1. (a) Describe an algorithm to find the poisoned bottle using at most $O(\log n)$ tests. (This is best possible in the worst case.)

Solution (binary search): Arbitrarily label the bottles from 0 to $n-1$. In the following algorithm, each Taster performs at most one test.

```
HALLOMYNAMEIS(n):
  lo ← 0
  hi ← n − 1
  while lo < hi
      mid ← ⌊(lo + hi)/2⌋
      T ← new Taster
      T tests bottles lo through mid
      if T is mostly dead
          hi ← mid
      else
          lo ← mid + 1
  return lo
```

The algorithm performs one test and then recursively searches half the bottles. So the number of tests obeys the recurrence $T(n) \le 1 + T(\lceil n/2 \rceil)$ with the base case $T(1) = 0$, which implies $T(n) \le \lceil \log_2 n \rceil = O(\log n)$. ■

Solution (one bit at a time): Arbitrarily label the bottles from 0 to $n-1$ in binary. For any non-negative integer i, let S_i denote the subset of all bottles whose label has its ith bit equal to 1. For example, S_0 is the set of all bottles with odd labels. Exactly $\lceil \log_2 n \rceil$ of these sets are non-empty. Similarly, arbitrarily label the first $\lceil \log_2 n \rceil$ Tasters from 0 to $\lceil \log_2 n \rceil - 1$.

```
INIGOMONTOYA(n):
  poison ← 0          ⟨⟨poisoned bottle's label⟩⟩
  for i ← 0 to ⌈log₂ n⌉ − 1
      Taster i tests subset S_i
      if Taster i is mostly dead
          poison ← poison + 2^i
  return poison
```

The algorithm clearly performs one test per iteration of the main loop, so the numebr of tests is $\lceil \log_2 n \rceil = O(\log n)$. ■

Rubric: 2 points. These are not the only correct solutions.

(b) Now suppose *two* of the n bottles have been poisoned. Describe an algorithm to identify *both* poisoned bottles, using as few tests as possible in the worst case.

Solution:

```
YouKilledMyFather(n):
  lo ← 0
  hi ← n − 1
  while lo < hi
      mid ← ⌊(lo + hi)/2⌋
      T1, T2 ← two new Tasters
      T1 tests bottles lo through mid
      T2 tests bottles mid + 1 through hi
      if T1 and T2 are both mostly dead
          find the poison in bottles lo through mid via part (a)
          find the poison in bottles mid + 1 through hi via part (a)
      else if only T1 is mostly dead
          hi ← mid
      else
          lo ← mid + 1
```

In each iteration of the main loop, the algorithm performs two tests and then *either* recursively searches half the bottles *or* invokes an algorithm for part (a). Thus, there number of tests obeys the recurrence

$$T(n) \leq 2 + \max\left\{T(\lceil n/2 \rceil),\ O(\log n)\right\},$$

which implies $T(n) = O(\log n)$. (More precisely, $T(n) \leq 2\lceil \log_2 n \rceil$.) ■

Rubric: 4 points. This is not the only correct solution.

(c) Finally, suppose the poisoners try to evade the Royal Tasters by reducing the amount of iocaine in each poisoned bottle. Now when a Taster tests a set S of wine bottles, they become mostly dead if and only if *both* poisoned bottles are in S. Describe an algorithm to find *both* poisoned bottles using as few tests as possible in the worst case.

Solution: Just for fun, I'll write this algorithm recursively instead of iteratively, using sets of bottles as arguments instead of indices.

```
PrepareToDie(S):
  n ← |S|
  if n = 2
      return both bottles in S
  T_1, T_2 ← two new Tasters
  A ← any subset of ⌊n/2⌋ bottles in S
  T_1 tests bottles in A
  T_2 tests bottles in S \ A
  if T_1 is mostly dead
      return PrepareToDie(A)
  else if T_2 is mostly dead
      return PrepareToDie(S \ A)
  else ⟨⟨A contains one poisoned bottle⟩⟩
      return Obviously(A, S \ A)
```

```
⟨⟨Find two poisoned bottles, one in A and one in B⟩⟩
Obviously(A, B):
  if |A| < |B|
      swap A ↔ B
  if |A| = 1 ⟨⟨and therefore |B| = 1⟩⟩
      return the only bottles in A and B
  T ← new Taster
  C ← any subset of ⌈|A|/2⌉ bottles in A
  T tests bottles in C ∪ B
  if T is mostly dead
      return Obviously(B, C)
  else
      return Obviously(B, A \ C)
```

PrepareToDie recurses at most $\lceil \log_2 n \rceil$ times, performing two tests at each level of recursion, before either finding both poisoned bottles or calling Obviously. Similarly, each recursive call to Obviously performs at most one test and shrinks one of the two sets by a factor of 2. In more detail, the number of tests performed by Obviously(A, B) satisfies the following recurrence, where $n = |A|$ and $m = |B|$, assuming $n/2 \le m \le n$:

$$T(n, m) \le \begin{cases} 0 & \text{if } n = 1 \\ 1 + T(m, \lceil n/2 \rceil) & \text{otherwise} \end{cases}$$

Expanding the recurrence once gives us $T(n, n) = 2 + T(\lceil n/2 \rceil, \lceil n/2 \rceil)$, which implies $T(n, n) \le 2\lceil \log_2 n \rceil$.

We conclude that the total number of tests is at most $2\lceil \log_2 n \rceil = O(\log n)$.

Sweet! Coins! ■

Solution (three subsets): We'll again write this algorithm using sets of bottles.

```
StopSayingThat(S):
  n ← |S|
  if n = 2
      return both bottles in S
  T1, T2 ← two new Tasters
  A ← any subset of ⌊n/3⌋ bottles in S
  B ← any subset of ⌈n/3⌉ bottles in S \ A
  C ← S \ (A ∪ B)
  T1 tests bottles in A ∪ B
  T2 tests bottles in B ∪ C
  if T1 is mostly dead
      return StopSayingThat(A ∪ B)
  else if T2 is mostly dead
      return StopSayingThat(B ∪ C)
  else ⟨⟨B contains neither poisoned bottle⟩⟩
      return StopSayingThat(A ∪ C)
```

The algorithm performs two tests and recursively tests a set containing at most $\lceil 2n/3 \rceil$ bottles. The number of tests obeys the recurrence $T(n) \le 2 + T(\lceil 2n/3 \rceil)$, implying the total number of tests is at most $2\lceil \log_{3/2} n \rceil = O(\log n)$. ■

Rubric: 4 points. As usual these are not the only solutions.

2. (•) ***Practice only. Do not submit solutions.***

Describe how to find an *arbitrary* bottle of rum (not necessarily closest to the northwest corner) in $O(\log n)$ time.

Solution: If $n = 1$, return the only bottle in the only square.

Otherwise, split the grid into four $(n/2) \times (n/2)$ quadrants, and test each quadrant using the Device. At least one of the quadrants contains a bottle of rum, so at least one of the tests is successful. Recurse in any quadrant that made the Device happy.

The running time of this algorithm on an $n \times n$ grid obeys the binary-search recurrence $T(n) = O(1) + T(n/2)$, so the algorithm runs in $\boldsymbol{O(\log n)}$ ***time***, as required. ■

Rubric: 0 points; practice only

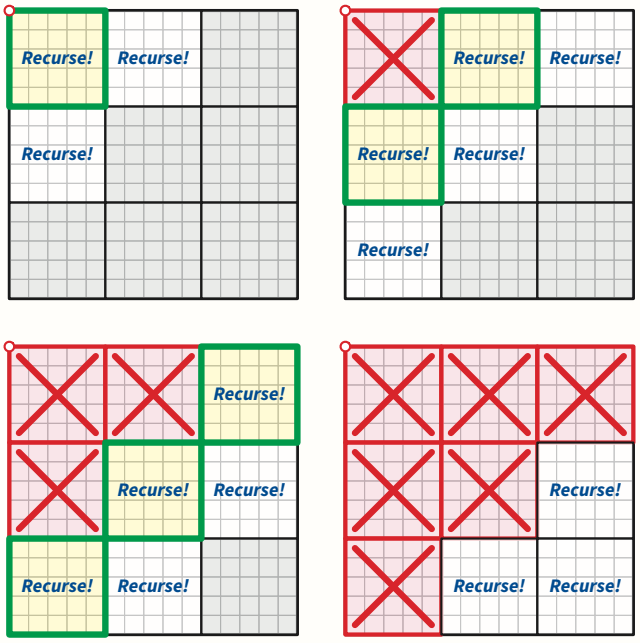
(a) What is the running time (the number of tests) of the algorithm that tests the northwest quadrant and then recursively searches three quadrants?

Solution: The running time obeys the recurrence $T(n) = 1 + 3T(n/2)$. The level sums of the recursion tree define an increasing geometric series, which is dominated by the number of leaves. The recursion tree has 3^ℓ nodes at level ℓ, and the depth of the recursion tree is $\log_2 n$. So the number of leaves is $3^{\log_2 n} = n^{\log_3 2}$. So the algorithm runs in $\boldsymbol{O(n^{\log_2 3}) = O(n^{1.585})}$ ***time***, just like Karatsuba's algorithm. ■

Rubric: 2 points. This is not the only correct solution.

(b) Now suppose we split Grid Island into nine $(n/3) \times (n/3)$ subgrids, test all nine subgrids using the Device, and then recurse on a smaller subset of those nine subgrids. How many recursive calls do we need to make in the worst case? Prove your answer is correct. What is the running time of the resulting algorithm?

Solution: We need to make at most ***five*** recursive calls in the worst case. There are four cases to consider, illustrated below.



- If the northwest block has any rum, then the closest rum is either in that block or one of its two neighbors, so we only need to recursively search three block.
- If the northwest block is empty, but at least one of its two neighbors is not, then the closest rum is in one of five block on or just above the diagonal.
- If the three block closest to the northwest corner are all empty, but at least one of the block along the diagonal is not, then the closest rum is in one of five block on or just below the diagonal.
- Finally, if the six block closest to the northwest corner are all empty, the closest rum is in one of the other three block.

We test at most six rectangles and make at most five recursive calls, so the running time of our obeys the recurrence $T(n) \leq 6 + 5T(n/3)$. The usual recursion-tree analysis implies $\boldsymbol{T(n) = O(n^{\log_3 5}) = O(n^{1.465})}$ time. ■

Rubric: 4 points = 2 for "at most five" + 2 for $O(n^{\log_3 5})$. This is not the only correct solution with this running time.

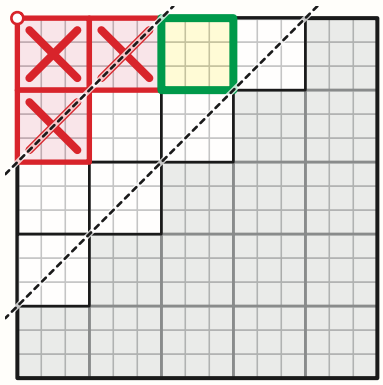
A correct and correctly analyzed algorithm with more recursive calls is worth ½ point less for each additional recursive call. For example, an algorithm that makes 8 recursive calls in the worst case is worth at most 2½ points.

*(c) Describe an even faster algorithm to find the closest bottle of rum.

Solution (Generalize part (b)): We can generalize the algorithm in part (b). For any integer $k > 1$, we can split the $n \times n$ grid into $(n/k) \times (n/k)$ blocks (ignoring rounding).

Let B be any block containing rum that is closest to the ship. Then the closest bottle to the ship must lie inside a block that is either on the same diagonal with B or the next lower diagonal. All higher blocks are known to be empty, and all lower blocks are too far away from the ship. Thus, we need to recursively search ***at most* $2k-1$** blocks.

In the figure below, suppose the block with the heavy green outline contains rum, but the red boxes with Xs do not. Then the closest bottle to the ship must lie between the dashed lines, and therefore not in any of the red Xed blocks above or in any of the shaded blocks below the dashed lines.



(Of course, we don't need to recursively search any block that the Device has already told us is empty, but in the worst case, every interesting block contains at least one bottle of rum.)

The running time of the resulting recursive algorithm obeys the recurrence

$$T(n) \le k^2 + (2k-1) \cdot T(n/k),$$

which has the solution $T(n) = O(k^2 n^{\log_k(2k-1)})$. By setting k to an appropriate constant, we can make the exponent arbitrarily close to 1, but we need a *very* large k to get an exponent close to 1. For example, if $k = 64$, the running time becomes $O(n^{\log_{64} 127}) = O(n^{1.1648})$, and if $k = 2^{32}$, the running time is $O(n^{\log(2^{33}-1)/\log(2^{32})}) = O(n^{33/32}) = O(n^{1.03125})$. ■

Rubric: 4 points. "Arbitrarily close to linear" is enough for full credit.

Solution (Generalize part (b) even more): We can use the same recursive algorithm as the previous solution, with the same running-time recurrence

$$T(n) \leq k^2 + (2k-1) \cdot T(n/k),$$

but now instead of using the same *constant* k at every level of recursion, we let k be an increasing function of n. Thus, we partition into fewer blocks at deeper levels of recursion.

The best function of n turns out to be $k = \sqrt{n}$. The resulting running time now obeys the recurrence

$$T(n) \leq n + 2\sqrt{n} \cdot T(\sqrt{n}).$$

(Yes, I intentionally dropped the -1.) We can solve this non-standard recurrence using recursion trees. The first few levels of the recursion tree have the following structure:

- Level 0 has one node with value n, so the level sum is n.
- Level 1 has $2\sqrt{n}$ nodes, each with value $\sqrt{n}$, so the level sum is $2n$.
- Level 2 has $4n^{3/4}$ nodes, each with value $n^{1/4}$, so the level sum is $4n$.

More generally, the sum of nodes at each level ℓ is $2^\ell n$. The level sums form an *increasing* geometric series, so the running time is dominated by the bottom level. Each node at level ℓ represents a recursive problem of size $n^{2^{-\ell}}$. If we arbitrarily declare problem size 2 to be our base case, the tree bottoms out when

$$n^{2^{-\ell}} = 2 \iff 2^{-\ell} \log_2 n = 1 \iff 2^\ell = \log_2 n \iff \ell = \log_2 \log_2 n$$

We conclude that $T(n) = O(2^{\log_2 \log_2 n} n) = \boldsymbol{O(n \log n)}$.

Intuitively, when $k = o(\sqrt{n})$, the recursion tree has larger depth, so the number of leaves $2^\ell n$ is bigger, so the overall running time is bigger. On the other hand, when k is significantly larger than $\sqrt{n}$, the level sums grow more quickly, so again the overall running time is bigger. ■

Rubric: 5 points. This is more detail than necessary for full credit.

Solution (binary search in each row): The closest bottle of rums must be the leftmost bottle in its row. We can find the leftmost bottle in each row in $O(\log n)$ time using binary search, as follows:

```
REALLYBADEGGS(n):
  bestdist ← ∞
  for i ← 1 to n
      if DEVICE(1, 1, i, n) farts
          continue ⟨⟨skip the rest of this iteration⟩⟩
      ⟨⟨Binary search!⟩⟩
      lo ← 1
      hi ← n
      while hi − lo < 3
          mid ← ⌊(lo + hi)/2⌋
          if DEVICE(1, 1, i, mid) chimes
              hi ← mid
          else
              lo ← mid + 1
      ⟨⟨leftmost bottle in row i is in column hi = lo⟩⟩
      if i + lo < bestdist
          bestdist ← i + lo
          best ← (i, lo)
  return best
```

The algorithm uses the device at most $O(\log n)$ times on each row, so the overall algorithm runs in $\boldsymbol{O(n \log n)}$ ***time***. ■

Rubric: 5 points. This is more detail than necessary for full credit.

Solution (either madness or brillance): We can combine the two previous solutions as follows. First, we split the grid into $O(k^2)$ blocks, each of size $(n/k) \times (n/k)$. Then for each row of blocks, we find the leftmost block that contains a bottle using binary search, as in the previous solution. From these $O(k)$ candidate blocks, let B be the block closest to the northwest corner. The analysis in part (a) implies that we can find the closest bottle by recursively searching at most $2k-1$ blocks.

The running time of the resulting algorithm satisfies therecurrence

$$T(n) \le O(k \log k) + (2k-1) \cdot T(n/k).$$

As in the first solution, we can make $T(n) = O(n^{1+\varepsilon})$ for any $\varepsilon > 0$ by choosing an appropriately large constant k (that depends on ε). But just like the second solution, we can do even better by allowing k to be an incerasing function of n.

The best choice of k for this algorithm turns out to be $k = n/\lg n$, meaning we partition the grid into blocks of size $\lg n \times \lg n$. (Here $\lg n$ is standard shorthand for $\log_2 n$.) The running time of the resulting algorithm satisfies the recurrence

$$T(n) \;\le\; O(n) \;+\; \frac{2n}{\lg n} \cdot T(\lg n).$$

As usual we can solve this recurrence using recursion trees. The first few levels of the recursion tree have the following structure:

- Level 0 has one node with value n, so the level sum is n.
- Level 1 has $\frac{2n}{\lg n}$ nodes, each with value $\lg n$, so the level sum is $2n$.
- Level 2 has $\frac{2n}{\lg n} \cdot \frac{2\lg n}{\lg \lg n} = \frac{4n}{\lg \lg n}$ nodes, each with value $\lg \lg n$, so the level sum is $4n$.

More generally, the sum of nodes at each level ℓ is $2^\ell n$. As before, the level sums form an increasing geometric series, so the running time is dominated by the deepest level. Each node at level ℓ represents a recursive problem of size $\lg^{(\ell)} n = \lg(\lg^{(\ell-1)} n)$. If we arbitrarily declare problem size 1 to be our base case, the tree bottoms out at level $\ell = \lg^* n$, where $\lg^*$ is the *iterated logarithm* function, defined recursively as follows

$$\lg^* n = \begin{cases} 0 & \text{if } n \le 1 \\ 1 + \lg^*(\lg n) & \text{otherwise} \end{cases}$$

Intuitively, $\lg^* n$ is the number of times we can take logarithms, starting with the value n, until the result is at most 1. For all practical purposes, $\lg^* n$ is a constant—in particular, $\lg^* n \le 4$ for all $n \le 2^{65536}$—but this is a *theory* class; we don't care about such insignificant values of n.

We conclude that our recursive algorithm runs in $\boldsymbol{O(n 2^{\lg^* n})}$ ***time***. ■

Rubric: 7 points. This is more detail than necessary for full credit.

Solution (Optimize part (a) nonrecursively): We can solve actually this problem in ***O(n) time*** by modifying the algorithm from part (a), using the analysis in the first solution. Instead of searching each block recursively, we *iteratively* consider finer and finer decompositions of the grid into blocks, and we use the results from each iteration to select a subset of smaller blocks to test in the next iteration.

Without loss of generality, assume that n is a power of 2; otherwise, we can imagine extra empty rows and columns to the south and east.

The algorithm from part (a) recursively searches a decomposition of the $n \times n$ grid called a *quadtree*. For each integer k, the kth level of the quadtree partitions the $n \times n$ grid into $(n/2^k) \times (n/2^k)$ squares, which I'll call ***k-blocks***. Thus, the entire grid is a 0-block; the four quadrants are 1-blocks, and more generally, each k-block is partitioned into four $(k+1)$-blocks.

For any integer k, the goal of the kth iteration of our algorithm finds a k-block closest to the ship that contains at least one bottle of rum. Let B_{k-1} be the $(k-1)$-block found in the $(k-1)$th iteration; in particular B_0 is the entire grid. The analysis in part (b) implies that our target bottle lies in a $(k-1)$-block that is either on the same diagonal as B_{k-1} or on the next diagonal below. There are at most $2(n/2^k) - 1 < 2n/2^k$ interesting $(k-1)$-blocks.

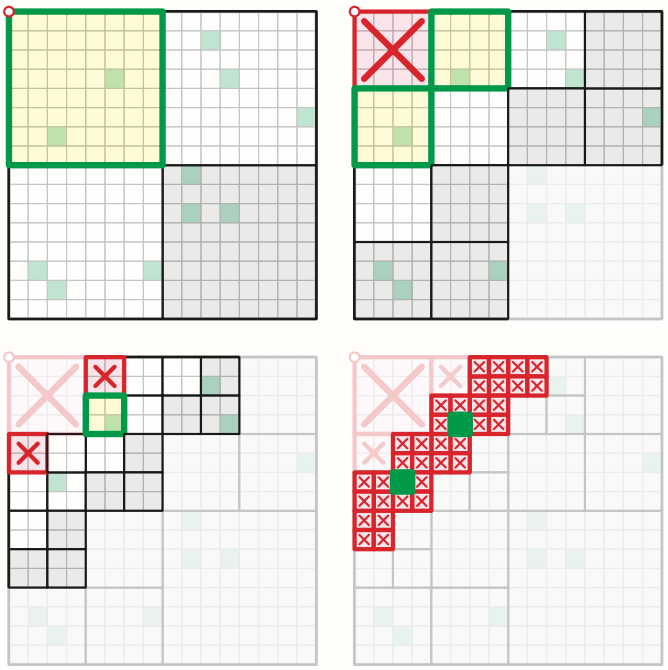
In the kth iteration, we break each of the interesting $(k-1)$-blocks into four k-blocks, use the Device to test each of those k-blocks, and finally define B_k to be the k-block closest to the ship that made the Device happy (breaking ties arbitrarily). Altogether we apply the Device to at most $8n/2^k$ k-blocks.

Summing over all iterations, we conclude that the total number of tests is (*very* conservatively) at most

$$\sum_{k=0}^{\log_2 n} 8n/2^k \;=\; 8n \cdot \sum_{k=0}^{\log_2 n} 1/2^k \;<\; 8n \cdot \sum_{k\geq 0} 1/2^k \;=\; 16n \;=\; \boldsymbol{O(n)}.$$

(Yes, we could reduce the constant factor 16 by improving the algorithm and/or applying more careful analysis, but why bother? $O(n)$ is already $O(n)$.)

The figure on the next page shows the algorithm in action.



■

Rubric: 9 points.

Solution (it's pronounced "egregious"): We once again follow a careful implementation the algorithm for part (a), but we keep track of the closest bottle that we have discovered so far, and we ignore any recursive call that explores a cell whose squares are further from the northwest corner than our candidate bottle.

Our recursive algorithm takes five input arguments:

- i and j are the indices of the top row and left column of the block we are exploring
- w is the width (= height) of the block we are exploring
- *besti* and *bestj* are the row and column indices for the closest bottle we found before making this recursive call.

The algorithm returns the new values of *besti* and *bestj*, that is, the coordinates of the closest bottle found either before or during the recursive call. The top-level function call is BLACKPEARL$(1, 1, n, \infty, \infty)$. To simplify the algorithm, assume without loss of generality that n is a power of 2; otherwise, we can imagine extra empty rows and columns to the south and east.

```
BLACKPEARL(i, j, w, besti, bestj):
  if i + j > besti + bestj                    ⟨⟨They're out of range!⟩⟩
      return besti, bestj
  if DEVICE(i, j, i + w, j + w) farts         ⟨⟨Why is the rum gone?⟩⟩
      return besti, bestj
  if w = 1
      return i, j                             ⟨⟨A better one.⟩⟩
  ⟨⟨Now where is that monkey? I want to shoot something.⟩⟩
  besti, bestj ← BLACKPEARL(i,     j,     w/2, besti, bestj)
  besti, bestj ← BLACKPEARL(i+w/2, j,     w/2, besti, bestj)
  besti, bestj ← BLACKPEARL(i,     j+w/2, w/2, besti, bestj)
  besti, bestj ← BLACKPEARL(i+w/2, j+w/2, w/2, besti, bestj)
  return besti, bestj
```

I claim that this algorithm runs in ***O(n) time***.

As in the previous solution, for each integer k, subdivide the grid into 4^k squares, each of width and height $n/2^k$, which I'll call ***k-blocks***. Each call BLACKPEARL$(i, j, w, \cdot, \cdot)$ *explores* a distinct k-block, where $k = \lg(n/w)$. We call a k-block *interesting* if BLACKPEARL explores that block and its four children; interesting blocks correspond to the non-leaf nodes in the recursion tree of BLACKPEARL. The overall running time of our algorithm is $O(1)$ times the number of interesting blocks.

Pick an integer k, and let $w = n/2^k$. Let B and B' be two interesting k-blocks with the same top row i, and left columns j and j', respectively. Without loss of generality, assume $j < j'$; because B and B are disjoint, we have $j' \geq j + w$.

The order of recursive calls implies that BLACKPEARL explores B. Because B is interesting, there is at least one bottle (bi, bj) such that $i \leq bi < i + w$ and $j \leq bj < j + w$. Thus, when BLACKPEARL finishes exploring BLACKPEARL, we

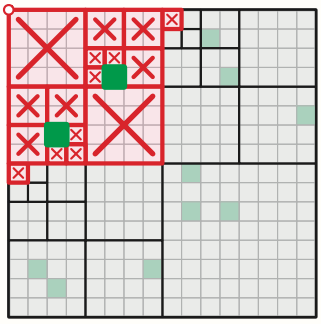
have $besti + bestj \le bi + bj < i + j + 2w$. Similarly, because B' is interesting, we must have $i + j' \le besti + bestj < i + j + 2w$ when BLACKPEARL starts exploring B', which implies $j' < j + 2w$. But $j' - j$ must be a multiple of w, so $j' = j + w$. We conclude that B' is directly to the right of B.

We have just argued that each row of k-blocks contains at most two interesting k-blocks, which implies that there are at most $2 \cdot 2^k$ interesting k-blocks altogether. Thus, the total number of interesting blocks of all sizes is at most

$$\sum_{k=0}^{\lg n} 2 \cdot 2^k \;\le\; 2 \cdot \sum_{k=0}^{\lg n} 2^k \;=\; 2 \cdot (2^{\lg n + 1} - 1) \;=\; 2(2n - 1) \;=\; 4n - 4.$$

We conclude that BLACKPEARL runs in ***O(n) time***, as claimed.

The following figure shows all the non-interesting children of interesting blocks (that is, the leaves of the recursion tree) in one run of BLACKPEARL. Gray blocks are out of range; red blocks with Xs are in range but the rum is gone; solid green blocks are bottles of rum.

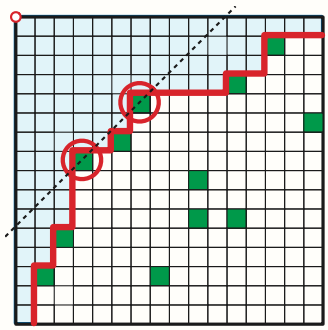


■

Rubric: 10 points = 6 points for the algorithm + 4 points for the time analysis.

Solution (more what you'd call "guidelines"): An even simpler non-recursive algorithm finds a small subset of bottles that is guaranteed to contain the closest bottle, using the Device only $O(n)$ times.

Let's call square (i, j) *pirate-optimal* if it contains the only bottle of rum in rectangle with corners $(1, 1)$ and (i, j). Each row or column of the grid contains most one pirate-optimal square. The pirate-optimal squares define a staircase shape called the *pirate-front*. In the figure below, the red staircase is the pirate-front, the green squares just below the corners of the pirate-front are the pirate-optimal bottles, and the two bottles closest to the ship are circled.



The following algorithm identifies all pirate-optimal bottles in $O(n)$ time.

```
IGotAJarOfDirt(n):
  row ← n
  col ← 1
  found ← False
  while row ≥ 1 and col ≤ n
      if Device(1, 1, row, col) farts
          if found
              (row + 1, col) is pirate-optimal
          found ← False
          col ← col + 1
      else ⟨⟨Device(1, 1, row, col) chimes⟩⟩
          found ← True
          row ← row − 1
  if row = 0 and found
      (1, col) is pirate-optimal
```

Each iteration of the while loop either increments col or decrements row, and therefore decrements $row - col$. Initially, $row - col = n - 1$. When the loop terminates. either $row = 0$ and $col \le n$, or $row \ge 1$ and $col = n + 1$; in either case, we have $row - col \ge -n$. We conclude that our algorithm uses the Device ***at most $2n - 1$ times***.

The closest bottle to our ship is clearly pirate-optimal, so Once we've identified all the pirate-optimal bottles, we can find the closest one by brute force, using no additional tests. ■

Rubric: 10 points. These are not the only $O(n)$-time solutions, but this is simplest one I know.

Solution (“You cheated.” “Pirate.”): We claim that no correct algorithm can use the Device fewer than $n-1$ times even if we're promised there is exactly one closest bottle and its distance is exactly n.

Imagine an all-powerful ***Adversary*** who *pretends* to present our algorithm with a fixed instance of the problem, but in fact changes their instance on the fly each time the algorithm uses the Device. You know, like how you used to cheat at 20 questions with your little brother. If the algorithm claims to find the closest bottle after using the Device less than $n-1$ times, the Adversary will reveal an actual instance that is consistent with all Device's responses, but where the algorithm's claimed closest bottle is incorrect.

Initially, the Adversary creates an “instance” of the problem where

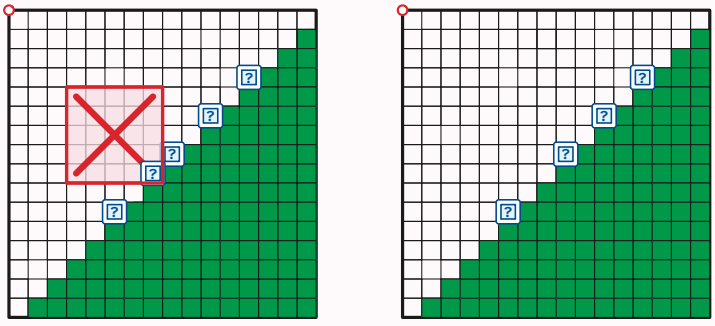
- every square of distance $n+1$ or greater holds a bottle of rum,
- every square of distance $n-1$ or less is empty, and
- every square of distance *exactly* n holds a *ghost bottle*, which the Adversary may later reveal to be a real bottle or nothing at all.

At the end of the algorithm, the Adversary will reveal that at least one ghost bottle is actually real.

Each time the algorithm uses the Device, the Adversary responds as follows:

- If the algorithms tests a rectangle containing no real bottles or ghost bottles, the Adversary makes the Device fart.
- If the algorithm tests a rectangle containing at least one real bottle, the Adversary makes the Device chime.
- The first $n-1$ times the algorithm tests a rectangle containing exactly one ghost bottle and no real bottles, the Adversary destroys that ghost bottle and makes the Device fart. See the figure below.
- Finally, the nth time the algorithm tests a rectangle containing exactly one ghost bottle and no real bottles, only one ghost bottle is left. The Adversary makes that last ghost bottle real and makes the Device chime.

Any rectangle that contains more than one ghost bottle also contains a real bottle, so these cases are exhaustive.



In the first two cases, the Device's response gives the algorithm no information whatsoever about which ghost bottles are real. At all times, the remaining ghost

bottles are precisely those that have never been in the southeast corner of a test rectangle.

Now consider what happens if the algorithm tries to report a closest bottle while there is still more than one ghost bottle (and in particular, after using the Device fewer than $n-1$ times). Because we are promised that the closest bottle has distance n, the algorithm must report a square at distance n.

- If the algorithm reports a square at distance n that does not contain a ghost bottle, the Adversary makes all ghost bottles real.
- If the algorithm reports a square that contains a ghost bottle, the Adversary destroys *that* ghost bottle and makes all other ghost bottles real.

In both cases, the Adversary reveals an instance that is consistent with all of the Device's responses, that has at least one bottle at distance n, and where the Algorithm has incorrectly reported an empty square instead of a bottle. As far as the algorithm can tell, that was the instance all along!

Too bad we insisted on only accepting the closest bottle to the port. ■

Rubric: 0 points. We never asked you to show a lower bound. This is not the only solution.

*3. ***Practice only. Do not submit solutions.***

The following variant of the infamous StoogeSort algorithm[1] was discovered by the British actor Patrick Troughton during rehearsals for the 20th anniversary *Doctor Who* special "The Five Doctors".[2]

```
WhoSort(A[1..n]) :
  if n < 13
      sort A by brute force
  else
      k = ⌈n/5⌉
      WhoSort(A[1..3k])          ⟨⟨Hartnell⟩⟩
      WhoSort(A[2k+1..n])        ⟨⟨Troughton⟩⟩
      WhoSort(A[1..3k])          ⟨⟨Pertwee⟩⟩
      WhoSort(A[k+1..4k])        ⟨⟨Davison⟩⟩
```

(a) Prove by induction that WhoSort correctly sorts its input. *[Hint: Where can the smallest k elements be?]*

Solution: Let n be an arbitrary no-negative integer, let $A[1..n]$ be an arbitrary array, and assume that WhoSort correctly sorts any array of size less than n. Assume that $n \geq 13$, since otherwise correctness is trivial.

Intuitively, WhoSort partitions the input array A into five chunks, four of size $k = \lceil n/5 \rceil$ and one of size $n - 4k \leq n/5 \leq k$:

$$A[1..k], \quad A[k+1..2k], \quad A[2k+1..3k], \quad A[3k+1..4k], \quad A[4k+1..n]$$

The inequality $n \geq 13$ implies $4\lceil n/5 \rceil \leq n$, so all five chunks lie within the bounds of the original input array. (When $n = 16$, the last chunk is empty, but that's not a problem.) Each recursive call to WhoSort processes three consecutive chunks, which comprise an subarray of size at most $3k$. The assumption $n \geq 13$ also implies $3\lceil k/5 \rceil < n$, so by the induction hypothesis, the recursive calls correctly sort their respective subarrays.

Call an element of the input array ***small*** if it is one of the k smallest elements, ***large*** if it is one of the $n - 4k \leq k$ largest elements, and ***medium*** otherwise. Consider the locations of these classes of elements after each recursive call to WhoSort.

- The inductive hypothesis implies that Hartnell moves all small elements in the first three chunks to chunk 1, and moves all large elements in the first three chunks to chunk 3. Thus, after Hartnell's sort, all small elements are in chunks 1, 4, and 5, and all large elements are in chunks 3, 4, and 5.
- The inductive hypothesis implies that after Troughton's sort, all small elements are in chunks 1 and 3, and all large elements (and nothing else)

[1] https://en.wikipedia.org/wiki/Stooge_sort

[2] Tom Baker, the fourth Doctor, declined to return for the reunion; hence, only four Doctors appeared in "The Five Doctors". (Well, okay, technically the BBC used excerpts of the unfinished episode "Shada" to include Baker, but he wasn't really *there*—to the extent that any fictional character in a television show about a time traveling wizard arguing with several other versions of himself about immortality can be said to be "really" "there".)

are in chunk 5. Moreover, chunk 5 is sorted; all large elements are in their correct final positions. The rest of the algorithm does not modify chunk 5.

- The inductive hypothesis implies that after Pertwee's sort, all small elements (and nothing else) are in chunk 1, in sorted order; thus, all small elements are in their correct final positions. The rest of the algorithm does not modify chunk 1. At this point, chunks 2, 3, and 4 contain all the medium elements and nothing else.
- Finally, the inductive hypothesis implies that Davison sorts chunks 2, 3, and 4, moving all the medium elements into their final correct positions.

We conclude that WHOSORT correctly sorts the array $A[1..n]$, as claimed. ■

> *I hate good wizards in fairy tales; they always turn out to be him.*
> — River Song [Alex Kingston], "The Pandorica Opens", *Doctor Who* 5(12): 2010.

(b) Would WHOSORT still sort correctly if we replaced "if $n < 13$" with "if $n < 4$"? Justify your answer.

Solution: ***No.*** When $n = 6$, the modified algorithm would fall into an infinite loop. Specifically, $k = \lceil 6/5 \rceil = 2$, so the first recursive call would attempt to recursively sort the entire array. ■

Solution: ***No.*** When $n = 11$, we have $k = \lceil 11/5 \rceil = 3$, so the last recursive call would attempt to recursively sort the "subarray" $A[4..12]$, which exceeds the bounds of the original input array. ■

(c) Would WHOSORT still sort correctly if we replaced "$k = \lceil n/5 \rceil$" with "$k = \lfloor n/5 \rfloor$"? Justify your answer.

Solution: ***No.*** The modified algorithm would fail when $n = 14$ (and therefore $k = 2$), because there is not enough overlap between the recursive subarrays to carry all large elements to the last chunk. For example, the algorithm would modify the array NMLKJIHGFEDCBA as follows:

Sort $A[1..6]$	IJKLMNHGFEDCBA
Sort $A[5..14]$	IJKLABCDEFGHMN
Sort $A[1..6]$	ABIJKLCDEFGHMN
Sort $A[3..8]$	ABCDIJKLEFGHMN

In contrast, the original algorithm (with $k = 3$) sorts the same array as follows.

Sort $A[1..9]$	FGHIJKLMNEDCBA
Sort $A[7..14]$	FGHIJKABCDELMN
Sort $A[1..9]$	ABCFGHIJKDELMN
Sort $A[4..12]$	ABCDEFGHIJKLMN

■

(d) What is the running time of WHOSORT? (Set up a running-time recurrence and then solve it, ignoring the floors and ceilings.)

Solution: The running time obeys the recurrence $T(n) = 4T(3n/5) + O(1)$. The level sums of the recursion tree form an increasing geometric series, so the solution is $T(n) = O(4^{\log_{5/3} n}) = O(n^{\log_{5/3} 4}) = O(n^{2.7138309})$. ■

(e) Forty years later, 15th Doctor Ncuti Gatwa discovered the following optimization to WHOSORT, which uses the standard MERGE subroutine from mergesort, which merges two sorted arrays into one sorted array.

```
⟨⟨Sort A⟩⟩
NUWHOSORT(A[1..n]):
  if n < 13
      sort A by brute force
  else
      k = ⌈n/5⌉
      NUWHOSORT(A[1..3k])              ⟨⟨Grant⟩⟩
      NUWHOSORT(A[2k+1..n])            ⟨⟨Whittaker⟩⟩
      MERGE(A[1..2k], A[2k+1..4k])     ⟨⟨Tennant⟩⟩
```

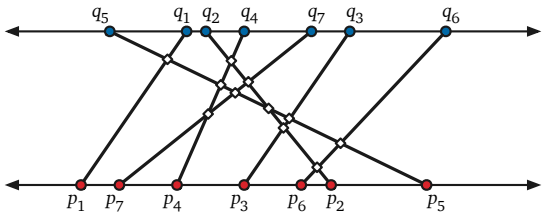
What is the running time of NUWHOSORT?

Solution: The running time of NUWHOSORT obeys the recurrence $T(n) = 2T(3n/5) + O(n)$. The level sums of the recursion tree still form an increasing geometric series, but growing only half as fast as the recursion tree for WHOSORT. The solution is $T(n) = O(2^{\log_{5/3} n}) = O(n^{\log_{5/3} 2}) = O(n^{1.3569155})$.

And yes, the algorithm is still correct. ■

Solved problems

4. Suppose we are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Consider the n line segments connecting each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time. See the example below.



Seven segments with endpoints on parallel lines, with 11 intersecting pairs.

Your input consists of two arrays $P[1..n]$ and $Q[1..n]$ of x-coordinates; you may assume that all $2n$ of these numbers are distinct. No proof of correctness is necessary, but you should justify the running time.

Solution: We begin by sorting the array $P[1..n]$ and permuting the array $Q[1..n]$ to maintain correspondence between endpoints, in $O(n \log n)$ time. Then for any indices $i < j$, segments i and j intersect if and only if $Q[i] > Q[j]$. Thus, our goal is to compute the number of pairs of indices $i < j$ such that $Q[i] > Q[j]$. Such a pair is called an ***inversion***.

We count the number of inversions in Q using the following extension of mergesort; as a side effect, this algorithm also sorts Q. If $n < 100$, we use brute force in $O(1)$ time. Otherwise:

- Color the elements in the Left half $Q[1..\lfloor n/2 \rfloor]$ bLue.
- Color the elements in the Right half $Q[\lfloor n/2 \rfloor + 1..n]$ Red.
- Recursively count inversions in (and sort) the blue subarray $Q[1..\lfloor n/2 \rfloor]$.
- Recursively count inversions in (and sort) the red subarray $Q[\lfloor n/2 \rfloor + 1..n]$.
- Count red/blue inversions as follows:
  - MERGE the sorted subarrays $Q[1..n/2]$ and $Q[n/2+1..n]$, maintaining the element colors.
  - For each blue element $Q[i]$ of the now-sorted array $Q[1..n]$, count the number of smaller red elements $Q[j]$.

The last substep can be performed in $O(n)$ time using a simple for-loop:

```
CountRedBlue(A[1..n]):
  count ← 0
  total ← 0
  for i ← 1 to n
      if A[i] is red
          count ← count + 1
      else
          total ← total + count
  return total
```

Merge and CountRedBlue each run in $O(n)$ time. Thus, the running time of our inversion-counting algorithm obeys the mergesort recurrence $T(n) = 2T(n/2) + O(n)$. (We can safely ignore the floors and ceilings in the recursive arguments.) We conclude that the overall running time of our algorithm is $O(n \log n)$, as required.

Rubric: This is enough for full credit.

In fact, we can execute the third merge-and-count step directly by modifying the Merge algorithm, without any need for "colors". Here changes to the standard Merge algorithm are indicated in red.

```
MergeAndCount(A[1..n], m):
  i ← 1; j ← m + 1; count ← 0; total ← 0
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i + 1; total ← total + count
      else if i > m
          B[k] ← A[j]; j ← j + 1; count ← count + 1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i + 1; total ← total + count
      else
          B[k] ← A[j]; j ← j + 1; count ← count + 1
  for k ← 1 to n
      A[k] ← B[k]
  return total
```

We can further optimize MergeAndCount by observing that *count* is always equal to $j - m - 1$, so we don't need an additional variable. (Proof: Initially, $j = m + 1$ and $count = 0$, and we always increment j and *count* together.)

```
MergeAndCount2(A[1..n], m):
  i ← 1; j ← m + 1; total ← 0
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i + 1; total ← total + j − m − 1
      else if i > m
          B[k] ← A[j]; j ← j + 1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i + 1; total ← total + j − m − 1
      else
          B[k] ← A[j]; j ← j + 1
  for k ← 1 to n
      A[k] ← B[k]
  return total
```

MergeAndCount2 still runs in $O(n)$ time, so the overall running time is still $O(n \log n)$, as required. ■

Rubric: 10 points = 2 for base case + 2 for divide (split and recurse) + 4 for conquer (merge and count) + 2 for time analysis. This is neither the only way to correctly describe this algorithm nor the only correct $O(n \log n)$-time algorithm. No proof of correctness is required.

Max 3 points for a correct $O(n^2)$-time algorithm.

Notice that each boxed algorithm is preceded by a clear English description of the task that algorithm performs—not how the algorithm works, but the relationship between its input and its output. **Each English description is worth 25% of the credit for that algorithm** (rounding to the nearest half-point). For example, the CountRedBlue algorithm is worth 4 points ("conquer"); the English description alone ("For each blue element $Q[i]$ of the now-sorted array $Q[1..n]$, count the number of smaller red elements $Q[j]$.") is worth 1 point.