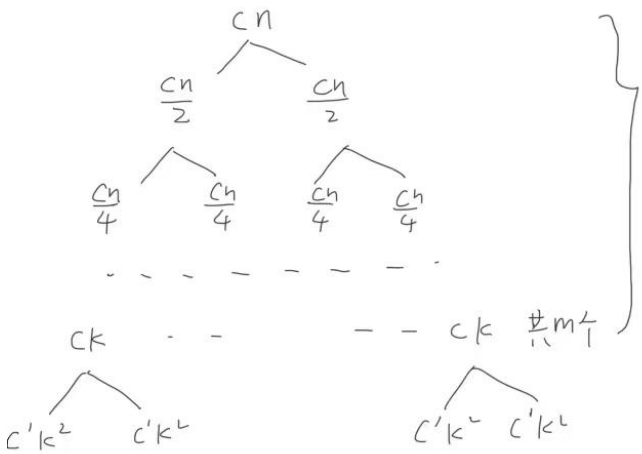
一、理论上对 K 的选择：

$f(n,k) = 2f(\frac{n}{2},k) + cn$　设插入排序为 $O(n) = c'n^2$



设 $ck = c\frac{n}{m}$

则 $m = \frac{n}{k}$

$2^{h-1} = m$

$h = \lg m + 1$

$f(n,k) = hcn + 2mc'k^2$

$= cn(\lg\frac{n}{k} + 1) + 2\frac{n}{k}c'k^2$

$= cn\lg\frac{n}{k} + cn + 2c'nk$

其中 $c_1 = c$，$c_2 = 2c'$ $= c_1 n\lg\frac{n}{k} + c_2 nk + c_1 n$

$O(f(n,k)) = O(n\lg\frac{n}{k} + nk)$

设 $g(k) = f(n,k) = c_1 n\lg n - c_1 n\lg k + c_2 nk + c_1 n$

$g'(k) = -\frac{c_1 n}{k} + c_2 n$　$g'(k)$ 在 $(0,+\infty)$ 单调↑

$g'(k_0) = 0$　$k_0 = \frac{c_1}{c_2}$　∴ $g(k)$ 在 $(0,k_0)$↓ $(k_0,+\infty)$↑

∴ $k = \frac{c_1}{c_2}$ 时 $g(k)$ 最小，最佳.

$= \frac{c}{2c'}$

二、实验中对 K 的选择：

多次实验得出发现，因为实验数据不可能做到无限大，把 n 无限分成一半，会出现以下情况：

(1) 区间长度分到成为奇数之后继续分下去就会出现两个区间长度不同的情况。

(2) 不是所有的 K 都有意义。例如：对于 n=2048 来说，K 只有取 1024,512,256 等 2 的幂次才有意义，若 K 取其他的存在于在这些数之间的数，实际上是和最大的比其小的 2 的幂次是一样的情况。

另外，为了减小误差：

1. 测试求最好的 K 时每次都使用一样的数据，并且使用了 4 个数据集测试，防止因数据问题造成时间上的差距。先用随机数产生足够多的数存入文档文件中，在测试时间时数中的数据就从这些存有足够多的文档文件中从头开始取。
2. 求取时间时多次尝试取平均，宏定义 CNT 为尝试次数，若数据量较大应该适量减小 cnt，若数据量较少可以适度增加 cnt 来提高准确性。

1、Try1:
因为理论得出的关于 K 的函数是一个先减后增的函数，所以首先取 left=0,right=n,k 为 left+(right-left)/2，再去比较取 k-1,k+1 的情况下得到的时间：
（1）若 t(k-1)>t(k)>t(k+1)则说明[k-1,k+1]属于减区间，则极小值 bestK>k,让 left=k，然后继续该操作；
（2）若 t(k-1)<t(k)<t(k+1)则说明[k-1,k+1]属于增区间，则极小值 bestK<k,让 right=k，，然后继续该操作；
（3）其他则说明 k 就是 bestK。
但是由于上面提到的情况（2）以及代码运行的时间误差，导致一般测出来 bestK 就是 n/2，此结果是根据 n 的变化而变化的，显然是不应该的，说明此方法并不适用于实际。
于是把 k-1,k+1 改成(k-left)/2 和(right-k)/2,但是和 k-1,k+1 一样仍然误差很大，算出来基本上是 n/2。

```
int selectK(int *array,int n)
{
    //try1:
    int left=0,right=n/2,k;
    while(left<right)
    {
        k=left+(right-left)/2;
        int
timel=getTime(array,n,left+(k-left)/2),timer=getTime(array,n,k
+(right-k)/2),time=getTime(array,n,k);
        if(timel<time&&time<timer)
        {
            right=k;
        }
        else if(timel>time&&time>timer)
        {
            left=k;
        }
        else
        {
            left=left+(k-left)/2;
            right=k+(right-k)/2;
        }
    }
    return k;
```

```
}
```

2、Try2:

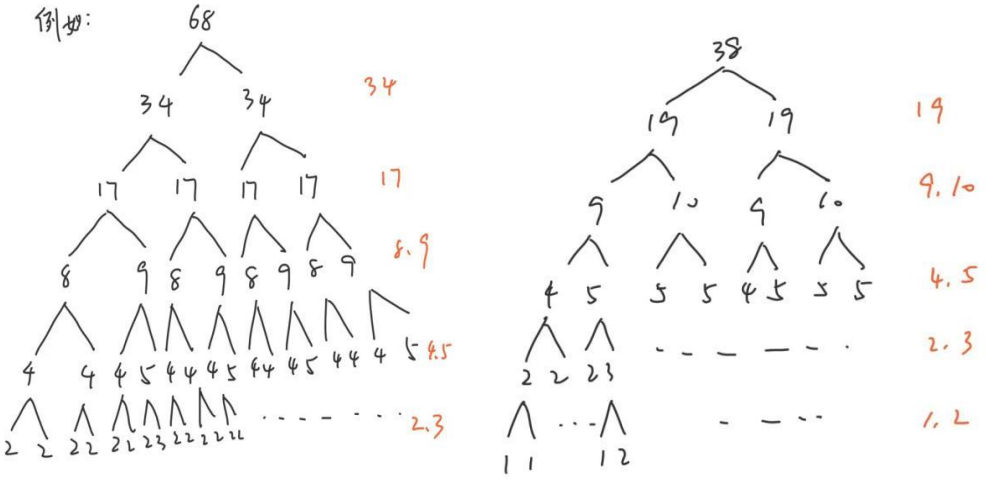
在没有考虑到情况（1）时采取了 k 取 n/2,n/4,n/8....的情况。但是后来因为考虑到上面提到情况（1），所以并不是只有对 n 不断除以 2 取整的数才是有意义的，其他数也可能是因为计算出奇数的情况而有意义，例如对于 17 来说，17 分为 8 和 9，再分为 4 和 4,4 和 5，再分为 2 和 2,2 和 2,2 和 2,2 和 3 的情况.....所以尝试了从 k=n 一直到 k=2 的枚举法，但是此方法效率过于低下，后面进行了优化。

```
int selectK(int *array,int n)
{
    //try2:
    unsigned long long int mintime=~0;
    int bestk;
    for(int k=n;k>1;k--)
    {
        int time=getTime(array,n,k);
        if(time<mintime)
        {
            bestk=k;
            mintime= time  ;
        }
    }
    return bestk;
}
```

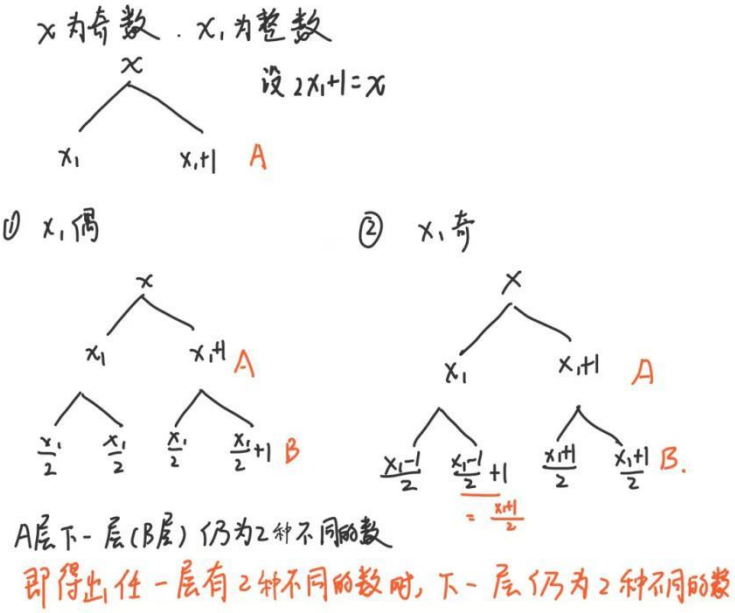
3、最终的方法：

最终发现在不断对于 n 分成两半的情况下，在未出现奇数时 k 都是取 n/2,n/4,n/8 有意义，一旦出现了奇数，就每次都考虑 k/2 和 k/2+1 的情况就可以了（即比之前多考虑一种加 1 的情况）

例如：



一旦出现奇数，下一层就都是 2 个数，且这两个数是对这个奇数除以二取整以及对这个奇数除以二取整并加一，后面层上也都是对上一层较小的数除以二并取整以及对其加一。
证明如下：




```
int selectK(int *array,int n)
{
    unsigned long long int mintime=~0;
    int bestk=n;
    int k=2*n;//k=n 的情况也要测试，而 getTime 的参数用的是 k/2，所以要初始化为 2n;
    while(k>=4&&k%2==0)//因为 k 不可以等于 1，若 k 为 3 则，k/2 是 1，所以 k 要>=4
    {
        int time=getTime(array,n,k/2);//getTime 的参数用的是 k/2 而不是 n 是因为:
        //k 为奇数时，才考虑 k/2 和 k/2+1 的情况,用 k/2 就可以在第一次循环时判断 k 是不是奇数就可以了
        if(time<mintime)
        {
            bestk=k/2;
            mintime= time  ;
        }
        k/=2;
    }
    while(k>=4)//出现了奇数后，每次都要多考虑一种加一的情况
    {
        int time1=getTime(array,n,k/2),time2=getTime(array,n,k/2+1);
        if(time1<mintime)
```

```
        {
            bestk=k/2;
            mintime= time1  ;
        }
        if(time2<mintime)
        {
            bestk=k/2+1;
            mintime= time2  ;
        }
        k/=2;
    }
    if(k==3)//也要把 k=2 考虑进去
    {
        int time=getTime(array,n,2);
        if(time<mintime)
        {
            bestk=2;
        }
    }
    return bestk;
}
```

其中出现的函数 getTime：

```
int getTime(int *array,int n ,int k)
{
    int* newarray=new int[n];
    unsigned long long int time=0;
    for(int count=0;count<CNT;count++)//多次求时间取平均，减小误差
    {
        for(int i=0;i<n;i++)
            newarray[i]=array[i];
        auto start = std::chrono::steady_clock::now();
        ImprovedSort(newarray,0,n-1,k);
        auto end = std::chrono::steady_clock::now();
        //cout<<std::chrono::duration_cast<std::chrono::microse
conds >(end - start).count()<<endl;
        // getchar();
        time+=std::chrono::duration_cast<std::chrono::microseco
nds>(end - start).count();
    }
    //getchar();
    delete[] newarray;
    cout<<k<<' '<<time<<endl;
```

```
    return time/CNT;
}
```

实验中发现的问题：

实验中发现每次第一次测试 K 的选取的时间都是最小值，且与之后的 K 选取其他情况都差很多。调试后发现是因为除了第一次，其他每次测试都用的是第一次排序完成好的数组。所以修改后在 getTime 中新建了 newarray 数组，防止修改原 array 数组，每次排序前都让 array 从头到尾赋值给 newarray。

但是这里并不清楚为什么去排排好序的数组会时间较长，且为了减小误差，每种 K 都测 100 遍，如果是因为 array 改变影响了时间，也应该只影响了第一次 K 的第一遍测试。

所以我尝试把每次的时间都打印出来，又发现在 n 取 1000 左右时时间为 0 纳秒的占大多数情况，所以 n 不够大时测试出来的时间其实都是不准确的。测试后发现 n>=8000 时测出时间出现 0 的情况几乎才会基本消失。

当我用 n=8192 去测试每次排 array（而不是 newarray）时发现并没有这样的情况，但是测出来最好的 K 是 2048 比每次排 newarray 测出来的 K 大很多，个人猜测可能是因为对于排好序的数组来说，insertsort 会快一些？

最好的 K：

实验发现在数据量>=8000 左右时测出来的时间误差较小（n 过小时测出的时间有多次出现 0 的情况）。最终，n 从 8000 到 100000 的多次测试得出最好的 K 在个人电脑上一般在 30 左右。（因为 n 不同，k 取某些值不一定有意义，所以每次求出的最好的 k 是与 n 有关的，上面已作详细解释）