Queue Reconstruction by Height

Suppose you have a random list of people standing in a queue. Each person is described by a pair of integers `(h, k)`, where `h` is the height of the person and `k` is the number of people in front of this person who have a height greater than or equal to `h`. Write an algorithm to reconstruct the queue.

Note:
The number of people is less than 1,100.

Example

```
Input:
[[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Output:
[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]
```

Solution 1

1. **Pick out tallest group of people and sort them** in a subarray (S). Since there's no other groups of people taller than them, therefore **each guy's index will be just as same as his k value**.
2. For 2nd tallest group (and the rest), insert each one of them into (S) by k value. So on and so forth.

E.g.
input: [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
subarray after step 1: **[[7,0], [7,1]]**
subarray after step 2: [[7,0], **[6,1]**, [7,1]]
...

It's not the most concise code, but I think it well explained the concept.

```
class Solution(object):
    def reconstructQueue(self, people):
        if not people: return []

        # obtain everyone's info
        # key=height, value=k-value, index in original array
        peopledct, height, res = {}, [], []

        for i in xrange(len(people)):
            p = people[i]
            if p[0] in peopledct:
                peopledct[p[0]] += (p[1], i),
            else:
                peopledct[p[0]] = [(p[1], i)]
                height += p[0],

        height.sort()      # here are different heights we have

        # sort from the tallest group
        for h in height[::-1]:
            peopledct[h].sort()
            for p in peopledct[h]:
                res.insert(p[0], people[p[1]])

        return res
```

EDIT:
Please also check:
@tlhuang 's concise Python code.
@tonygogogo 's 8 lines C++ solution.
@zeller2 's Java version.

written by YJL1228 original link here

Solution 2

```
vector<pair<int, int>> reconstructQueue(vector<pair<int, int>>& people) {
    auto comp = [](const pair<int, int>& p1, const pair<int, int>& p2)
                    { return p1.first > p2.first || (p1.first == p2.first && p1.s
econd < p2.second); };
    sort(people.begin(), people.end(), comp);
    vector<pair<int, int>> res;
    for (auto& p : people)
        res.insert(res.begin() + p.second, p);
    return res;
}
```

written by zyoppy008 original link here

Solution 3

Based on this solution which I first saw brought up by @bigoffer4all **here** (and which I **explained here**):

```
def reconstructQueue(self, people):
    queue = []
    for p in sorted(people, key=lambda (h, t): (-h, t)):
        queue.insert(p[1], p)
    return queue
```

That takes O(n^2) because each `insert` into the list takes O(n).

Instead of just one long list of all people, I break the queue into O(sqrt(n)) blocks of size up to sqrt(n). Then to insert at the desired index, I find the appropriate block, insert the person into that block, and potentially have to break the block into two. Each of those things takes O(sqrt(n)) time.

```
def reconstructQueue(self, people):
    blocks = [[]]
    for p in sorted(people, key=lambda (h, t): (-h, t)):
        index = p[1]

        for i, block in enumerate(blocks):
            m = len(block)
            if index <= m:
                break
            index -= m
        block.insert(index, p)
        if m * m > len(people):
            blocks.insert(i + 1, block[m/2:])
            del block[m/2:]

    return [p for block in blocks for p in block]
```

"Unfortunately", Python's `list.insert` is really fast compared to doing things in Python myself, and with the inputs allowed here (less than 1100 people), the O(n^2) solution wins. Locally I tested with larger inputs, and around 200000 people the two solutions were about equally fast. With 300000 people, the O(n sqrt(n)) solution was about factor 1.25 faster, and with a million people, the O(n sqrt(n)) solution was about factor 2.2 faster

The testing code:

```
# The original O(n^2) solution.
class Solution(object):
    def reconstructQueue(self, people):
        queue = []
        for p in sorted(people, key=lambda (h, t): (-h, t)):
            queue.insert(p[1], p)
        return queue
nsquared = Solution().reconstructQueue

# The O(n sqrt(n)) solution.
class Solution(object):
    def reconstructQueue(self, people):
        blocks = [[]]
        for p in sorted(people, key=lambda (h, t): (-h, t)):
            index = p[1]

            for i, block in enumerate(blocks):
                m = len(block)
                if index <= m:
                    break
                index -= m
            block.insert(index, p)
            if m * m > len(people):
                blocks.insert(i + 1, block[m/2:])
                del block[m/2:]

        return [p for block in blocks for p in block]
nsqrtn = Solution().reconstructQueue

# Generate a large test case and time it.
from bisect import bisect
from random import randint, shuffle
from timeit import timeit
n = 300000
heights = [randint(1, n) for _ in range(n)]
standing = []
people = []
for h in heights:
    i = bisect(standing, -h)
    standing.insert(i, -h)
    people.append([h, i])
shuffle(people)
for solution in nsquared, nsqrtn, nsquared, nsqrtn:
    print timeit(lambda: solution(people), number=1)
```

written by StefanPochmann original link here