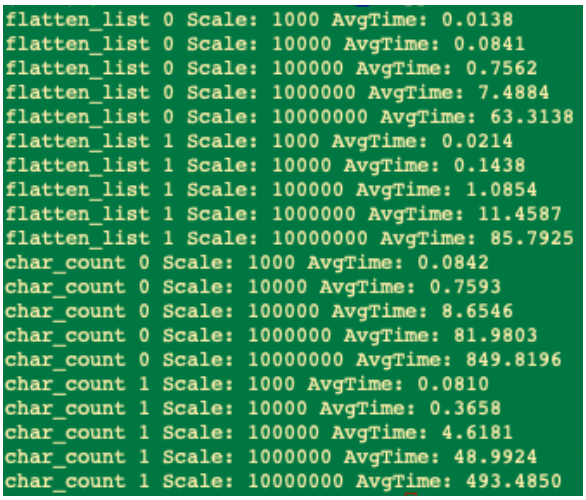
Task 1 报告

1 实验环境

使用的CPU为Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz、操作系统为Linux 5.15.0-117-generic、使用的Python版本为Python 3.9.18。

2 实验数据

使用`time.perf_counter()`测量被测量函数运行时间，被测量函数分别为`flatten_list`和`char_count`，每个函数有两种实现，分别在1e3, 1e4, 1e5, 1e6, 1e7这五个数据规模上进行测试，每个实现在每个数据规模上的测试会被重复10次，最终取平均值。测试时使用的脚本为efficiency_test.py，在repo中task_1目录下，运行时的输出如下图。



2.1 `flatten_list`

第一种实现使用了Python list自带的`extend()`函数。

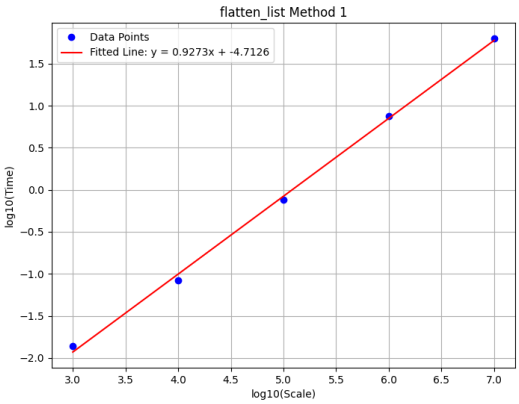
```
def flatten_list(nested_list: list):
    result_list = []
    for lst in nested_list:
        result_list.extend(lst)
    return result_list
```

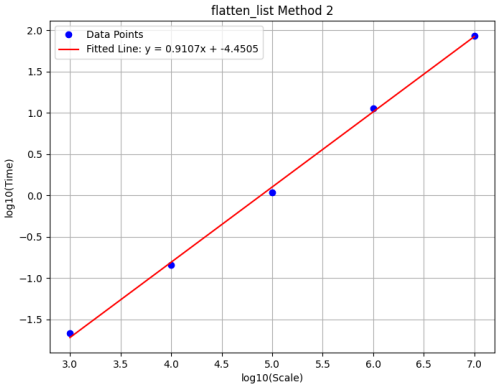
第二种实现使用了`itertools.chain()`函数。

```
def flatten_list(nested_list: list):
    return list(itertools.chain(*nested_list))
```

实验数据如下（时间单位为毫秒）：

实现	1e3耗时	1e4耗时	1e5耗时	1e6耗时	1e7耗时
第一种	0.0138	0.0841	0.7562	7.4884	63.3138
第二种	0.0214	0.1438	1.0854	11.4587	85.7925
第二种/第一种	1.55	1.71	1.44	1.53	1.36





可以看到第一种实现明显比第二种实现要快一些，两者所耗费的时间随着数据规模的增大基本上等比例增大，数据规模每变为原来10倍，第一种方法耗时变为原来约8.46倍、第二种方法耗时变为原来约8.14倍。

2.2 `char_count`

第一种实现直接使用一个dict对字符串中的字符进行计数：

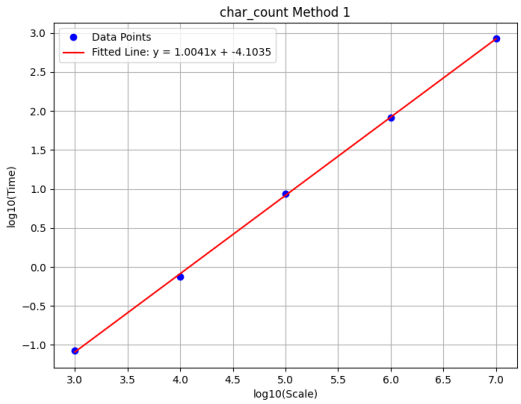
```
def char_count(s: str):
    result_dict = {}
    for char in s:
        if char in result_dict:
            result_dict[char] += 1
        else:
            result_dict[char] = 1
    return result_dict
```

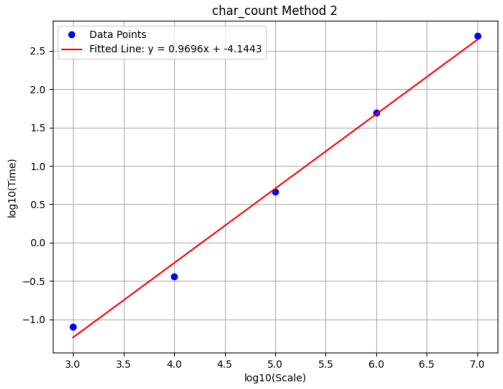
第二种实现使用了`collections`中的`Counter`：

```
def char_count(s: str):
    return dict(Counter(s))
```

实验数据如下（时间单位为毫秒）：

实现	1e3耗时	1e4耗时	1e5耗时	1e6耗时	1e7耗时
第一种	0.0842	0.7593	8.6546	81.9803	849.8196
第二种	0.0810	0.3658	4.6181	48.9924	293.4850
第一种/第二种	1.03	2.08	1.87	1.67	2.90





可以看到第二种实现明显比第一种实现要快一些，尤其是在大规模的数据上。两者所耗费的时间随着数据规模的增大基本上等比例增大，数据规模每变为原来10倍，第一种方法耗时变为原来约10.09倍、第二种方法耗时变为原来约9.32倍。

3 改进方法

不难发现上面的这两个函数执行的任务实际上可以被分成独立的子任务，比如在flatten_list中将nested_list拆分为两半分别合并之后再将两个的结果进行合并也是可以的，在char_count中也可以将字符串分为两半后分别计数最后再把得到的两个dict进行合并。

因此，在规模较大时可以创建多个子进程处理独立的子任务，最后再将结果合并，这样或许可以更快，不过需要考虑建立新进程和最后合并的开销。