《算法设计与分析》

课程实验报告

学院：　计算机学院（国家示范性软件学院）

班级：　2019211308

姓名：　曾世茂 顾天阳

学号：　2019211532 2019211539

贡献：　50% 50%

目录

快速排序、合并排序

一、实验要求

1、 随机生成最大数值为 30000、长度分别为 2000、5000、10000、15000、20000、30000 的多组 ADD 不同的数组

2、 分别采用递归合并排序、非递归合并排序、快速排序 1 、快速排序 2 对其进行由小到大，并记录采用各种排序方法的运行时间，递归算法的递归层次。

3、 分别考察

(1) 问题规模 n、输入完全相同时，各个算法的运行时间差异

(2) 问题规模 n 相同、输入不同时，同一算法的运行时间随 ADD、DD 增长的变化情况

(3) 输入 I 取平均时，问题规模 n 对算法运行时间的差异

(4) 考察排序难度相同时，问题规模 n 对算法运行时间的影响

二、算法实现

1、 递归合并排序

利用分治策略，将待排序的数组分为两个大小大致相同的子集合，分别对两个子集合进行排序，随后将两个已排序的子集合合并到新数组中。

```
void merge_sort_by_recursion(int a[], int b[], int l, int r)
{
    if (l < r){
        int i = (l + r) / 2;
        merge_sort_by_recursion(a, b, l, i);
        merge_sort_by_recursion(a, b, i + 1, r);
        merge_rec(a, b, l, i, r);
        copy(a, b, l, r);
    }
}
```

2、 非递归合并排序

递归调用程序十分耗费时间。所以可以采用迭代的思想对合并排序进行优化。依次将数组中的元素排序成长度为 2、4、8. . . . 的有序对，直至整体有序。为了减少回写，偶数次将 a 数组写到 b 数组中，奇数次将 b 数组写入 a 数组中，可以减少一半的反复写入

```
int nonReCusiveMerge(int *a, int N, int M)
{
    int *b = new int[N];
    int s = 1;
    while (s < N)
    {
        merge_pass(a, b, s, N);
        s += s;
        merge_pass(b, a, s, N);
        s += s;
    }
```

```
}
void merge_pass(int x[], int y[], int s, int n)
{
    int i = 0;
    while (i <= n - 2 * s)
    {
        merge_non(x, y, i, i + s - 1, i + 2 * s - 1);
        i = i + 2 * s;
    }
    if (i + s < n)
        merge_non(x, y, i, i + s - 1, n - 1);
    else
        for (int j = i; j <= n - 1; j++){
            y[j] = x[j];
        }
}
```

3、 快速排序 1

利用分治的思想，将数组分解成三段，中间为一个基准元素，左段所有元素都比基准元素小，右段所有元素都比基准元素大。接着对左右段分别递归调用快速排序，即可对整体进行排序。

```
void quickSort(int *a, int l, int r){
    if (l < r) {
        int m = partition(a, l, r);
        quickSort(a, l, m - 1);
        quickSort(a, m + 1, r);
    }
}
```

4、 快速排序 2

快速排序要点为将数组分为左右大致长度相等的两个子问题。故基准元素的选择较为重要。当原数组接近倒序时，采取第一个元素作为基准元素分段将失效。故可以采用随机采取基准元素的方式进行优化

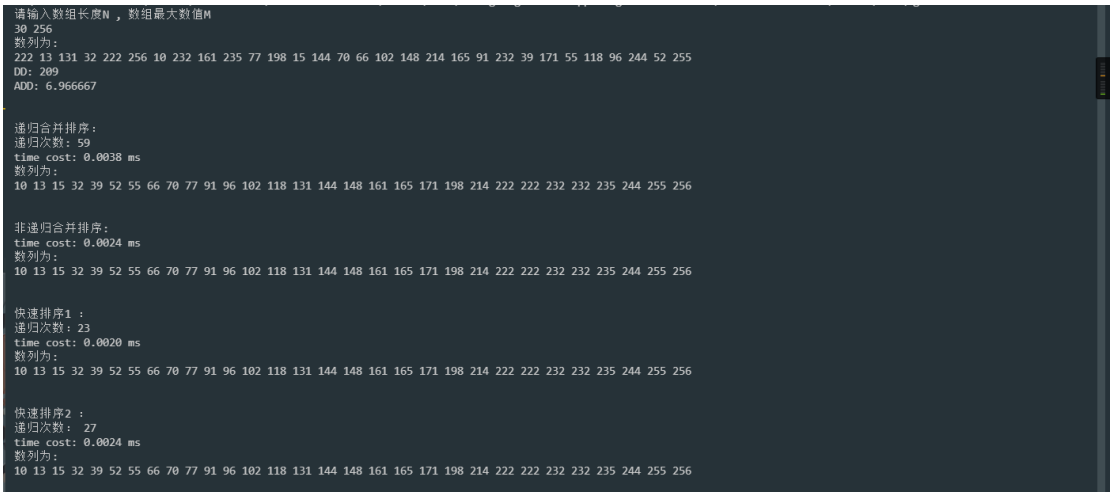
```
int randomized_partition(int *q, int l, int r)
{
   int i = l + rand() % (r - l + 1);
    swap(q[i], q[l]);
    return partition(q, l, r);
}
```

三、运行结果及其分析

1、 检验程序正确性

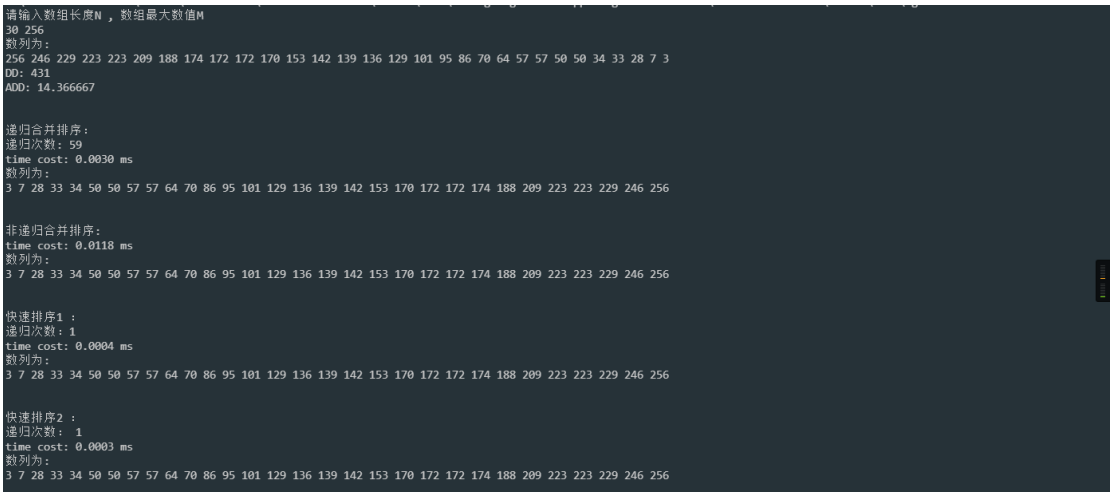
(1) 测试样例 1：较短的数组

(2) 测试样例 2：更长的数组

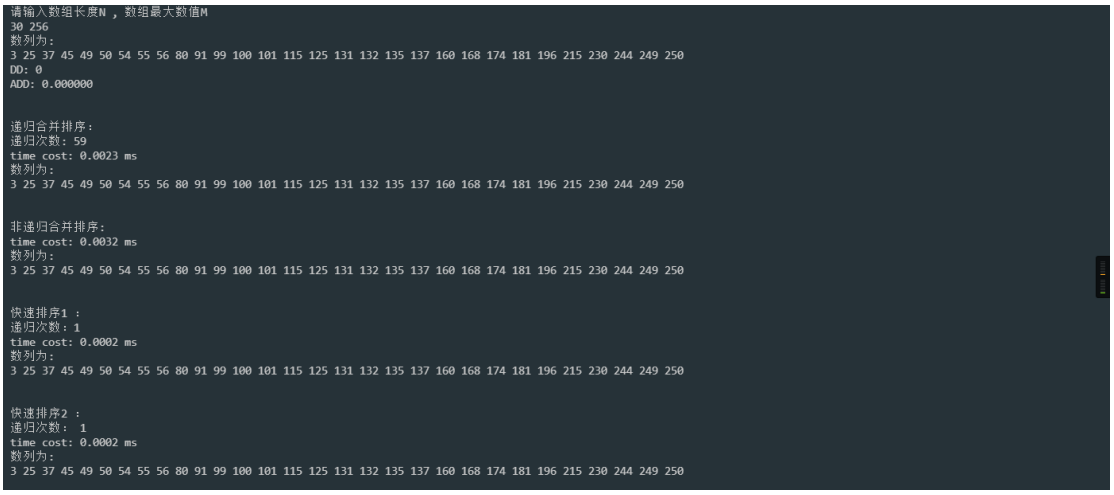


(3) 测试样例 3：长度为 1 的数组

(4) 测试样例 4：逆序的数组



(5) 测试样例 5：正序的数组



测试结果：

各个排序算法在不同的条件下均运行良好，可以正确地对数组进行排序

2、采用 30 组数据得到的不同问题规模 n、ADD 以及各个算法的运行时间复杂度关系如下所示

编号	长度 n	组号	DD	ADD	递归合并时间(ms)/递归层次	非递归合并时间(ms)	快排 1 时间(ms)/递归层次	快排 2 时间(ms)/递归层次
1	2000	1	805625	402.8125	0.2695 / 12	0.15	0.1731/28	0.2402/23
2	2000	1	834094	417.047	0.213/12	0.1495	0.172 / 24	0.2502 / 24
3	2000	1	989436	494.718	0.2187/12	0.1527	0.2473/25	0.2117/24
4	2000	1	1000117	500	0.2158 / 12	0.1515	0.1783 / 26	0.2131/25
5	2000	1	1175342	587.671	0.2126 / 12	0.1714	0.1954 / 22	0.2088 / 26
6	5000	2	4681113	936.2226	0.602 / 14	0.4094	0.4706 / 28	0.5959 / 30
7	5000	2	5556819	1111.3638	0.5973 / 14	0.4255	0.4934 / 34	0.5524 / 26
8	5000	2	6217493	1243.4986	0.5985 / 14	0.4213	0.4817 / 29	0.5902 / 25
9	5000	2	7290007	1458.0014	0.581 / 14	0.4119	0.4925 / 28	0.6327 / 29
10	5000	2	7796765	1559.353	0.5863 / 14	0.4187	0.4934 / 31	0.5637 / 26

11	10000	3	19438730	1943.873	1.242 / 15	0.8984	1.0186 / 30	1.166 / 35
12	10000	3	22138740	2213.874	1.2506 / 15	1.2035	1.0966 / 32	1.2921 / 31
13	10000	3	24488472	2448.8472	1.6052 / 15	0.877	1.0571 / 32	1.2111 / 29
14	10000	3	29113568	2911.3568	1.2428 / 15	0.8672	1.1563 / 35	1.2681 / 30
15	10000	3	30762415	3076.2415	1.4874 / 15	0.9164	1.0478 / 28	1.1623 / 30
16	15000	4	47218415	3147.894333	1.9810 / 15	1.4592	1.678 / 33	2.4575 / 33
17	15000	4	56394851	3759.656733	1.9498 / 15	1.9506	1.6191 / 33	2.0332 / 32
18	15000	4	56518264	3767.884267	2.4684 / 15	1.3603	1.7555 / 36	2.0389 / 33
19	15000	4	66028851	4401.9234	1.9944 / 15	1.6415	1.6492 / 31	1.9248 / 32
20	15000	4	70605810	4707.054	1.9244 / 15	1.3318	1.8197 / 31	2.0220 / 30
21	20000	5	76869971	3843.49855	2.6621 / 16	1.9189	2.2563 / 32	2.4408 / 30
22	20000	5	83104259	4155.21295	2.6374 / 16	1.9022	2.4922 / 32	2.5444 / 32
23	20000	5	100106407	5005.32035	2.9901 / 16	1.9008	2.2361 / 33	2.5918 / 32
24	20000	5	116774551	5838.72755	3.6323 / 16	1.8883	2.2535 / 33	2.5144 / 34
25	20000	5	123419197	6170.95985	2.8397 / 16	1.8781	3.2616 / 34	2.4726 / 32
26	30000	6	217251334	7241.711133	4.1148 / 16	3.1047	3.8763 / 36	3.8692 / 35
27	30000	6	223230258	7441.0086	4.1945 / 16	2.9773	3.8464 / 35	3.9204 / 36
28	30000	6	225192654	7506.4218	4.1809 / 16	2.9778	4.0101 / 33	4.0156 / 33
29	30000	6	248341961	8278.065367	4.2764 / 16	2.9773	3.8104 / 39	3.8412 / 37
30	30000	6	265037615	8834.587167	4.0893 / 16	3.4248	4.2246 / 35	4.1923 / 36

表 1

3、 考察当问题规模 n，输入 I 完全相同时，四种算法运行时间 T(n,I)差异选取的五行数据如下：

编号	长度 n	组号	DD	ADD	递归合并时间(ms)/递归层次	非递归合并时间(ms)	快排 1 时间(ms)/递归层次	快排 2 时间(ms)/递归层次
26	30000	6	217251334	7241.711133	4.1148 / 16	3.1047	3.8763 / 36	3.8692 / 35
21	20000	5	76869971	3843.49855	2.6621 / 16	1.9189	2.2563 / 32	2.4408 / 30
16	15000	4	47218415	3147.894333	1.9810 / 15	1.4592	1.678 / 33	2.4575 / 33
11	10000	3	19438730	1943.873	1.242 / 15	0.8984	1.0186 / 30	1.166 / 35
1	2000	1	805625	402.8125	0.2695 / 12	0.15	0.1731/ 28	0.2402/ 23
6	5000	2	4681113	936.2226	0.602 / 14	0.4094	0.4706 / 28	0.5959 / 30

平均	/	/	/	/	1.8	1.47	1.57	1.79

表 2

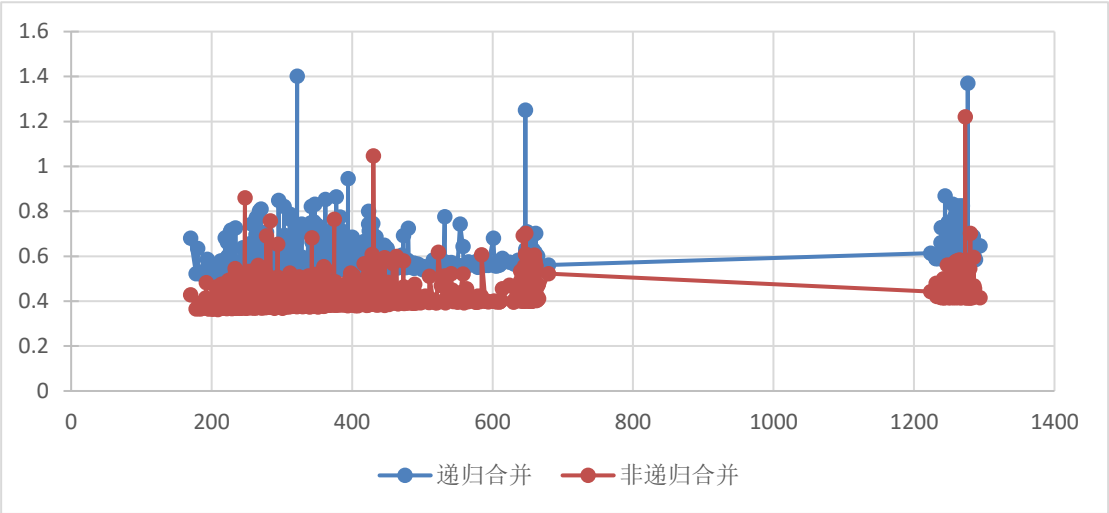
故当问题规模 n，输入 I 完全相同时，四种算法的运行速度由快到慢的顺序为：

非递归合并排序->快速排序 1->快速排序 2->递归合并排序。

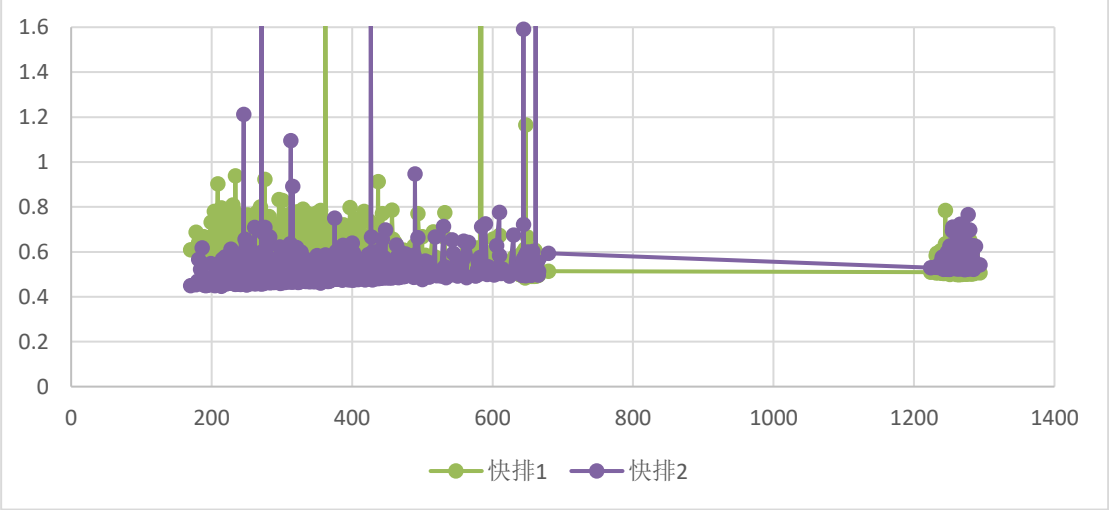
在实际使用中，添加有序序列检测优化后，快速排序 1 与快速排序 2 的递归次数相近，但考虑到快速排序 2 每次递归均需要调用 rand 等函数，导致时间略微增加。

4、 考察问题规模 n 相同，输入 I 不相同，比较算法运行时间 T(n, I)差异

由于上述数据样本量过小，相同规模下每组只有 5 组数据，偶然性较大，无法直接得出输入的 ADD 于算法运行时间的差异。故选择将样本量增加到 1000，考察 n 在 30000，M 在 30000 下 ADD 对算法运行时间的影响，结果如下（横坐标为 ADD，纵坐标为运行时间 ms）：



图表 1



图表 2

可见算法的运行时间随着 ADD 的增加缓慢上升，但变化幅度并不大，ADD 提高几倍时，运行时间只增加了大约百分之 10，在该测试条件下，测试数据的偶然性对排序时间的影响更大。这是由于归并排序、快速排序 2 从算法上并不受数据乱序程度的影响，快速排序 1 在添加了检测有序序列的优化后，递归次数与快速排序 2 的相近，可见受到有序序列的影响

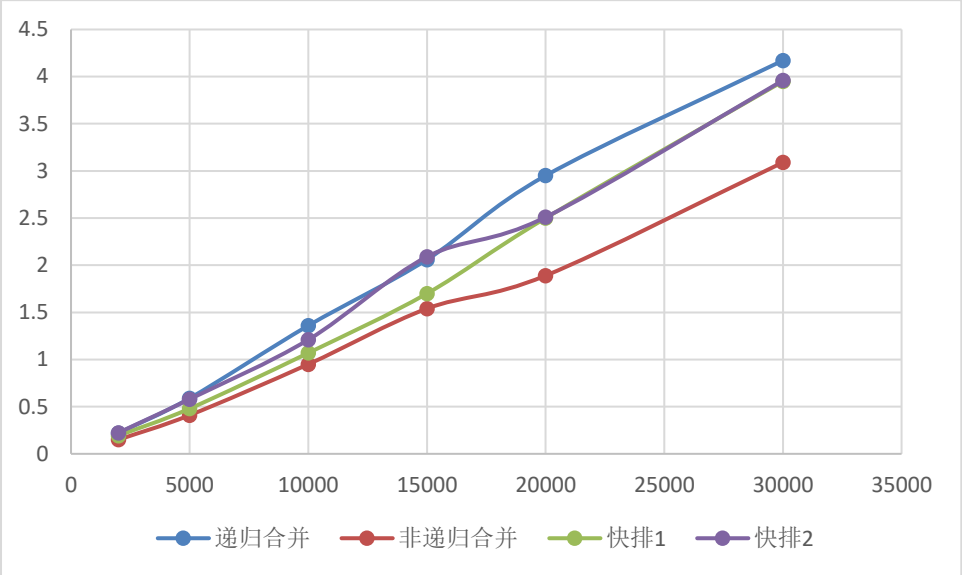
较小。故四种算法的运行时间对 ADD 的敏感程度均较低。

5、 考察问题规模 n 不同，输入 I 取平均，比较算法运行时间 T(n, I) 差异

长度 n	组号	avgDD	avgADD	递归归并平均时间	非递归归并平均时间	快排 1 平均时间	快排 2 平均时间
2000	1	960922.8	480.44	0.22	0.15	0.193	0.224
5000	2	6308439	1261	0.59	0.41	0.48	0.58
10000	3	25188385	2518	1.36	0.95	1.07	1.21
15000	4	59353238	3956	2.06	1.54	1.70	2.09
20000	5	100054877	5002	2.95	1.89	2.50	2.51
30000	6	235810764	7860	4.171	3.09	3.95	3.96

表 3

得到运行时间与 n 的关系图如下：



图表 3

可见相对于 ADD 对运行时间的轻微影响，问题规模对运行时间的影响较大，运行时间与问题规模 n 基本成线性增加。且各个排序算法增长速度与 n 大致一样，与各算法的实际平均复杂度 O(nlogn)一致，即大约呈线性关系。

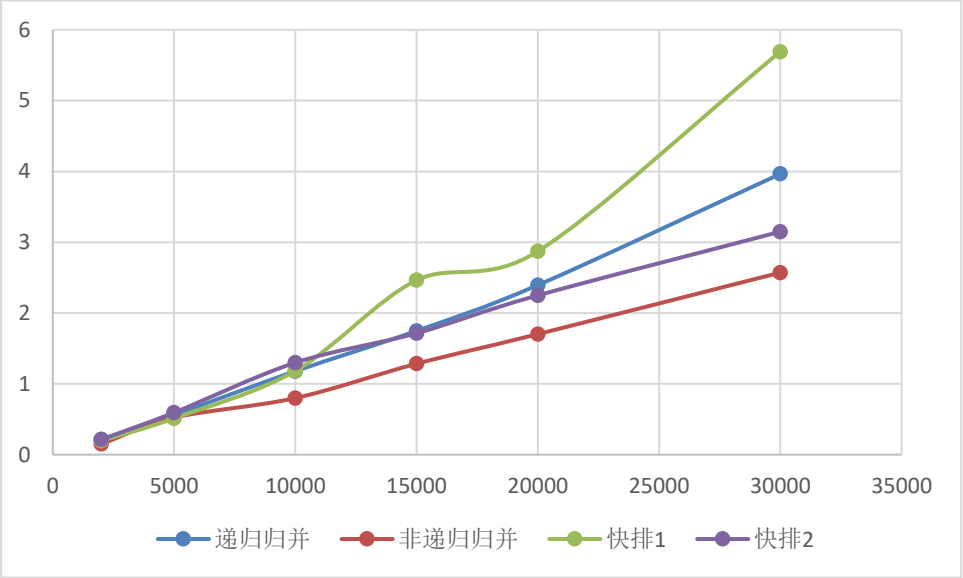
- 选出 6 组数据如下图所示

长度 n	组号	aDD	ADD	递归合并时间	非递归合并时间	快排 1 时间	快排 2 时间
2000	1	1059209	529.605	0.2138	0.1503	0.1939	0.2127
5000	2	3397597	679.519	0.5615	0.5228	0.5134	0.5941
10000	3	7002609	700.261	1.1786	0.7987	1.1829	1.2985

15000	4	10509519	700.635	1.7486	1.2835	2.464	1.7149
20000	5	15651454	782.573	2.3945	1.702	2.8717	2.2492
30000	6	28104288	936.81	3.9649	2.5713	5.6917	3.1493

表 4

由此得出的当排序难度相近时，问题规模 n 对运行时间的影响如下：



图表 4

拟合得到各算法时间 t(ms)与问题规模 n 的线性关系为：

非递归归并：

$$t = 8E-5*n + 0.0112$$

递归归并：

$$t = 0.0001n - 0.1315$$

快速排序 1：

$$t = 0.0002n - 0.4964$$

快速排序 2：

$$t = 0.0001n + 0.1123$$

由上图可看出，在排序难度相近的时候，对于递归归并排序、非递归归并排序、快速排序 2 来说，数据规模 n 与排序时间复杂度基本呈现线性关系，而快速排序 1 受到样本偶然性的可能性较大，排序时间较不稳定，对样本要求较高。但总体上来看，四种算法的数据规模 n 与排序时间复杂度基本呈现线性关系。

四、算法改进

1、 对快速排序基准元素选择的改进

由于基准元素的选择直接影响到划分子问题的均等性，所以基准元素的选择对

于快速排序十分重要。最理想的情况为每次选择都恰好选择到数组的中位数，以将数组划分成两个规模一样的子数组。但当数组有序的情况下，选择数组第一个数作为中位数往往会导致递归层次剧增。故可以通过随机数随机挑选数组中的某一个数作为基准元素，以减少有序数组对快速排序算法的影响。

由 表 1 计算得到两种快速排序的递归层次平均值如下

长度	快速排序 1	快速排序 2
2000	25	24.4
5000	30	27.2
10000	31.4	31
15000	32.8	32
20000	32.8	32
30000	35.6	35.4

表 5

可见仅由最深递归层次来看，快速排序 2 仅仅比快速排序 1 的最深递归层次略微减少一点。但由图表 2 可看出，在 ADD 较小时，快速排序 2 的时间整体上都比快速排序 1 低一些。更重要的是，快速排序 2 的波动性（紫色区域面积）明显小于快速排序 1，这说明快速排序 2 受到数据的影响程度较低，运行时间稳定。

线性时间选择

一、实验要求

1、采用线性时间选择算法，根据基站 k-dist 距离，分别挑选出 k-dist 值最小、第 5 小、第 50 小、最大的基站。
2、在排序过程中设置全局变量，记录选择划分过程的递归层次。
3、将教科书上的“一分为二”的子问题划分方法，改进为“一分为三”，比较这两种划分方式下，选择过程递归层次的差异。

二、随机选择算法

1、算法实现

该算法实际上是模仿快速排序算法设计出来的，基本思想也是对输入数组进行递归划分。与快速排序算法不同的是，它只对划分出的子数组之一进行处理。算法用到了 RandomizedPartition 随机划分函数，因此该算法是一个随机化算法。首先给出 RandomizedPartition 的简单实现：

```
template <class Type>
int RandomizedPartition(Type a[], int p, int r)
{
    int i = Random(p, r);
    Swap(a[i], a[p]);
    return Partition(a, p, r);
```

```
}
```

要找数组 a[0:n-1]中第 k 小元素，只要调用 RandomizedSelect(a, 0, n - 1, k)即可，具体算法如下：

```
template <class Type>
Type RandomizedSelect(Type a[], int p, int r, int k)
{
    if (p == r)
        return a[p];
    int i = RandomizedPartition(a, p, r);
    j = i - p + 1;
    if (k == j)
        return a[i];
    else if (k < j)
        return RandomizedSelect(a, p, i - 1, k);
    else
        return RandomizedSelect(a, i + 1, r, k - j);
}
```

在算法 RandomizedSelect 中执行 RandomizedPartition 后，数组 a[p:r]被划分成两个子数组 a[p:i]和 a[i+1:r]，使 a[p:i]中每个元素都不大于 a[i+1:r]中每个元素。接着算法计算子数组 a[p:i]中元素个数 j。如果 k=j，说明 a[i]即为所求；如果 k<j，则 a[p:r]中第 k 小元素落在子数组 a[p:i-1]中；如果 k>j，则要找的第 k 小元素落在子数组 a[i+1:r]中。由于此时已知道子数组 a[p:i]中元素均小于要找的第 k 小元素，因此要找的 a[p:r]中第 k 小元素是 a[i+1:r]中的第 k-j 小元素。

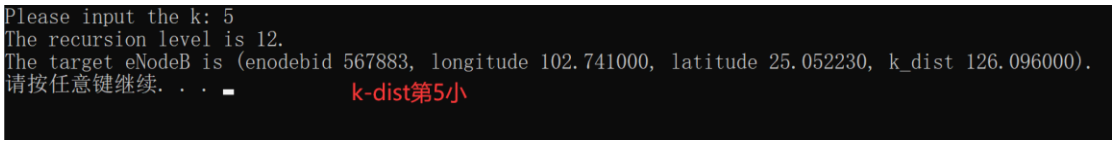
2、复杂度分析

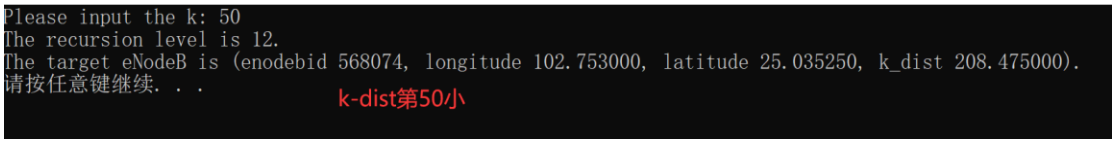
由于随机划分函数 RandomizedPartition()使用了一个随机产生器 Random，能随机地产生 p 和 r 之间的一个随机整数，因此 RandomizedPartition()产生的划分基准是随机的。在这种条件下，算法 RandomizedSelect 的平均时间复杂度为$O(n)$。

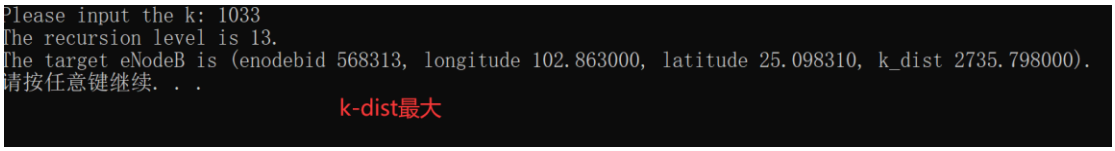
然而，在最坏情况下，例如，在找最小元素时，总是在最大元素处划分，此时时间复杂度为$O(n^2)$。

3、运行结果及分析

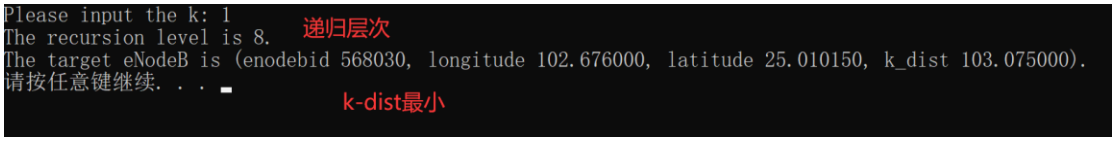
（1）一分为三

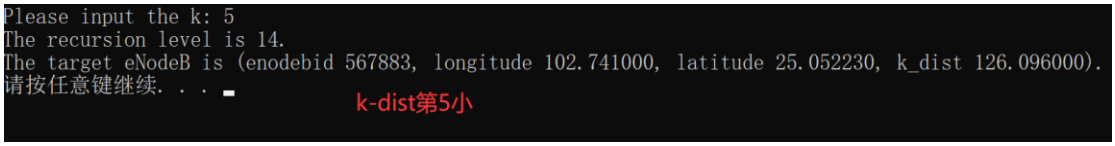


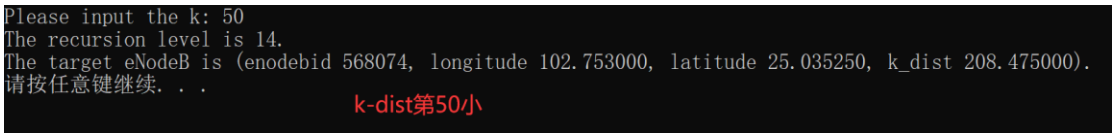


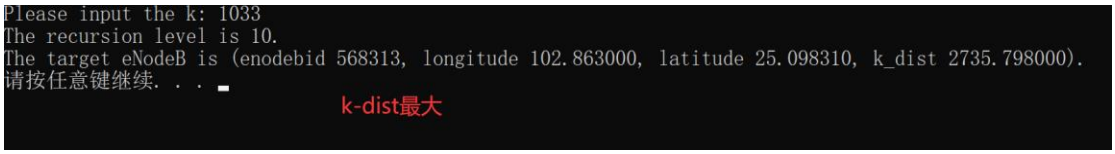


（2）一分为二









（3）结果分析

对比“一分为三”和“一分为二”的4次运行结果，可以发现，整体上“一分为三”的递归层次少于“一分为二”，这主要是因为“一分为三”的减治法有利于降低递归深度，速度快。但同时也发现，当k=1033时，“一分为三”的递归层次大于“一分为二”，这是因为该算法的RandomizedSelect中调用了Random，存在偶然性，这里属于“一分为二”的结果出现了更为合理的随机划分，导致递归层次减少。

三、随机选择算法的优化

下面讨论一个类似RandomizedSelect()但可以在最坏情况下用$O(n)$时间就完成选择任务的算法Select。如果能在线性时间内找到一个划分基准，使得按这个基准划分出的两个子数组的长度都至少为原数组的ε倍（$0<\varepsilon<1$），那么在最坏情况下用$O(n)$时间就可以完成选择任务。

1、算法实现

（1）将n个输入元素划分成⌈n/5⌉个组，每组5个元素，除可能一个组不是5个元素外。用任意一种排序算法，将每组中的元素排好序，并取出每组的中位数，共⌈n/5⌉个。

（2）递归调用Select找出这⌈n/5⌉个元素的中位数。如果⌈n/5⌉是偶数，就找它的两个中位数中较大的一个。然后以这个元素作为划分基准。

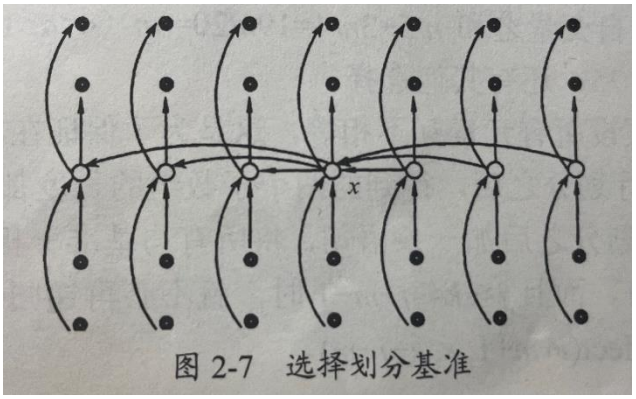


图 2-7　选择划分基准

这种情况下选出的划分基准 x 至少比 a[0:n-1] 中的$\frac{3}{10}n-1$个元素大，若要求划分后的 2 个子数组长度至少缩减 1/4，则：

$$\frac{3}{10}n-1\geq\frac{n}{4}\Rightarrow n\geq 20$$

据此，我们给出算法 Select 如下：

```
1.  template <class Type>
2.  Type Select(Type a[], int l, int r, int k)
3.  {
4.      if (r - l < 20)
5.      {
6.          quick_sort(l, r);
7.          return a[p + k - 1];
8.      }
9.
10.     for (int i = 0; i <= (r - l - 4) / 5; i++)
11.     {
12.         int s = l + 5 * i;
13.         int t = s + 4;
14.         quick_sort(s, t);
15.         swap(a, p + i, s + 2);
16.     }
17.
18.     Type x = select(a, l, l + (r - l - 4) / 5, (r - l + 6) / 10);
19.
20.     int i = Partition(a, l, r, x);
21.     int j = i - l + 1;
22.     if (k == j)
23.         return a[i];
24.     else if (k < j)
25.         return Select(a, l, i - 1, k);
26.     else
27.         return Select(a, i + 1, r, k - j);
28. }
```

2、复杂度分析

设 n=r-l+1，即 n 为输入数组的长度，算法的递归调用只要在 n>=20 时才执行，因此，当 n<20 时，算法 Select 所用的计算时间不超过一个常数 C1。找到中位数的中位数 x 后，算法 Select 以 x 为划分基准调用函数 Partition()对数组 a[p:r]进行划分，这需要$O(n)$时间。算法 Select 的 for 循环共执行约 n/5 次，每次需要$O(1)$时间，因此，执行 for 循环需要$O(n)$时间。

设对 n 个元素的数组调用 Select 需要 T(n)时间，那么找中位数的中位数 x 至多需要 T(n/5)时间。现已证明，按照算法所选的基准 x 进行划分所得到的两个子数组分别至多有 3n/4 个元素。所以无论对哪一个子数组调用都至多用 T(3n/4)时间。

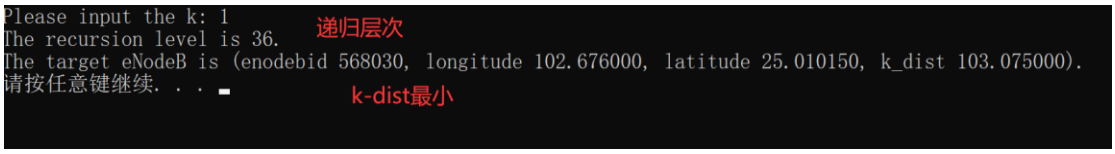
总之，可以得到关于 T(n)的递归式：

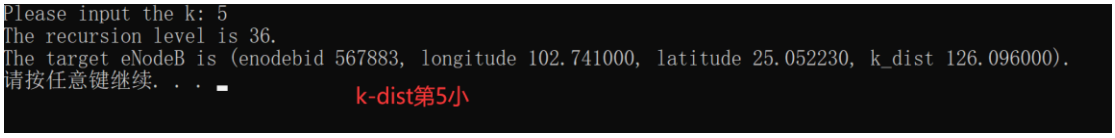
$$T(n) \leq \begin{cases} C_1 & n < 75 \\ C_2 n + T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) & n \geq 75 \end{cases}$$

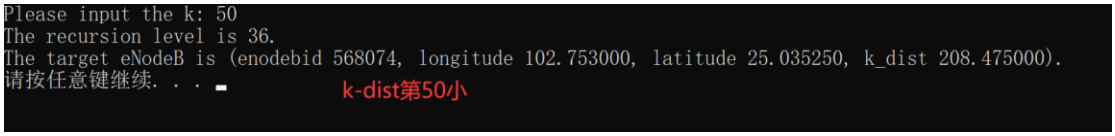
解此递归式，可得$T(n) = O(n)$。

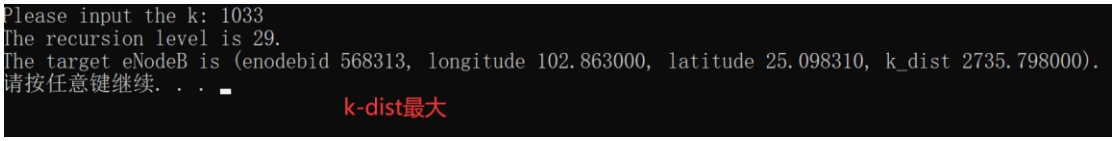
3、运行结果及分析

（1）一分为三

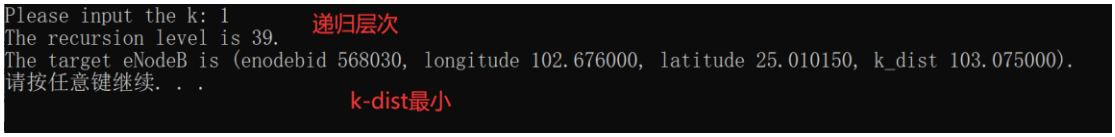


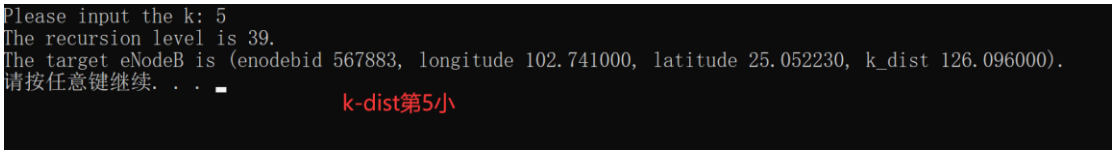






（2）一分为二





```
Please input the k: 50
The recursion level is 39.
The target eNodeB is (enodebid 568074, longitude 102.753000, latitude 25.035250, k_dist 208.475000).
请按任意键继续. . .
```
k-dist第50小

```
Please input the k: 1033
The recursion level is 37.
The target eNodeB is (enodebid 568313, longitude 102.863000, latitude 25.098310, k_dist 2735.798000).
请按任意键继续. . .
```
k-dist最大

（3）结果分析

对比“一分为三”和“一分为二”的4次运行结果，可以发现，整体上“一分为三”的递归层次少于“一分为二”，这主要是因为“一分为三”的减治法有利于降低递归深度，速度快。

最近平面点对

一、实验要求

1、采用平面最近点对算法，根据基站经纬度，挑选出距离非零且最近的两个基站、距离非零且次最近的两个基站。

2、返回最近/次最近的 2 个基站间距离、最近/次最近的 2 个基站点对（用基站 ENodeBID 表示）。

二、算法实现

1、设 S 中的点为平面上的点，它们都有两个坐标值 x 和 y。为了将平面上点集 S 线性分割为大小大致相等的两个子集 S1 和 S2，选取一垂直线 l:x=m 来作为分割直线。其中，m 为 S 中各点 x 坐标的中位数。由此将 S 分割为$S_1 = \{p \epsilon S | x(p) \leq m\}$和$S_1 = \{p \epsilon S | x(p) > m\}$。从而使 S1 和 S2 分别位于直线 l 的左侧和右侧。由于 m 是 S 中各点 x 坐标值的中位数，因此 S1 和 S2 的点数大致相等。

2、递归地在 S1 和 S2 上解最接近点对问题，分别得到 S1 和 S2 中的最小距离 d1 和 d2。现设 d=min{d1,d2}。若 S 的最接近点对(p,q)之间的距离小于 d，则 p 和 q 必分属于 S1 和 S2。不妨设$p \in S_1$，$q \in S_2$。那么 p 和 q 距直线 l 的距离均小于 d。因此，若用 P1 和 P2 分别表示直线 l 的左侧和右侧宽为 d 的两个垂直长条区域，则$p \in P_1$且$q \in P_2$。

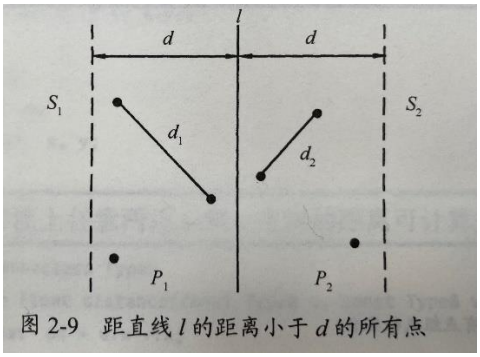


图 2-9　距直线 l 的距离小于 d 的所有点

3、满足 distance(p,q)<d 的 P2 的点一定落在一个 dx2d 的矩形 R 中，如图所示。由 d 的意

义可知，P2 中任何两个 S 中的点的距离都不小于 d，由此可以推出矩形 R 中最多只有 6 个 S 中的点。因此在分治法的合并步骤中，最多只需要检查 6xn/2=3n 个候选者。

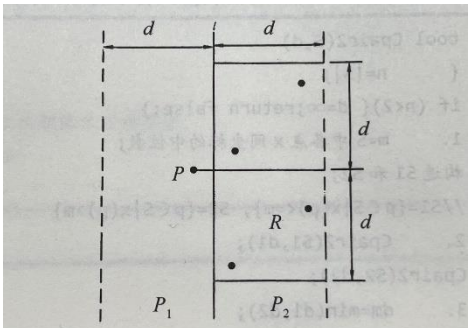


图 2-10　包含点 q 的 $d\times 2d$ 矩形 R

4、进一步的，我们将 p 和 P2 中所有 S2 的点投影到垂直线 l 上。由于能与 p 点一起构成最接近点对候选者的 S2 中点一定在矩形 R 中，所以它们在直线 l 上的投影点距 p 在 l 上投影点的距离小于 d。由上面的分析可知，这种投影点最多只有 6 个。因此，若将 P1 和 P2 中所有 S 中点按其 y 坐标排好序，则对 P1 中所有点，对排好序的点列做一次扫描，就可以找出所有最接近点对候选者。对 P1 中每点最多只要检查 P2 中排好序的相继 6 个点。给出分治法求二维点集最接近点对的算法如下：

```
1.  bool Cpair2(PointX X[], int n, PointX &a, PointX &b, double &d)
2.  {
3.      if (n < 2)
4.          return false;
5.
6.      PointX *tmp = new PointX[n];
7.      merge_sort(X, tmp, 0, n - 1);
8.      delete[] tmp;
9.
10.     PointY *Y = new PointY[n];
11.     for (int i = 0; i < n; i++)
12.         Y[i].set_point(X[i].get_x(), X[i].get_y(), i, X[i].get_enodebid());
13.
14.     PointY *tmp1 = new PointY[n];
15.     merge_sort(Y, tmp1, 0, n - 1);
16.     delete[] tmp1;
17.
18.     PointY *Z = new PointY[n];
19.     closest(X, Y, Z, 0, n - 1, a, b, d);
20.     delete[] Y;
21.     delete[] Z;
22.
23.     return true;
24. }
```

5、算法 Cpair2 中，具体计算最接近点对的工作有函数 closest 完成。

```
void closest(PointX X[], PointY Y[], PointY Z[], int l, int r, PointX &a, PointX &b, double &d)
{
    if (r - l == 1)
    {
        a = X[l];
        b = X[r];
        d = distance(X[l], X[r]);
        return;
    }

    if (r - l == 2)
    {
        double d1 = distance(X[l], X[l + 1]);
        double d2 = distance(X[l + 1], X[r]);
        double d3 = distance(X[l], X[r]);
        if (d1 <= d2 && d1 <= d3)
        {
            a = X[l];
            b = X[l + 1];
            d = d1;
            return;
        }
        if (d2 <= d3)
        {
            a = X[l + 1];
            b = X[r];
            d = d2;
        }
        else
        {
            a = X[l];
            b = X[r];
            d = d3;
        }
        return;
    }

    // 多于 3 点的情形，用分治法
    int m = (l + r) / 2;
    int f = l, g = m + 1;
    for (int i = l; i <= r; i++)
    {
        if (Y[i].p > m)
```

```
            Z[g++] = Y[i];
        else
            Z[f++] = Y[i];
    }

    closest(X, Z, Y, l, m, a, b, d);
    double dr;
    PointX ar, br;
    closest(X, Z, Y, m + 1, r, ar, br, dr);

    if (dr < d)
    {
        a = ar;
        b = br;
        d = dr;
    }

    // 重构数组 Y
    Merge(Z, Y, l, m, r);
    // d 矩形条内的点置于 Z 中
    int k = l;
    for (int i = l; i <= r; i++)
    {
        if (fabs(Y[m].x - Y[i].x) < d)
            Z[k++] = Y[i];
    }

    for (int i = l; i < k; i++)
    {
        for (int j = i + 1; j < k && Z[j].y - Z[i].y < d; j++)
        {
            double dp = distance(Z[i], Z[j]);
            if (dp < d)
            {
                d = dp;
                a = X[Z[i].p];
                b = X[Z[j].p];
            }
        }
    }
}
```

三、复杂性分析

经过预排序处理后的算法 Cpair2 所需的计算时间 T(n)满足递归方程：

$$T(n)=\begin{cases}O(1) & n<4\\ 2T\left(\frac{n}{2}\right)+O(n) & n\geq 4\end{cases}$$

解得$T(n)=O(nlogn)$。在渐近的意义下，此算法已是最优算法。

四、运行结果及分析

通过对 Cpair2 的预排序处理，程序可以在较快时间内获得结果。这里存在不同频点基站位置相同的情况，即经纬度相同，为了简便起见，程序只输出了其中一对基站，其余基站对遍历数组找取经纬度相同的即可。

```
The closest enodebs pair are eNodeB 567389 and eNodeB 566803.
Their distance is 5.788965.
```

最接近点对

```
The second closest enodebs pair are eNodeB 566784 and eNodeB 567222.
Their distance is 7.569397.
请按任意键继续. . .
```

次接近点对