform (FFT)** enables us to accelerate the calculation of the generating function and to reduce its time complexity from $\Theta(n^2)$ to $\Theta(n \log n)$.
Consider

$$
\begin{aligned}
A(x) &= a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} \\
&= a_0 + a_2x^2 + a_4x^4 + \cdots + a_{n-2}x^{n-2} + \\
&+ x(a_1 + a_3x^2 + a_5x^4 + \cdots + a_{n-1}x^{n-1}) \\
&= A_0(x^2) + xA_1(x^2) \\
where\ A_0(x) &= a_0 + a_2x + \cdots + a_{n-2}x^{\frac{n-2}{2}} \\
and\ A_1(x) &= a_1 + a_3x + \cdots + a_{n-1}x^{\frac{n-2}{2}}
\end{aligned}
$$

Let $x = \omega_n^k,\ 0 \le k \le \frac{n}{2} - 1,\ k \in \mathbb{Z}$, we have

$$
A(\omega_n^k) = A_0(\omega_n^{2k}) + \omega_n^k \cdot A_1(\omega_n^{2k})
$$

which simplifies to

$$
A(\omega_n^k) = A_0(\omega_{\frac{n}{2}}^k) + \omega_n^k \cdot A_1(\omega_{\frac{n}{2}}^k)
$$

Then for $\frac{n}{2} \le k + \frac{n}{2} \le n - 1,\ k \in \mathbb{Z}$, we have

$$
A(\omega_n^{k+\frac{n}{2}}) = A_0(\omega_n^{2k+n}) + \omega_n^{k+\frac{n}{2}} \cdot A_1(\omega_n^{2k+n})
$$

which simplifies to

$$
A(\omega_n^{k+\frac{n}{2}}) = A_0(\omega_{\frac{n}{2}}^k) - \omega_n^k \cdot A_1(\omega_{\frac{n}{2}}^k)
$$

Then it can be divided into two subproblems: If we've already known the value of $A_0(x)$ and $A_1(x)$ when $x = \omega_{\frac{n}{2}}^0, \omega_{\frac{n}{2}}^1, \cdots, \omega_{\frac{n}{2}}^{\frac{n}{2}-1}$, we can get $A(x)$ in $O(n)$. That's the main idea of Fast Fourier Transform.

Detailed introduction to FFT can be found in the link in the References part. Or just search `FFT` on the internet.

2.4 Too Large for `long long`?

Sadly, for n that is not small enough, the ordinary FFT method will give out a negative number, which is clearly not the right answer. Consider the largest number we could possibly meet in one `polymul()`.

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

There might be $i+1$ terms, each of which can be as large as the product of two integers no larger than p, the modulus. And i can be at most n. Therefore, we can estimate how large the result can be.

$$MaxValue \approx p \times p \times n \approx 10^{18} n$$

The largest capable number for `long long` is $2^{63} - 1 \approx 9.22 \times 10^{18}$, clearly not suitable for the demand.
The solution is to split the extreme large number. For each a_i, we split it into two pieces, the lower one a_{il} and the higher one a_{ih}:

$$a_i = a_{il} + Ma_{ih}, where\ M = 2^{15} = 32768$$

Then we have $a_{il}, a_{ih} < 2^{15}$, and $a_{il}^2 n, a_{ih}^2 n$ are apparently all in the range of `long long`, which solves this problem.
To calculate $a_i \times b_i$,

$$\begin{aligned} a_i \times b_i &= (a_{il} + Ma_{ih})(b_{il} + Mb_{ih}) \\ &= a_{il}b_{il} + Ma_{il}b_{ih} + Ma_{ih}b_{il} + M^2 a_{ih}b_{ih} \end{aligned}$$

Calculating these four parts separately would work.

Chapter 3 : Testing Results

3.1 Correctness

Fortunately, the number of black-rooted red-black trees with n internal nodes is a very classical problem, which has been included in the On-Line Encyclopedia of Integer Sequences as A001137. We can download the first 900 terms of this sequence as `b001137.txt` to check whether our program gives the correct answer. For all the input values from 1 to 900, the three programs give the identical outputs, which proves the correctness of three programs.
As n is guaranteed to be no more than 500, we actually have covered all the input cases by comparing our outputs with `b001137.txt`.

3.2 Time

To guarantee a fair competition, we use the same compile options as below:

```
gcc code.c -o code.exe "-Wl,--stack=268435456" -Ofast
```

Use the timer below, we check their performances for different inputs.

```
#include<time.h>
clock_t start,stop;
start=clock();
//the main part
stop=clock();//end timing
printf("%lf",(double)(stop-start)/CLK_TCK);
```

Inputs	`DP`	`Ordinary Generating Function`	`FFT`
500	0.006s	0.018s	0.006s
1000	0.020s	0.057s	0.013s
2000	0.070s	0.231s	0.030s
5000	0.498s	1.698s	0.118s

From the table above we can see that FFT is the fastest. Detailed disucssion on time complexity will be included in Chapter 4.

Chapter 4 : Analysis and Comments

4.1 Dynamic Programming

4.1.1 Time Complexity

Clearly there's three loops. In the outer loop we decide the size of the red-black tree, which will be applied for n times; The middle loop enumerates the black height, which is $[\log_2(n+1)]$ times; The inner loop enumerates the sizes of two subtrees and has i cases. Therefore, the overall time complexity for dynamic programming is $\Theta(n^2 \log n)$.

4.1.2 Space Complexity

Two arrays are used. So the space complexity for dynamic programming is $\Theta(n \log n)$. However, in the recurrence relation only `red/black[][j]` and `red/black[][j-1]` are used, which indicates that reducing space complexity to $\Theta(n)$ is possible. But there's just no practical use to do so.

4.2 Ordinary Generating Function Method

4.2.1 Time Complexity

In the outer loop, we need to enumerate h from 1 to $[\log_2(n+1)]$. That's $\Theta(\log n)$. Inside each loop apply several polynomial multiplications. Each `polymul()`
So the overall time complexity is $\Theta(n^2 \log n)$.

4.2.2 Space Complexity

Several arrays of length n are used and there's no recursion or anything else that would add up the space complexity. So the space complexity for the ordinary generating function method is $\Theta(n)$.

4.3 Fast Fourier Transform

4.3.1 Time Complexity

Firstly, we'd like to prove the time complexity of the multiplication of two n-length polynomials by Fast Fourier Transform to be $\Theta(n \log n)$. Basically, Fast Fourier Transform divides the . The and the multiplication step take $\Theta(n)$. Then we have

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

According to **Master Theorem**, we have $T(n) = \Theta(n \log n)$.
Although splitting the large number in order not to exceed the range of `long long` forces us to apply FFT for four times, that's just constant-level complexity and will do no harm.
The outer loop is identical as ordinary generating function. We'll do that for $\log_2(n + 1)$ times. So the overall time complexity is $\Theta(n \log^2 n)$.

4.3.2 Space Complexity

In Fast Fourier Transform, we just employed several assisstant arrays of length n for help. And the total space complexity is still $\Theta(n)$.

4.4 Comments

From the theoretical analysis above we see Dynamic Programming and the Ordinary Generating Function Method shares the same time complexity of $\Theta(n^2 \log n)$, while Fast Fourier Transform, with the time complexity of $\Theta(n \log^2 n)$, is more effective. Our test results support the theoretical analysis, and shows the Ordinary Generating Function Method is slower than Dynamic Programming in practice. One possible reason is that the polynomial multiplications may add too much constant-scale time complexity to it and pull it back.

References

- https://oeis.org/A001137

- 一小时学会快速傅里叶变换（Fast Fourier Transform）
  https://zhuanlan.zhihu.com/p/31584464

Declaration

> *We hereby declare that all the work done in this research project titled "Red-Black Tree" is of our independent effort.*