算法讨论班 第九期 2015-10-23

主讲人：韩冬奇

31. Next Permutation

Implement next permutation, which rearranges numbers into the lexicographically next greater permutation of numbers.

If such arrangement is not possible, it must rearrange it as the lowest possible order (ie, sorted in ascending order).

The replacement must be in-place, do not allocate extra memory.

Here are some examples. Inputs are in the left-hand column and its corresponding outputs are in the right-hand column.

`1,2,3 → 1,3,2`

`3,2,1 → 1,2,3`

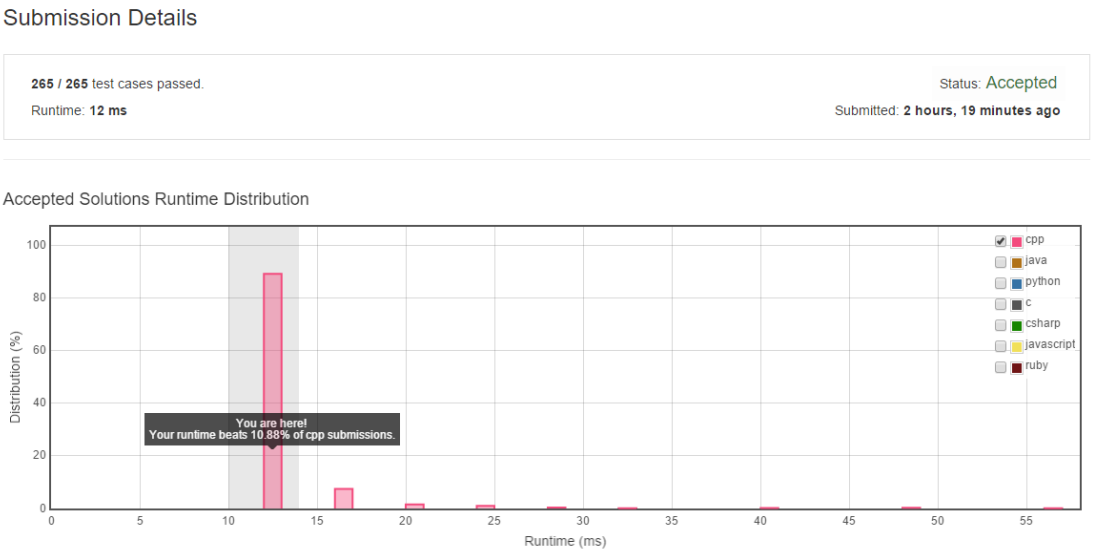
`1,1,5 → 1,5,1`

Tags: Array

```
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        if (nums.size() <= 1) return;
        int i, j;
        for (i = nums.size() - 2;i >= 0 && nums[i] >= nums[i + 1];i--)
            ;
        if (i < 0) {
            reverse(nums.begin(), nums.end());
            return;
        }
        for (j = nums.size() - 1;j >= 0 && nums[j] <= nums[i];j--)
            ;
        swap(nums[i], nums[j]);
        reverse(nums.begin() + i + 1, nums.end());
    }
};
```

思路:

1. 对于一个序列来说，如果序列中任意一个数都比它前面的数小，即该序列是降序排列，那么按照字典序来说，这个序列是最后一个排列。因此，在一个序列中，如果其末尾由 k 个元素组成的子序列是降序排列，那么该子序列已经达到了局部最大值，即这 k 个元素形成了字典序排列的最大。
2. 如果要找到比一个排列大的下一个排列，那么对于排列大小的改变要尽可能的小。若想使得排列增长的最小，则需要去交换排列中尽可能靠后的位置，因为排列的位置越靠后就意味着对于排列的大小的影响越小。比如，排列 `1 2 3 4`，交换 `3` 和 `4` 明显比交换 `2` 和 `4` 或者 `2` 和 `3` 与原排列相比所带来的改变更小，或者说 `1 2 4 3` 小于 `1 4 2 3` 和 `1 3 2 4`。
3. 那么，如 1 中所述，既然全降序的子序列已无增大的可能，而且交换的位置越靠后影响越小。因此，就找到了倒数 k+1 个元素组成的序列，该子序列是非全降序的序列，这也就意味着该子序列还有再排列的空间，直到该子序列排列为全降序为止。
4. 所以，现在需要做的就是从降序的 k 个元素中选取一个放到倒数第 k+1 个位置上(倒数第 k+1 个元素设为 L)。不过元素的选择是有条件的，首先，该元素必须要比 L 大。如果比 L 小，就意味着交换后的序列是小于原序列的，这跟目的相悖。然后，该元素是要比 L 大的元素中最小的，这样才能保证得到的新序列跟原序列的差距最小。
5. 接下来，还有剩下的 k-1 个降序的元素还有 L 需要排。为了保证剩下的这些元素组成的序列是最小的，需要把它们按照升序排列。即比 L 大的按升序依次放到 L 的后面，比 L 小的按升序依次放到 L 的前面。而对于程序的实现上，恰好是交换 L 和比 L 大的元素中最小的，并将这 k 个元素组成的序列反转。
6. 简而言之，整个操作步骤为:
    1. 反向查找第一个增序，设较小值的位置为 i
    2. 反向查找找到比 nums[i]大的第一个数，设位置为 j，交换位置 i 和 j 的值
    3. 将第 i 个位置以后的数列反转



39. Combination Sum

Given a set of candidate numbers **(C)** and a target number **(T)**, find all unique combinations in **C** where the candidate numbers sums to **T**.
The **same** repeated number may be chosen from **C** unlimited number of times.
Note:

- All numbers (including target) will be positive integers.
- Elements in a combination (a1, a2, … , ak) must be in non-descending order. (ie, a1 ≤ a2 ≤ … ≤ ak).
- The solution set must not contain duplicate combinations.

For example, given candidate set `2,3,6,7` and target `7`, A solution set is:
`[7]`
`[2, 2, 3]`
Tags: Array, Backtracking

```
class Solution {
public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> re;
        if (candidates.empty()) return re;
        sort(candidates.begin(), candidates.end());
        vector<int> comVec;
        backtrack(candidates, re, comVec, target, 0);
        return re;
    }
    void backtrack(vector<int>& candidates, vector<vector<int>>& re, vector<int>&
comVec, int target, int i) {
        if (target == 0) {
            re.push_back(comVec);
            return;
        }
        for (;i < candidates.size() && target >= candidates[i];i++) {
            comVec.push_back(candidates[i]);
            backtrack(candidates, re, comVec, target - candidates[i], i);
            comVec.pop_back();
        }
    }
};
```

思路:

1. 题目是让我们得到和为所给目标值的候选子集，且为多重集，对于候选元素个数不限。而通过选取候选集里的元素去不断接近目标值的过程是一个不断试验的过程。为了能够得到全部可能的结果，试验的方法需要按一个特定的顺序，以保证不会漏掉或者重复。因此候选元素的选择应该按照递增或者递减的顺序，我们以递增为例。
2. 因而我们很自然的想到可以用递归的方法去做，不断的减小问题的规模，而且子问题是完全独立的。按照候选元素选取的原则，优先选取数值小的元素。说到递归，就分为两部分：第一部分是递归停止条件，第二部分是子问题划分。显然，该问题的递归停止条件就是 `target <= 0`，当 `target = 0` 时说明找到问题的一个解，当 `target < 0` 时说明元素选取已不符合要求。第二个问题就是子问题拆分。很直观的想法就是，我们按照规则从中选取了一个数，然后递归给下一层去处理剩下的问题。当下一层处理完跳出以后，就说明当前选取的这个元素的解空间已经处理完，再按照规则选择下一个元素，接着递归进去寻找可行解。需要注意的是，递归到下一层的时候，初始元素的选择是有要求的，它的值必须要大于等于当前选取的元素的值，以保证我们生成的试验的顺序不会产生重复。
3. 简而言之，递归的步骤如下：

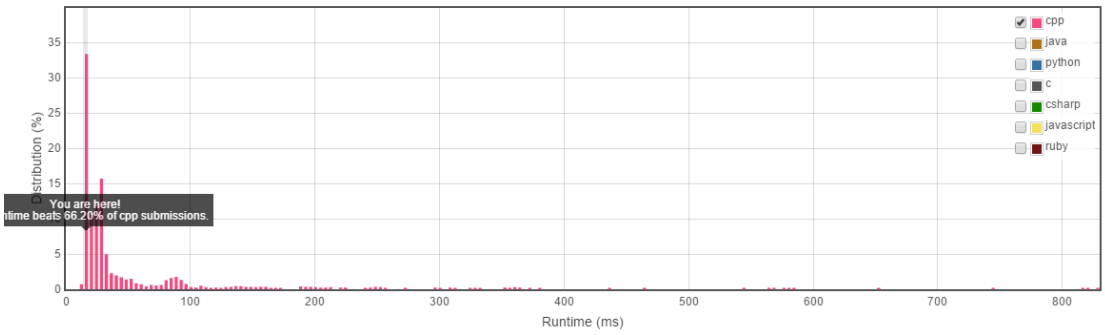
```
recursion (target, combination) {
    if (target == 0)
        return combination;
    for i in candidates:
        add new i to combination;
        recursion(target - i, combination);
        remove i from combination;
}
```

Submission Details

168 / 168 test cases passed.
Runtime: **16 ms**

Status: Accepted
Submitted: **11 hours, 37 minutes ago**

Accepted Solutions Runtime Distribution



57. Insert Internal

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start times.

Example 1:
Given intervals `[1,3],[6,9]`, insert and merge `[2,5]` in as `[1,5],[6,9]`.
Example 2:
Given `[1,2],[3,5],[6,7],[8,10],[12,16]`, insert and merge `[4,9]` in as `[1,2],[3,10],[12,16]`.
This is because the new interval `[4,9]` overlaps with `[3,5],[6,7],[8,10]`.
Tags: Array, Sort

```
class Solution {
public:
    vector<Interval> insert(vector<Interval>& intervals, Interval newInterval) {
        vector<Interval> re;
        int i, start, end;

        if (intervals.empty()) {
            re.push_back(newInterval);
            return re;
        }
        if (newInterval.start > intervals.back().end) {
            intervals.push_back(newInterval);
            return intervals;
        }

        i = 0;
        while (i < intervals.size() && intervals[i].end < newInterval.start)
            re.push_back(intervals[i++]);

        start = min(newInterval.start, intervals[i].start);

        while (i < intervals.size() && intervals[i].start <= newInterval.end)
            i++;

        end = (i>0) ? max(newInterval.end, intervals[i - 1].end) : newInterval.end;
```

```
        re.push_back(Interval(start, end));

        while (i < intervals.size())
            re.push_back(intervals[i++]);

        return re;
    }
};
```

思路:

问题的关键就在于把 `newInterval` 放到合适的位置。显然，如果新区间在区间序列的最前或者最后，只需要插入到 `intervals` 的头部或者尾部就可以了。更一般的情况则是新区间和区间序列中的区间有重叠或者是在区间之间。

可大致分为如下几种情况：

```
[     ]   [     ]   [     ]
  ↑_↑
[     ]       [     ]   [     ]
          ↑_↑
[     ]   [     ]   [     ]
  ↑__________↑
[     ]     [     ]   [     ]
          ↑__________↑
```

那么合并的结果为：左边界是 `min(newInterval.start, intervals[i].start)`，右边界是 `max(newInterval.end, intervals[i].end)`。