Experiment 4

BY YANGZHE PB17000025

1 基站放置

1.1 Description

现有 N 个村庄排成一排，在这些村庄里需要挑选若干村庄搭建基站，其中基站的覆盖半径为 R。假设在村庄 i 处放置基站，村庄 i 的坐标为 X_i，则处在 $[X_i - R, X_i + R]$ 范围内的村庄都能被该基站所覆盖。在给出所有村庄坐标的情况下，求使所有村庄都能被基站覆盖所需要的最少的基站个数。

1.2 Input Description

第一行包含两个整数 N, R 分别表示村庄个数和基站覆盖半径。接下来的 N 行每行一个整数 X_i 表示村庄 i 的坐标。

数据规模：

$N \leqslant 100,000$

$R \leqslant 1,000$

$X_i \leqslant 1,000,000$

1.3 Output Description

输出一个整数表示最少需要搭建的基站数目。

1.4 Question Analysis

首先把N个村庄的位置，按照从小到大排序，相当于把它们放到数轴上。

根据贪心的想法，当我们放一个基站在村庄i上时，我们希望$[X_i - R, X_i + R]$这个区间的最左边，即点$X_i - R$，是当前所有没有被基站覆盖的村庄中，最左边村庄的位置。

所以我们用一个变量i记录上一次放基站时，最左边的没有覆盖到的村庄。

那么算法的开始，$i = 0$，因为第一个村庄还没被覆盖。然后我们在$[X_i, X_i + R]$中，找到最右边的村庄，记为j，X_j就是我们要放基站的位置。这是因为我们希望基站的所覆盖的范围内，i尽可能的出现在靠近左部边界的地方，那么右边就有更多的空间来覆盖其他的村庄，即$(X - R) - X_i$尽可能小，且$X - R \leqslant X_i$，$X \geqslant X_i$。转化一下，得到$X \leqslant R + X_i$且$X - (R + X_i)$尽可能小，也就等价于说，寻找$[X_i, X_i + R]$内，最右边的村庄X_j。

在找到基站放置的位置X_j后，我们要更新i，用一个新的循环变量k，从X_j开始，向右走R的距离，找到$[X_j, X_j + R]$之间最右边的村庄X_{k-1}，然后把k赋给i，完成i的更新。

如此循环往后，直到N个村庄都被覆盖时，结束。

具体形成代码，即如下形式：

```
int num=0;
for (int i = 0; i < N;) {
  int j,k;
  for (j = i; j<N&&Village[j] - Village[i] <=R; j++);
  for (k = j; k<N&&Village[k] - Village[j - 1] <= R; k++);
  i=k;
  num++;
```

```
}
```

AC：https://202.38.86.171/status/e8ab02db37a92de5bec47a29c641cfeb

1.5 复杂度分析

首先排序的时间复杂度为$N\log(N)$。

在for循环中，每次j从i开始，增加后赋值给k，然后k增加后再赋值给i。由于i最多增加到N，所以总共的执行次数为$O(N)$。

故总时间复杂度为$T(N)=O(N)+N\log(N)=N\log(N)$。

空间复杂度为$O(N)$，因为只开了一个大小为N的数组。

1.6 遇到的问题

1. 开始写的时候，没有注意到基站必须放在村庄上，每次就直接把基站的最左端置成X_i，然后算出基站的位置后再更新i，导致结果不正确。

2. 在计算j和k的时候，开始忘了加边界保护，导致跑到N后，非法访问数组。

   RE：https://202.38.86.171/status/a29f1eebd891b0274dd7d6b42c90a8dc

2 任务调度

2.1 Description

现有 N 个任务需要在一台机器上进行调度，其中每个任务有相应的到达时间 r_i 和执行时间 p_i。任务必须在到达之后才能进行执行，该机器同时只能执行一个任务。当一个任务在机器上执行 p_i 时间 后，该任务即视为完成。若该任务的完成时刻为 f_i，则该任务的响应时间为 f_i-r_i+1。我们需要找到一个调度方案使所有任务的响应时间之和最短。

需要说明的是，在这个问题里我们假设已经将时间划分为若干不可再分的时隙，即一个任务最少的执行时间为 1 个单位时间，且任务只会在整数时刻到达。此外，调度是可抢占的，这意味着在任意时隙开始时可以通过挂起正在执行的任务 i 转而执行另一项任务 j，随后可以恢复任务 i 并执行其剩余的工作。

2.2 Input Description

第一行一个整数 N 表示总共的任务个数。接下来的 N 行，每行两个整数 r_i, p_i 表示任务的到达时间和执行时间。

数据规模：

$N\leqslant 100,000$

$r_i\leqslant 10,000,000$

$p_i\leqslant 1,000$

2.3 Output Description

输出一个整数表示最小的所有任务的总响应时间。

2.4 Question Analysis

根据操作系统的课程，我们知道，可抢占式的SJF可以达到等待时间最优。

而本题定义的每个任务的响应时间$f_i - r_i + 1$，正是完成时间与到达时间之差，即等待时间。

首先，输入的任务不是按照到达时间排序的，我们要先以到达时间为排序变量，对整体数据进行排序。

考虑到可抢占式的算法，可能有多重抢占，这里使用递归实现，我们定义一个递归函数RunTask()，它接受index参数，调用这个函数表示执行该任务。

为了表示一个任务执行了多少，是否结束，我们给每个任务一个结构体类型：

```
typedef struct Task {
  int arrive_time;
  int continue_time;
  int remain_time;
} Task;
```

其中arrive_time和continue_time为输入的到达时间和执行时间。而remain_time则表示该任务剩余的执行时间，它被初始化为continue_time。

此外，我们还定义了一个全局变量global_time，来标识当前时间，这个变量主要用于嵌套抢占时，后面的任务到达的情况。

然后，当执行某个任务i的RunTask()时，一定说明该任务**目前**剩余时间最短，这代表两种情况：一种是它的确是剩余执行时间最短的，另一种情况是剩余时间更短的目前还没有到。

所有在处理某个任务的时候，我们有两种选择：一种是直接把该任务执行完，这种情况出现在就算该任务执行完了，下一个任务也不会到来，或者该任务是最后一个的情况，这时直接把remain_time加到global_time上，把remain_time归零即可：

```
if (next_task >= N || //no more tasks to arrive
    global_time + T[index].remain_time <= T[next_task].arrive_time) {
    // task can be exec directly
    global_time += T[index].remain_time;
    T[index].remain_time = 0;
    break;
}
```

而另一种是，执行到下一个任务的到达时间，比较当前任务和到达任务的剩余时间，如果当前任务短，继续执行，如果到达任务短，调用到达任务的RunTask()。这时应该更新global_time为下一个任务的到达是时间，并把remain_time减去两者的差值，即：

```
else{// have tasks to arrive
    T[index].remain_time -= T[next_task].arrive_time - global_time;
    global_time = T[next_task].arrive_time;
    }
    if (T[next_task].remain_time < T[index].remain_time) {
        RunTask(T, next_task, N);
        next_task++;
    } else {
        next_task++;
    }
```

此外，由于这是递归函数，可能调用其他的递归函数回来后，global_time已经很大了，后面的任务已经到达了，这个时候直接比较剩余时间，不需要更新global_time和remain_time。

```
else if(global_time>T[next_task].arrive_time)
{//just do nothing(this case is when come back from recursion)
}
```

为了保证任务执行结束，我们在外面套一层while()，循环判断条件为remain_time是否为0,当跳出循环时，表示任务结束，计算等待时间即可。

然后在`main()`函数内用一个循环,N个任务都执行`runTask()`,有些任务前面已经执行完了,`remain_time`为0，就会直接`return`。

2.5 复杂度分析

这个时间复杂度还挺复杂的，因为涉及到多重递归，但我们可以注意到，每一次循环，肯定会执行完一个任务，或者执行到下一个任务的到达时间。随后，两种情况的结果都是执行下一个任务。但复杂的地方在于，递归结束，回到原函数的时候，会继续执行原任务，并且可能进入新的递归。

我们可以设想一种最坏的情况，第一个任务执行一段后，被第二个任务抢占，随后被第三个抢占，...，直到第N个任务结束，重新执行$N-1, N-2, ..., 1$，这样的总执行时间为$O(2N)$。

我们可以证明最坏情况下为$O(2N) = O(N)$

那么加上排序的$O(N\log(N))$，总时间复杂度为$O(N\log(N))$。

空间复杂度为$O(N)$，因为对于每个任务，给了3个int大小的空间。

2.6 遇到的问题

本题用递归实现，思路很简单，难在`global_time`这个变量的维护，由于会递归执行，执行的过程中会更改`global_time`，而执行`RunTask()`函数时，`global_time`的值应该**至少为**这个任务的`arrive_time`，才能保证任务的往后执行，这就给`global_time`的维护产生了问题。

我后来在`main()`函数里增加了：

```
for (int i = 0; i < N; i++) {
    global_time = (global_time>T[i].arrive_time)?global_time:T[i].arrive_time;
    RunTask(T, i, N);
}
```

即`global_time`应该为两者的最大值，最后成功AC：

https://202.38.86.171/status/3b40f6dbe0ee45e30f8ac76274481706

3 数据缓存

3.1 Description

现有一个长为 N 的数据访问序列和一块大小为 K 的缓存，请设计算法使数据访问的 MISS 次数最少。假设所有可能访问的数据在一个确定的范围内，这里用整数来表示存放数据的地址。假设无论是否命中，这次访问的数据都需要存放在缓存内。

3.2 Input Description

第一行两个整数 N, K 表示共有 N 个数据访问的请求，缓存大小为 K。接下来的 N 行每行一个整数 i，表示访问内存地址为 i 的数据。

数据规模：

$N \leqslant 10,000$

$K \leqslant 1,000$

内存地址的范围为$0 \sim 10,000$

3.3 Output Description

输出一个整数表示最少的 MISS 次数。

3.4 Question Analysis

这题也是操作系统的一个算法，在已知所有访问请求的情况下，我们可以用OPT计算最少的MISS次数，OPT的思想是，把未来最长时间没有使用的那一块，替换成新的。（当然通常情况下我们不会知道未来有哪些请求，所以会用LRU等算法，利用过去的请求处理Cache。）

因为我们知道所有的请求，我这里直接把每个块对应请求的位置，作为一个vector存储，比如1，2，3，1，2，那么出现1的位置就是0和3。

这样当我们处理新的请求时，只需要比较K个元素所对应vector的第一个元素，把第一个元素最大的换掉（第一个元素表示下一次调用它的时间），并更新这个元素对应的vector即可。当出现空vector的时候，直接选择它即可。

实现的时候利用了STL库，使用了

```
std::vector<std::queue<int>> V(INF);
std::vector<int> Address(N);
std::set<int,std::greater<int>> Cache;
```

其中`V`表示每一个块的出现序列，`Address`读入块的请求序列，`Cache`模拟一个Cache，利用了`std::set`的内部是红黑树这个结构。

3.5 复杂度分析

对于每个请求，首先删除它在对应vector的记录，时间为$O(1)$，(`std::queue::pop()`)。

然后判断在不在Cache内，time cost为$O(\log(K))$(使用`std::set::count()`)，如果在Cache内，结束。否则，在Cache的K元素对应的vector内，找第一个元素最大的，把它换掉。找最大值的复杂度为$O(K)$，去除这个值的复杂度为$O(1)$(`std::set::erase`，摊还分析)，而再增加一个元素的复杂度为$O(\log(K))$。

故每一次请求的时间复杂度为$O(\log(K)+K+1+\log(K))=O(K)$

因为总共N个请求，总时间复杂度为$O(NK)$

空间复杂度为$O(NK+N+K)=O(NK)$

3.6 遇到的问题

对于找K个元素中的最大值，尝试过优先队列，但由于每次处理请求的同时，也要删去优先队列中的对应的值，但`C++`的`priority_queue`并不支持这样的操作，也就没有改。

如果使用有限队列，时间复杂度应该能压到$O(N\log(K))$，当然又增加了很多push和pop操作，不一定会比现在的快。

AC:https://202.38.86.171/status/2ca474c41aa29370669bcf20754dd767

4 晾衣服

4.1 Description

小明在一个很长的晾衣架上挂了 N 件等待被晾干的衣服，相邻的两件衣服之间若距离过近会影响衣服的晾干时间。例如，我们可以用相邻两件衣服的距离的最小值来衡量全部衣服晾干的速度。现在小明选择将其中的 M 件衣服撤走，以使得剩余的衣服可以更快晾干。给定 N 件衣服在晾衣架上的位置，请设计算法使得撤走 M 件衣服后相邻衣服距离的最小值最大。

4.2 Input Description

第一行两个整数 N, M 表示共有 N 件衣服，其中可以撤走 M 件。接下来的 N 行，每行一个整数 X_i 表示衣服的坐标。

数据规模：

$M < N \leqslant 200,000$

$X_i \leqslant 20,000,000$

4.3 Output Description

输出一个整数表示撤走 M 件衣服后最大的相邻衣服距离的最小值。

4.4 Question Analysis

这题看起来复杂，解法却出乎意料的简单。

直接把N件衣服的坐标排序，然后以`a[n-1]-a[0]`为上界，0为下界，使用二分法枚举结果。

判断一个结果是否合格，取决于给定这个值，我们需要删掉几个点才能使最小值为它，如果删的点大于M说明这个点不合格，我们找更小的最小值（因为最小值越小，需要删的点越少，删点会把距离增大）。

而如果删的点小于等于M，说明这个点合格，但不能保证最优，所以我们检验更大的最小值。

AC:https://202.38.86.171/status/893ec0ebc1ad7f7ecd156d46066e8e38

4.5 复杂度分析

二分查找的话，复杂度为$O(\log(n))$，这个n是范围大小，在本题就是$x_{\max} - x_{\min}$

而每一次判断是否合格需要最多N次，所以时间复杂度为$O(N(x_{\max} - x_{\min}))$

空间复杂度为$O(N)$，因为只开了一个大小为N的vector。

4.6 遇到的问题

1. 本题我最开始的思路是，寻找最短边，然后根据条件判断是删除最短边的左边还是右边。

   但在决定这个条件时出了问题，我尝试过：删除后最短边最大、删除与其相邻边最短的那个点、一次看5个点，通过一个很复杂的条件计算。结果都失败。

2. 后来转换思路，想着不如选出$N - M$个点，但是选哪个点又陷入了条件的确定，最开始肯定先选两边的点，第三个点选离两个点的中点最近的点。但这样问题就来了，如果现在有n个点被选中，我要再选一个点，就需要计算$n - 1$个区间的中点，然后判断离中点最近的点的距离，再比较$n - 1$个距离，选择最小。这样操作太复杂且时间复杂度高。

3. 最后才想到这种最简单暴力，却比上面两种都快的方法，是参考网上相关题目的思路。