Divide-and-Conquer Method, Searching and Sorting

1 Introduction

Divide-and-conquer In divide-and-conquer method, a problem is divided into one or more subproblems, each subproblem is solved recursively, and the solutions to the subproblems are combined to arrive at a solution to the problem.

A number of considerations play in the design of algorithms using the divide-and-conquer method.

- It may be necessary to reduce the given problem to a problem so it is amenable for the application of the divide-and-conquer method.
- Each problem is associated with a size and the size of each subproblem should be smaller than the problem to ensure termination.
- The time complexity of a divide-and-conquer solution is a function of the following factors:
  1. the number of subproblems and their sizes, and
  2. the overhead involved in dividing the problem into subproblems and putting together the solution from the solutions of the subproblems.

  Time complexity of a divide-and-conquer algorithm can be expressed in terms of recurrence relations.
- Techniques for solving recurrence relations are useful in the study of design and analysis of divide-and-conquer algorithms.

Searching Problems are usually stated in the form of searching for an object in a collection. In these situations, it may be useful to explicitly identify the possible scenarios. A typical algorithm for a search problem consists of a series of steps where in each step we perform some computation and ask a question to narrow down the possibilities. Each possible answer to the question reduces the problem to a subproblem where we have fewer possibilities for the object we are seeking. Moreover, the set of possible scenarios is partitioned along the possible answers to the question. For efficiency (that is, for minimizing the number of questions in the worst-case), it is important to design the algorithm so that the number of possibilities in each case is as equal as it can be to the number of possibilities in other cases.

2 Problems

Problem 1: Tromino Puzzle

Cover a $2^n \times 2^n$ $(n \geq 1)$ board missing one square with right tromnioes, which are L-shaped tiles formed by three adjacent squares. The missing square can be any of the board squares. Trominoes should cover all the squares except the missing ones with no overlaps.

A right tromino can also be viewed as a 2×2 board with exactly one missing square.

Problem 2: Tromino Tilings

For each of the three cases, prove or disprove that for every $n > 0$ all the boards of the following can be tiled by right trominoes.

A *tiling* is a cover of the of board with no overlaps.

1. $3^n \times 3^n$
2. $5^n \times 5^n$
3. $6^n \times 6^n$

Recall that right trominoes are L-shaped tiles formed by three adjacent squares. In a tiling, trominoes can be oriented in different ways, but they should cover all the squares of the board exactly with no overlaps.

Problem 3: Coin Removal

There is a line of n coins on the table; some of them are heads up and the rest are tails up, in no particular order. The object of the puzzle is to remove all the coins by a sequence of moves. On each move, one can remove any head-up coin, after which its neighboring coin or coins, if any, must be turned over. Coins are considered "neighbors" if they are next to each other in the original line; if there is a gap between coins after some moves, the coins are no longer considered neighbors.

Determine the property of the starting line that is necessary and sufficient for the puzzle to have a solution. For those lines that can be removed by the puzzle's rules, design an algorithm for doing so.

Problem 4: Finding Minimum and Maximum

Given $n > 1$ items and a two pan balance scale with no weights, determine the lightest and the heaviest items in $\lceil 3n/2 \rceil - 2$ weighings.

Problem 5: Maximum contiguous sum

Use the divide-and-conquer approach to write an efficient recursive algorithm that finds the maximum sum in any contiguous sublist of a given list of n real (positive or negative) values. Analyse your algorithm, and show the results in order notation. Can you do better? Obtain a linear-time algorithm.

Problem 6: Stooge sort

Professors Howard, Fine, and Howard have proposed the following 'elegant' sorting algorithm:

```
STOOGE-SORT (A, i, j)
1    if A[i] > A[j]
2      then exchange A[i] ↔ A[j]
3    if i + 1 ≥ j
4      then return
5    k ← ⌊(j − i + 1)/3⌋
6    STOOGE-SORT (A, i, j − k)
7    STOOGE-SORT (A, i + k, j)
8    STOOGE-SORT (A, i, j − k)
```

1. Argue that STOOGE-SORT$(A, 1, length[A])$ correctly sorts the input array $A[1..n]$, where $n = length[A]$
2. Give a recurrence for the worst-case running time of STOOGE-SORT and a tight asymptotic (Θ-notation) bound on the worst-case running time.
3. Compare the worst-case running time of STOOGE-SORT with that of insertion sort, merge sort, heapsort, and quicksort. Do the professors deserve tenure?

Problem 7: Computing mode

The mode of a set of numbers is the number that occurs most frequently in the set. The set (4,6,2,4,3,1) has a mode of 4.

1. Give an efficient and correct algorithm to compute the mode of a set of n numbers;
2. Suppose we know that there is an (unknown) element that occurs $\lfloor n/2 \rfloor + 1$ times in the set. Give a worst-case linear-time algorithm to find the mode. For partial credit, your algorithm may run in expected linear time.

Problem 8: Frequent elements

Design an algorithm that, given a list of n elements in an array, finds all the elements that appear more than $n/3$ times in the list. The algorithm should run in linear time. n is a nonnegative integer.

You are expected to use comparisons and achieve linear time. You may not use hashing. Neither can you use excessive space. Linear space is sufficient. Moreover, you are expected to design a deterministic algorithm. You may also not use the standard linear-time deterministic selection algorithm. Your algorithm has to be more efficient (in terms of constant factors) than the standard linear-time deterministic selection algorithm.

Problem 9: Testing chips

Professor Diogenes has supposedly identical VLSI chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports wheter it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the answer of a bad chip cannot be trusted. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B Says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

1. Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

2. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

3. Show that the good chips can be identified with $\Theta(n)$ (proportional to n) pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Problem 10: Integer multiplication (DPV book)

Use Karatasuba integer multiplication algorithm to multiply the two binary integers 10011011 and 10111010.

Problem 11: Multiplication in threes

Describe and analyze a divide-and-conquer algorithm for integer multiplication based on partitioning up each input into 3 blocks of digits. Is this better than the divide and conquer approach in the standard Karatsuba algorithm? If so, is it possible to get further improvements along the same lines?

Problem 12: Counting inversions (KT)

Let $a_1, \ldots, a_n$ be a sequence of integers. Two indices $1 \leq i < j \leq n$ form an *inversion* in the sequence if $a_i > a_j$. Design an efficient algorithm to count the number of inversions in the sequence. There is an $O(n^2)$

time naive algorithm for solving this problem. However, there is a more efficient algorithm that runs in time an $O(n \log n)$.
Consider the sequence $a_1 = 22, a_2 = 14, a_3 = 16, a_4 = 42, a_5 = 11, a_6 = 18, a_7 = 20, a_8 = 17$ of numbers. In this sequence, the pair of indices $(1, 2)$ is an inversion since $a_1 > a_2$. Similarly, (2,5) is an inversion. Here is the list of all inversions in this sequence:

$$(1,2), (1,3), (1,5), (1,6), (1,7), (1,8), (2,5), (3,5), (4,5), (4,6), (4,7), (4,8), (6,8), (7,8)$$

Altogether, there are 14 inversions in the sequence.

Problem 13: Counting significant inversions (KT)

Let $a_1, \ldots, a_n$ be a sequence of integers. Two indices $1 \leq i < j \leq n$ form an *significant inversion* in the sequence if $a_i > 2a_j$. Design an efficient algorithm to count the number of inversions in the sequence.

Problem 14: Diameter of a tree (CLRS)

The *diameter* of a tree $T = (V, E)$ is given by

$$\max_{u,v \in V} \delta(u, v)$$

where $\delta(u, v)$ is the shortest path length between the vertices u and v. That is, the diameter is the largest of all shortest-path length in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

Problem 15: Skylines

In a video game, a city's buildings are rectangles along an axis, specified as $B_i = (s_i, f_i, h_i)$, for $1 \leq i \leq n$, for a building starting at $x = s_i$ and going to $x = f_i$ at height h_i. When seen from a distance, the graphics is supposed to plot the skyline, which at each vertical point x has height the height of the tallest building B_i with $s_i \leq x \leq f_i$. Give an algorithm to plot the skyline, as a series of points connected by line segments. Assume n is a power of 2.

Problem 16: Maximum overlap of two intervals

Design an algorithm that takes as input a list of intervals $[a_i, b_i]$ for $1 \leq i \leq n$ and outputs the length of the maximum overlap of two distinct intervals in the list.

Problem 17: Black jack

Give the best algorithm you can for the following problem:

Name Blackjack Hand Card Counting

Instance An array A of n positive integers (cards with face values) with values from 1 to k, and positive integers $l \leq n, v \leq kn$.

Problem Count the number of sets of l array positions (hands of l cards) whose total value is equal to v.

Analyze your algorithm in terms of n, k and l. Your algorithm should take time polynomial in all 3 parameters.

Problem 18: Linear 3SAT (DPV book)

The 3SAT problem is defined in section 8.1 of the DPV book. Briefly, the input is a Boolean formula - expressed as a set of clauses - over some set of variables and the goal is to determine whether there is an assignment (of true/false values) to these variables that makes the entire formula evaluate to true.

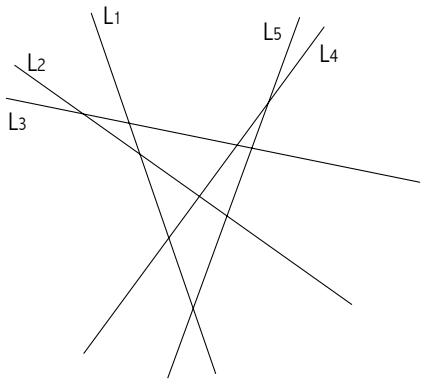


Figure 1: An instance of hidden surface removal with five lines (labeled L_1 to L_5 in the figure). All the lines except for L_2 are visible.

Consider a 3SAT instance with the following special locality property. Suppose there are n variables in the Boolean formula, and that they are numbered $1, 2, \ldots, n$ in such a way that each clause involves variables whose numbers are within ± 10 of each other. Give a linear-time algorithm for solving such an instance of 3SAT.

Problem 19: k-th quantiles

The k-th quantiles of an n-element set are the $k-1$ order statistics that divide the sorted set into k equal-sized (to within 1.) Give an $O(n \lg k)$-time algorithm to list the k-th quantiles of a set.

The k-th quantiles of a sequence of elements of length n are its $\lfloor in/k \rfloor$-th smallest elements for $1 \leq i \leq k-1$.

The *order statistic* of rank j for a set of elements is the j-th smallest element in the set.

Problem 20: Hidden surface removal (KT 5.5)

Hidden surface removal is a problem in computer graphics that scarcely needs an introduction: when Woody is standing in front of Buzz, you should be able to see Woody but not Buzz; when Buzz is standing in front of Woody, ... well, you get the idea.

The magic of hidden surface removal is that you can often compute things faster than your intuition suggests. Here's a clean geometric example to illustrate a basic speed-up that can be achieved. You are given n nonvertical lines in the plane, labeled $L_1, L_2, \ldots, L_n$, with the i^{th} line specified by the equation $y = a_i x + b_i$. We will make the assumption that no three of the lines all meet at a single point. We say line L_i is *uppermost* at a given x-coordinate x_0 if its y-coordinate at x_0 is greater than the y-coordinates of all the other lines at $x_0 : a_i x_0 + b_i > a_j x_0 + b_j$ for all $j \neq i$. We say line L_i is *visible* if there is some x-coordinate at which it is uppermost – intuitively, some portion of it can be seen if you look down from "$y = \infty$".

Give an algorithm that takes n lines as input and in $O(n \log n)$ time returns all the ones that are visible. Figure 1 gives an example.

Problem 21: 132 pattern

Given a sequence of n distinct positive integers $a_1, \ldots, a_n$, a 132-pattern is a subsequence a_i, a_j, a_k such that $i < j < k$ and $a_i < a_k < a_j$. For example: the sequence $31, 24, 15, 22, 33, 4, 18, 5, 3, 26$ has several 132-patterns including 15, 33, 18 among others. Design an algorithm that takes as input a list of n numbers and checks whether there is a 132-pattern in the list.

Problem 22: GCD (DPV)

Consider the following divide-and-conquer approach for finding the greatest common divisor of two positive integers.

1. Show that the following rule is true.

$$\gcd(a,b) = \begin{cases} 2\gcd(a/2, b/2) & \text{if } a, b \text{ are even} \\ \gcd(a, b/2) & \text{if } a \text{ is odd and } b \text{ are even} \\ \gcd((a-b)/2, b) & \text{if } a, b \text{ are odd} \end{cases}$$

2. Give an efficient divide-and-conquer algorithm for greatest common divisor.

3. How does the efficiency of your algorithm compare to Euclid's algorithm if a and b are n-bit integers? (In particular, since n might be large you cannot assume that basic arithmetic operations like addition take constant time.)

Problem 23: Weighted median

For n distinct elements $x_1, x_2, \cdots, x_n$ with positive weights $w_1, w_2, \cdots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the weighted median is the element x_k satisfying

$$\sum_{x_i < x_k} w_i \leq 1/2 \text{ and } \sum_{x_i > x_k} w_i \leq 1/2$$

1. Argue that the median of $x_1, x_2, \cdots, x_n$ is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \cdots, n$.

2. Show how to compute the weighted median of n elements in $O(n \lg n)$ worst-case time using sorting.

3. Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm.

Problem 24: Closest pair

Let $p_i = (a_i, b_i)$ for $1 \leq i \leq n$ be points in the plane. For points $p_i = (a_i, b_i)$ and $p_j = (a_j, b_j)$, we define the distance betwee them to be $d(p_i, p_j) = \sqrt{(a_i - a_j)^2 + (b_i - b_j)^2}$. Find $1 \leq i \neq j \leq n$ such that the distance between them is the minimum.

Problem 25: Polynomial multiplication in subquadratic time

Give a divide-and-conquer algorithm for multiplying polynomials of degree $n-1$ in the same variable, $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$ and $q(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$. Assume that the inputs are each given as arrays of integers of dimension n, and assume that n is a power of 2. Also assume that integer operations on the co-efficients are $O(1)$ time. Your output should be a dimension $2n-1$ array representing the polynomial $r = c_0 + c_1 x + \cdots c_{2n-2} x^{2n-2}$ so that $r(x) = p(x)q(x)$. Can you speed your algorithm up by using tricks like in integer multiplication?

Problem 26: Maximum weight subtree

The maximum weight subtree problem is as follows. You are given a tree T together with (not necessarily positive) weights $w(i)$ for each node $i \in T$. A subtree of T is any connected subgraph of T, (so a subtree is not necessarily the entire subtree rooted at a node.) You wish to find a subtree S of T that maximizes $\sum_{i \in S} w(i)$. Give the fastest algorithm you can to solve this problem. Note that there is a linear (in the number of vertices of the tree) time algorithm for this problem.

Problem 27: Tree isomorphism

Give an efficient algorithm to decide, given two rooted binary trees T and T', whether T and T' are isomorphic as rooted trees.

Problem 28: Binary tree isomorphism

Two rooted trees T_1 and T_2 are *isomorphic* if there is a 1-1 onto map $f : T_1 \to T_2$ so that $f(root_1) = f(root_2)$ and $p_2(f(x)) = f(p_1(x))$, for every $x \in T_1$ except $root_1$. (Here, $root_1$ is the root of T_1, $root_2$ is the root of T_2, and p_1, p_2, represent the parents in the respective trees.) Give an efficient algorithm to determine whether two n node rooted binary trees are isomorphic. (Hint: it is possible to do better than $O(n^2)$. Note that f is NOT GIVEN AS INPUT– your job is to find th isomporphism if it exists. Binary tree means that each node has at most two children, say $left(x)$ and $right(x)$ where one or more could be null. Do not assume the trees are balanced.)

Problem 29: Detecting triangles

Let G be an undirected graph with nodes $v_1, ..v_n$. The *adjacency matrix* for G is the $n \times n$ matrix M given by: $M_{i,j} = 1$ if there is an edge from v_i to v_j, and $M_{i,j} = 0$ otherwise. A useful property of this matrix is that, if M is the adjacency matrix for G, there is a path of length k from v_i to v_j iff the i, j'th entry of M^k is positive. Use this fact to design a fast algorithm for telling if a graph contains a triangle, three nodes that are all connected.

Problem 30: Column minima

Let $M[i, j]$ be a matrix of numbers for $1 \leq i, j \leq n$. Let $\mu(j)$ is the row index of the minimum element in column j. Compute $\mu(j)$ for all $1 \leq j \leq n$.

Show that the funciton μ can be computed in linear time if the following conditions hold

1. Given $1 \leq i, j \leq n$, $M[i, j]$ can be computed in constant time.
2. The matrix M is totally monotone, i.e., for all $i < j$ and $k < l$

$$M[i, k] + M[j, l] \leq M[i, l] + M[j, k]$$

2.1 Searching

Problem 31: Lighter or Heavier

You have $n > 2$ identical-looking coins and a two-pan balance scale with no weights. One of the coins is a fake, but you do not know whether it is lighter or heavier than the genuine coins, which all weigh the same. Design an algorithm to determine in the minimum number of weighings whether the fake coin is lighter or heavier than the others.

Problem 32: Finding the peak in a unimodal sequence

Let $a_i, \ldots, a_n$ be a sequence of distinct integers. Moreover, the sequence is unimodal, that is, there is an index $1 \leq p \leq n$ such that the values a_i increase up to position p and decrease from then on. Design an efficient algorithm to find the peak value.

Problem 33: A Fake among n Coins

There are $n \geq 1$ identical-looking coins; one of these coins is counterfeit and is known to be lighter than the genuine coins. All genuine coins have the same weight. What is the minimum number of weighings needed to identify the fake coin with a two-pan balance scale without weights? Describe your algorithm to identify the counterfeit. How many comparisons (as a function of n) does your algorithm require in the worst case?

Problem 34: A Stack of Fake Coins

There are 10 stacks of 10 identical-looking coins. All of the coins in one of these stacks are counterfeit, and all the coins in the other stacks are genuine. Every genuine coin weighs 10 grams, and every fake weighs 11 grams. You have an analytical scale that can determine the exact weight of any number of coins. What is the minimum number of weighings needed to identify the stack with the fake coins?

Problem 35: Ternary searching

Write an algorithm that searches a sorted list of n items by dividing it into three sublists of almost $n/3$ items. This algorithm finds the sublist that might contain the given item, and divides it into three smaller sublists of almost equal size. The algorithm repeats this process until it finds the item or concludes that the item is not in the list. Analyze your algorithm, and give the results using order notation.

Problem 36: Fixpoint

Given a sorted array of distinct integers $A[1 \ldots n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

Problem 37: Finding the missing integer

An array $A[1..n]$ contains all the integers from 0 to n except one. It would be easy to determine the missing integer in $O(n)$ time by using an auxiliary array $B[0..n]$ to record which numbers appear in A. In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of $A[i]$," which takes constant time.

Show that if we use only this operation, we can still determine the missing integer in $O(n)$ time.

Problem 38: Searching in a fuzzy sorted list

Given an array of integers $A[1..n]$, such that, for all $i, 1 \leq i < n$, we have $|A[i] - A[i+1]| \leq 1$. Let $A[1] = x$ and $A[n] = y$, such that $x < y$. Design an efficient search algorithm to find j such that $A[j] = z$ for a given value $z, x \leq z \leq y$. If no there is no such j, declare as such. Analyze its worst case time complexity (in terms of the number of comparisons).

Problem 39: Searching in a rotated array

Given a sorted array of n elements that has been rotated an unknown number of times, develop an algorithm to find an element in the array. You may assume that the array was sorted originally in increasing order.

Problem 40: Two sum in sorted lists

Consider the following recursive algorithm. Its inputs are two *sorted* n element arrays A and B (so the algorithm assumes $A[1] < A[2] < \cdots < A[n]$ and $B[1] < B[2] < \cdots < B[n]$) and a target T. The following algorithm computes whether $T = A[I] + B[J]$ for some $1 \leq I, J \leq n$. Give a recurrence relation for the time the algorithm takes, and give a brief explanation of your answer. Use the recurrence to give a time analysis.

1. Program: TargetSum(A[1..n], B[1..n]: Arrays of Integers, T: integer):
2. IF $n = 0$ return False;
3. IF $n = 1$
   THEN IF $T = A[1] + B[1]$
   THEN return True;
   ELSE return False;
4. $k \leftarrow \lceil n/2 \rceil$
5. $k' \leftarrow \lceil (n+1)/2 \rceil$
6. IF $A[k] + B[k] = T$ THEN return True;
7. IF $A[k] + B[k] > T$ THEN return
   ($TargetSum(A[1..k], B[1..k], T)$
   OR $TargetSum(A[1..k], B[k'..n], T)$
   OR $TargetSum(A[k'..n], B[1..k], T)$);
8. ELSE return
   ($TargetSum(A[k'..n], B[1..k], T)$ OR
   $TargetSum(A[1..k], B[k'..n], T)$ OR
   $TargetSum(A[k'..n], B[k'..n], T)$);

Here, OR is the Boolean OR operation on the values of the returned calls.

Example: say $n = 4, A[1..4] = 3, 8, 13, 21; B[1..4] = 1, 2, 3, 5; T = 13$. Then the algorithm would first compare $A[2] + B[2] = 10$ to T. Since it is smaller, it would recursively check TargetSum[A[1..2],B[3..4],13], TargetSum[A[3..4],B[1..2],13] and TargetSum[A[3..4],B[3..4],13]. (It could ignore the case $I, J \leq 2$, since all such sums are less than T). After computing recursively, which I won't show here, the sub-procedures would return $True$, $False$ and $False$. Thus, the main procedure would return $True$.

Problem 41: The Josephus Problem

We have n people numbered (clockwise) 1 to n around a circle. We eliminate every second person until only one survives. Determine the survivor's number.

Problem 42: Median of $2n$ elements

Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.

Problem 43: k-th smallest element

You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the k'th smallest element in the union of the two lists.

Problem 44: Sorted matrix search

Given an $m \times n$ matrix in which each row and column is sorted in ascending order, design an algorithm to find an element.

Problem 45: Local minimum (KT 5.7)

Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, l) are joined by an edge if and only if $|i - k| + |j - l| = 1$.)

Each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum (A node v of G is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.) of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Problem 46: Stress testing (KT 2.8)

You are doing some stress-testing on various models of glass jars to determine the height from which they can be dropped and still not break. The setup for this experiment, on a particular type of jar, is as follows. You have a ladder with n rungs, and you want to find the highest rung from which you can drop a copy of the jar and not have it break. We call this the *h*ighest safe rung.

It might be natural to try binary search: drop a jar from the middle rung, see if it breaks, and then recursively try from rung $n/4$ or $3n/4$ depending on the outcome. But this has the drawback that you could break a lot of jars in finding the answer.

If your primary goal were to conserve jars, on the other hand, you could try the following strategy. Start dropping a jar from the first rung, then the second rung, and so forth, climbing one higher each time until the jar breaks. In this way, you only need a single jar — at the moment it breaks, you have the correct answer — but you may have to drop it n times (rather than $\log n$ as in the binary search solution).

So, here is the trade-off: it seems that you can perform fewer drops if your are willing to break more jars. To understand better how this trade-off works at a quantitative level, let us consider how to run this experiment given a fixed *b*udget of $k \geq 1$ jars. In other words, you have to determine the correct answer — the highest stage rung — and can use at most k jars in doing so.

1. Suppose you are given a budget of $k = 2$ jars. Describe a strategy for finding the highest safe rung that requires you to drop a jar at most $f(n)$ times, for some function $f(n)$ that grows slower than linearly. In other words, it should be the case that $\lim_{n\to\infty} f(n)/n = 0$.

2. Now suppose you have a budget of $k > 2$ jars, for some given k. Describe a strategy for finding the highest safe rung using at most k jars. If $f_k(n)$ denotes the number of items you need to drop a jar according to your strategy, then the functions $f, f_2, f_3, \ldots$ should have the property that each grows asymptotically slower than the previous one: $\lim_{n\to\infty} f_k(n)/f_{k-1}(n) = 0$ for each k.

3 Sorting and Selection

Problem 47: Pancake Sorting

There are n pancakes, all of different sizes, that are stacked on top of each other. You are allowed to slip a spatula under one of the pancakes and flip over the whole stack above the spatula. The objective is to arrange the pancakes according to their size with the biggest at the bottom. Design an algorithm for solving this puzzle and determine the number of flips made by the algorithm in the worst case.

Problem 48: Insertion sort

Insertion sort can be expressed as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively

sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write a recurrence for the running time of this recursive version of insertion sort.

Problem 49: Almost sorted arrays

Suppose we are given an 'almost' sorted list $x_1, x_2, \ldots, x_n$ with the property that the correct position of each element is at most a distance of k away from the current position.

More formally, let $A[1..n]$ be an array of integers, and let $A[\sigma(1)] \leq A[\sigma(2)] \leq ...A[\sigma(n)]$ be its sorted order. Say A is *k-almost sorted* if for every i, $|i - \sigma(i)| \leq k$. Show that, in the comparison model, the problem of sorting a k-almost sorted list of size n takes $\Theta(n \lg k)$ comparisons. (You must provide an $O(n \lg k)$ algorithm, and also prove an $\Omega(n \lg k)$ lower bound for an arbitrary comparison algorithm.)

Problem 50: Modular stable sort

Let $a_1, \cdots, a_n$ be a sequence of elements and let p and q be positive integers. Consider the subsequences formed by selecting every p-th element. Sort these subsequences. Repeat the process for q. Prove that the subsequences of distance p remain sorted.

Problem 51: k-way merging

Give an $O(n \lg k)$ algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. Hint: Use a heap for k-way merging. Analyze the time complexity of your algorithm.

Prove a tight lower bound on the number of comparisons for k-way merging.

Problem 52: Sorting variable length integers

You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over all the integers in the array is n. Show how to sort the array in $O(n)$ time.

Problem 53: Faster sorting

The input is a sequence of n integers with many duplications, such that the number of distinct integers in the sequence is $O(\lg n)$.

1. Design a sorting algorithm to sort such sequences using at most $O(n \lg \lg n)$ comparisons in the worst case.
2. Why is the lower bound of $\Omega(n \lg n)$ not satisfied in this case?

Problem 54: Comparisons for i extreme elements

Suppose that an algorithms uses only comparisons to find the ith smallest element in a set of n elements. Show that it can also find the $i-1$ smaller elements and the $n-i$ larger elements without performing any additional comparisons.

Problem 55: Close to median

Describe an $O(n)$-time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S.

Problem 56: Combine intervals

Given a set of time intervals in any order, merge all overlapping intervals into one. Output the result,

which should have only mutually exclusive intervals. Let the intervals be represented as pairs of integers for simplicity. For example, the intervals $[1, 5]$, $[3, 8]$ and $[6, 10]$ should be merged into $[1, 10]$.

Problem 57: Matrix stable sort

Consider a rectangular array. Sort the elements in each row into increasing order. Next sort the elements in each column into increasing order. Prove that the elements in each row remain sorted.

4 Sorting or selection as a preprocessing step

Problem 58: Disjoint sets

Design an algorithm to determine whether two sets of numbers (presented as arrays) are disjoint. State the complexity of your algorithm in terms of the sizes m and n of the given sets. Make sure to consider the case where m is substantially smaller than n.

Problem 59: Minmax

Take as input a sequence of $2n$ real numbers. Design an $O(n \lg n)$ algorithm that partitions the numbers into n pairs, with the property that the partition minimizes the maximum sum of a pair.

Problem 60: Duplicates (DPV book)

You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$.

Problem 61: Good soldier Schweik

The good solider Schweik had been ordered to line up a band of new recruits before their officer gave them a speech. The desired line sought to miminize the sum total of differences of heights of adjacent recruits in the line. Schweik put the tallest recruit first, the shortest last, and let the remaining recruits stand between them in an arbitrary order.

1. Did Schweik execute his order as stated? Prove or provide a counter example.
2. How would you arrange the recruits to minimize the sum of the differences of adjacent recruits? Argue the correctness of your algorithm.
3. Argue the correctness of your solution.

Example: Assume that the recruits are arranged so that their heights form the following sequence: $4, 8, 3, 15, 1$. Then the sum of the differences of heights of adjacent recruits would be $4 + 5 + 12 + 14 = 35$.

Problem 62: Two-sum

The input is an array A containing n real numbers, and a real number x.

1. Design an algorithm to determine whether there are two elements of A whose sum is exactly x. The algorithm should run in time $O(n \lg n)$.
2. Suppose now that the array A is given in a sorted order. Design an algorithm to solve this problem in time $O(n)$.

Problem 63: Summing triples

Let $A[1..n]$ be an array of positive integers. A *summing triple* in A is 3 distinct indices $1 \leq i, j, k \leq n$ so that $A[i] + A[j] = A[k]$. Give and analyze an algorithm that, given A, determines whether there is any summing triple in A.

Problem 64: Largest set of indices within a given distance

You are given a sequence of numbers $a_1, \ldots, a_n$ in an array. You are also given a number k. Design an efficient algorithm to determine the size of the largest subset $L \subseteq \{1, 2, \cdots, n\}$ of indices such that for all $i, j \in L$ the difference between a_i and a_j is less than or equal to k. There is an $O(n \log n)$ algorithm for this problem.

For example, consider the sequence of numbers $a_1 = 7, a_2 = 3, a_3 = 10, a_4 = 7, a_5 = 8, a_6 = 7, a_7 = 1, a_8 = 15, a_9 = 8$ and let $k = 3$. $L = \{1, 3, 4, 5, 6, 9\}$ is the largest such set of indices. Its size is 6.

Problem 65: k-sum

Given a set S of n integers and an integer T, give an $O(n^{k-1} \lg n)$ algorithm to test whether k of integers in S sum up to T.

Problem 66: Olay

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. From each well, a spur pipeline is to be connected directly to the main pipeline along a shortest path (either north or south). Given x and y coordinates of the wells, how should the professor pick the optimal location of the main pipeline (the one that minimizes the the total length of the spurs?) Show that the optimal location can be determined in linear time.

5 Heaps

Problem 67: Second smallest element

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

Problem 68: i-th largest element

Show that, for any constant i, the i-th largest element of an array can be found with $n+O(\lg n)$ comparisons.

Problem 69: k-th largest in a heap

The input is a heap of size n (in which the largest element in on top), given as an array, and a real number x. Design an algorithm to determine whether the kth largest element in the heap is less than or equal to x. The worst-case running time of your algorithm should be $O(k)$, independent of the size of the heap.

Problem 70: k-th smallest element in $O(k)$ space

The input is a sequence of elements $x_1, x_2, \ldots, x_n$, given one at a time. Design an $O(n)$ expected time algorithm to compute the kth smallest element using only $O(k)$ memory cells. The value of k is known ahead of time, but the value of n is not known until the last element is seen.

6 Fast Fourier Transform

Problem 71: Fast Fourier transform (Problem 2.8, page 72, DPV book)

Practice with the fast Fourier transform.

1. What is the FFT of (1,0,0,0)? What is the appropriate value of ω in this case? And of which sequence is (1,0,0,0) the FFT?
2. Repeat for (1,0,1,-1).

Problem 72: Polynomial multiplication (Problem 2.9, page 72, DPV book)

Practice with polynomial multiplication by FFT.

1. Suppose that you want to multiply the two polynomials $x + 1$ and $x^2 + 1$ using the FFT. Choose an appropriate power of two, find the FFT of the two sequences, multiply the results componentwise, and compute the inverse FFT to get the final result.
2. Repeat for the pair of polynomials $1 + x + 2x^2$ and $2 + 3x$.

Problem 73: Hadamard matrix vector multiplication (DPV)

The *Hadamard* matrices $H_0, H_1, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix [1].
- For $k > 0$, H_k is the $2^k \times 2^k$ matrix

$$\begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

Show that v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

Problem 74: Interpolation (Problem 2.10, page 72, DPV book)

Find the unique polynomial of degree 4 that takes on values $p(1) = 2$, $p(2) = 1$, $p(3) = 0$, $p(4) = 4$ and $p(5) = 0$. Write your answer in the coefficient representation.

Problem 75: Fourier transform in modular arithmetic (DPV)

This problem illustrates how to Fourier Transfrom (FT) in modular arithmetic, for example, modulo 7.

(a). There is a number ω such that all powers $\omega, \omega^2, \ldots, \omega^6$ are distinct (modulo 7). Find this ω, and show that $\omega + \omega^2 + \cdots + \omega^6 = 0$. Interestingly, for any prime modulus there is such a number.

(b). Using the matrix form of FT, produce the transform of the sequence $(0, 1, 1, 1, 5, 2)$ modulo 7; that is, multiply this vector by the matric $M_6(\omega)$, for the value of ω you found earlier. In the matrix multiplication, all calculations should be performed modulo 7.

(c). Write down the matrix necessary to perform the inverse FT. Show that multiplying by this matrix returns the original sequence. Again all arithmetic should be performed modulo 7.

(d). Now show how to multiply the polynomials $x^2 + x + 1$ and $x^3 + 2x - 1$ using the FT modulo 7.

Problem 76: FFT using modular arithmetic

As defined, the Discrete Fourier Transform requires the use of complex number, which can result in a loss of precision due to round-off errors. For some problems, the answer is known to contain only integers, and it is

desirable to utilize a variant of the FFT based on modular arithmetic in order to guarantee that the answer is calculated exactly. An example of such a problem is that of multiplying two polynomials with integer coefficients. Exercise 30.2-6 (CLRS) gives one approach, using a modulus of length $\Omega(n)$ bits to handle a DFT on n points. The problem gives another approach that uses a modulus of the more reasonable length $O(\lg n)$; it require that you understand the material of Chapter 31 (CLRS). Let n be a power of 2.

1. Suppose that we search for the smallest k such that $p = kn + 1$ is prime. Give a simple heuristic argument why we might expect k to be approximately $\lg n$. (The value of k might be much larger or smaller, but we can reasonably expect to examine $O(\lg n)$ candidate values of k on average.) How does the expected length of p compare to the length of n?

   Let g be a generator of $\mathbb{Z}_p^*$, and let $w = g^k \mod p$.

2. Argue that the DFT and inverse DFT are well-defined inverse operations modulo p, where w is used as a principal root of unity.

3. Argue that the FFT and its inverse can be made to work modulo p in time $O(n \lg n)$, where operations on words of $O(\lg n)$ bits take unit time. Assume that the algorithm is given p and w.

4. Compute the DFT modulo $p = 17$ of the vector $(0, 5, 3, 7, 7, 2, 1, 6)$. Note that $g = 3$ is a generator of $\mathbb{Z}_{17}^*$.

Problem 77: Polynomial multiplication

Give a divide-and-conquer algorithm for multiplying polynomials of degree $n - 1$ in the same variable, $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{n-1}x^{n-1}$ and $q(x) = b_0 + b_1x + b_2x^2 + \ldots + b_{n-1}x^{n-1}$. Assume that the inputs are each given as arrays of integers of dimension n, and assume that n is a power of 2. Also assume that integer operations on the co-efficients are $O(1)$ time. Your output should be a dimension $2n - 1$ array representing the polynomial $r = c_0 + c_1x + ...c_{2n-2}x^{2n-2}$ so that $r(x) = p(x)q(x)$. Can you speed your algorithm up by using tricks like in integer multiplication?

Problem 78: Probabilities for multi-faceted dice

There are N dice, and the i-th die has its faces numbered from 1 to k_i where each k_i is a positive integer less than K. Assuming that on each die, each number is equally likely, we want to find the probability that when we roll all N dice, the sum of the numbers is exactly T. To do so, we need to count exactly how many ways we can choose integers $d_1, d_2, \ldots, d_N$ with $d_i \leq k_i$, such that $\sum d_i = T$. (The probability is this count divided by the product of the k_i's.)

Find an efficient algorithm, and give its time analysis in terms of N and K.

Problem 79: Dice pools

This problem arises from calculating success probabilities for certain role-playing games, where players roll dice in proportion to their character's abilities, and each die is either a "Success ", a "Failure" or "Neutral", and the outcome is determined by the number of successes minus the number of failures. (For example, in one game, dice take random values from 1 to 10, with 1 being a "Failure" and 8-10 being a "Success".) Abstractly, the problem is: there are n independent random variables, $X_1...X_n$. Each variable is $+1$ with probability p, -1 with probability q and 0 otherwise, where $0 \leq p, q \leq 1$ and $p + q \leq 1$. (In the above example, $p = 3/10, q = 1/10$.) We want to calculate, given n, an array of probabilities: for all k with $-n \leq k \leq n$ compute the probability that $\sum_{i=1}^{i=n} X_i = k$. Your algorithm should be polynomial-time in n. Assume arithmetic operations are constant time.

Problem 80: Base conversion

Give an algorithm that inputs an array of n base b_1 digits representing a positive integer in base b_1 and

outputs an array of base b_2 digits representing the same integer in base b_2. Get as close as possible to linear time. Assume b_1, b_2 are fixed constants.

If we do the conversion digit-by-digit, the number of operations is really $O(n^2)$, since, for example, once we convert each digit we need to add up all the resulting $O(n)$ bit numbers. We can do better using a divide-and-conquer algorithm using a FFT integer multiplication algorithm as a sub-routine. Assume n is a power of 2; otherwise, pad with 0's in the most significant digits.

Problem 81: Cartesian sum

Consider two sets A and B, each having n integers in the range from 0 to $10n$. We wish to compute the *Cartesian sum* of A and B, defined by

$$C = \{x + y : x \in A \text{ and } y \in B\}.$$

Note that the integers in C are in the range from 0 to $20n$. We want to find the elements of C and the number of times each element of C is realized as a sum of elements in A and B. Show that the problem can be solved in $O(n \lg n)$ time. (*Hint:* Represent A and B as polynomials of degree at most $10n$.)

Problem 82: Coulomb forces (KT 5.4)

You've been working with some physicists who need to study, as part of their experimental design, the interactions among large numbers of very small charged particles. Basically, their setup works as follows. They have an inert lattice structure, and they use this for placing charged particles at regular spacing along a straight line. Thus we can model their structure as consisting of the points $\{1, 2, 3, \ldots, n\}$ on the real line; and at each of these points j, they have a particle with charge q_j. (Each charge can be either positive or negative.)

They want to study the total force on each particle, by measuring it and then comparing it to a computational prediction. This computational part is where they need your help. The total net force on particle j, by Coulomb's Law, is equal to

$$F_j = \sum_{i<j} \frac{Cq_iq_j}{(j-i)^2} - \sum_{i>j} \frac{Cq_iq_j}{(j-i)^2}.$$

They've written the following simple program to compute F_j for all j:

---

for $j = 1, 2, \ldots, n$ **do**
  Initialize F_j to 0
  for $i = 1, 2, \ldots, n$ **do**
    if $i < j$ **then**
      Add $\frac{Cq_iq_j}{(j-i)^2}$ to F_j
    else if $i > j$ **then**
      Add $-\frac{Cq_iq_j}{(j-i)^2}$ to F_j
    end if
  end for
  Output F_j
end for

It's not hard to analyze the running time of this program: each invocation of the inner loop, over i, takes $O(n)$ time, and this inner loop is involved $O(n)$ times total, so the overall running time is $O(n^2)$.

The trouble is, for the large values of n they're working with, the program takes several minutes to run. On the other hand, their experimental setup is optimized so that they can throw down n particles, perform the measurements, and be ready to handle n more particles within a few seconds. So they'd really like it if there were a way to compute all the forces F_j much more quickly, so as to keep up with the rate of the experiment.

Help them out by designing an algorithm that computes all the forces F_j in $O(n \log n)$ time.

Problem 83: Evaluating all derivatives

Given a polynomial $A(x)$ of degree bound n, its t^{th} derivative is defined by

$$A^{(t)}(x) = \begin{cases} A(x) & \text{if } t = 0, \\ \frac{d}{dx}A^{(t-1)}(x) & \text{if } 1 \le t \le n-1, \\ 0 & \text{if } t \ge n. \end{cases}$$

From the coefficient representation $(a_0, a_1, \ldots, a_{n-1})$ of $A(x)$ and a given point x_0, we wish to determine $A^{(t)}(x_0)$ for $t = 0, 1, \ldots, n$.

1. Given coefficients $b_0, b_1, \ldots, b_{n-1}$ such that

$$A(x) = \sum_{j=0}^{n-1} b_j (x - x_0)^j,$$

   show how to compute $A^{(t)}(x_0)$, for $t = 0, 1, \ldots, n-1$, in $O(n)$ time.

2. Explain how to find $b_0, b_1, \ldots, b_{n-1}$ in $O(n \lg n)$ time, given $A(x_0 + \omega_n^k)$ for $k = 0, 1, \ldots, n-1$.

3. Prove that

$$A(x_0 + \omega_n^k) = \sum_{r=0}^{n-1} \left(\frac{\omega_n^{kr}}{r!} \sum_{j=0}^{n-1} f(j) g(r-j) \right),$$

   where $f(j) = a_j \cdot j!$ and

$$g(l) = \begin{cases} x_0^{-l}/(-l)! & \text{if } -(n-1) \le l \le 0, \\ 0 & \text{if } 1 \le l \le (n-1). \end{cases}$$

4. Explain how to evaluate $A(x_0 + \omega_n^k)$ for $k = 0, 1, \ldots, n-1$ in $O(n \lg n)$ time. Conclude that all nontrivial derivatives of $A(x)$ can be evaluated at x_0 in $O(n \lg n)$ time.

Problem 84: Polynomial roots

Given a list of values $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions), show how to find the coefficients of a polynomial $P(x)$ of degree bound $n+1$ that has zeros only at $z_0, z_1, \ldots, z_{n-1}$ (possibly with repetitions). Your procedure should run in time $O(n \lg^2 n)$. (*Hint:* The polynomial $P(x)$ has a zero at z_j if and only if $P(x)$ is a multiple of $(x - z_j)$.)

Problem 85: Toeplitz matrices

A *Toeplitz matrix* is an $n \times n$ matrix $A = (a_{ij})$ such that $a_{ij} = a_{i-1,j-1}$ for $i = 2, 3, \ldots, n$ and $j = 2, 3, \ldots, n$.

1. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?

2. Describe how to represent a Toeplitz matrix so that two $n \times n$ Toeplitz matrices can be added in $O(n)$ time.

3. Give an $O(n \lg n)$-time algorithm for multiplying an $n \times n$ Toeplitz matrix by a vector of length n. Use your representation from part (b).

4. Give an efficient algorithm for multiplying two $n \times n$ Toeplitz matrices. Analyze its running time.

7 Lower Bounds

Problem 86: Communication complexity

Let A and B two sets with both with n elements, such that A resides in computer P and B in Q. P and Q can communicate by sending messages, and they can perform any kind of local computation. Design an algorithm to find the nth smallest element of the union of A and B (that is, the median). You can assume that all the elements are distinct. Your goal is to minimize the number of messages, where a message can contain one element or one integer. What is the number of messages in the worst case?

Problem 87: Max and min

Show that $\lceil 3n/2 \rceil - 2$ comparisons are necessary in the worst case to find both the maximum and minimum of n numbers. (Hint: Consider how many numbers are potentially either the maximum or minimum, and investigate how a comparison affects both these counts.)

Problem 88: Lower bound on merging

Show a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items.

Problem 89: Merging k lists

Give tight upper and lower bounds in the comparison model for merging k sorted lists with a total of n elements.

Problem 90: Not-in-the-set

The input is an array A of n real numbers. Design an $O(n)$ algorithm to find a number that is not in the array A. Prove that $\Omega(n)$ is a lower bound on the number of steps required to solve this problem.

8 Miscellaneous

Problem 91: Bachet's Weights

Find an optimal set of n weights $\{w_1, w_2, \cdots, w_n\}$ so that it would be possible to weigh on a two-pan balance scale any integral load in the largest possible range from 1 to W, assuming the following:

1. Weights can be put only on the free pan of the scale.
2. Weights can be put on both pans of the scale.

Problem 92: Sum to zero

Given a set of integers $S = \{x_1, x_2, \ldots, x_n\}$, find a nonempty subset $R \subseteq S$ such that

$$\sum_{x_i \in R} x_i = 0 \pmod{n}.$$

9 Solved Problems

Problem 93: Tromino Puzzle

Cover a $2^n \times 2^n$ ($n \geq 1$) board missing one square with right tromnioes, which are L-shaped tiles formed by three adjacent squares. The missing square can be any of the board squares. Trominoes should cover all the squares except the missing ones with no overlaps.

A right tromino can also be viewed as a 2×2 board with exactly one missing square.

Solution: Tromino Puzzle

High-level description

We will present an algorithm that covers a $2^n \times 2^n$ board with a missing square with L-shaped trominoes for all $n \geq 1$.

If $n = 1$, we have a 2×2 square with a missing square, which can be covered with one tromino.

If $n \geq 2$, we divide the board into four smaller boards, each of size $2^{n-1} \times 2^{n-1}$. However, of the four smaller boards, three of them do not miss any squares. We use an appropriately oriented tromino to cover the three corner squares (which meet at the center of the board) of the three smaller boards (the ones that do not miss any squares). We now have four smaller boards, each with exactly one missing square. We cover them recursively with L-shaped trominoes.

Problem 94: Coin Removal

There is a line of n coins on the table; some of them are heads up and the rest are tails up, in no particular order. The object of the puzzle is to remove all the coins by a sequence of moves. On each move, one can remove any head-up coin, after which its neighboring coin or coins, if any, must be turned over. Coins are considered "neighbors" if they are next to each other in the original line; if there is a gap between coins after some moves, the coins are no longer considered neighbors.

Determine the property of the starting line that is necessary and sufficient for the puzzle to have a solution. For those lines that can be removed by the puzzle's rules, design an algorithm for doing so.

Solution: Coin removal

Characterization of the existence of solutions

We will show that all coins can be removed if and only if there are an odd number of coins which are heads-up. Implicit in the argument is an algorithm for removing all coins when it is possible to do so.

Let us first define a few terms. We call a *contiguous sequence* of coins a *string* of coins. An empty string is a string with no coins in it. When we remove a coin from a string, we get two strings, the string to the left of the removed coin and the string to the right of the removed coin. Any one of the two strings can be empty.

Number of heads-up coins is even: We argue that if there are an even number of coins which are heads-up in any non-empty string of coins, then it is not possible to remove all coins using the coin removal rule.

Consider a non-empty string of coins such that no coin is head-up. All coins in such a string are tails-up. Hence, no coin removal step can take place. Since the string in non-empty, we will not succeed in removing all coins using the coin removal rule.

Now consider the case when the string has k coins which are heads-up such that k is even and $k \geq 2$. If we remove any head from such a string, it must result in a non-empty string of smaller size with an even number coins which are heads-up. By inductive hypothesis, we argue that we cannot remove all coins from such a string.

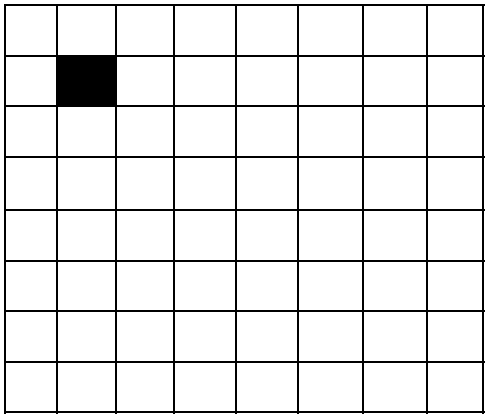
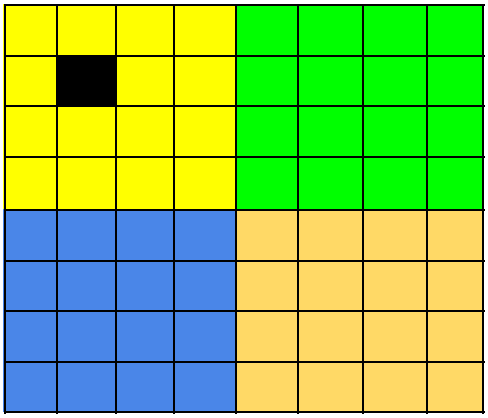


Figure 2: Tromino Puzzle

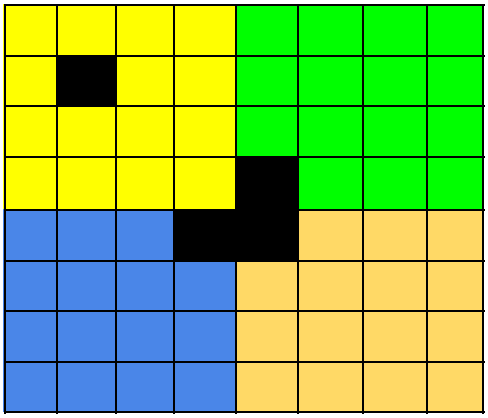


8 x 8 Board with one missing square

Division into 4 equal quadrants (4 x 4 boards)

The subproblems of covering the quadrants are not exactly the same type as the original problem

Figure 3: Division into four subproblems- I



8 x 8 Board with one missing square

Division into 4 equal smaller size boards (4 x 4 boards)

The subproblems of covering the smaller boards are not exactly the same type as the original problem since three of them do not have a missing square

Use a tromino to cover the three corner squares of the three smaller boards which do not have a missing square. The three corner squares meet at the center of the board.

Figure 4: Division into four subproblems- II

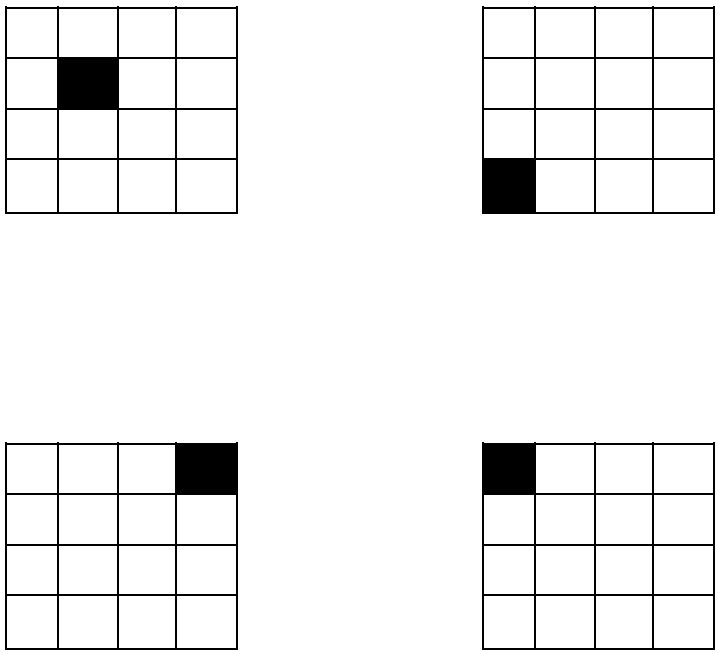


Figure 5: Division into four subproblems- III

Therefore, we conclude that we cannot succeed in removing all coins from a non-empty string of coins with an even number of coins heads-up.

Number of heads-up coins is odd: Let x be a string of coins with an odd-number of coins heads-up. Consider the left-most coin which is head-up. There must exist such a coin since there is at least one coin which is head-up. Let x_L be the string to the left of the left-most head-up coin and x_R be the string to the right of the left-most head-up coin. x_L and x_R both must have an even number of coins which are heads-up. If x_L is non-empty, let x'_L be the string obtained after flipping the right-most coin in x_L. x'_L must have an odd number of coins which are heads-up and we deal with x'_L recursively.

Similarly, if x_R is non-empty, let x'_R be the string obtained after flipping the left-most coin in x_R. x'_R must have an odd number of coins which are heads-up and we deal with x'_R recursively.

Problem 95: Lighter or Heavier

You have $n > 2$ identical-looking coins and a two-pan balance scale with no weights. One of the coins is a fake, but you do not know whether it is lighter or heavier than the genuine coins, which all weigh the same. Design an algorithm to determine in the minimum number of weighings whether the fake coin is lighter or heavier than the others.

Solution: Ligther or Heavier

A suboptimal solution: Create 3 groups of $\lfloor \frac{n}{3} \rfloor$ coins each. Call these groups A, B, and C. The remaining coins form the group E. E contains 0, 1, or 2 coins. Compare groups A and B and groups A and C. Comparison of any two groups of coins using a two-pan balance would produce one of three outcomes: the first group is lighter, the first group is heavier, or both groups have equal weight. The information obtained as a result of the two comparisons is best represented as a 3-way tree with $3 \times 3 = 9$ leaves. Each leaf corresponds to a set of possibilities. The following table captures the two-level 3-way tree. We use the notation $A < B$ to denote that the weight of coins in group A is less than the weight of the coins in group B. $A > B$ and $A = B$ are used with an analogous meaning.

A vs B	A vs C	Analysis
$A < B$	$A < C$	$B = C$. The fake coin is lighter and is in A
	$A = C$	$A = C$. The fake coin is heavier and is in B
	$A > C$	A, B, and C have distinct weights, which is impossible
$A = B$	$A < C$	$A = B$. The fake coin is heavier and is in C
	$A = C$	$A = B = C$. The fake coin is in E
	$A > C$	$A = B$. The fake coin is lighter and is in C
$A > B$	$A < C$	A, B, and C have distinct weights, which is impossible
	$A = C$	$A = C$. The fake coin is lighter and is n B
	$A > C$	$B = C$. The fake coin is heavier and is in A

For row 5, we concluded that all the coins in the groups A, B, and C are genuine and the fake coin is in the group E. To figure out whether the fake coin is lighter or heavier, take $|E|$ genuine coins from among the coins in A, B, and C. Compare this group with E to determine whether the fake coin is lighter or heavier. In total, we need a maximum of 3 comparisons.

An optimal solution: It turns out that we can devise an algorithm for this problem that uses just two comparisons.

Algorithm **DetermineLighterOrHeavier**: Let n be equal to $3k$ or $3k + 1$ or $3k + 2$ for some $k \geq 1$. This is without loss of generality since $n > 2$. If $n = 3k$ or $3k+1$, form two groups of k coins each. Otherwise (that is, if $n = 3k + 2$), form two groups of $k + 1$ coins each. In either case, place the remaining coins in the third group. Call these groups L, R, and E respectively. All groups have at least one coin since $k \geq 1$. We use the notation $|\cdot|$ to denote the size of a group. For example, $|L|$ denotes the number of coins in L.

Compare the weights of groups L and R. If the weights are equal (denoted by $L = R$), we know then that each coin in groups L and R is genuine. Select $|E|$ many coins from the groups L and R and form a

new group E'. Compare the weights of the groups E and E'. If $E < E'$, then the fake coin is lighter. If $E > E'$, the fake coin is heavier. It is not possible that $E = E'$. Observe that $|L| + |R| \geq |E|$ for all $n > 2$, so it is always possible to select $|E|$ many coins from the groups L and R together.

Now consider the case $L < R$. We know that each coin in E is genuine. If $|L|$ is odd, add a genuine coin from E to L. Without loss of generality, assume that $|L| \geq 2$ is even. Split L into two subgroups of equal size. Call the subgroups L' and L''. Compare L' and L''. If $L' = L''$, we infer that the fake coin is heavier since all the coins in L are genuine. If $L' < L''$ or $L' > L''$, we infer that the fake coin is lighter since otherwise we will have two fake coins.

The case $L > R$ can be handled similarly.

Thus, in call cases, we have shown that we need only two comparisons to determine whether the fake coin is lighter or heavier.

Problem 96: A Stack of Fake Coins

There are 10 stacks of 10 identical-looking coins. All of the coins in one of these stacks are counterfeit, and all the coins in the other stacks are genuine. Every genuine coin weighs 10 grams, and every fake weighs 11 grams. You have an analytical scale that can determine the exact weight of any number of coins. What is the minimum number of weighings needed to identify the stack with the fake coins?

Solution: A Stack of Fake Coins

One weighing is sufficient. Let $S_1, S_2, \ldots, S_{10}$ denote the ten stacks. Create a group of coins by selecting i coins from stack S_i for every $1 \leq i \leq 10$. Use the scale to determine their weight. If the weight is w, the counterfeit stack is then S_{w-550}.

There are $\frac{(10)(10+1)}{2} = 55$ coins in the group. Each genuine coin weighs 10 grams, so the entire group, would weigh 550 grams if every S_i is a stack of genuine coins. However, there are between 1 and 10 fake coins in the group depending on which stack contains the fake coins. For any $1 \leq i \leq 12$, if S_i is the stack of fake coins, the weight of the group would be $550 + i$. Hence, subtracting 550 from the weight of the group should give us the index of the stack of fake coins.

Problem 97: Local minimum (KT 5.7)

Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, l) are joined by an edge if and only if $|i - k| + |j - l| = 1$.)

Each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum (A node v of G is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.) of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Solution: Local minimum (KT 5.7)

Let A be a $m \times n$ matrix of reals with indexes (i, j) for $1 \leq i \leq m$ and $1 \leq j \leq n$. The *neighbors* of entry (i, j) are those (i', j') with $|i - i'| + |j - j'| = 1$. A *local minimum* is some (i, j) such that for each neighbor (i', j'), $A_{i,j} \leq A_{i',j'}$. An inefficient greedy method to find a local minimum is to start anywhere and while you have a smaller neighbor, move to that neighbor. This process must terminate since each move results in a strictly smaller number. While greedy method may not lead to an efficient algorithm, it is nevertheless useful since it establishes the existence of a local minimum within a region of the matrix.

We will show how to find a local minimum using $O(m + n)$ queries to A. There are many solutions to this problem, but there are many appealing non-solutions as well, so be careful.

We will solve a slightly more general problem: we will also be given as input the index c of an element in A and find some local minimum $\leq A_c$. The greedy method starting at c shows such a local minimum exists. It is necessary to require the algorithm to find a local minimum subject to certain upperbound to ensure correctness of the algorithm.

If $m, n < 3$, use exhaustive search. Otherwise, suppose wlog that $n \geq m$ and let M be the middle column of A (if $n < m$, we would use the middle row). More precisely, we will let M to be the set of indexes corresponding to the middle column. Find an index a in $M \cup \{c\}$ such that A_a is the smallest number among all A_x where x in $M \cup \{c\}$. If a is a local minimum, we are done. Otherwise, there is some neighbor $b \notin M \cup \{c\}$ of a with $A_b < A_a$. M divides A into 2 halves. Let B be the half of A, not including M, that contains b.

We claim that B contains a local minimum x such that $A_x \leq A_b$ and that any such local minimum is a local minimum of A such that it is less than or equal to A_c. To see the first claim, note that the greedy method starting at $b \in B$ cannot cross M since every value in M is strictly larger than A_b. To see the second claim, let d be a local minimum of B such that $A_d \leq A_b$. It is clear that $A_d \leq A_c$ since $A_b \leq A_c$. To prove that d is a local minimum of A, assume for the sake of contradiction that d is not a local minimum of A. Then d has a neighbor $e \in M$ with $A_e < A_d$. But then $A_e < A_d \leq A_b < A_a$, contradicting that A_a is a smallest element among all A_x for x in $M \cup \{c\}$.

Each recursive call uses $O(\max\{m, n\})$ queries to A and cuts the larger of $\{m, n\}$ in half, so the total number of queries is at most a constant times $(m + m/2 + m/4 + \cdots) + (n + n/2 + n/4 + \cdots)$, which is $O(m + n)$.

Problem 98: Next greater element

Given an array, print the Next Greater Element (NGE) for every element. The Next Greater Element of an item x is the first greater element to its right in the array.

Solution: Next greater element

Algorithm description: Let $a_1, a_2, \ldots, a_n$ be a list of n numbers. Let us assume without loss of generality that a_i is a positive integer for $1 \leq i \leq n$. If a_i does not have any greater element to its right, we define its next greater element to be -1. We compute the next greater element (NGE) for every element a_i by scanning the list from left to right and storing the elements for which we have not yet found the next greater element on the stack.

Here are the details of the algorithm. For each element a_i starting with $i = 1$ and an empty stack, we process a_i as follows until it is pushed onto the stack.

We push a_i onto the stack if the stack is empty or $a_i \leq t$ where t is the top of the stack. Otherwise, if $a_i > t$, we pop t from the stack and mark a_i as its next greater element and repeat the process of pushing a_i onto the stack.

After a_n is processed, we pop all the elements of the stack and mark -1 as their next greater element.

Pseudocode :

Algorithm 1: Next Greater Element

```
input A[1...n]
S ← Empty Stack
for i = 1 to n do
  while S is not empty AND A[S.top] < A[i] do
    t = S.pop()
    NGE(t) = A[i]
  end while
  S.push(i)
end for
while S is not empty do
  t = S.pop()
  NGE(t) = −1
end while
output NGE[1...n]
```

Proof of Correctness: We prove the correctness of the algorithm by induction on the number of iterations of the algorithm. Each iteration exactly involves processing one element of the sequence. More precisely,

Claim 9.1. *For $0 \leq i \leq n$, at the end of processing a_i, the following holds:*

1. *If the stack is not empty, indexes of the elements on the stack belong to the set $\{1, 2, \ldots, i\}$, are increasing and the values are decreasing as we go from the bottom of the stack to the top of the stack.*
2. *The elements on the stack do not have a next greater element among $a_1, \ldots, a_i$.*
3. *If an element e is popped out of the stack during iteration i, then a_i is the next greater element for e.*

We regard the beginning of iteration j as the end of iteration $j-1$ for $1 \leq j \leq n$. Although we only push indexes onto the stack, with a slight abuse of the notation, we use the phrase top value on the stack to refer to the value at the index given by the top element of the stack.

Proof.
Base case ($i = 0$): At the end of iteration 0 the stack is empty, so properties $1, 2$, and 3 are satisfied vacuously.
Inductive Step: Assume that the properties hold at the end of iteration j for all $0 \leq j \leq i < n$.

We will prove that the properties hold at the end of iteration $i+1$. Consider the state of the program at the beginning of the iteration $i+1$. There are several possibilities: the stack is empty; if not, a_{i+1} is less than the top value on the stack; or a_{i+1} is greater than the top value on the stack. In all scenarios, we prove that all the three properties will hold.

- The stack is empty: In this case, we simply push a_{i+1} onto the stack during iteration $i+1$. Property 1 holds trivially since there is only one element on the stack. Property 2 holds since the element a_{i+1} cannot have its next greater element in the list $a_1, \ldots, a_{i+1}$. Property 3 holds vacuously since no element has been popped during iteration $i+1$ since the stack is empty at the beginning of the iteration.

- a_{i+1} is less than the top value on the stack: We push a_{i+1} onto the stack in this case. Since $i+1$ is the largest index of all the indices considered and a_{i+1} is smaller than the top value on the stack, it follows from induction hypothesis that Property 1 is satisfied. Again by induction hypothesis and since a_{i+1} is smaller than the top value we conclude that none of the elements on the stack have a next greater element in $a_1, \ldots, a_{i+1}$. Since no elements have been popped from the stack, Property 3 holds vacuously.

- a_{i+1} is greater than the top value on the stack: Let $x_1, \ldots, x_k$ be the indices of the elements on the stack with x_1 at the bottom and x_k at the top of the stack. Clearly $i+1 > x_k$ since $i+1$ is largest index we have accessed so far. We pop elements from the stack till the stack is empty or a_{i+1} is less than the top value on the stack. Hence, property 1 still holds at the end of iteration $i+1$.

  Let $a_{x_j}, \ldots, a_{x_k}$ be the elements that were popped during the iteration. This implies that $a_{i+1} > a_{x_j} > \cdots > a_{x_k}$. Additionally, we claim that for every element a_{x_m} where $m \in [j, k]$, there is no other element with index y such that $i+1 > y > m$ and $a_y > a_m$, since from property 2 at the end of iteration i we know that the elements on the stack do not contain their next greater element in the array $a_1, \ldots, a_i$. Therefore all the popped elements have found their next greater element. Since a_{i+1} is smaller than the remaining elements on the stack, we conclude that values on the stack (including a_{i+1}) do not have their next greater element in $a_1, \ldots, a_{i+1}$.

□

Terminating Step: At the end of the loop, if the stack contains any elements (with indices $x_1, x_2, \ldots, x_k$) then $x_1 < x_2 < \cdots < x_k$ and $a_{x_1} > a_{x_2} > \cdots > a_{x_k}$. Since the properties in the claim hold at the end of iteration n, NGE does not exist for the elements on the stack in $a_1, \ldots, a_n$ so we set their NGE values to -1.

Complexity Analysis: Every element in the array is pushed exactly once onto the stack and popped exactly once from the stack. Overall, there are only constant number of operations per element. Therefore, the algorithm runs in $\Theta(n)$ time.

Problem 99: Median of $2n$ elements

Let $X[1..n]$ and $Y[1..n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays X and Y.

Solution: Median of $2n$ elements

Let $X = (X_1, \ldots, X_n), Y = (Y_1, \ldots, Y_n)$ be two sorted lists of n elements each. For the purposes of this problem, the *median* of a list of size $2n$ is the nth smallest element. The problem is to find the median of the combiness lists X and Y.

Let's generalize the problem a little to finding the kth smallest element of two lists, X of size m and Y of size n. If $m = 0$, then kth smallest element is Y_k. If $n = 0$, then it is X_k. Otherwise, $m, n \geq 1$. Without loss of generality, we can assume $X_{\lceil \frac{m}{2} \rceil} \leq Y_{\lceil \frac{n}{2} \rceil}$. If $X_{\lceil \frac{m}{2} \rceil} > Y_{\lceil \frac{n}{2} \rceil}$, we can swap X, Y (and m, n). If $k > m + \lfloor \frac{n}{2} \rfloor$, then we can discard the first $\lceil \frac{m}{2} \rceil$ elements of X from consideration and find the $k - \lceil \frac{m}{2} \rceil$th element of what remains. If $k \leq m + \lfloor \frac{n}{2} \rfloor$, then we can discard the last $\lceil \frac{n}{2} \rceil$ elements of Y from consideration and find the kth element of what remains. One of these two conditions must occur, otherwise we would have the contradiction $m + \lfloor \frac{n}{2} \rfloor < k \leq m + leq \lfloor \frac{n}{2} \rfloor$. Either way, we at least halve one of the lists and so the algorithm takes time $O(\lg m + \lg n)$. In the special case where $m = n$ and $k = n$, we can solve the problem in time $O(\lg n)$.

Students are advised to supply the missing arguments to prove the correctness of the algorithm.

Solution: Median of $2n$ elements

Let $X = (X_1, \ldots, X_n), Y = (Y_1, \ldots, Y_n)$ be two sorted lists of n elements each. For the purposes of this problem, the *median* of a list of size $2n$ is the nth smallest element. The problem is to find the median of the combiness lists X and Y.

The following is an alternative solution.

```
1  A_n(X, Y) // X, Y both have size n
2    if n = 1, return min{X_1, Y_1}
3    if X_{⌊n/2⌋} > Y_{⌊n/2⌋}, swap X, Y
4    X' ← (X_{⌊n/2⌋+1}, ..., X_n)
5    Y' ← (Y_1, ..., Y_{⌈n/2⌉}))
6    return A_{⌈n/2⌉}(X', Y')
```

We prove correctness of A_n by induction on n. If $n = 1$, then the median of $X \cup Y$ is the smaller of its 2 elements, and so A_1 is correct. If $n \geq 2$, then, due to line 3 of the code, we may assume without loss of generality that $X_{\lfloor \frac{n}{2} \rfloor} \leq Y_{\lfloor \frac{n}{2} \rfloor}$. So the first $\lfloor \frac{n}{2} \rfloor$ elements of X and the last $\lceil \frac{n}{2} \rceil$ elements of Y need not be considered. Nonetheless, we will only discard the last $\lfloor \frac{n}{2} \rfloor$ elements from Y. The median of $X \cup Y$ is the nth smallest element of $X \cup Y$, which is the same as the $\lceil \frac{n}{2} \rceil$th element of $X' \cup Y'$, which is the median of $X' \cup Y'$, which, by the inductive hypothesis, is correctly returned by $A_{\lceil \frac{n}{2} \rceil}$. This completes the induction.

To implement this, we don't actually copy arrays, since that would take linear time, but just pass pointers and the start and end indexes into the arrays. The running time T satisfies $T(n) = T(\lceil \frac{n}{2} \rceil) + O(1)$, which is clearly $O(\lg n)$.

Solution: Median of $2n$ elements

Let $X = (X_1, \ldots, X_n), Y = (Y_1, \ldots, Y_n)$ be two sorted lists of n elements each. For the purposes of this problem, the *median* of a list of size $2n$ is the nth smallest element. The problem is to find the median of the combiness lists X and Y.

We want to find the median of the union of two sorted lists, $X[1..n]$ and $Y[1..n]$, each of length n, i.e., the n'th largest element. The algorithm goes as follows: We'll maintain three values, $L1, L2, K$ and maintain the invariant that the element we are looking for is the median (K'th largest) of the sets $X[L1..L1+K-1]$ and $Y[L_2..L_2+K-1]$ (the important feature is that the two sub-arrays we are examing are the same size, K). We first compare the medians of the sub-arrays, $X[L_1+\lceil K/2\rceil-1]$ and $Y[L_2+\lceil K/2\rceil-1]$. If $X[L_1+\lceil K/2\rceil-1] \geq Y[L_2+\lceil K/2\rceil-1]$, we know $X[M] \geq Y[M']$ for every $M \geq L_1+\lceil K/2\rceil-1$ and every $M' \leq L_2+\lceil K/2\rceil-1$. Thus, for each of the $\lfloor K/2\rfloor$ values of M with $L_1+K-1 \geq M \geq L_1+\lceil K/2\rceil$, we know that it is larger than at least $\lceil K/2\rceil$ elements in the Y list, and at least $M-L_1+1 \geq \lceil K/2\rceil+1$ elements in the X list. Therefore, it is greater than $K+1$ elements total, and is therefore greater than the median. Similarly, for all $\lfloor K/2\rfloor$ values of M' in the range $L_2 \leq M' \leq L_2+\lfloor K/2\rfloor-1$, we know that M' is smaller than $\lceil K/2\rceil$ elements in the X list and $\lceil K/2\rceil+1$ elements in the X list. So each such M' is smaller than the median. Thus, if we delete $X[M]$ and $Y[M']$ for M and M' in the range above, we have deleted $\lfloor K/2\rfloor$ elements smaller than the median and the same number of elements larger than the median. Thus, the median will be the median of the remaining elements. To do this deletion, we reset $K \leftarrow \lceil K/2\rceil$ and $L_2 \leftarrow L_2+\lfloor K/2\rfloor$.

The other case is handled symmetrically. When $K=1$, we return the larger element between $X[L_1]$ and $Y[L_2]$.

Since our algorithm uses only one comparison before recursing, and each recursion halves K and K starts at n, we perform at most $\log n$ comparisons total.