14. (a) Suppose you 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 $y = 1$. Create a set of n line segments by connect 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. *[Hint: See the previous problem.]*

    (b) Now suppose you are given two sets $\{p_1, p_2, \ldots, p_n\}$ and $\{q_1, q_2, \ldots, q_n\}$ of n points *on the unit circle*. Connect 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 $O(n \log^2 n)$ time. *[Hint: Use your solution to part (a).]*

    ♥(c) Describe an algorithm for part (b) that runs in $O(n \log n)$ time. *[Hint: Use your solution from part (b)!]*

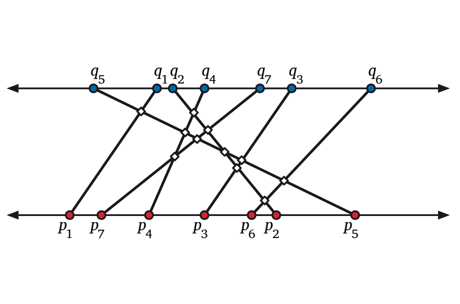


Figure 1.18. Eleven intersecting pairs of segments with endpoints on parallel lines, and ten intersecting pairs of segments with endpoints on a circle.

15. (a) Describe an algorithm that sorts an input array $A[1\,..\,n]$ by calling a subroutine SqrtSort(k), which sorts the subarray $A\big[k+1\,..\,k+\sqrt{n}\big]$ in place, given an arbitrary integer k between 0 and $n - \sqrt{n}$ as input. (To simplify the problem, assume that $\sqrt{n}$ is an integer.) Your algorithm is ***only*** allowed to inspect or modify the input array by calling SqrtSort; in particular, your algorithm must not directly compare, move, or copy array elements. How many times does your algorithm call SqrtSort in the worst case?

    ♣(b) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SqrtSort, prove that no algorithm can sort using $o(f(n))$ calls to SqrtSort.

    (c) Now suppose SqrtSort is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k}, so that repeated square roots are always integers.)

Selection

16. Suppose we are given a set S of n items, each with a *value* and a *weight*. For any element $x \in S$, we define two subsets

    - $S_{<x}$ is the set of elements of S whose value is less than the value of x.
    - $S_{>x}$ is the set of elements of S whose value is more than the value of x.

    For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The ***weighted median*** of R is any element x such that $w(S_{<x}) \le w(S)/2$ and $w(S_{>x}) \le w(S)/2$.

    Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

17. (a) Describe an algorithm to determine in $O(n)$ time whether an arbitrary array $A[1..n]$ contains more than $n/4$ copies of any value.

    (b) Describe and analyze an algorithm to determine, given an arbitrary array $A[1..n]$ and an integer k, whether A contains more than k copies of any value. Express the running time of your algorithm as a function of both n and k.

    Do not use hashing, or radix sort, or any other method that depends on the precise input values, as opposed to their order.

18. Describe an algorithm to compute the median of an array $A[1..5]$ of distinct numbers using at most 6 comparisons. Instead of writing pseudocode, describe your algorithm using a ***decision tree***: A binary tree where each internal node contains a comparison of the form "$A[i] \gtrless A[j]$?" and each leaf contains an index into the array.

19. Consider the generalization of the Blum-Floyd-Pratt-Rivest-Tarjan Select algorithm shown in Figure 1.20, which partitions the input array into $\lceil n/b \rceil$ blocks of size b, instead of $\lceil n/5 \rceil$ blocks of size 5, but is otherwise identical.

    (a) State a recurrence for the running time of Mom$_b$Select, assuming that b is a constant (so the subroutine MedianOfB runs in $O(1)$ time). In particular, how do the sizes of the recursive subproblems depend on the constant b? Consider even b and odd b separately.

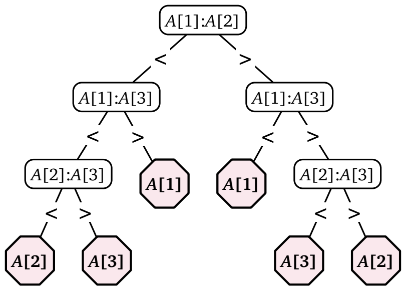


Figure 1.19. Finding the median of a 3-element array using at most 3 comparisons

```
MOM_b SELECT($A[1..n], k$):
    if $n \le b^2$
        use brute force
    else
        $m \leftarrow \lceil n/b \rceil$
        for $i \leftarrow 1$ to $m$
            $M[i] \leftarrow$ MEDIANOFB($A[b(i-1)+1..bi]$)
        $mom_b \leftarrow$ MOM_b SELECT($M[1..m], \lfloor m/2 \rfloor$)
        $r \leftarrow$ PARTITION($A[1..n], mom_b$)
        if $k < r$
            return MOM_b SELECT($A[1..r-1], k$)
        else if $k > r$
            return MOM_b SELECT($A[r+1..n], k-r$)
        else
            return $mom_b$
```

Figure 1.20. A parametrized family of selection algorithms; see problem 19.

(b) What is the worst-case running time of Mom$_1$Select? *[Hint: This is a trick question.]*

♣♥(c) What is the worst-case running time of Mom$_2$Select? *[Hint: This is an unfair question!]*

♥(d) What is the worst-case running time of Mom$_3$Select? Finding an upper bound on the running time is straightforward; the hard part is showing that this analysis is actually tight. *[Hint: See problem 10.]*

♥(e) What is the worst-case running time of Mom$_4$Select? Again, the hard part is showing that the analysis cannot be improved.[15]

[15]The median of four elements is either the second smallest or the second largest. In 2014, Ke Chen and Adrian Dumitrescu proved that if we modify Mom$_4$Select to find second-smallest elements when $k < n/2$ and second-largest elements when $k > n/2$, the resulting algorithm runs in $O(n)$ time! See their paper "Select with Groups of 3 or 4 Takes Linear Time" (WADS 2015, arXiv:1409.3600) for details.

(f) For any constants $b \ge 5$, the algorithm MOM$_b$SELECT runs in $O(n)$ time, but different values of b lead to different constant factors. Let $M(b)$ denote the minimum number of comparisons required to find the median of b numbers. The exact value of $M(b)$ is known only for $b \le 13$:

b	1	2	3	4	5	6	7	8	9	10	11	12	13
$M(b)$	0	1	3	4	6	8	10	12	14	16	18	20	23

For each b between 5 and 13, find an upper bound on the running time of MOM$_b$SELECT of the form $T(n) \le \alpha_b n$ for some explicit constant α_b. (For example, on page 38 we showed that $\alpha_5 \le 16$.)

(g) Which value of b yields the smallest constant α_b? *[Hint: This is a trick question!]*

20. Prove that the variant of the Blum-Floyd-Pratt-Rivest-Tarjan SELECT algorithm shown in Figure 1.21, which uses an extra layer of small medians to choose the main pivot, runs in $O(n)$ time.

```
MOMOMSELECT($A[1..n], k$):
  if $n \le 81$
      use brute force
  else
      $m \leftarrow \lceil n/3 \rceil$
      for $i \leftarrow 1$ to $m$
          $M[i] \leftarrow$ MEDIANOF3($A[3i-2..3i]$)
      $mm \leftarrow \lceil m/3 \rceil$
      for $j \leftarrow 1$ to $mm$
          $Mom[j] \leftarrow$ MEDIANOF3($M[3j-2..3j]$)
      $momom \leftarrow$ MOMOMSELECT($Mom[1..mm], \lfloor mm/2 \rfloor$)
      $r \leftarrow$ PARTITION($A[1..n], momom$)
      if $k < r$
          return MOMOMSELECT($A[1..r-1], k$)
      else if $k > r$
          return MOMOMSELECT($A[r+1..n], k-r$)
      else
          return $momom$
```

Figure 1.21. Selection by median of moms; see problem 20).

21. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$. Describe an algorithm to find the median element in the union of A and B in $\Theta(\log n)$ time. You can assume that the arrays contain no duplicate elements.

(b) Suppose we are given two sorted arrays $A[1..m]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in

$A \cup B$ in $\Theta(\log(m+n))$ time. For example, if $k=1$, your algorithm should return the smallest element of $A \cup B$.) *[Hint: Use your solution to part (a).]*

♥(c) Now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B \cup C$ in $O(\log n)$ time.

(d) Finally, suppose we are given a two dimensional array $A[1..m, 1..n]$ in which every row $A[i, \cdot]$ is sorted, and an integer k. Describe an algorithm to find the kth smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? *[Hint: Solve problem 16 first.]*

Arithmetic

22. In 1854, archaeologists discovered Sumerians clay tablets, carved around 2000BCE, that list the squares of integers up to 59. This discovery led some scholars to conjecture that ancient Sumerians performed multiplication by reduction to squaring, using an identity like $x \cdot y = (x^2 + y^2 - (x-y)^2)/2$. Unfortunately, those same scholars are silent on how the Babylonians supposedly squared larger numbers. Four thousand years later, we can finally rescue these Sumerian mathematicians from their lives of drudgery through the power of recursion!

(a) Describe a variant of Karatsuba's algorithm that squares any n-digit number in $O(n^{\lg 3})$ time, by reducing to squaring three $\lceil n/2 \rceil$-digit numbers. (Karatsuba actually did this in 1960.)

(b) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 6})$ time, by reducing to squaring six $\lceil n/3 \rceil$-digit numbers.

♥(c) Describe a recursive algorithm that squares any n-digit number in $O(n^{\log_3 5})$ time, by reducing to squaring only *five* $(n/3 + O(1))$-digit numbers. *[Hint: What is $(a+b+c)^2 + (a-b+c)^2$?]*

23. (a) Describe and analyze a variant of Karatsuba's algorithm that multiplies any m-digit number and any n-digit number, for any $n \geq m$, in $O(nm^{\lg 3 - 1})$ time.

(b) Describe an algorithm to compute the decimal representation of 2^n in $O(n^{\lg 3})$ time, using the algorithm from part (a) as a subroutine. (The standard algorithm that computes one digit at a time requires $\Theta(n^2)$ time.)

(c) Describe a divide-and-conquer algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(n^{\lg 3})$ time. *[Hint: Watch out for an extra log factor in the running time.]*

♥(d) Suppose we can multiply two n-digit numbers in $O(M(n))$ time. Describe an algorithm to compute the decimal representation of an arbitrary n-bit binary number in $O(M(n)\log n)$ time. *[Hint: The analysis is the hard part; use a domain transformation.]*

24. Consider the following classical recursive algorithm for computing the factorial $n!$ of a non-negative integer n:

```
FACTORIAL(n):
  if n = 0
      return 1
  else
      return n · FACTORIAL(n − 1)
```

(a) How many multiplications does this algorithm perform?

(b) How many bits are required to write $n!$ in binary? Express your answer in the form $\Theta(f(n))$, for some familiar function $f(n)$. *[Hint: $(n/2)^{n/2} < n! < n^n$.]*

(c) Your answer to (b) should convince you that the number of multiplications is *not* a good estimate of the actual running time of FACTORIAL. We can multiply any k-digit number and any l-digit number in $O(k \cdot l)$ time using either the lattice algorithm or duplation and mediation. What is the running time of FACTORIAL if we use this multiplication algorithm as a subroutine?

(d) The following recursive algorithm also computes the factorial function, but using a different grouping of the multiplications:

```
FALLING(n, m):                ⟨⟨Compute n!/(n − m)!⟩⟩
  if m = 0
      return 1
  else if m = 1
      return n
  else
      return FALLING(n, ⌊m/2⌋) · FALLING(n − ⌊m/2⌋, ⌈m/2⌉)
```

What is the running time of FALLING(n, n) if we use grade-school multiplication? *[Hint: As usual, ignore the floors and ceilings.]*

(e) Describe and analyze a variant of Karastuba's algorithm that multiplies any k-digit number and any l-digit number, for any $k \geq l$, in $O(k \cdot l^{\lg 3 - 1}) = O(k \cdot l^{0.585})$ time.

♥(f) What are the running times of FACTORIAL(n) and FALLING(n, n) if we use the modified Karatsuba multiplication from part (e)?

25. The ***greatest common divisor*** of two positive integer x and y, denoted $\mathbf{gcd}(x, y)$, is the largest integer d such that both x/d and y/d are integers.

Euclid's *Elements*, written around 300BC, describes the following recursive algorithm to compute gcd(x, y):[16]

```
EuclidGCD($x, y$):
  if $x = y$
      return $x$
  else if $x > y$
      return EuclidGCD($x - y, y$)
  else
      return EuclidGCD($x, y - x$)
```

(a) Prove that EuclidGCD correctly computes gcd(x, y). Specifically:

  i. Prove that EuclidGCD(x, y) divides both x and y.

  ii. Prove that every divisor of x and y is a divisor of EuclidGCD(x, y).

(b) What is the worst-case running time of EuclidGCD(x, y), as a function of x and y? (Assume that computing $x - y$ requires $O(\log x + \log y)$ time.)

(c) Prove that the following algorithm also computes gcd(x, y):

```
FastEuclidGCD($x, y$):
  if $x = y$
      return $x$
  else if $x > y$
      return FastEuclidGCD($x \bmod y, y$)
  else
      return FastEuclidGCD($x, y \bmod x$)
```

(d) What is the worst-case running time of FastEuclidGCD(x, y), as a function of x and y? (Assume that computing $x \bmod y$ takes $O(\log x \cdot \log y)$ time.)

(e) Prove that the following algorithm also computes gcd(x, y):

```
BinaryGCD($x, y$):
  if $x = y$
      return $x$
  else if $x$ and $y$ are both even
      return $2 \cdot$ BinaryGCD($x/2, y/2$)
  else if $x$ is even
      BinaryGCD($x/2, y$)
  else if $y$ is even
      BinaryGCD($x, y/2$)
  else if $x > y$
      return BinaryGCD($(x - y)/2, y$)
  else
      return BinaryGCD($x, (y - x)/2$)
```

[16]Euclid's algorithm is sometimes incorrectly described as the *first* recursive algorithm, or even the first *nontrivial* algorithm, even though the Egyptian duplation and mediation algorithm—which is both nontrivial and recursive—predates Euclid by at least 1500 years.

(f) What is the worst-case running time of FastEuclidGCD(x, y), as a function of x and y? (Assume that computing $x-y$ takes $O(\log x + \log y)$ time, and computing $z/2$ requires $O(\log z)$ time.)

Arrays

26. Suppose you are given a $2^n \times 2^n$ chessboard with one (arbitrarily chosen) square removed. Describe and analyze an algorithm to compute a tiling of the board by without gaps or overlaps by **L**-shaped tiles, each composed of 3 squares. Your input is the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time. *[Hint: First prove that such a tiling always exists.]*

27. You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates; each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the *same* party by introducing them to each other. Members of the same political party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.[17]

    (a) Suppose more than half of the delegates belong to the same political party. Describe an efficient algorithm that identifies all members of this majority party.

    (b) Now suppose there are more than two parties, but one party has a *plurality*: more people belong to that party than to any other party. Present a practical procedure to precisely pick the people from the plurality political party as parsimoniously as possible, presuming the plurality party is composed of at least p people. Pretty please.

28. Smullyan Island has three types of inhabitants: *knights* always speak the truth; *knaves* always lie; and *normals* sometimes speak the truth and sometimes don't. Everyone on the island knows everyone else's name and type (knight, knave, or normal). You want to learn the type of every inhabitant.

    You can ask any inhabitant to tell you the type of any other inhabitant. Specifically, if you ask "Hey X, what is Y's type?" then X will respond as follows:

[17]Real-world politics is much messier than this simplified model, but this is a theory book!

- If X is a knight, then X will respond with Y's correct type.
- If X is a knave, then X could respond with *either* of the types that Y is *not*.
- If X is a normal, then X could respond with *any* of the three types.

The inhabitants will ignore any questions not of this precise form; in particular, you may not ask an inhabitant about their own type. Asking the same inhabitant the same question multiple times always yields the same answer, so there's no point in asking any question more than once.

(a) Suppose you know that a strict majority of inhabitants are knights. Describe an efficient algorithm to identify the type of every inhabitant.

(b) Prove that if at most half the inhabitants are knights, it is impossible to determine the type of every inhabitant.

29. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

Suppose we want to rotate an $n \times n$ pixel map 90° clockwise. One way to do this, at least when n is a power of two, is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. (Why five? For the same reason the Tower of Hanoi puzzle needs a third peg.) Alternately, we could *first* recursively rotate the blocks and *then* blit them into place.

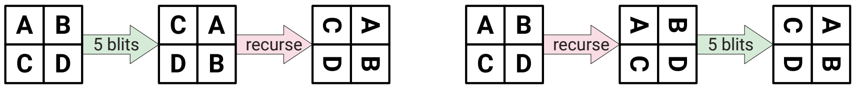


Figure 1.22. Two algorithms for rotating a pixel map.

(a) Prove that both versions of the algorithm are correct when n is a power of 2.

(b) *Exactly* how many blits does the algorithm perform when n is a power of 2?

(c) Describe how to modify the algorithm so that it works for arbitrary n, not just powers of 2. How many blits does your modified algorithm perform?

(d) What is your algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

(e) What if a $k \times k$ blit takes only $O(k)$ time?

30. An array $A[0\,..\,n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and

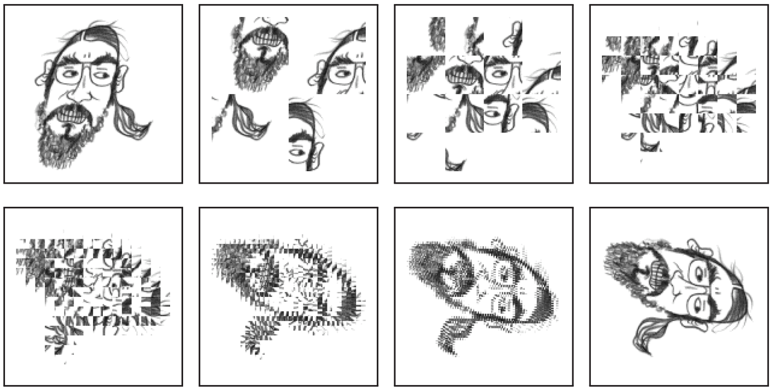


Figure 1.23. The first rotation algorithm (blit then recurse) in action. (See Image Credits at the end of the book.)

$A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3	is bitonic, but
3	6	9	8	7	5	1	2	4	is *not* bitonic.

Describe and analyze an algorithm to find the *smallest* element in an n-element bitonic array in $O(\log n)$ time. You may assume that the numbers in the input array are distinct.

31. Suppose we are given an array $A[1..n]$ of n distinct integers, which could be positive, negative, or zero, sorted in increasing order so that $A[1] < A[2] < \cdots < A[n]$.

    (a) Describe a fast algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists.

    (b) Suppose we know in advance that $A[1] > 0$. Describe an even faster algorithm that either computes an index i such that $A[i] = i$ or correctly reports that no such index exists. *[Hint: This is **really** easy.]*

32. Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are six local minima in the following array:

9	**7**	7	2	**1**	3	7	5	**4**	7	**3**	**3**	4	8	**6**	9
	▲			▲				▲		▲	▲			▲	

We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array **must** have at least one local minimum. Why?]*

33. Suppose you are given a sorted array of n distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, you are given an array $A[1\,..\,n]$ such that some prefix $A[1\,..\,k]$ is sorted in increasing order, the corresponding suffix $A[k+1\,..\,n]$ is sorted in increasing order, and $A[n] < A[1]$.

    For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	1	3	4	5	7	8

(a) Describe and analyze an algorithm to compute the unknown integer k.

(b) Describe and analyze an algorithm to determine if the given array contains a given number x.

34. At the end of the second act of the action blockbuster *Fast and Impossible XIII$\frac{3}{4}$: The Last Guardians of Expendable Justice Reloaded*, the villainous Dr. Metaphor hypnotizes the entire Hero League/Force/Squad, arranges them in a long line at the edge of a cliff, and instructs each hero to shoot the closest taller heroes to their left and right, at a prearranged signal.

    Suppose we are given the heights of all n heroes, in order from left to right, in an array $Ht[1\,..\,n]$. (To avoid salary arguments, the producers insisted that no two heroes have the same height.) Then we can compute the Left and Right targets of each hero in $O(n^2)$ time using the following brute-force algorithm.

```
WHOTARGETSWHOM($Ht[1..n]$):
  for $j \leftarrow 1$ to $n$
      ⟨⟨Find the left target $L[j]$ for hero $j$⟩⟩
      $L[j] \leftarrow$ NONE
      for $i \leftarrow 1$ to $j-1$
          if $Ht[i] > Ht[j]$
              $L[j] \leftarrow i$
      ⟨⟨Find the right target $R[j]$ for hero $j$⟩⟩
      $R[j] \leftarrow$ NONE
      for $k \leftarrow n$ down to $j+1$
          if $Ht[k] > Ht[j]$
              $R[j] \leftarrow k$
  return $L[1..n]$, $R[1..n]$
```

(a) Describe a divide-and-conquer algorithm that computes the output of WhoTargetsWhom in $O(n \log n)$ time.

(b) Prove that at least $\lfloor n/2 \rfloor$ of the n heroes are targets. That is, prove that the output arrays $R[0\,..\,n-1]$ and $L[0\,..\,n-1]$ contain at least $\lfloor n/2 \rfloor$ distinct values (other than None).

(c) Alas, Dr. Metaphor's diabolical plan is successful. At the prearranged signal, all the heroes simultaneously shoot their targets, and all targets fall over the cliff, apparently dead. Metaphor repeats his dastardly experiment over and over; after each massacre, he forces the remaining heroes to choose new targets, following the same algorithm, and then shoot their targets at the next signal. Eventually, only the shortest member of the Hero Crew/Alliance/Posse is left alive.[18]

Describe and analyze an algorithm to compute the number of rounds before Dr. Metaphor's deadly process finally ends. For full credit, your algorithm should run in $O(n)$ time.

35. You are a contestant on the hit game show "Beat Your Neighbors!" You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than any of your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni™, to which you are hopelessly addicted.

(a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than either of its neighbors. How many boxes does your algorithm open in the worst case?

♥(b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

♣♥(c) Prove that your solution to part (b) is optimal up to a constant factor.

36. (a) Let $n = 2^\ell - 1$ for some positive integer ℓ. Suppose someone claims to hold an unsorted array $A[1\,..\,n]$ of *distinct* ℓ-bit strings; thus, exactly one ℓ-bit string does *not* appear in A. Suppose further that the ***only*** way we can access A is by calling the function FetchBit(i, j), which returns the jth bit of the string $A[i]$ in $O(1)$ time. Describe an algorithm to find the missing string in A using only $O(n)$ calls to FetchBit.

[18]In the thrilling final act, Retcon the Squirrel, the last surviving member of the Hero Team/Group/Society, saves everyone by traveling back in time and retroactively replacing the other $n-1$ heroes with lifelike balloon sculptures. So, yeah, basically it's *Avengers: Endgame*.

♥(b) Now suppose $n = 2^\ell - k$ for some positive integers k and ℓ, and again we are given an array $A[1..n]$ of *distinct* ℓ-bit strings. Describe an algorithm to find the k strings that are missing from A using only $O(n \log k)$ calls to FetchBit.

Trees

37. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return both the root and the depth of this subtree. See Figure 1.24 for an example.

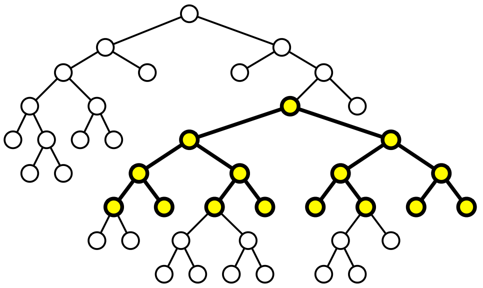


Figure 1.24. The largest complete subtree of this binary tree has depth 3.

38. Let T be a binary tree with n vertices. Deleting any vertex v splits T into at most three subtrees, containing the left child of v (if any), the right child of v (if any), and the parent of v (if any). We call v a ***central*** vertex if each of these smaller trees has at most $n/2$ vertices. See Figure 1.25 for an example.

    Describe and analyze an algorithm to find a central vertex in an arbitrary given binary tree. *[Hint: First prove that every tree has a central vertex.]*

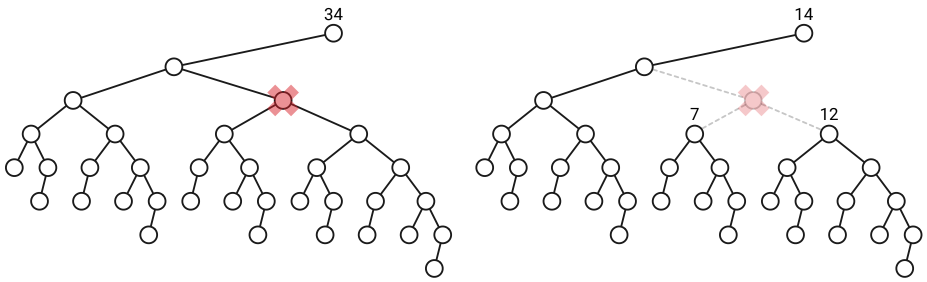


Figure 1.25. Deleting a central vertex in a 34-node binary tree, leaving subtrees with 14, 7, and 12 nodes.

39. (a) Professor George O'Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character **&**. Preorder and postorder traversals of the tree visit the nodes in the following order:

    - Preorder: `I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X`
    - Postorder: `H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I`

    Draw George's binary tree.

    (b) Recall that a binary tree is *full* if every non-leaf node has exactly two children.

    i. Describe and analyze a recursive algorithm to reconstruct an arbitrary *full* binary tree, given its preorder and postorder node sequences as input.

    ii. Prove that there is no algorithm to reconstruct an *arbitrary* binary tree from its preorder and postorder node sequences.

    (c) Describe and analyze a recursive algorithm to reconstruct an *arbitrary* binary tree, given its preorder and *inorder* node sequences as input.

    (d) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence.

    ♥(e) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence, ***in $O(n)$ time***.

    In parts (b)–(e), assume that all keys are distinct and that the input is consistent with at least one binary tree.

40. Suppose we have n points scattered inside a two-dimensional box. A ***kd-tree***[19] recursively subdivides the points as follows. If the box contains no points in its interior, we are done. Otherwise, we split the box into two smaller boxes with a *vertical* line, through a median point inside the box (*not* on its boundary), partitioning the points as evenly as possible. Then we recursively build a kd-tree for the points in each of the two smaller boxes, *after rotating them 90 degrees*. Thus, we alternate between splitting vertically and splitting horizontally at each level of recursion. The final empty boxes are called *cells*.

[19]The term "kd-tree" (pronounced "kay *dee* tree") was originally an abbreviation for "k-dimensional tree", but modern usage ignores this etymology, in part because nobody in their right mind would *ever* use the letter k to denote dimension instead of the *obviously* superior d. Etymological consistency would require calling the data structure in this problem a "2d-tree" (or perhaps a "2-d tree"), but the standard nomenclature is now "two-dimensional kd-tree". See also: B-tree (maybe), alpha shape, beta skeleton, epsilon net, Potomac River, Mississippi River, Lake Michigan, Lake Tahoe, Manhattan Island, the La Brea Tar Pits, Sahara Desert, Mount Kilimanjaro, South Vietnam, East Timor, the Milky Way Galaxy, the City of Townsville, and self-driving automobiles.

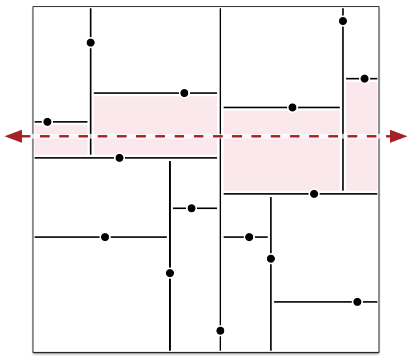


Figure 1.26. A kd-tree for 15 points. The dashed line crosses the four shaded cells.

(a) How many cells are there, as a function of n? Prove your answer is correct.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[Hint: There is more than one function f such that $f(16) = 4$.]*

(c) Suppose we are given n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

(d) Describe an analyze an efficient algorithm that counts, given a kd-tree containing n points, the number of points that lie inside a rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

♥41. Bob Ratenbur, a new student in CS 225, is trying to write code to perform preorder, inorder, and postorder traversals of binary trees. Bob sort-of understands the basic idea behind the traversal algorithms, but whenever he actually tries to implement them, he keeps mixing up the recursive calls. Five minutes before the deadline, Bob frantically submits code with the following structure:

```
PreOrder(v):
    if v = Null
        return
    else
        print label(v)
        ████Order(left(v))
        ████Order(right(v))
```

```
InOrder(v):
    if v = Null
        return
    else
        ████Order(left(v))
        print label(v)
        ████Order(right(v))
```

```
PostOrder(v):
    if v = Null
        return
    else
        ████Order(left(v))
        ████Order(right(v))
        print label(v)
```

Each ████ in this pseudocode hides one of the prefixes Pre, In, or Post. Moreover, each of the following function calls appears exactly once in Bob's submitted code:

```
PREORDER(left(v))      PREORDER(right(v))
INORDER(left(v))       INORDER(right(v))
POSTORDER(left(v))     POSTORDER(right(v))
```

Thus, there are precisely 36 possibilities for Bob's code. Unfortunately, Bob accidentally deleted his source code after submitting the executable, so neither you nor he knows which functions were called where.

Now suppose you are given the output of Bob's traversal algorithms, executed on some ***unknown*** binary tree T. Bob's output has been helpfully parsed into three arrays $Pre[1..n]$, $In[1..n]$, and $Post[1..n]$. You may assume that these traversal sequences are consistent with exactly one binary tree T; in particular, the vertex labels of the unknown tree T are distinct, and every internal node in T has exactly two children.

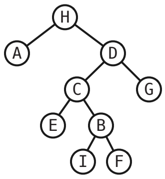
(a) Describe an algorithm to reconstruct the unknown tree T from the given traversal sequences.

(b) Describe an algorithm that either reconstructs Bob's code from the given traversal sequences, or correctly reports that the traversal sequences are consistent with more than one set of algorithms.

For example, given the input

$$Pre[1..n] = [\mathtt{H\ A\ E\ C\ B\ I\ F\ G\ D}]$$
$$In[1..n] = [\mathtt{A\ H\ D\ C\ E\ I\ F\ B\ G}]$$
$$Post[1..n] = [\mathtt{A\ E\ I\ B\ F\ C\ D\ G\ H}]$$

your first algorithm should return the following tree:



and your second algorithm should reconstruct the following code:

```
PREORDER(v):
  if v = NULL
      return
  else
      print label(v)
      PREORDER(left(v))
      POSTORDER(right(v))
```

```
INORDER(v):
  if v = NULL
      return
  else
      POSTORDER(left(v))
      print label(v)
      PREORDER(right(v))
```

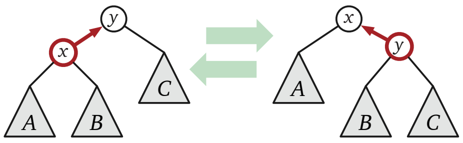
```
POSTORDER(v):
  if v = NULL
      return
  else
      INORDER(left(v))
      INORDER(right(v))
      print label(v)
```

♥42. Let T be a binary tree whose nodes store distinct numerical values. Recall that T is a ***binary search tree*** if and only if either (1) T is empty, or (2) T satisfies the following recursive conditions:

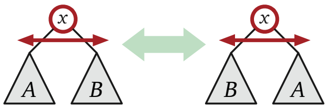
- The left subtree of T is a binary search tree.
- All values in the left subtree are smaller than the value at the root.
- The right subtree of T is a binary search tree.
- All values in the right subtree are larger than the value at the root.

Consider the following pair of operations on binary trees:

- ***Rotate*** an arbitrary node upward.[20]



- ***Swap*** the left and right subtrees of an arbitrary node.



In both of these operations, some, all, or none of the subtrees A, B, and C could be empty.

(a) Describe an algorithm to transform an *arbitrary* n-node binary tree with distinct node values into a binary search tree, using at most $O(n^2)$ rotations and swaps. Figure 1.27 shows a sequence of eight operations that transforms a five-node binary tree into a binary search tree.

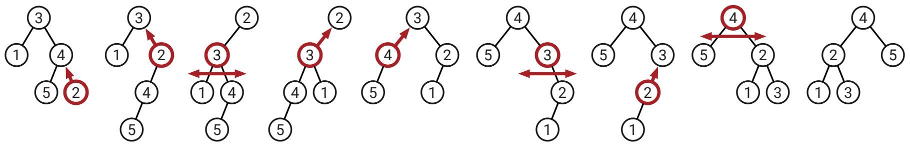


Figure 1.27. "Sorting" a binary tree: rotate 2, rotate 2, swap 3, rotate 3, rotate 4, swap 3, rotate 2, swap 4.

Your algorithm is not allowed to directly modify parent or child pointers, create new nodes, or delete old nodes; the *only* way to modify the tree is through rotations and swaps.

On the other hand, you may *compute* anything you like for free, as long as that computation does not modify the tree; the running time of your algorithm is *defined* to be the number of rotations and swaps that it performs.

♥(b) Describe an algorithm to transform an arbitrary n-node binary tree into a binary search tree, using at most $O(n \log n)$ rotations and swaps.

[20]Rotations preserve the inorder sequence of nodes in a binary tree. Partly for this reason, rotations are used to maintain several types of balanced binary search trees, including AVL trees, red-black trees, splay trees, scapegoat trees, and treaps. See http://algorithms.wtf for lecture notes on all of these data structures.

(c) Prove that any n-node binary *search* tree can be transformed into any other binary *search* tree with the same node values, using only $O(n)$ rotations (and *no* swaps).

♥(d) **Open problem:** Either describe an algorithm to transform an arbitrary n-node binary tree into a binary search tree using only $O(n)$ rotations and swaps, or prove that no such algorithm is possible. *[Hint: I don't think it's possible.]*

Where, however, the ambiguity cannot be cleared up, either by the rule of faith or by the context, there is nothing to hinder us to point the sentence according to any method we choose of those that suggest themselves.

— Augustine of Hippo, *De doctrina Christiana* (397CE)
Translated by Marcus Dods (1892)

I dropped my dinner, and ran back to the laboratory. There, in my excitement, I tasted the contents of every beaker and evaporating dish on the table. Luckily for me, none contained any corrosive or poisonous liquid.

— Constantine Fahlberg on his discovery of saccharin,
Scientific American (1886)

The greatest challenge to any thinker is stating the problem in a way that will allow a solution.

— attributed to Bertrand Arthur William Russell

When you come to a fork in the road, take it.

— Yogi Berra (giving directions to his house)

2 Backtracking

This chapter describes another important recursive strategy called ***backtracking***. A backtracking algorithm tries to construct a solution to a computational problem incrementally, one small piece at a time. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it recursively evaluates ***every*** alternative and then chooses the best one.

2.1 N Queens

The prototypical backtracking problem is the classical ***n* *Queens Problem***, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym “Schachfreund”) for the standard 8×8 board and by François-Joseph Eustache Lionnet in 1869 for the more general $n \times n$ board. The problem is to place n queens on an $n \times n$ chessboard, so that no two queens are attacking each other.

For readers not familiar with the rules of chess, this means that no two queens are in the same row, the same column, or the same diagonal.

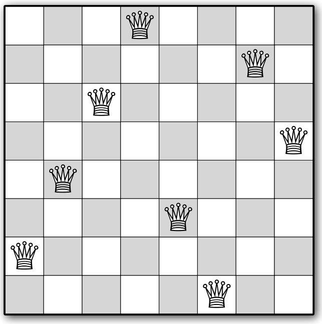


Figure 2.1. One solution to the 8 queens problem, represented by the array $[4,7,3,8,2,5,1,6]$

In a letter written to his friend Heinrich Schumacher in 1850, the eminent mathematician Carl Friedrich Gauss wrote that one could easily confirm Franz Nauck's claim that the Eight Queens problem has 92 solutions by trial and error in a few hours. ("*Schwer ist es übrigens nicht, durch ein methodisches Tatonnieren sich diese Gewifsheit zu verschaffen, wenn man eine oder ein paar Stunden daran wenden will.*") His description *Tatonnieren* comes from the French *tâttoner*, meaning to feel, grope, or fumble around blindly, as if in the dark. Unfortunately, Gauss did not describe the mechanical groping method he had in mind, but he did observe that any solution can be represented by a permutation of the integers 1 through 8 satisfying a few simple arithmetic properties.

Following Gauss, let's represent possible solutions to the n-queens problem using an array $Q[1..n]$, where $Q[i]$ indicates which square in row i contains a queen. Then we can find solutions using the following recursive strategy, described in 1882 by the French recreational mathematician Édouard Lucas, who attributed the method to Emmanuel Laquière. We place queens on the board one row at a time, starting with the top row. To place the rth queen, we methodically try all n squares in row r from left to right in a simple for loop. If a particular square is attacked by an earlier queen, we ignore that square; otherwise, we tentatively place a queen on that square and *recursively* grope for consistent placements of the queens in later rows.

Figure 2.2 shows the resulting algorithm, which recursively enumerates *all* complete n-queens solutions that are consistent with a given partial solution. The input parameter r is the index of the first empty row; the prefix $Q[1..r-1]$ contains the position of the first $r-1$ queens. In particular, to compute all n-queens solutions with no restrictions, we would call PLACEQUEENS$(Q[1..n], 1)$. The outer for-loop considers all possible placements of a queen on row r; the inner for-loop checks whether a candidate placement of row r is consistent with the queens that are already on the first $r-1$ rows.

```
PLACEQUEENS($Q[1\,..\,n], r$):
  if $r = n + 1$
      print $Q$
  else
      for $j \leftarrow 1$ to $n$
          $legal \leftarrow$ TRUE
          for $i \leftarrow 1$ to $r - 1$
              if $(Q[i] = j)$ or $(Q[i] = j + r - i)$ or $(Q[i] = j - r + i)$
                  $legal \leftarrow$ FALSE
          if $legal$
              $Q[r] \leftarrow j$
              PLACEQUEENS($Q[1\,..\,n], r + 1$)        ⟨⟨Recursion!⟩⟩
```

Figure 2.2. Laquière's backtracking algorithm for the n-queens problem.

The execution of PLACEQUEENS can be illustrated using a ***recursion tree***. Each node in this tree corresponds to recursive subproblem, and thus to a legal partial solution; in particular, the root corresponds to the empty board (with $r = 0$). Edges in the recursion tree correspond to recursive calls. Leaves correspond to partial solutions that cannot be further extended, either because there is already a queen on every row, or because every position in the next empty row is attacked by an existing queen. The backtracking search for complete solutions is equivalent to a depth-first search of this tree.

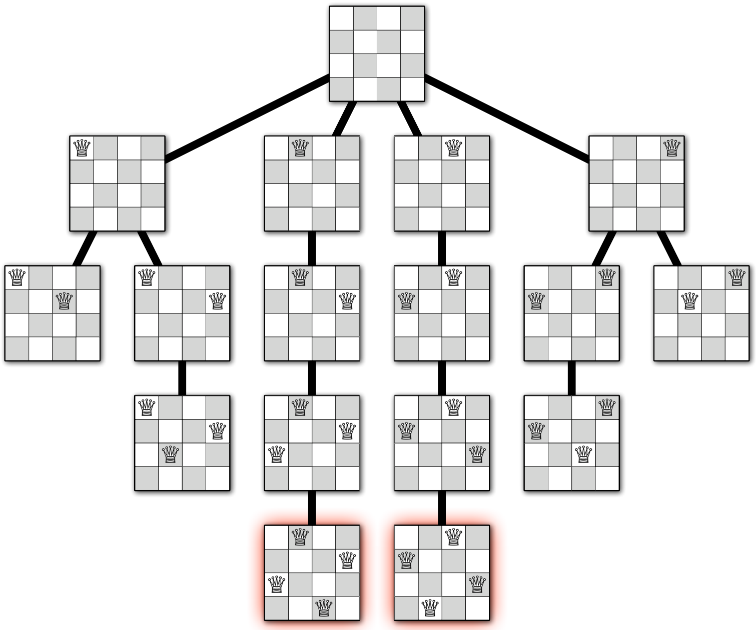


Figure 2.3. The complete recursion tree of Laquière's algorithm for the 4 queens problem.

2.2 Game Trees

Consider the following simple two-player game[1] played on an $n \times n$ square grid with a border of squares; let's call the players Horace Fahlberg-Remsen and Vera Rebaudi.[2] Each player has n tokens that they move across the board from one side to the other. Horace's tokens start in the left border, one in each row, and move ***hor***izontally to the right; symmetrically, Vera's tokens start in the top border, one in each column, and move ***ver***tically downward. The players alternate turns. In each of his turns, Horace either *moves* one of his tokens one step to the right into an empty square, or *jumps* one of his tokens over exactly one of Vera's tokens into an empty square two steps to the right. If no legal moves or jumps are available, Horace simply passes. Similarly, Vera either moves or jumps one of her tokens downward in each of her turns, unless no moves or jumps are possible. The first player to move all their tokens off the edge of the board wins. (It's not hard to prove that as long as there are tokens on the board, at least one player can legally move, so someone eventually wins.)

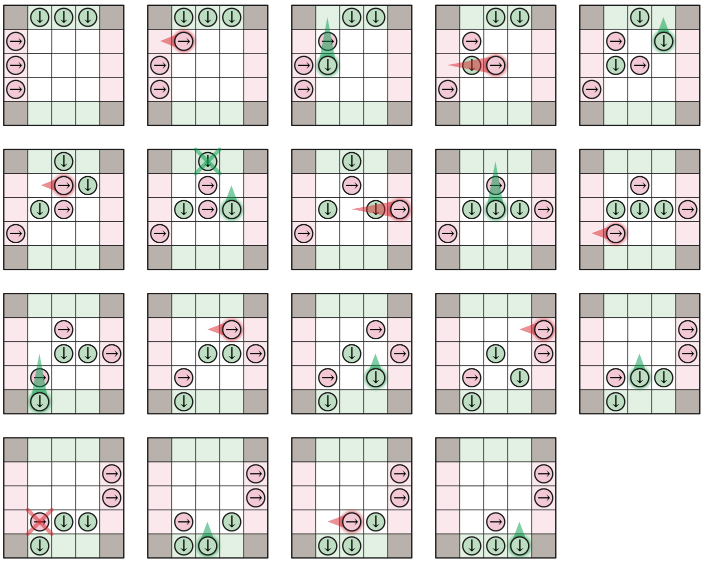


Figure 2.4. Vera wins the 3×3 fake-sugar-packet game.

[1] I don't know what this game is called, or even if I'm remembering the rules correctly; I learned it (or something like it) from Lenny Pitt, who recommended playing it with fake-sugar packets at restaurants.

[2] Constantin Fahlberg and Ira Remsen synthesized saccharin for the first time in 1878, while Fahlberg was a postdoc in Remsen's lab investigating coal tar derivatives. In 1900, Ovidio Rebaudi published the first chemical analysis of *ka'a he'ê*, a medicinal plant cultivated by the Guaraní for more than 1500 years, now more commonly known as *Stevia rebaudiana*.

Unless you've seen this game before[3], you probably don't have any idea how to play it well. Nevertheless, there is a relatively simple backtracking algorithm that can play this game—or any two-player game without randomness or hidden information—*perfectly*. That is, if we drop you into the middle of a game, and it is *possible* to win against another perfect player, the algorithm will tell you how to win.

A ***state*** of the game consists of the locations of all the pieces and the identity of the current player. These states can be connected into a *game tree*, which has an edge from state x to state y if and only if the current player in state x can legally move to state y. The root of the game tree is the initial position of the game, and every path from the root to a leaf is a complete game.

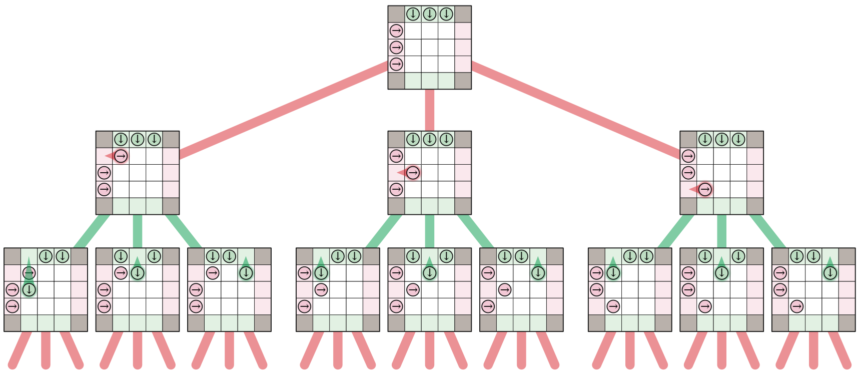


Figure 2.5. The first two levels of the fake-sugar-packet game tree.

To navigate through this game tree, we recursively define a game state to be ***good*** or ***bad*** as follows:

- A game state is *good* if either the current player has already won, or if the current player can move to a bad state for the opposing player.
- A game state is *bad* if either the current player has already lost, or if every available move leads to a good state for the opposing player.

Equivalently, a non-leaf node in the game tree is good if it has at least one bad child, and a non-leaf node is bad if all its children are good. By induction, any player that finds the game in a good state on their turn can win the game, even if their opponent plays perfectly; on the other hand, starting from a bad state, a player can win only if their opponent makes a mistake.

This recursive definition immediately suggests the following recursive backtracking algorithm to determine whether a given game state is good or bad. At its core, this algorithm is just a depth-first search of the game tree; equivalently, the game tree is the recursion tree of the algorithm. A simple modification of

[3]If you have, please tell me where!