1

将字符串的每一位作为一个子排序码，按照高位优先法进行基数排序.

注意到，每个桶经过一轮排序会产生 27 个子桶，分别对应 26 个字母和空字符，而空字符对应的桶无需参与后续排序——因为其中元素的后续字符都是空字符，注定相等，没有排序的必要.

像这样，长度为 l_i 的字符串只会参与 l_i 轮排序，因此这个算法的时间是 $O(\sum\limits_{i=1}^{m} l_i)$ 的.

最后，应当指出，由于字符串的复制并不是 $O(1)$ 的，有必要采用索引排序，在移动元素时只改变索引数组，这样元素移动就是 $O(1)$ 的了.

2

(1)

对于非流数据，可以在数组上用筛选法原地建立最大堆，再删除 k 次堆顶，所删除的 k 个元素就是前 k 大的元素. 建堆是 $O(n)$ 的，单次删除堆顶是 $O(\log n)$ 的，因此整个算法是 $O(n + k\log n)$ 的.

对于流数据，可以先读入前 k 个元素并以之建立最小堆，再依次读入新元素.

读入新元素时：

- 如果该元素小于堆顶元素，它一定不是整个数组中前 k 大的元素，因此我们直接忽略这个值.
- 如果该元素大于堆顶元素，堆顶元素就一定不是整个数组中前 k 大的元素，因此直接用该元素替换堆顶元素，并向下调整以维护最小堆.
- 该元素与堆顶元素相等时，替不替换都可以.

当输入结束后，堆中的 k 个元素就是前 k 大的元素.

建堆是 $O(k)$ 的，单次调整是 $O(\log k)$ 的，因此整个算法是 $O(n\log k)$ 的.

虽然从复杂度上看这个算法要低效一些，但流数据的输入较慢，这个算法可以同时等待输入和维护最小堆，因此实践中在流数据上可能会优于前一个算法.

(2)

对整个数组按从大到小的顺序进行快速排序.

在第一轮排序结束后，原区间被分成左右两个子区间，长度分别为 l_1, l_2，其中 $l_1 + l_2 = n$.

如果 $l_1 < k$，那么左子区间中都是前 k 大的元素，因此左子区间不必继续排序，只对右子区间进行排序并寻找其中的前 $k - l_1$ 大元素即可.

如果 $l_1 > k$，那么前 k 大的元素都在左子区间中，因此右子区间不必继续排序，只对左子区间进行排序并寻找其中的前 k 大元素即可.

如果 $l_1 = k$，左子区间的全部元素即为所求.

继续递归地做下去，即可找到全部前 k 大的元素.

设在 n 个元素中寻找前 k 大的元素的平均时间为 $T(n, k)$.

第一轮排序的时间消耗是 $O(n)$，排序后原区间被分成左右两个子区间，不妨假设子区间长度是均匀分布的，于是有

$$T(n,k) = \frac{1}{n}(\sum_{i=0}^{k-1} T(n-i,k-i) + \sum_{i=k+1}^{n-1} T(i,k)) + O(n).$$

这个递推方程很难求解，但用第二数学归纳法可以证明 $T(n,k) = \Theta(n)$. 只需按数学归纳法的步骤作一些初等的代数计算即可，这里不再赘述.

对于不同的快速排序实现，上述递推方程的形式可能略有差异，但不会影响最终结果.

3

首先，我们对算法的正确性给出一个直觉性的证明.

我们对区间长度 n 归纳地去考虑，对于较小的 n，容易验证算法的正确性. 在归纳的过程中，由于 t 一定为正，三个子区间大小严格小于原区间大小，根据归纳假设，三个子区间都被正确排序了. 因此，我们唯一要证明的就是：三个子区间被正确排序会导致整个区间被正确排序.

记 $L = [i, j-t], M = [i+t, j-t], R = [j-t, j]$. 三次子区间排序分别使 $L \cup M, M \cup R, L \cup M$ 中元素有序.

对于正确排序后应当属于 R 的元素，如果初始在 $L \cup M$，第一次排序后一定会落在 M. 如果初始在 R，第一次排序后仍在 R. 无论如此，第二次排序后它都会落在 R 中的正确位置，且不被第三次排序影响.

对于正确排序后应当属于 M 的元素，如果初始在 $L \cup M$，第一次排序后一定会落在 M. 如果初始在 R，第一次排序后仍在 R，第二次排序后落在 M. 无论如此，第三次排序后它都会落在 M 中的正确位置.

对于正确排序后应当属于 L 的元素，如果初始在 $L \cup M$，第一次排序后一定会落在 L，且不被第二次排序影响. 如果初始在 R，第一次排序后仍在 R，第二次排序后落在 M. 无论如此，第三次排序后它都会落在 L 中的正确位置.

这是一个粗糙的定性分析，用第二数学归纳法可以给出一个严格的证明，但会占用更多篇幅，而且仍是以上述直觉为基础，因此这里不再赘述.

接下来，我们证明该算法的渐进时间复杂度为 $\Theta(n^{\log_3 \frac{3}{2}})$.

设排序长度为 n 的区间所需时间为 $T(n)$，从代码实现中容易看出

$$T(n) = 3T(\frac{2}{3}n) + c,$$

其中 c 为一个常数.

由于取整导致的些微误差，这个递推方程并不是完全精确的，不过当 n 充分大时，这点误差无伤大雅，不会对复杂度的数量级造成影响.

设 $(\frac{3}{2})^k < n < (\frac{3}{2})^{k+1}$，有

$$
\begin{aligned}
T(n) &= 3T(\frac{2}{3}n) + c \\
&= 9T(\frac{4}{9}n) + 4c \\
&= 27T(\frac{8}{27}n) + 13c \\
&= \cdots \\
&= 3^k T(1) + \frac{1}{2}(3^k - 1)c.
\end{aligned}
$$

由 $(\frac{3}{2})^k < n$ 可知 $3^k < 3^{\log_{\frac{3}{2}} n} = 3^{\frac{\log_3 n}{\log_3 \frac{3}{2}}} = n^{\frac{1}{\log_3 \frac{3}{2}}}$.

同理由 $n < (\frac{3}{2})^{k+1}$ 可知 $3^k > (\frac{2}{3}n)^{\frac{1}{\log_3 \frac{3}{2}}} \approx \frac{1}{3} n^{\log_3 \frac{3}{2}}$.

因此，$T(n) = \Theta(n^{\log_3 \frac{3}{2}}) \approx \Theta(n^{2.7})$.

这个排序算法的思路很新颖，不过其时间效率低得令人发指，甚至不如最基本的冒泡、选择、插入排序.

4

显然不是所有序列都有可行的折叠方案，下面给出了一个判定函数 `solve`.

```
vector<pair<int, int>> odd, even;    // odd intervals, even intervals. Elements have form {left end, right end}
int input[N];    // input sequence, N elements, 0~N-1

// check if intervals in v don't overlap
bool noOverlap(vector<pair<int, int>> v){
    sort(v.begin(), v.end());   // ensures v[i].first <= v[i+1].first for all i

    int re = -1;    // right end
    for (auto t: v){
        if (re <= t.first)  // [  ] {  }
            re = t.second;
        else if (re < t.second) // [ {  ] }
            return false;
    }
    return true;
}

// check if the input sequence has a valid solution
bool solve(){
    // build intervals, even->odd->even->...
    for (int i = 0; i < N-1; ++i)
        (i&1? odd: even).push_back({input[i], input[i+1]});

    return noOverlap(odd) && noOverlap(even);
}
```

对于一个给定的输入序列 $k_1, k_2, \cdots, k_n$，我们在数轴上画出区间 $[k_1, k_2]. [k_2. k_3], \cdots, [k_{n-1}, k_n]$.

i 为奇数时，称 $[k_i, k_{i+1}]$ 为奇区间，i 为偶数时则称为偶区间.

数轴上的整点 $1, 2, \cdots, n$ 对应着纸条的格子，这些区间就对应着纸条的边线（区间的一奇一偶正是从边线的一左一右得到启发）. 容易发现，一个序列有可行折叠方案，当且仅当所有奇区间都互不交叉且所有偶区间都互不交叉（即任意两区间或者交为空，或者有包含关系）.

上述 `noOverlap` 函数便是用于判定区间是否互不交叉的，其时间开销为 $O(n\log n)$，因此整个判定过程为 $O(n\log n)$ 的.

正因为纸条与数轴的对应关系，当奇偶区间都互不交叉时，按照这些区间的"形状"去折叠，就自然得到了一种可行的折叠方案.

5

这是一个经典问题. 将区间等分为左子区间和右子区间，逆序对可以分为左子区间内的、右子区间内的、左右子区间之间的. 前两种可以递归处理，最后一种如果左右区间都有序则非常容易计算. 这个过程与归并排序完全相同，因此直接对整个区间作归并排序，即可计算出逆序对数，时间复杂度为 $\Theta(n\log n)$.

```
int cnt = 0;     // result
void merge_sort(int l, int r){  // [l, r]
    if (l == r) return;

    int mid = (l+r)/2;
    merge_sort(l, mid);
    merge_sort(mid+1, r);

    // R.Sedgewick optimization
    for (int i = l; i <= mid; ++i)
        t[i] = a[i];
    for (int i = 0; i < r-mid; ++i)
        t[mid+1+i] = a[r-i];

    for (int k = l, i = l, j = r; k <= r; ++k)
        if (t[i] <= t[j]) a[k] = t[i++];
        else cnt += mid-i+1, a[k] = t[j--];
}
```