算法基础 Lab 1

快速排序算法及其优化

王世炟

PB20151796

October 17, 2022

Part 1: 实验要求

本次实验要求我们完成**快速排序及其优化算法**的模型实现，并对 data.txt 中的数据进行排序测试和优化选择。具体需要完成以下部分：

- 完成基础的快速排序算法
- 对快速排序进行优化，主要方法有：选择不同的基准，在几乎有序时进行插入排序与聚集元素三种方法

Part 2: 快速排序基础算法

Algorithm 1 Algorithm of QuickSort

QUICKSORT(A,p,r)

1: if $p < r$ then
2: $q = PARTITION(A, p, r)$
3: $QUICKSORT(A, p, q-1)$
4: $QUICKSORT(A, q+1, r)$
5: end if

为了排序一个数组 A 的全部元素，初始调用是 QUICKSORT(A,1,A.length).
可以容易的将上述伪代码转化成代码：

```
1 void QuickSort::Sort(int p, int r, string base = "fixed", string
2 optimization = "normal")
3 {
4     int q;
5     if (p < r)
6     {
7         q = (*this).Partition(p, r, base);
```

```
        (*this).Sort(p, q - 1, base);
        (*this).Sort(q + 1, r, base);
    }
    return;
}
```

数组的划分

Algorithm 2 Algorithm of Partition

PARTITION(A,p,r)

1: $x = A[r]$
2: $i = p - 1$
3: for $j = p$ ***to*** $r - 1$ do
4: if $A[j] < x$ then
5: $i = i + 1$
6: *exchange* $A[i]$ *with* $A[j]$
7: end if
8: end for*exchange* $A[i + 1]$ *with* $A[r]$
9: return i+1

上述伪代码转化成代码为：

```
int QuickSort::Partition(int p, int r, string base)
{
    int x, i;
    x = Arr[r];
    i = p - 1;
    for (int j = p; j < r; j++)
    {
        if (Arr[j] <= x)
        {
            i += 1;
            swap(Arr[i], Arr[j]);
        }
    }
    swap(Arr[i + 1], Arr[r]);
    return i + 1;
}
```

Part 3: 优化算法

优化算法主要有以下几种：

- 基准的选择
  - 固定基准
  - 随机基准
  - 三数取中
- 在几乎有序时进行插入排序
- 聚集元素

3.1 基准的选择

3.1.1 固定基准

固定基准即我们基础的快速排序算法中的基准，每次均选择最后一个数作为基准，，这个时候数组数据呈现出一边倒的趋势此时快速排序的时间复杂度达到最坏的情况逼近 $O(n^2)$ 甚至达到为 $O(n^2)$，这样的快速排序远远达不到我们的需求

3.1.2 随机基准

为了解决上面提到的问题，我们采取在每次 PARTITION 前进行在 [p,r] 中随机选取一个数作为基准，这样能有效的解决上述问题，虽然有些情况下会导致算法变慢，但对于某些情况来说的确有很大的提升：

源代码：

```
int QuickSort::Partition(int p, int r, string base)
{
    int x, i;
    if (base == "random")
    {
        srand((unsigned)time(NULL));
        i = rand() % (r - p + 1) + p;
        swap(Arr[r], Arr[i]);
    }
    x = Arr[r];
    i = p - 1;
    for (int j = p; j < r; j++)
    {
```

```
        if (Arr[j] <= x)
        {
            i += 1;
            swap(Arr[i], Arr[j]);
        }
    }
    swap(Arr[i + 1], Arr[r]);
    return i + 1;
}
```

3.1.3 三数取中

对于快速排序的 PARTITION 来说，假如每次的基准都是中位数，那么算法的效果将会很好。虽然我们不能达到这个要求，但是我们可以尽可能贴近这个效果，三数取中的方法自然地成为选择之一。

三数取中：这里面的三数分别为 $A[p], A[r], A[(p+r)/2]$，我们需要对这三个数排序，位于中间的数作为基准。

源代码：

```
int QuickSort::Partition(int p, int r, string base)
{
    int x, i, mid;
    if (base == "middle")
    {
        mid = p + ((r - p) >> 1);
        if (Arr[mid] > Arr[p])
        {
            swap(Arr[mid], Arr[p]);
        }
        if (Arr[r] > Arr[p])
        {
            swap(Arr[r], Arr[p]);
        }
        if (Arr[mid] > Arr[r])
        {
            swap(Arr[mid], Arr[r]);
        }
    }
    x = Arr[r];
    i = p - 1;
    for (int j = p; j < r; j++)
    {
```

```
24          if (Arr[j] <= x)
25          {
26              i += 1;
27              swap(Arr[i], Arr[j]);
28          }
29      }
30      swap(Arr[i + 1], Arr[r]);
31      return i + 1;
32  }
```

3.2 在几乎有序时进行插入排序

教材中习题 7.4-5 告诉我们，当输入数据已经“几乎有序”时，使用插入排序速度很快，我们可以利用这一特点来提高快速排序的速度。当对一个长度小于 k 的子数组调用快速排序时，让它不做任何排序就返回。上层的快速排序调用返回后，对整个数组运行插入排序。

源代码：

```
1  void QuickSort::Sort(int p, int r, string base = "fixed", string
2  optimization = "normal")
3  {
4      int q;
5      if (r - p < 15)
6      {
7          (*this).InsertSort(p, r);
8          return;
9      }
10
11     if (p < r)
12     {
13         q = (*this).Partition(p, r, base);
14         (*this).Sort(p, q - 1, base);
15         (*this).Sort(q + 1, r, base);
16     }
17     return;
18 }
```

3.3 聚集元素

在一次分割结束后，将与本次基准相等的元素聚集在一起，再分割时，不再对聚集过的元素进行分割。

源代码：

```
void QuickSort::GatherPartition(int p, int r)
{
    int i = p, x, k;
    for (int j = p; j < r; j++)
    {
        if (Arr[j] == Arr[r])
        {
            swap(Arr[j], Arr[i]);
            i++;
        }
    }
    swap(i, p);
    x = Arr[r];
    k = p - 1;
    for (int j = p; j < r; j++)
    {
        if (Arr[j] <= x)
        {
            k += 1;
            swap(Arr[k], Arr[j]);
        }
    }
    swap(Arr[k + 1], Arr[r]);
    for (int j = i; j < p; j++)
    {
        swap(Arr[j], Arr[k - (j - i)]);
    }
    P.begin = i;
    P.leftMid = k - (p - i);
    P.rightMid = k + 2;
    P.end = r;
    return;
}
```

Part 4: 实验结果分析

由于在本地速度过快，均在 5ms 左右，差异不明显，本次选择在 vlab 提供的虚拟机上运行算法，进行对比.

为了排除机器状态对实验结果的影响，本次实验对于数据进行了 50 轮排序，得到如下结果：

表 1: Characteristics of the buck converter

算法	50 **次排序时间** (ms)	**平均时间** (ms)
固定基准	2801.5	56.03
随机基准	8659.36	173.19
三数取中	2692.11	53.84
三数取中 + 插入排序	1984.67	39.69
三数取中 + 插入排序 + 聚集	172.664	3.45
c++ algorithm 中的 sort	2189.87	43.80
c++ algorithm 中的 sort_heap	5469.78 ms	109.40

通过以上对比，我们可以明显观察到优化带来的效率提高，尤其是**三数取中 + 插入排序 + 聚集**，效率有惊人的提高（震惊）。在虚拟机上也仅仅耗费了 3.45ms 就完成了对于 100000 个数据的排序，于是我在本机上又运行了一次，时间达到了惊人的 0.99ms!

Part 5 实验总结

本次实验完成了快速排序算法的实现及其优化。在实验过程中获得了以下收获：

- 感受到了当数据量巨大时不同算法的效率
- 对于快速排序的理解更加深刻，学会了几种优化算法
- 体会到了学习算法与实践的重要性