寻找最优初始值

对于$\sqrt{c}$，我们期望在二进制中找到最近的数作初始值，即将2进制转化为对应平方数：

$$2^{s-1} < \sqrt{c} \le 2^s, s = 1, 2, \dots, 16$$

则

$$(2^{s-1}+1)^2 \le c < (2^s+1)^2, s = 1, 2, \dots, 16$$

c	$\sqrt{c}$	s	$x_0 = 2^s$
$[4, 9)$	2	1	2
$[9, 25)$	$(2, 4]$	2	4
...	...	...	...
$[(2^{15}+1)^2, 2^{32})$	$(2^{15}, 2^{16}]$	16	65536

定义索引表

```
unsigned ss[16] = {
    4, 9, 25, 81,
    17 * 17, 33 * 33, 65 * 65, 129 * 129,
    257 * 257, 513 * 513, 1025 * 1025, 2049 * 2049,
    4097 * 4097, 8193 * 8193, 16385 * 16385, 32769 * 32769,
};
```

给定c，二分搜索可找到对应的s，即最接近的2^s

```
unsigned* temp = upper_bound(ss, ss + 16, c);
int s = temp - ss;
unsigned x0 = 1 << s;
```

查找二叉树

采用树结构

按照以上划分区间的方法，可以发现刻度并不均匀

考虑最后一段区间

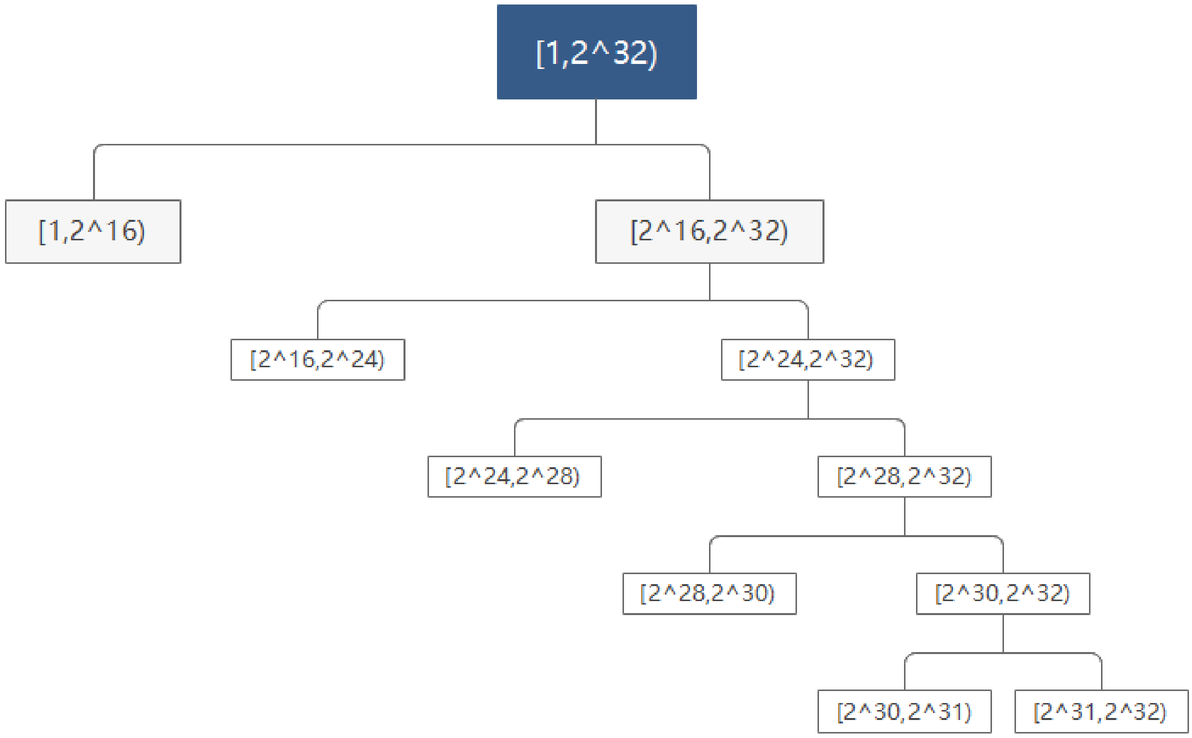
$$[2^{31}, 2^{32})$$

发现它占整个输入空间大小的一半，即

$$2^{32} - 2^{31} = \frac{1}{2}(2^{32} - 1)$$

如果采用二分搜索，要确定该区间点的位置，需要多次比较，遍历到最后一层

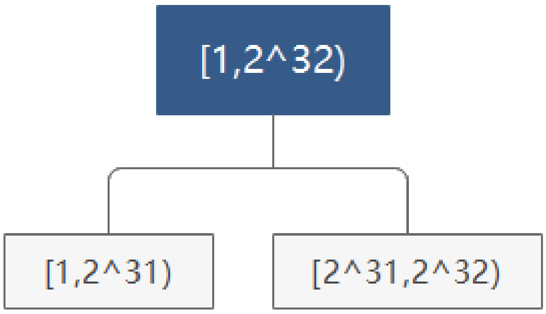
也就是说对于不同频率的点，需要的比较次数相同：



因此，我们可以用树的结构来优化

- 将频率较高的点对应的区间，放在深度较低的树节点，以较少的比较次数访问它
- 将频率较低的点对应的区间，放在深度较高的树节点，以较多的比较次数访问它

这样的好处是，频率较高的点能够快速访问，从而减少查找时间：



频率与概率

上面提到的频率，是在整个输入空间

$$[1, 2^n)$$

各区间点c出现的个数

$$[1, 2^1), [2^1, 2^2), \ldots, [2^{n-1}, 2^n)$$

我们可以进一步将**输入c的频率**转化为**取初始值s的概率**

对于每段区间对应的s

$$\text{区间} s : (2^{s-1}, 2^s], s = 1, 2, \ldots, \frac{n}{2}$$

输入空间c满足

$$\begin{gathered} 2^{s-1} < \sqrt{c} \leq 2^s \\ (2^{s-1}+1)^2 \leq c < (2^s+1)^2 \end{gathered}$$

因为c的输入空间为

$$[1, 2^n)$$

所以$\sqrt{c}$的输入空间为

$$[1, 2^{\frac{n}{2}})$$

考虑$\sqrt{c}$输入空间点的分布，定义**选取初始值s的概率**为

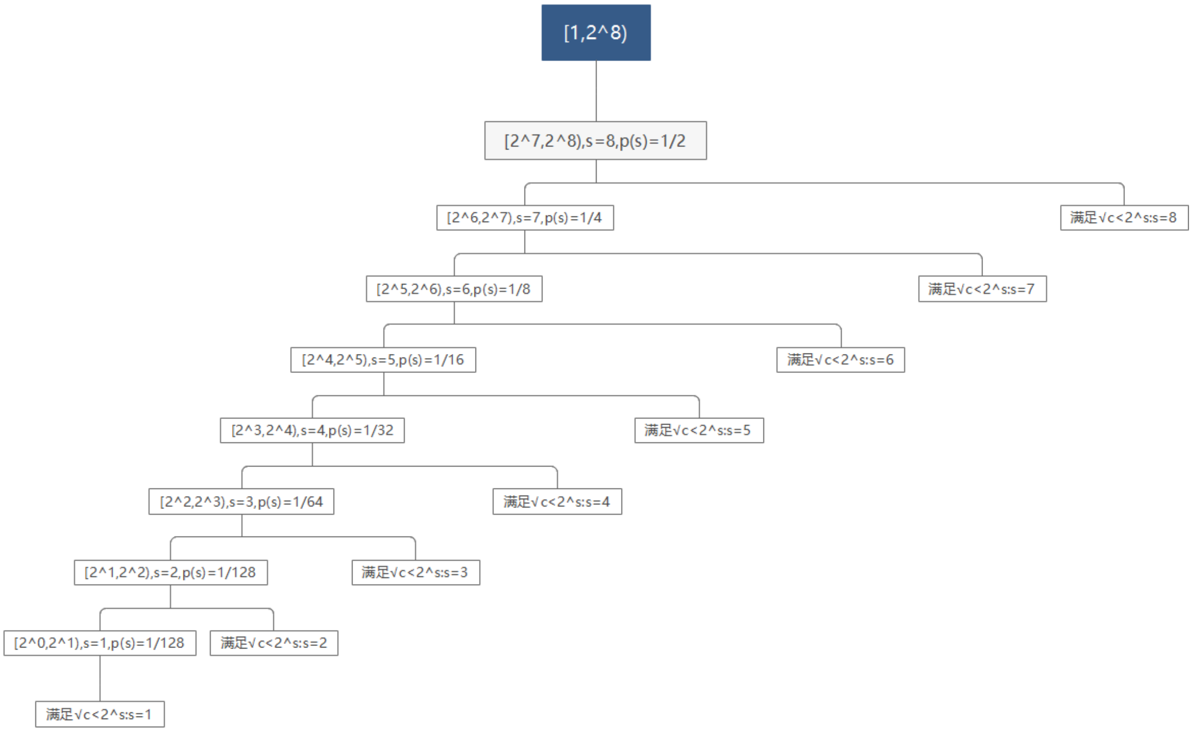
$$p(s) = \frac{2^s - 2^{s-1}}{2^{\frac{n}{2}} - 1}, s = 1, 2, \ldots, \frac{n}{2}$$

由于**输入规模n足够大**，且我们考虑的是**整个输入空间**$[1, 2^n)$

所以我们可直接从概率最大的区间查找初始值s，令$x_0 = 2^s$

例：令$n = 16$

s	1	2	3	4	5	6	7	8
概率	$\frac{1}{128}$	$\frac{1}{128}$	$\frac{1}{64}$	$\frac{1}{32}$	$\frac{1}{16}$	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$



最优查找二叉树

定义树的节点

```
struct node {
    int key;
    int value;
};
```

`key`为$s-1$，用于查找；`value`为s，为查找结果

对于每个节点，我们定义**期望**为**概率**和**查找深度**的乘积：

$$expection(s) = p(s) \cdot depth(s)$$

考虑**期望和**

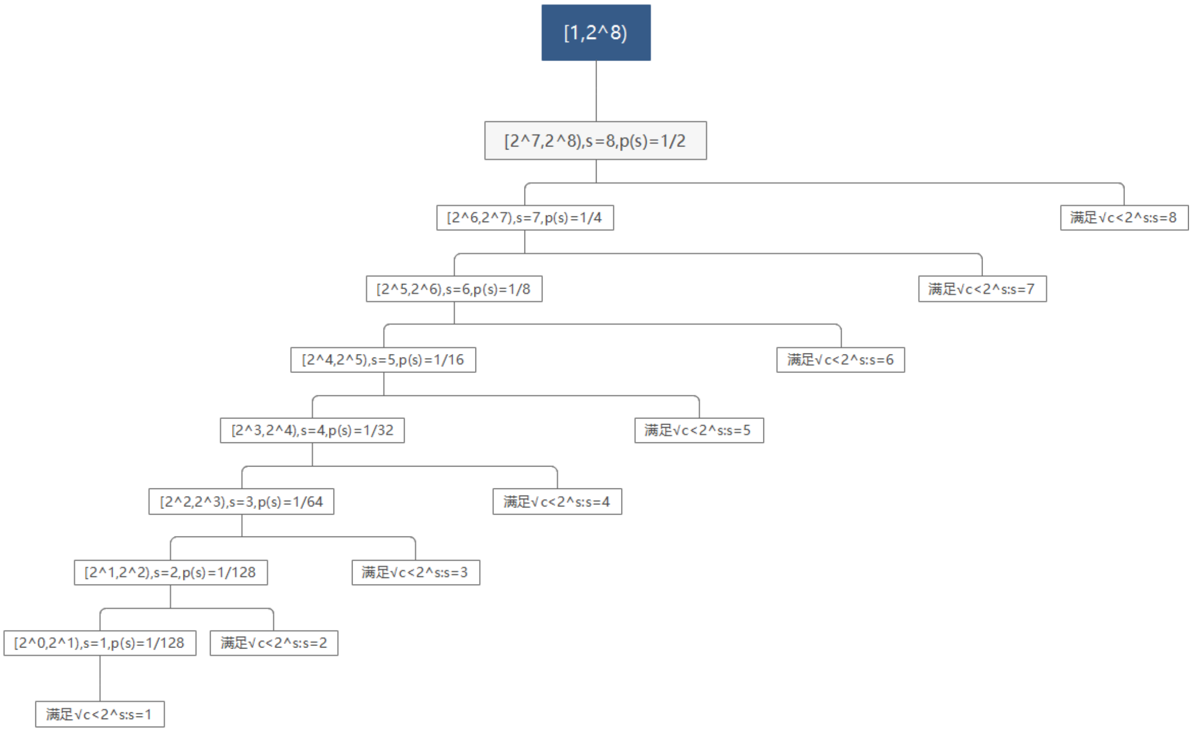
$$Expection(s) = \sum_{s=1}^{\frac{n}{2}} expection(s)$$

我们定义：当期望和最小时为**最优**

$$tree_{optimal} = min(\bigcup_{Tree} Expection(s))$$

为了达到最优，我们需要将概率较大的s值放在深度较低的树节点上

经过枚举验证，我找到最优的树结构为：



效率比较

范围选取$[1,2^{24})$

算法	是否有误差	平均用时	平均迭代次数
sqrt	无	0.0613183us	
my_isqrt	无	0.639274us	2.57015
isqrt2	无	0.135822us	15.8283
isqrt3	无	0.110519us	2.57181
isqrt4	无	0.14771us	15

```
迭代开始: 00000000000000000000000000000001(32位)
迭代结束: 00000001000000000000000000000000(32位)

cmath sqrt: 0.0613183us
my_isqrt: 平均迭代次数: 2.57015 平均时间: 0.639274us
isqrt2: 平均迭代次数: 15.8283 平均时间: 0.135822us
isqrt3: 平均迭代次数: 2.57181 平均时间: 0.110519us
isqrt4: 平均迭代次数: 15 平均时间: 0.14771us
```

范围选取$[1,2^{16})$

算法	是否有误差	平均用时	平均迭代次数
sqrt	无	0.0887451us	
my_isqrt	无	1.05694us	2.06581
my_isqrt_op	无	0.221732us	2.08838
isqrt2	无	0.202795us	9.65709
isqrt3	无	0.181285us	2.08813
isqrt4	无	0.336838us	14.9998

```
迭代开始: 00000000000000000000000000000001(32位)
迭代结束: 00000000000000010000000000000000(32位)

cmath sqrt: 0.0887451us
my_isqrt: 平均迭代次数: 2.06581 平均时间: 1.05694us
my_isqrt_op: 平均迭代次数: 2.08838 平均时间: 0.221732us
isqrt2: 平均迭代次数: 9.65709 平均时间: 0.202795us
isqrt3: 平均迭代次数: 2.08813 平均时间: 0.181285us
isqrt4: 平均迭代次数: 14.9998 平均时间: 0.336838us
```

代码

```
#include<cstdio>
#include<iostream>
#include<algorithm>
#include<cmath>
#include<cstdlib>
#include<ctime>
#include<bitset>
#include<windows.h>

using namespace std;

unsigned ss[16] = {
    4, 9, 25, 81,
    17 * 17, 33 * 33, 65 * 65, 129 * 129,
    257 * 257, 513 * 513, 1025 * 1025, 2049 * 2049,
    4097 * 4097, 8193 * 8193, 16385 * 16385, 32769 * 32769,
};

struct node {
    int key;
    int value;
    node* left;
    node* right;
};

struct tree {
    node* root;
```

```
};

tree t;

unsigned my_isqrt_num = 0;
unsigned my_isqrt_op_num = 0;
unsigned isqrt2_num = 0;
unsigned isqrt3_num = 0;
unsigned isqrt4_num = 0;

unsigned my_isqrt(unsigned);
unsigned my_isqrt_op(unsigned);
unsigned isqrt2(unsigned);
unsigned isqrt3(unsigned);
unsigned isqrt4(unsigned);
double tickTock(unsigned, unsigned(*func)(unsigned));
double tickTock(double, double(*func)(double));

void init();
int search(int, node*);

int main() {
    init();

    double time = 0;
    unsigned start = 1;
    unsigned top = (1 << 24);
    cout << "迭代开始: " << bitset<32>(start) << "(32位)" << endl;
    cout << "迭代结束: " << bitset<32>(top) << "(32位)" << endl << endl;

    for (unsigned c = start; c < top; c++) {
        time += tickTock((double)c, sqrt);
    }
    cout << "cmath sqrt: ";
    cout << (time / top) << "us" << endl;

    time = 0;
    for (unsigned c = start; c < top; c++) {
        time += tickTock(c, my_isqrt);
    }
    cout << "my_isqrt: ";
    cout << "平均迭代次数: ";
    cout << ((float)my_isqrt_num / top);
    cout << " 平均时间: ";
    cout << (time / top) << "us" << endl;

    time = 0;
    for (unsigned c = start; c < top; c++) {
        time += tickTock(c, my_isqrt_op);
    }
    cout << "my_isqrt_op: ";
    cout << "平均迭代次数: ";
    cout << ((float)my_isqrt_op_num / top);
    cout << " 平均时间: ";
    cout << (time / top) << "us" << endl;

    time = 0;
    for (unsigned c = start; c < top; c++) {
```

```
        time += tickTock(c, isqrt2);
    }
    cout << "isqrt2: ";
    cout << "平均迭代次数: ";
    cout << ((float)isqrt2_num / top);
    cout << " 平均时间: ";
    cout << (time / top) << "us" << endl;

    time = 0;
    for (unsigned c = start; c < top; c++) {
        time += tickTock(c, isqrt3);
    }
    cout << "isqrt3: ";
    cout << "平均迭代次数: ";
    cout << ((float)isqrt3_num / top);
    cout << " 平均时间: ";
    cout << (time / top) << "us" << endl;

    time = 0;
    for (unsigned c = start; c < top; c++) {
        time += tickTock(c, isqrt4);
    }
    cout << "isqrt4: ";
    cout << "平均迭代次数: ";
    cout << ((float)isqrt4_num / top);
    cout << " 平均时间: ";
    cout << (time / top) << "us" << endl;
    return 0;
}

unsigned my_isqrt(unsigned c) {
    // 牛顿法
    if (c <= 1) return c;
    unsigned* temp = upper_bound(ss, ss + 16, c);
    int s = temp - ss;
    unsigned x0 = 1 << s;
    unsigned x1 = (x0 + (c >> s)) >> 1;
    while (x1 < x0)
    {
        x0 = x1;
        x1 = (x0 + c / x0) >> 1;
        my_isqrt_num++;
    }
    return x0;
}

unsigned my_isqrt_op(unsigned c) {
    int s = search(c, t.root);
    unsigned x0 = 1 << s;
    unsigned x1 = (x0 + (c >> s)) >> 1;
    while (x1 < x0)
    {
        x0 = x1;
        x1 = (x0 + c / x0) >> 1;
        my_isqrt_op_num++;
    }
    return x0;
}
```

```
unsigned isqrt2(unsigned x)
{
    unsigned a = 1; //e.
    unsigned b = (x >> 5) + 8; //þ.
    if (b > 65535) b = 65535; //a <= sqrt(x) <= b
    do {
        unsigned m = (a + b) >> 1;
        if (m * m > x) b = m - 1;
        else a = m + 1;
        isqrt2_num++;
    } while (b >= a);
    return a - 1;
}

unsigned isqrt3(unsigned x)
{
    if (x <= 1) return x;
    unsigned x1 = x - 1;
    int s = 1;
    if (x1 > 65535) { s += 8; x1 >>= 16; }
    if (x1 > 255) { s += 4; x1 >>= 8; }
    if (x1 > 15) { s += 2; x1 >>= 4; }
    if (x1 > 3) { s += 1; }
    unsigned x0 = 1 << s;
    x1 = (x0 + (x >> s)) >> 1;
    while (x1 < x0) {
        x0 = x1;
        x1 = (x0 + x / x0) >> 1;
        isqrt3_num++;
    }
    return x0;
}

unsigned isqrt4(unsigned M)
{
    unsigned int N, i;
    unsigned long tmp, ttp;
    if (M == 0)
        return 0;
    N = 0;
    tmp = (M >> 30);
    M <<= 2;
    if (tmp > 1)
    {
        N++;
        tmp -= N;
    }
    for (i = 15; i > 0; i--)
    {
        N <<= 1;
        tmp <<= 2;
        tmp += (M >> 30);
        ttp = N;
        ttp = (ttp << 1) + 1;
        M <<= 2;
        if (tmp >= ttp)
        {
```

```
            tmp -= ttp;
            N++;
        }
        isqrt4_num++;
    }
    return N;
}

void init() {
    node* n1, * n2, * n3, * n4, * n5, * n6, * n7, * n8;
    n1 = (node*)malloc(sizeof(node));
    n2 = (node*)malloc(sizeof(node));
    n3 = (node*)malloc(sizeof(node));
    n4 = (node*)malloc(sizeof(node));
    n5 = (node*)malloc(sizeof(node));
    n6 = (node*)malloc(sizeof(node));
    n7 = (node*)malloc(sizeof(node));
    n8 = (node*)malloc(sizeof(node));

    n1->key = 7;
    n1->value = 8;
    n2->key = 6;
    n2->value = 7;
    n3->key = 5;
    n3->value = 6;
    n4->key = 4;
    n4->value = 5;
    n5->key = 3;
    n5->value = 4;
    n6->key = 2;
    n6->value = 3;
    n7->key = 1;
    n7->value = 2;
    n8->key = 0;
    n8->value = 1;

    t.root = n1;
    n1->left = n2;
    n2->left = n3;
    n3->left = n4;
    n4->left = n5;
    n5->left = n6;
    n6->left = n7;
    n7->left = n8;
}

int search(int c, node* root) {
    if ((1 << (root->key*2)) <= c)
        return root->value;
    else
        return search(c, root->left);
}

double tickTock(unsigned n, unsigned(*func)(unsigned))
{
    double sum = 0;

    double run_time;
```

```
    _LARGE_INTEGER time_start;  //开始时间
    _LARGE_INTEGER time_over;   //结束时间
    double dqFreq;      //计时器频率
    LARGE_INTEGER f;    //计时器频率
    QueryPerformanceFrequency(&f);
    dqFreq = (double)f.QuadPart;
    QueryPerformanceCounter(&time_start);   //计时开始
    sum = func(n);//调用函数A
    QueryPerformanceCounter(&time_over);    //计时结束
    run_time = 1000000 * (time_over.QuadPart - time_start.QuadPart) / dqFreq;
    // 乘以1000000把单位由秒化为微秒，精度为1000 000/（cpu主频）微秒
    // printf("result: %lf, run_time: %fus\n", sum, run_time);
    return run_time;
}

double tickTock(double n, double(*func)(double))
{
    double sum = 0;

    double run_time;
    _LARGE_INTEGER time_start;  //开始时间
    _LARGE_INTEGER time_over;   //结束时间
    double dqFreq;      //计时器频率
    LARGE_INTEGER f;    //计时器频率
    QueryPerformanceFrequency(&f);
    dqFreq = (double)f.QuadPart;
    QueryPerformanceCounter(&time_start);   //计时开始
    sum = func(n);//调用函数A
    QueryPerformanceCounter(&time_over);    //计时结束
    run_time = 1000000 * (time_over.QuadPart - time_start.QuadPart) / dqFreq;
    // 乘以1000000把单位由秒化为微秒，精度为1000 000/（cpu主频）微秒
    // printf("result: %lf, run_time: %fus\n", sum, run_time);
    return run_time;
}
```