We know from the text that polynomials (i.e. a sum of terms where n is raised to fixed powers, even if they are not integers) grow slower than exponentials. Thus, we will consider f_1, f_2, f_3, f_6 as a group, and then put f_4 and f_5 after them.

For polynomials f_i and f_j, we know that f_i and f_j can be ordered by comparing the highest exponent on any term in f_i to the highest exponent on any term in f_j. Thus, we can put f_2 before f_3 before f_1. Now, where to insert f_6? It grows faster than n^2, and from the text we know that logarithms grow slower than polynomials, so f_6 grows slower than n^c for any $c > 2$. Thus we can insert f_6 in this order between f_3 and f_1.

Finally come f_4 and f_5. We know that exponentials can be ordered by their bases, so we put f_4 before f_5.

[1] ex831.202.488

The right ordering (from smallest to largest) is:
$g_1, g_3, g_4, g_5, g_2, g_7, g_6$

(i) This is false in general, since it could be that $g(n) = 1$ for all n, $f(n) = 2$ for all n, and then $\log_2 g(n) = 0$, whence we cannot write $\log_2 f(n) \leq c \log_2 g(n)$.

On th other hand, if we simply require $g(n) \geq 2$ for all n beyond some n_1, then the statement holds. Since $f(n) \leq cg(n)$ for all $n \geq n_0$, we have $\log_2 f(n) \leq \log_2 g(n) + \log_2 c \leq (\log_2 c)(\log_2 g(n))$ once $n \geq \max(n_0, n_1)$.

(ii) This is false: take $f(n) = 2n$ and $g(n) = n$. Then $2^{f(n)} = 4^n$, while $2^{g(n)} = 2^n$.

(iii) This is true. Since $f(n) \leq cg(n)$ for all $n \geq n_0$, we have $(f(n))^2 \leq c^2(g(n))^2$ for all $n \geq n_0$.

[1]ex66.350.972

(a) Suppose for simplicity that n is a perfect square. We drop the first jar from heights that are multiples of $\sqrt{n}$ (i.e. from $\sqrt{n}, 2\sqrt{n}, 3\sqrt{n}, \ldots$) until it breaks.

If we drop it from the top rung and it survives, then we're also done. Otherwise, suppose it breaks from height $j\sqrt{n}$. Then we know the highest safe rung is between $(j-1)\sqrt{n}$ and $j\sqrt{n}$, so we drop the second jar from rung $1 + (j-1)\sqrt{n}$ on upward, going up by one each time.

In this way, we drop each of the two jars at most $\sqrt{n}$ times, for a total of at most $2\sqrt{n}$. If n is not a perfect square, then we drop the first jar from heights that are multiples of $\lfloor \sqrt{n} \rfloor$, and then apply the above rule for the second jar. In this way, we drop the first jar at most $2\sqrt{n}$ times (quite an overestimate if n is reasonably large) and the second jar at most $\sqrt{n}$ times, still obtaining a bound of $O(\sqrt{n})$.

(b) We claim by induction that $f_k(n) \le 2kn^{1/k}$. We begin by dropping the first jar from heights that are multiples of $\lfloor n^{(k-1)/k} \rfloor$. In this way, we drop the first jar at most $2n/n^{(k-1)/k} = 2n^{1/k}$ times, and thus narrow the set of possible rungs down to an interval of length at most $n^{(k-1)/k}$.

We then apply the strategy for $k-1$ jars recursively. By induction it uses at most $2(k-1)(n^{(k-1)/k})^{1/(k-1)} = 2(k-1)n^{1/k}$ drops. Adding in the $\le 2n^{1/k}$ drops made using the first jar, we get a bound of $2kn^{1/k}$, completing the induction step.

[1] ex291.532.145

We'll define a recursive divide-and-conquer algorithm ALG which takes a sequence of distinct numbers $a_1, \ldots, a_n$ and returns N and $a'_1, \ldots, a'_n$ where

- N is the number of significant inversions
- $a'_1, \ldots, a'_n$ is same sequence sorted in the increasing order

ALG is similar to the algorithm from the chapter that computes the number of inversions. The difference is that in the 'conquer' step we merge twice: first we merge $b_1, \ldots, b_k$ with $b_{k+1}, \ldots, b_n$ just for sorting, and then we merge $b_1, \ldots, b_k$ with $2b_{k+1}, \ldots, 2b_n$ for counting significant inversions.

Let's define ALG formally. For $n = 1$ ALG just returns $N = 0$ and $\{a_1\}$ for the sequence. For $n > 1$ ALG does the following:

- let $k = \lfloor n/2 \rfloor$.
- call $\mathsf{ALG}(a'_1, \ldots, a'_k)$. Say it returns N_1 and $b_1, \ldots, b_k$.
- call $\mathsf{ALG}(a'_{k+1}, \ldots, a'_n)$. Say it returns N_2 and $b_{k+1}, \ldots, b_n$.
- compute the number N_3 of significant inversions (a_i, a_j) where $i \le k < j$.
- return $N = N_1 + N_2 + N_3$ and $a'_1, \ldots, a'_n = \mathsf{MERGE}(b_1, \ldots, b_k;\ b_{k+1}, \ldots, b_n)$

MERGE can be implemented in $O(n)$ time. According to the discussion in the book, it remains to find a way to compute N_3 in $O(n)$ time. We implement a variant of merge-count of $b_1, \ldots, b_k$ and $2\,b_{k+1}, \ldots, 2\,b_n$ as follows.

- Initialize counters: $i \leftarrow k$, $j \leftarrow n$, $N_3 \leftarrow 0$.
- If $b_i \le 2b_j$ then
  - if $j > k + 1$ decrease j by 1.
  - if $j = k + 1$ return N_3.
- If $b_i > 2b_j$ then increase N_3 by $j - k$. Then
  - if $i > 1$ decrease i by 1.
  - if $i = 1$ return N_3.

Explanation For every i we count the number of significant inversions between b_i and all b_j's. If $b_i \le 2b_j$ then there are no significant inversions between b_i and any b_m s.t. $m \ge j$, so we decrease j. If $b_i > 2b_j$ then $b_i > 2b_m$ for all m s.t. $k < m \le j$. In other words, we have detected $j - k$ significant inversions involving b_i. So we increase N_3 by $j - k$. Finally, when we are down to $i = 1$ and have counted significant inversions involving b_1, there are no more significant inversions to be detected.

[1] ex499.218.598

We give two solutions for this problem. The first solution is a divide and conquer algorithm, which is easier to think of. The second solution is a clever linear time algorithm.

Via divide and conquer: Let $e_1, \ldots, e_n$ denote the equivalence classes of the cards: cards i and j are equivalent if $e_i = e_j$. What we are looking for is a value x so that more than $n/2$ of the indices have $e_i = x$.

Divide the set of cards into two roughly equal piles: a set of $\lfloor n/2 \rfloor$ cards and a second set for the remaining $\lceil n/2 \rceil$ cards. We will recursively run the algorithm on the two sides, and will assume that if the algorithm finds an equivalence class containing more than half of the cards, then it returns a sample card in the equivalence class.

Note that if there are more than $n/2$ cards that are equivalent in the whole set, say have equivalence class x, than at least one of the two sides will have more than half the cards also equivalent to x. So at least one of the two recursive calls will return a card that has equivalence class x.

The reverse of this statement is not true: there can be a majority of equivalent cards in one side, without that equivalence class having more than $n/2$ cards overall (as it was only a majority on one side). So if a majority card is returned on either side we must test this card against all other cards.

```
If $|S| = 1$ return the one card
If $|S| = 2$
    test if the two cards are equivalent
    return either card if they are equivalent
Let $S_1$ be the set of the first $\lfloor n/2 \rfloor$ cards
Let $S_2$ be the set of the remaining cards
Call the algorithm recursively for $S_1$.
If a card is returned
    then test this against all other cards
If no card with majority equivalence has yet been found
    then call the algorithm recursively for $S_2$.
    If a card is returned
        then test this against all other cards
Return a card from the majority equivalence class if one is found
```

The correctness of the algorithm follows from the observation above: that if there is a majority equivalence class, than this must be a majority equivalence class for at least one of the two sides.

To analyze the running time, let $T(n)$ denote the maximum number of tests the algorithm does for any set of n cards. The algorithm has two recursive calls, and does at most $2n$ tests outside of the recursive calls. So we get the following recurrence (assuming n is divisible by 2):

$$T(n) \leq 2T(n/2) + 2n.$$

[1]ex628.974.324

As we have seen in the chapter, this recurrence implies that $T(n) = O(n \log n)$.

In linear time: Pair up all cards, and test all pairs for equivalence. If n was odd, one card is unmatched. For each pair that is not equivalent, discard both cards. For pairs that are equivalent, keep one of the two. Keep also the unmatched card, if n is odd. We can call this subroutine ELIMINATE.

The observation that leads to the linear time algorithm is as follows. If there is an equivalence class with more then $n/2$ cards, than the same equivalence class must also have more than half of the cards after calling ELIMINATE. This is true, as when we discard both cards in a pair, then at most one of them can be from the majority equivalence class. One call to ELIMINATE on a set of n cards takes $n/2$ tests, and as a result, we have only $\leq \lceil n/2 \rceil$ cards left. When we are down to a single card, then its equivalence is the only candidate for having a majority. We test this card against all others to check if its equivalence class has more than $n/2$ elements.

This method takes $n/2 + n/4 + \ldots$ tests for all the eliminates, plus $n - 1$ tests for the final counting, for a total of less than $2n$ tests.

This can be accomplished directly using a convolution. Define one vector to be $a = (q_1, q_2, \ldots, q_n)$. Define the other vector to be $b = (n^{-2}, (n-1)^{-2}, ..., 1/4, 1, 0, -1, -1/4, ... - n^{-2})$. Now, for each j, the convolution of a and b will contain an entry of the form

$$\sum_{i<j} \frac{q_i}{(j-i)^2} + \sum_{i>j} \frac{-q_i}{(j-i)^2}.$$

From this term, we simply multiply by Cq_j to get the desired net force F_j.

The convolution can be computed in $O(n \log n)$ time, and reconstructing the terms F_j takes an additional $O(n)$ time.

[1] ex726.26.783

For simplicity, we will say u *is smaller than* v, or $u \prec v$, if $x_u < x_v$. We will extend this to sets: if S is a set of nodes, we say $u \prec S$ if u has a smaller value than any node in S.

The algorithm is the following. We begin at the root r of the tree, and see if r is smaller than its two children. If so, the root is a local minimum. Otherwise, we move to any smaller child and iterate.

The algorithm terminates when either (1) we reach a node v that is smaller than both its children, or (2) we reach a leaf w. In the former case, we return v; in the latter case, we return w.

The algorithm performs $O(d) = O(\log n)$ probes of the tree; we must now argue that the returned value is a local minimum. If the root r is returned, then it is a local minimum as explained above. If we terminate in case (1), v is a local minimum because v is smaller than its parent (since it was chosen in the previous iteration) and its two children (since we terminated). If we terminate in case (2), w is a local minimum because w is smaller than its parent (again since it was chosen in the previous iteration).

[1] ex739.448.876

Let B denote the set of nodes on the *border* of the grid G — i.e. the outermost rows and columns. Say that G has *Property $(*)$* if it contains a node $v \notin B$ that is adjacent to a node in B and satisfies $v \prec B$. Note that in a grid G with Property $(*)$, the *global minimum* does not occur on the border B (since the global minimum is no larger than v, which is smaller than B) — hence G has at least one local minimum that does not occur on the border. We call such a local minimum an *internal local minimum*

We now describe a recursive algorithm that takes a grid satisfying Property $(*)$ and returns an internal local minimum, using $O(n)$ probes. At the end, we will describe how this can be easily converted into a solution for the overall problem.

Thus, let G satisfy Property $(*)$, and let $v \notin B$ be adjacent to a node in B and smaller than all nodes in B. Let C denote the union of the nodes in the middle row and middle column of G, not counting the nodes on the border. Let $S = B \cup C$; deleting S from G divides up G into four sub-grids. Finally, let T be all nodes adjacent to S.

Using $O(n)$ probes, we find the node $u \in S \cup T$ of minimum value. We know that $u \notin B$, since $v \in S \cup T$ and $v \prec B$. Thus, we have two cases. If $u \in C$, then u is an internal local minimum, since all of the neighbors of u are in $S \cup T$, and u is smaller than all of them. Otherwise, $u \in T$. Let G' be the sub-grid containing u, together with the portions of S that border it. Now, G' satisfies Property $(*)$, since u is adjacent to the border of G' and is smaller than all nodes on the border of G'. Thus, G' has an internal local minimum, which is also an internal local minimum of G. We call our algorithm recursively on G' to find such an internal local minimum.

If $T(n)$ denotes the number of probes needed by the algorithm to find an internal local minimum in an $n \times n$ grid, we have the recurrence $T(n) = O(n) + T(n/2)$, which solves to $T(n) = O(n)$.

Finally, we convert this into an algorithm to find a local minimum (not necessarily internal) of a grid G. Using $O(n)$ probes, we find the node v on the border B of minimum value. If v is a corner node, it is a local minimum and we're done. Otherwise, v has a unique neighbor u not on B. If $v \prec u$, then v is a local minimum and again we're done. Otherwise, G satisfies Property $(*)$ (since u is smaller than every node on B), and we call the above algorithm.

[1] ex624.352.598