算法导论课程
第一次上机实验报告

班级：1413014
姓名：乔新文
学号：14130140393

2017.3.26

Contents

1 综述

本文档将阐述《算法导论》上机实验代码的详细设计及实现。

本次上机实验所用代码均为 F# 代码，主要算法和辅助定义均包含在 AlgorithmLib.fs 所定义的 AlgorithmLib 模块中。

算法测试驱动部分在 Program.fs 中，同时整个程序的入口也在 Program.fs 中。

本次实验所有代码均运行在.Net Core 上，project.json 和 project.lock.json 为项目配置文件，可在安装有.Net Core 的环境中在项目文件下使用 dotnet run 命令运行。

2 题目一

2.1 题目

Describe a Θ(n lg n)-time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. (Implement exercise 2.3-7.)

2.2 实现思路

首先对集合 S 进行排序，然后遍历有序集合 S 中的元素，使其在有序集合 S 中对所有的

$$x - S[i]$$

进行二分查找。

2.3 实现代码

排序算法选择快速排序。

```
let Partition (A: int []) p r =
   let x = A.[r]
   let mutable i = p - 1
   for j = p to r - 1 do
      if A.[j] < x then
         i <- i + 1
         &A.[i] <-> &A.[j]
   &A.[i + 1] <-> &A.[r]
   i + 1
let rec QuickSort (A:int []) p r =
   if p < r then
      let q = Partition A p r
      QuickSort A p (q - 1)
      QuickSort A (q + 1) r
```

也可以使用更为简练的链表快速排序算法

```
let rec qsort (A:int list) =
   match A with
      | [] -> []
      | [a] -> [a]
      | head :: tail ->
```

```
        let smaller = List.filter ( (>=) head) tail
        let larger = List.filter ( (<) head) tail
        (qsort smaller) @ [head] @ (qsort larger)
```

二分查找算法如下

```
let rec HalfSearch (A: 'T []) (a: 'T) =
   match A.Length with
      | 1 -> (A.[0] = a, 0)
      | 0 -> (false, 0)
      | _ ->
         let mid = A.Length / 2
         if a = A.[mid] then (true,mid)
         elif a < A.[mid] then HalfSearch A.[0 .. mid - 1] a
         else HalfSearch A.[mid + 1 .. A.Length - 1] a
```

最终的总体算法如下

```
let SumOfTwoNumber (A:int []) (a:int) =
   QuickSort A 0 (A.Length - 1)
   let mutable i = 0
   let mutable result = (false, 0)
   while i < A.Length && not (fst result) do
      result <- HalfSearch A (a - A.[i])
      i <- i + 1
   (result,A.[i],a - A.[i])
```

2.4 算法分析

- 快速排序算法的时间复杂度为

$$O(nlgn)$$

- 折半查找的的时间复杂度为

$$O(lgn)$$

- 则算法总体的时间复杂度为

$$O(nlgn) + n * O(lgn) = O(nlgn)$$

3 题目二

3.1 题目

Implement priority queue.

3.2 实现思路

优先级队列的实现建立在大顶堆上，故需要先实现大顶堆的 MaxHeapify , BulidHeap , HeapSort 算法。

使用数组来保存队列，优先级队列的 HeapMaximum , HeapExtractMax , HeapIncreaseKey , MaxHeapInsert 操作作为优先级队列对象的实例方法。

3.3 实现代码

堆操作使用的辅助函数。

```
let ParentNode i =
   (i + 1 ) / 2 - 1

let LeftNode i =
   i * 2 + 1

let RightNode i =
   i * 2 + 2
```

自定义全局运算符 <-> 用来交换两个相同对象的值

```
let inline (<->) (x:'T byref) (y:'T byref) =
   let temp = x
   x <- y
   y <- temp
   ()
```

堆的 MaxHeapify 操作如下

```
let rec MaxHeapify (A: int []) i heapsize =
   let l = LeftNode i
   let r = RightNode i
   let mutable largest = 0
   if l < heapsize && A.[l] > A.[i] then largest <- l
   else largest <- i
   if r < heapsize && A.[r] > A.[largest] then largest <- r
   if largest <> i then
      &A.[i] <-> &A.[largest]
      MaxHeapify A largest heapsize
```

堆的 BulidHeap 操作如下

```
let BulidHeap (A:int []) =
   for i = A.Length / 2 downto 0 do MaxHeapify A i A.Length
```

堆的 HeapSort 操作如下

```
let HeapSort (A:int []) =
   BulidHeap A
   for i = A.Length - 1 downto 1 do
      &A.[0] <-> &A.[i]
      MaxHeapify A 0 i
```

最终的优先级队列对象定义如下

```
type PriorityQuene(A:int []) =
   let mutable quene = Array.empty<int>
   do
      quene <- A.[0 .. A.Length - 1]
      BulidHeap quene
   member this.Quene
      with get() = quene
   member this.Item
      with get(index) = quene.[index]
      and set index value = quene.[index] <- value
   member this.HeapMaximum () =
      if quene.Length = 0 then raise (System.Exception ("empty quene"))
```

```
            quene.[0]
        member this.HeapExtractMax () =
            if quene.Length = 0 then raise (System.Exception ("empty␣quene"))
            let max = quene.[0]
            quene <- quene.[1 .. quene.Length - 1]
            MaxHeapify quene 0 quene.Length
            max
        member this.HeapIncreaseKey i k =
            let mutable j = i
            if k < quene.[j] then raise (System.Exception ("new␣k␣is␣smaller␣than␣curren␣key"))
            quene.[j] <- k
            while j > 0 && quene.[ParentNode j] < quene.[j] do
                &quene.[j] <-> &quene.[ParentNode j]
                j <- ParentNode j
        member this.MaxHeapInsert key =
            quene <- Array.append quene [|0|]
            this.HeapIncreaseKey (quene.Length - 1) key
```

3.4 算法分析

- 堆的 MaxHeapify 操作的时间复杂度为
$$O(lgn)$$
- 堆的 BulidHeap 操作的时间复杂度为
$$O(n)$$
- 堆的 HeapSort 操作的时间复杂度为
$$O(nlgn)$$
- 优先级队列的 HeapMaximum 操作的时间复杂度为
$$O(1)$$
- 优先级队列的 HeapExtractMax 操作的时间复杂度为
$$O(lgn)$$
- 优先级队列的 HeapIncreaseKey 操作的时间复杂度为
$$O(lgn)$$
- 优先级队列的 MaxHeapInsert 操作的时间复杂度为
$$O(lgn)$$
- 由以上可得到优先级队列的所有操作都可以在
$$O(lgn)$$
的时间内完成

4 题目三

4.1 题目

Implement Quicksort and Randomized Quicksort. Answer the following questions.
(1) How many comparisons will Quicksort do on a list of n elements that all have the same value?
(2) What are the maximum and minimum number of comparisons will Quicksort do on a list of n elements, give an instance for maximum and minimum case respectively.

4.2 实现思路

使用分治递归的思想来实现快速排序。
就地排序的快速排序算法中，使用数组来储存数据以提高效率，并使用最后一个元素作为哨兵元素。
非就地排序的快速排序算法中，使用链表来储存数据以让代码更加简练，并使用第一个元素作为哨兵元素。
随机化快速排序中，哨兵元素的选择是随机的。

4.3 实现代码

全局的随机数生成对象。

```
let random = System.Random()
```

就地排序的快速排序。

```
let Partition (A: int []) p r =
    let x = A.[r]
    let mutable i = p - 1
    for j = p to r - 1 do
        if A.[j] < x then
            i <- i + 1
            &A.[i] <-> &A.[j]
    &A.[i + 1] <-> &A.[r]
    i + 1
let rec QuickSort (A:int []) p r =
    if p < r then
        let q = Partition A p r
        QuickSort A p (q - 1)
        QuickSort A (q + 1) r
```

随机化的就地快速排序

```
let RandomizePartition (A:int []) p r =
    let i = random.Next (p, r)
    &A.[r] <-> &A.[i]
    Partition A p r
let rec RandomizeQuickSort (A:int []) p r =
    if p < r then
        let q = RandomizePartition A p r
        RandomizeQuickSort A p (q - 1)
        RandomizeQuickSort A (q + 1) r
```

非就地排序的快速排序。

```
let rec qsort (A:int list) =
    match A with
```

```
        | [] -> []
        | [_] -> A
        | head :: tail ->
          let smaller = List.filter ( (>=) head) tail
          let larger = List.filter ( (<) head) tail
          (qsort smaller) @ [head] @ (qsort larger)
```

随机化的非就地快速排序

```
let rec rqsort (A:int list) =
    match A with
    | [] -> []
    | [_] -> A
    | _ ->
      let rmidnum = random.Next (0, A.Length)
      let temp = A.[0 .. rmidnum - 1] @ A.[rmidnum + 1 .. A.Length - 1] //temp is A
          - [A.[rmidnum]]
      let smaller = List.filter ( (>=) A.[rmidnum]) temp
      let larger = List.filter ( (<) A.[rmidnum]) temp
      (rqsort smaller) @ [A.[rmidnum]] @ (rqsort larger)
```

4.4 算法分析

- 当所有元素的值都相同时，哨兵元素需要与剩下的所有元素进行比较，且划分成一个空列表和一个长度当前列表长度 -1 的列表，故需要进行

$$n^2$$

次比较。

- 当需要进行最大次数的比较时，快排陷入最坏情况，即所有元素均相同或每次取到的哨兵元素为当前列表最大值或最小值，也就是每次划分的列表都有一个是空列表。此时比较次数达到最大值，需要进行

$$n^2$$

次比较。

- 当需要进行最小次数的比较时，快排为最好情况，即每次的哨兵元素均为当前列表的中位数，也就是每次划分的两个子列表大小相同或大小相差为一。此时比较次数为最小值，需要进行

$$nlgn$$

次比较。

5 题目四

5.1 题目

Sorting in place in linear time.
Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:
1. The algorithm runs in O(n) time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to

the original array.
a. Give an algorithm that satisfies criteria 1 and 2 above.
b. Give an algorithm that satisfies criteria 1 and 3 above.
c. Give an algorithm that satisfies criteria 2 and 3 above.
d. Can you use any of your sorting algorithms from parts (a)–(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b-bit keys in O(bn) time? Explain how or why not.
e. Suppose that the n records have keys in the range from 1 to k. Show how to modify counting sort so that it sorts the records in place in O(n+k) time. You may use O(k) storage outside the input array. Is your algorithm stable? (Hint: How would you do it for k = 3?)

5.2 实现思路

首先对 int(System.Int32) 类型进行扩展，扩展出一个 Tag 方法，该方法接受一个整形参数，返回该 int 型数的某一位上的位数。(例如参数为 0 时返回个位上的数，参数为 1 时返回十位上的数)。
在排序算法中使用基数排序，基数排序使用稳定的桶排序，而桶排序中使用稳定的插入排序。

5.3 实现代码

对 int(System.Int32) 类型进行扩展。

```
type System.Int32 with
   member this.Tag (x:int) =
      let temp = int (10.0**float x)
      this / temp % 10
```

插入排序算法

```
let rec InsertSort (A:int list) (s:int)=
   match A with
      | [] -> [s]
      | [a] ->
         if s < a then [s] @ [a]
         else [a] @ [s]
      | head :: tail ->
         if s < head then [s] @ A
         else [head] @ (InsertSort tail s)
```

桶排序算法

```
let RadixSort (A:int []) (x: int)=
   let bucket = Array.create 10 List.empty<int>
   for i = 0 to A.Length - 1 do
      let temp = int (A.[i].Tag x)
      bucket.[temp] <- InsertSort bucket.[temp] A.[i]
   let mutable resurt = List.empty<int>
   for i = 0 to bucket.Length - 1 do
      resurt <- resurt @ bucket.[i]
   for i = 0 to A.Length - 1 do
      A.[i] <- resurt.[i]
   ()
```

基数排序算法

```
let BuckerSort (A:int []) (x: int)=
   for i = 0 to x do
```

```
        RadixSort A i
    ()
```

5.4 算法分析

- 基数排序算法的时间复杂度为

$$O(n)$$

- 桶排序是稳定的排序算法，故保证了外层的基数排序也是稳定的。

- 而且对整数进行桶排序，只需准备 10 个桶，空间复杂度为

$$O(n)$$

  即该算法是就地排序。

- 综上所述，该排序算法同时满足 1、2、3 条件。