

# 程序设计语言认知实验

姓 名： 高 月

学 号： 1120141944

班 级： 07121402

指导老师： 计 卫 星

## 目录

一、实验目的 .....	1
二、实验内容 .....	1
三、实验环境 .....	1
3.1. 硬件信息 .....	1
3.2. 环境信息 .....	1
3.3. 各语言（编译器/解释器/虚拟机）版本 .....	2
四、实验步骤 .....	2
4.1 复习归并排序算法实现原理 .....	2
4.2 生成待排序的整型随机数 .....	2
4.3 编写各语言版本的归并排序代码 .....	3
4.3.1 C++ .....	3
4.3.2 Java .....	4
4.3.3 Python .....	5
4.3.4 Haskell .....	5
4.4 运行各语言版本程序并记录结果 .....	6
4.4.1 C++ .....	6
4.4.2 Java .....	6
4.4.3 Python .....	6
4.4.4 Haskell .....	6
3.4.5 记录程序运行时间 .....	7
五、实验结果 .....	8
5.1 实验效果截图 .....	8
5.1.1 生成指定数量的随机数 .....	8
5.1.2 C++ .....	8
5.1.3 Java .....	9
5.1.4 Python .....	9
5.1.5 Haskell .....	9
5.2 语言易用性及程序规模对比 .....	10
5.2.1 语言易用性 .....	10
5.2.2 程序规模对比 .....	10
5.3 程序运行性能对比 .....	11
六、实验心得 .....	12

## 一、实验目的

了解程序设计语言的发展历史,了解不同程序设计语言的各自特点;感受编译执行和解释执行两种不同的执行方式,初步体验语言对编译器设计的影响,为后续编译程序的设计和开发奠定良好的基础。

## 二、实验内容

分别使用 C++、Java、Python 和 Haskell 实现归并排序,对采用这几种语言实现的编程效率,程序的规模,程序的运行效率进行对比分析。

## 三、实验环境

### 3.1. 硬件信息

本次实验电脑硬件信息如下表所示:

电脑型号	MacBook Pro (Retina, 13-inch, Mid 2014)
CPU 型号及主频	Intel Core i5 2.6 GHz
CPU 核心数	2
L2 Cache (per Core)	256 KB
L3 Cache	3 MB
内存	8 GB 1600 MHz DDR3
磁盘型号	APPLE SSD SM0256F

### 3.2. 环境信息

本次实验系统环境等信息如下表所示

OS	Editor	Shell
macOS Sierra 10.12.3	Vim 8	Bash 3.2

### 3.3. 各语言（编译器/解释器/虚拟机）版本

本次实验的四种语言版本信息如下表所示：

语言	版本
C++	GCC 4.2.1
Java	Java 8 Update 121
Python	Python 3.6.0
Haskell	GHC 8.0.2

## 四、实验步骤

### 4.1 复习归并排序算法实现原理

归并排序算法是基于归并操作的一种有效的排序算法，主要思想是分治法。归并排序有多路归并排序、两路归并排序，可用于内排序，也可以用于外排序。实现方法有两种：迭代法、递归法。本次实验采用递归法实现两路归并排序。本次实验中部分归并排序算法代码的优化参考了维基百科（<https://zh.wikipedia.org/wiki/归并排序>）中的实现，如 Python 语言中 deque 的使用，简化了 list 的操作。

### 4.2 生成待排序的整型随机数

为便于对比分析各语言程序的性能，本次实验采用 C++ 编写随机生成指定数量整型数据的程序 nums\_generator.cpp。循环中每生成一个随机数就写到文件 not\_sorted.txt 中（其中每个随机数范围在  $[0, \text{RAND\_MAX}]$ ， $\text{RAND\_MAX} = 2^{31}-1 = 2147483647$ ）一共排序程序使用。程序代码主要部分如下图所示：

```
16 // File io
17 fstream fs;
18 fs.open("not_sorted.txt", fstream::out);
19
20 // Prepare rand
21 srand(time(0));
22
23 // Get array length
24 int n = 0;
25 cout << "Please input the number of random numbers: \n";
26 cin >> n;
27
28 // Prompt user random value range
29 cout << "Random value on [0 " << RAND_MAX << "].\n";
30
31 int random_variable = 0;
32 for (int i = 0; i < n; i++) {
33     // Generate random number and append to file
34     random_variable = rand();
35     fs << random_variable << endl;
36 }
```

## 4.3 编写各语言版本的归并排序代码

本次实验中实现归并排序整型数组,数据的输入输出均采用文件操作,防止手动输入数据对实验结果造成影响。

### 4.3.1 C++

归并排序算法代码如下所示：

```
7 void merge_sort_recursive(int arr[], int result[], int start, int end){
8     if (start >= end)
9         return;
10    int len = end - start, mid = (len >> 1) + start;
11    int start1 = start, end1 = mid;
12    int start2 = mid + 1, end2 = end;
13    merge_sort_recursive(arr, result, start1, end1);
14    merge_sort_recursive(arr, result, start2, end2);
15    int ptr = start;
16    while (start1 <= end1 && start2 <= end2)
17        result[ptr++] = arr[start1] < arr[start2] ? arr[start1++] : arr[start2++];
18    while (start1 <= end1)
19        result[ptr++] = arr[start1++];
20    while (start2 <= end2)
21        result[ptr++] = arr[start2++];
22    for (int i = start; i <= end; i++)
23        arr[i] = result[i];
24 }
25
26 void merge_sort(int arr[], const int length) {
27     int result[length];
28     merge_sort_recursive(arr, result, 0, length-1);
29 }
```

文件操作代码如下所示，排序好的数据保存在 sorted\_cpp.txt 中：

```
39 // File io
40 fstream ifs, ofs;
41 ifs.open("not_sorted.txt", fstream::in);
42 ofs.open("sorted_cpp.txt", fstream::out);
43
44 // Read file
45 int num = 0, len = 0;
46 int arr[MAX_LENGTH];
47 while (ifs >> num) {
48     arr[len++] = num;
49 }
50
51 // Prompt user
52 cout << "Sorting...\n";
53
54 // Merge sort
55 merge_sort(arr, len);
56
57 // Write sorted data to file
58 for (int i = 0; i < len; i++) {
59     ofs << arr[i] << endl;
60 }
61
62 // Prompt user
63 cout << "Done!\n";
```

### 4.3.2 Java

归并排序算法代码如下:

```
static void mergeSortRecursive(int[] arr, int[] result, int start, int end) {
    if (start >= end)
        return;
    int len = end - start, mid = (len >> 1) + start;
    int start1 = start, end1 = mid;
    int start2 = mid + 1, end2 = end;
    mergeSortRecursive(arr, result, start1, end1);
    mergeSortRecursive(arr, result, start2, end2);
    int ptr = start;
    while (start1 <= end1 && start2 <= end2)
        result[ptr++] = arr[start1] < arr[start2] ? arr[start1++] : arr[start2++];
    while (start1 <= end1)
        result[ptr++] = arr[start1++];
    while (start2 <= end2)
        result[ptr++] = arr[start2++];
    for (int i = start; i <= end; i++)
        arr[i] = result[i];
}

public static void mergeSort(int[] arr, final int length) {
    int[] result = new int[length];
    mergeSortRecursive(arr, result, 0, length-1);
}
```

文件操作代码如下图所示, 排序好的数据保存在 sorted\_java.txt 中:

```
43      // Read data from file
44      try {
45          br = new BufferedReader(new FileReader("not_sorted.txt"));
46          String line;
47          while ((line = br.readLine()) != null) {
48              num = Integer.parseInt(line);
49              arr[len++] = num;
50          }
51      } catch (Exception e) {
52          e.printStackTrace();
53      }
54
55      // Prompt user
56      System.out.println("Sorting...");
57
58      // Merge sort
59      mergeSort(arr, len);
60
61      // Write sorted data to file
62      try {
63          pw = new PrintWriter(new FileWriter("sorted_java.txt"));
64          for (int i = 0; i < len; i++) {
65              pw.println(arr[i]);
66          }
67      } catch (Exception e) {
68          e.printStackTrace();
```

### 4.3.3 Python

归并排序算法代码如下：

```
8 def merge(left, right):
9     merged, left, right = deque(), deque(left), deque(right)
10    while left and right:
11        merged.append(left.popleft() if left[0] <= right[0] else right.popleft())
12        print(merged[-1])
13    merged.extend(right if right else left)
14    return list(merged)
15
16 mid = int(len(lst) // 2)
17 left = merge_sort(lst[:mid])
18 right = merge_sort(lst[mid:])
19 return merge(left, right)
```

文件操作代码如下图所示，排序好的数据保存在 sorted\_python.txt 中：

```
22 with open('not_sorted.txt') as inf:
23     nums = inf.readlines()
24     lst = [int(x.strip()) for x in nums]
25
26 # Prompt user
27 print("Sorting...")
28
29 # Merge Sort
30 sorted_lst = merge_sort(lst)
31
32 with open('sorted_python.txt', 'w+') as outf:
33     for x in sorted_lst:
34         outf.write(str(x) + '\n')
35
36 # Prompt user
37 print("Done!")
```

### 4.3.4 Haskell

归并排序算法代码如下：

```
1 mergesort'merge :: (Ord a) => [a] -> [a] -> [a]
2 mergesort'merge [] xs = xs
3 mergesort'merge xs [] = xs
4 mergesort'merge (x:xs) (y:ys)
5   | (x < y) = x:mergesort'merge xs (y:ys)
6   | otherwise = y:mergesort'merge (x:xs) ys
7
8 mergesort'splitinhalf :: [a] -> ([a], [a])
9 mergesort'splitinhalf xs = (take n xs, drop n xs)
10    where n = (length xs) `div` 2
11
12 mergesort :: (Ord a) => [a] -> [a]
13 mergesort xs
14   | (length xs) > 1 = mergesort'merge (mergesort ls) (mergesort rs)
15   | otherwise = xs
16    where (ls, rs) = mergesort'splitinhalf xs
```

文件操作代码如下图所示，排序好的数据保存在 sorted\_haskell.txt 中：

```
18 readLines :: FilePath -> IO [String]
19 readLines = fmap lines . readFile
20 makeInteger :: [String] -> [Int]
21 makeInteger = map read
22 makeString :: [Int] -> [String]
23 makeString = map show
24
25 main :: IO ()
26 main = do
27     contents <- readLines "not_sorted.txt"
28     let lst = makeInteger contents
29     putStrLn "Sorting..."
30     let sorted_lst = mergesort lst
31
32     let outFileName = "sorted_haskell.txt"
33     writeFile outFileName $ unlines (makeString sorted_lst)
34     putStrLn "Done!"
```

## 4.4 运行各语言版本程序并记录结果

### 4.4.1 C++

本次实验中采用 gcc 编译链接 .cpp 程序源文件，并执行得到的 merge\_sort\_cpp\_bin 可执行文件，在 bash 中命令如下：

```
$ gcc -std=c++11 merge_sort.cpp -o merge_sort_cpp_bin
$ ./merge_sort_cpp_bin
```

### 4.4.2 Java

本次实验中使用 javac 编译 .java 程序源文件，并用 java 执行程序，在 bash 中命令如下：

```
$ javac MergeSort.java
$ java MergeSort
```

### 4.4.3 Python

本次实验中使用 python3 执行 .py 程序源文件，在 bash 中命令如下：

```
$ python3 merge_sort.py
```

### 4.4.4 Haskell

本次实验中使用 runhaskell 解释执行 .hs 程序源文件，在 bash 中命令如下：

```
$ runhaskell merge_sort.hs
```

另外 Haskell 也可以采用 ghc --make 的方式编译链接得到可执行程序，在 bash



中命令如下：

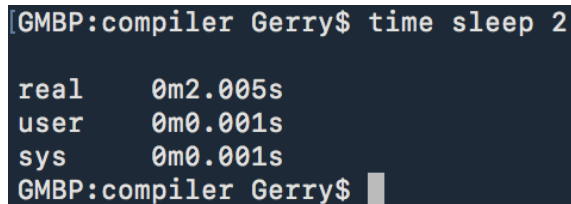
```
$ ghc --make merge_sort.hs -o merge_sort_haskell_bin  
$ ./merge_sort_haskell_bin
```

### 3.4.5 记录程序运行时间

本次实验中为了更好地监控程序(进程),体现程序间差异,使用系统命令 `time` 记录进程运行时间,而不是在程序中调用时间函数来记录时间。另外本次实验中所记录的时间包含文件 IO 的时间,由于实验主要目的是考察各语言的性能等差异、每个程序所读入的数据都是一样的、磁盘也一致,故此方法记录时间较好。各语言版本程序在 `bash` 中分别使用如下命令：

```
$ time merge_sort_cpp  
$ time java MergeSort  
$ time python3 merge_sort.py  
$ time runhaskell merge_sort.hs  
$ time merge_sort_haskell_bin
```

`time` 指令输出如下图所示：



```
GMBP:compiler Gerry$ time sleep 2  
  
real    0m2.005s  
user    0m0.001s  
sys     0m0.001s  
GMBP:compiler Gerry$
```

如上图所示：`real` `user` `sys` 三个值分别代表：进程时钟时间、用户 CPU 时间、系统 CPU 时间，本次实验中主要讨论各语言版本进程的时钟时间。

## 五、实验结果

### 5.1 实验效果截图

#### 5.1.1 生成指定数量的随机数

bash 中 nums\_generator 执行效果如下:

```
GMBP:compiler Gerry$ ./nums_generator
Please input the number of random numbers:
1000
Random value on [0 2147483647].
Complete random numbers generation!
GMBP:compiler Gerry$
```

保存到文件 not\_sorted.txt , 内容如下 ( 只截取了前 20 个随机数 ):

```
GMBP:compiler Gerry$ head -20 not_sorted.txt
1521134668
2065031188
1495957549
1967470614
325412992
1722791282
431064073
1431533580
1525581719
1644689700
2037767363
686867585
1458898470
1885787491
1866698811
1018317454
1564266435
1131166471
1989634853
1325106934
GMBP:compiler Gerry$
```

#### 5.1.2 C++

在 bash 中 C++ 版程序执行效果如下:

```
GMBP:compiler Gerry$ time merge_sort_cpp_bin
CPP
Sorting...
Done!

real    0m0.010s
user    0m0.002s
sys     0m0.003s
GMBP:compiler Gerry$
```

### 5.1.3 Java

在 bash 中 Java 版程序执行效果如下：

```
[GMBP:compiler Gerry$ time java MergeSort
Java
Sorting...
Done!

real    0m0.127s
user    0m0.112s
sys     0m0.027s
GMBP:compiler Gerry$
```

### 5.1.4 Python

在 bash 中 Python 版程序执行效果如下：

```
[GMBP:compiler Gerry$ time python3 merge_sort.py
Python
Sorting...
Done!

real    0m0.124s
user    0m0.053s
sys     0m0.018s
GMBP:compiler Gerry$
```

### 5.1.5 Haskell

在 bash 中 Haskell 版程序解释执行效果如下：

```
[GMBP:compiler Gerry$ time runhaskell merge_sort.hs
Haskell
Sorting...
Done!

real    0m1.422s
user    0m0.303s
sys     0m0.267s
GMBP:compiler Gerry$
```

在 bash 中 Haskell 版程序编译执行效果如下：

```
[GMBP:compiler Gerry$ time merge_sort_haskell_bin
Haskell
Sorting...
Done!

real    0m0.023s
user    0m0.011s
sys     0m0.009s
GMBP:compiler Gerry$
```

## 5.2 语言易用性及程序规模对比

本次实验中采用打分的方法对比分析这四种语言的易用性和程序规模（打分具有个人主观性，分数 1-10，10 分最好），其中程序规模按代码行数对比（仅去除注释及空行，包含归并排序算法，文件操作，错误处理等）。

### 5.2.1 语言易用性

易用性分程序实现容易度（占 70%）、代码可读性（占 30%）两方面评分，结果如下表

语言	程序实现容易度	代码可读性	总分
C++	7	8	7.3
Java	6	8	6.6
Python	8	9	8.3
Haskell	4	5	4.3

由表可见，Python 易用性最高，Haskell 易用性最差，C++易用性相对较好，Java 的错误处理比较麻烦但是程序安全性很好。

### 5.2.2 程序规模对比

统计结果如下表：

语言	代码行数
C++	46
Java	67
Python	28
Haskell	33

由表可见，Python 程序规模最优不到 Java 的一半，Haskell 程序规模比较小几乎为 Java 的一半；而 C++ 程序规模一般；Java 的程序规模最大。

综合上面两方面可得到，解释执行的 Python 在语言易用性和程序规模都有很大的优势。

### 5.3 程序运行性能对比

使用 time 命令记录进程运行时间，运行 7 次不同的 1,000 个数据的排序，和 5 次不同的 1,000,000 个数据的排序。

1,000 个数据排序结果如下表所示(单位为 s , Haskell(i)表示解释执行，Haskell(c)表示编译执行)：

语言	7 次 1,000 个数据排序耗时							平均值
C++	0.009	0.010	0.009	0.009	0.008	0.007	0.009	0.009
Java	0.365	0.158	0.343	0.130	0.127	0.132	0.130	0.198
Python	0.120	0.076	0.109	0.055	0.054	0.055	0.054	0.075
Haskell(i)	1.296	0.387	1.185	0.372	0.366	0.374	0.374	0.622
Haskell(c)	0.031	0.016	0.028	0.016	0.019	0.022	0.024	0.022

1,000,000 个数据排序结果如下表所示(单位为 s , Haskell(i)表示解释执行，Haskell(c)表示编译执行)：

语言	5 次 1,000,000 个数据排序耗时						平均值
C++	2.819	2.822	2.805	2.639	2.661		2.749
Java	0.722	0.915	0.931	0.790	0.935		0.859
Python	10.908	10.870	10.727	10.884	11.070		10.892
Haskell(i)	22.986	23.515	23.977	24.309	24.026		23.763
Haskell(c)	10.587	9.889	9.532	9.941	9.951		9.980

由上表可见，在少量数据时，编译执行的 C++ 有明显的优势。而在大量数据时，编译解释混合的 Java 居然出奇的快。主要是解释执行的 Python 在不论数据量大小的情况下均比较慢：比 C++ 慢 5-10 倍，勉强可以接受。而 Haskell 在解释执行编译执行两种情况下速度差异较大：编译执行比解释执行快 2-10 倍、在小数据量情况下仅比 C++ 慢、大数据量时略微比 Python 快。而解释执行时不论是大数据量或大数据量都是最慢的，大数据量时甚至比 Java 慢 20 多倍。

由此可见，编译执行在性能上一般比解释执行好，但具体情况的看编译器解释器的优化程度以及实际的应用情况，不能一概而论。

## 六、实验心得

本次实验中我了解了程序设计语言的发展历史：C/C++ 语言在这四种语言中最早被开发，且一直受工业学术界的推崇；Python Haskell 几乎同时诞生，但是现在的发展趋势却是大相径庭：Python 越来越流行，其性能也比 Haskell 好很多，但是 Haskell 的函数式编程思想在新开发的语言中越来越被认可。

了解了不同程序设计语言的各自特点：解释执行的语言普遍代码规模比较小，也比较易用（Haskell 除外）。

感受到了编译执行和解释执行两种不同的执行方式：编译执行时不需要源代码执行，而解释执行不需要；解释执行的语言调试起来也比较方便，不需要反复编译，修改源代码即可。

体会到了语言对编译器设计的影响。不同语言对编译器的设计影响很大：解释执行的语言需要解释器，编译执行的语言需要编译器。编译器的设计对语言性能的影响也很大：同样是解释执行或编译执行，可能由于编译器/解释器的设计导致性能差异巨大。