Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word "research"*. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term* "research" *in his presence. You can imagine how he felt, then, about the term* "mathematical"*. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?*

— Richard Bellman, on the origin of his term "dynamic programming", in *Eye of the Hurricane: An Autobiography* (1984)

If we all listened to the professor, we may be all looking for professor jobs.

— Pittsburgh Steelers' head coach Bill Cowher, responding to David Romer's dynamic-programming analysis of football strategy (2003)

Chapter 3

Dynamic Programming

3.1 Fibonacci Numbers

Recursive Definitions Are Recursive Algorithms

The Fibonacci numbers F_n, named after Leonardo Fibonacci Pisano, one of the mathematicians who popularized "algorism" in Europe in the 13th century, are defined as follows: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them:

```
RecFibo(n):
  if (n < 2)
      return n
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

How long does this algorithm take? Except for the recursive calls, the entire algorithm requires only a constant number of steps: one comparison and possibly one addition. If $T(n)$ represents the number of recursive calls to RecFibo, we have the recurrence

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1.$$

This looks an awful lot like the recurrence for Fibonacci numbers! Writing out the first several values of $T(n)$ quickly suggests the closed-form solution $\boldsymbol{T(n) = 2F_{n+1} - 1}$, which we can quickly verify by induction (hint, hint). So computing F_n using this algorithm takes about twice as long as just counting to F_n! Methods beyond the scope of this book imply that $F_n = \Theta(\phi^n)$, where $\phi = (\sqrt{5} + 1)/2 \approx 1.61803$ is the so-called *golden ratio*. In short, the running time of this this naive recursive algorithm is exponential in n.

We can also see this exponential growth intuitively as follows. We can think of the recursion tree for RecFibo as a big binary tree of additions, with only 0s and 1s at the leaves. Since the eventual output is F_n, exactly F_n of the leaves must have value 1; these leaves represent the calls to RecRibo(1). An easy inductive argument (hint, hint) implies that RecFibo(0) is called exactly F_{n-1} times, but if we just want an asymptotic bound, it's enough to observe that the number of calls to RecFibo(0) is at most the number of calls to RecFibo(1). Thus, the recursion tree has $F_n + F_{n-1} = F_{n+1} = O(F_n)$ leaves, and therefore, because it's a full binary tree, $2F_{n+1} - 1 = O(F_n)$ nodes altogether.

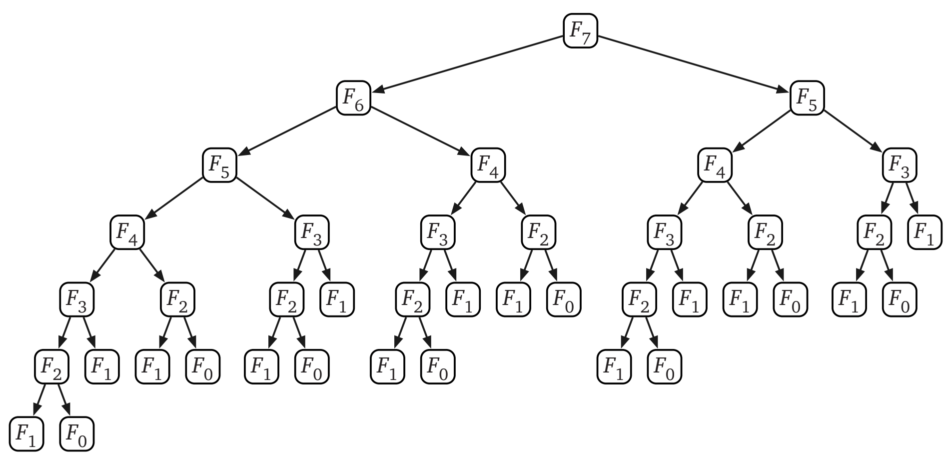


Figure 3.1. The recursion tree for computing F_7; arrows represent recursive calls.

Memo(r)ization: Remember Everything

The obvious reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers over and over and over. A single call to RecFibo(n) results in one recursive call to RecFibo($n-1$), two recursive calls to RecFibo($n-2$), three recursive calls to RecFibo($n-3$), five recursive calls to RecFibo($n-4$), and in general F_{k-1}

recursive calls to RecFibo($n-k$) for any integer $0 \le k < n$. Each call is recomputing some Fibonacci number from scratch.

We can speed up our recursive algorithm considerably just by writing down the results of our recursive calls and looking them up again if we need them later. This process was dubbed *memoization* by Richard Michie in the late 1960s.[1]

```
MemFibo(n):
  if (n < 2)
      return n
  else
      if F[n] is undefined
          F[n] ← MemFibo(n − 1) + MemFibo(n − 2)
      return F[n]
```

Memoization clearly decreases the running time of the algorithm, but by how much? If we actually trace through the recursive calls made by MemFibo, we find that the array $F[\,]$ is filled from the bottom up: first $F[2]$, then $F[3]$, and so on, up to $F[n]$. This pattern can be verified by induction: Each entry $F[i]$ is filled only after its predecessor $F[i-1]$. If we ignore the time spent in recursive calls, it requires only constant time to evaluate the recurrence for each Fibonacci number F_i. But by design, the recurrence for F_i is evaluated only once for each index i. We conclude that MemFibo performs only $O(n)$ additions, an *exponential* improvement over the naïve recursive algorithm!

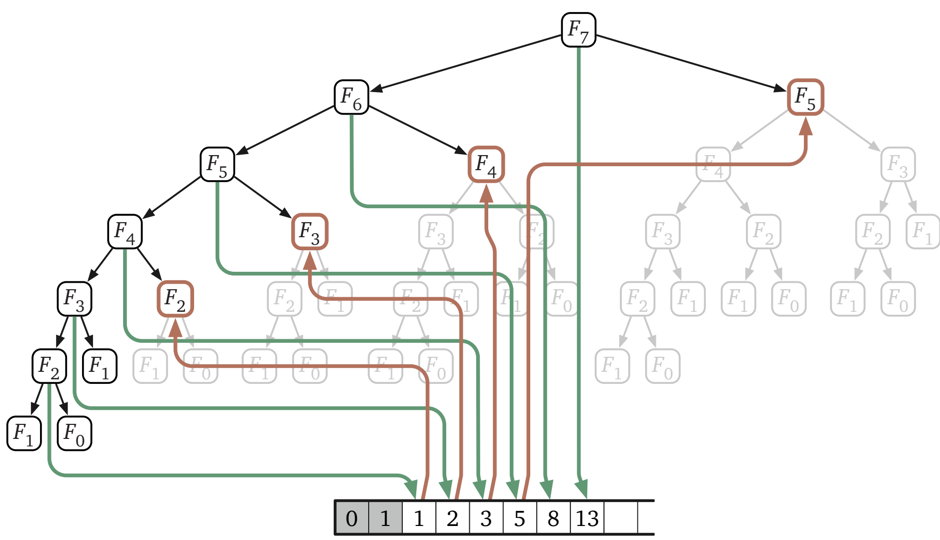


Figure 3.2. The recursion tree for F_7 trimmed by memoization. Green arrows indicate writing into the memoization array; read arrows indicate reading from the memoization array.

[1]“My name is Elmer J. Fudd, millionaire. I own a mansion and a yacht.”

Dynamic Programming: Fill Deliberately

But once we see how the array $F[\,]$ is filled, we can replace the recursion with a simple loop that intentionally fills the array in order, instead of relying on the complicated recursion to do it for us 'accidentally'.

```
ITERFIBO(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

Now the time analysis is immediate: IterFibo clearly uses ***$O(n)$ additions*** and stores $O(n)$ *integers*.

This gives us our first explicit *dynamic programming* algorithm. The dynamic programming paradigm was formalized and popularized by Richard Bellman in the mid-1950s, while working at the RAND Corporation, although he was not the first to use the technique.[2] Many years after the fact, Bellman claimed to have deliberately chosen the name "dynamic programming" to hide the mathematical character of his work from his military bosses, who were actively hostile toward anything resembling mathematical research.[3] The word "programming" does not refer to writing code, but rather to the older sense of *planning* or *scheduling*, typically by filling in a table. For example, sports programs and theater programs are schedules of important events (with ads); television programming involves filling each available time slot with a show (and ads); degree programs are schedules of classes to be taken (with ads). The Air Force funded Bellman and others to develop methods for constructing training and logistics schedules, or as they called them, "programs". The word "dynamic" originally referred to the multistage, time-varying processes that Bellman and his colleagues were attempting to optimize.

Don't Remember Everything After All

In many dynamic programming algorithms, it is not necessary to retain *all* intermediate results through the entire computation. For example, we can significantly reduce the space requirements of our algorithm IterFibo by maintaining only the two newest elements of the array:

---

[2]For example, Pierre Massé used dynamic programming techniques to optimize the operation of hydroelectric dams in France during the Vichy regime; Massé's work was published in 1944, six years before Bellman coined the phrase "dynamic programming".

[3]Bellman's quotation at the start of this chapter refers to Charles Erwin Wilson, who became Secretary of Defense started in January 1953, after a dozen years as the president of General Motors. "Engine Charlie" reorganized the Department of Defense and significantly decrease its budget in his first year in office, with the explicit goal of running the Department much more like an industrial corporation. However, Bellman's first published use of the term "dynamic programming" appeared in 1952, months before Wilson took office, so Bellman's history is at least slightly embellished.

```
ITERFIBO2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

(This algorithm uses the non-standard but perfectly consistent base case $F_{-1} = 1$ so that IterFibo2(0) returns the correct value 0.)

Faster! Faster!

Even this algorithm can be improved further, using the following wonderful fact:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x+y \end{bmatrix}$$

In other words, multiplying a two-dimensional vector by the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$ does exactly the same thing as one iteration of the inner loop of IterFibo2. This might lead us to believe that multiplying by the matrix n times is the same as iterating the loop n times:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$

A quick inductive argument proves this fact. So if we want the nth Fibonacci number, we just have to compute the nth power of the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$. If we use repeated squaring, computing the nth power of something requires only $O(\log n)$ multiplications. In this case, that means $O(\log n)$ 2×2 matrix multiplications, each of which reduces to a constant number of integer multiplications and additions. Thus, we can compute F_n in only ***$O(\log n)$ integer arithmetic operations***.

This is an exponential speedup over the standard iterative algorithm, which was already an exponential speedup over our original recursive algorithm. Right?

Whoa! Not so fast!

Well, not exactly. Fibonacci numbers grow exponentially fast. The nth Fibonacci number is approximately $n \log_{10} \phi \approx n/5$ decimal digits long, or $n \log_2 \phi \approx 2n/3$ bits. So we can't possibly compute F_n in logarithmic time — we need $\Omega(n)$ time just to write down the answer!

The way out of this apparent paradox is to observe that ***we can't perform arbitrary-precision arithmetic in constant time***. Let $M(n)$ denote the time required to multiply two n-digit numbers. The matrix-based algorithm's actual running time obeys the recurrence $T(n) = T(\lfloor n/2 \rfloor) + M(n)$, which solves to $\boldsymbol{T(n) = O(M(n))}$ using recursion trees. The fastest known multiplication algorithm runs in time $O(n \log n 2^{O(\log^* n)})$, so that is also the running time of the fastest algorithm known to compute Fibonacci numbers.

Is this algorithm slower than our initial "linear-time" iterative algorithm? No! Addition isn't free, either. Adding two n-digit numbers takes $O(n)$ time, so the running time of the iterative algorithm is $\boldsymbol{O(n^2)}$. (Do you see why?) The matrix-squaring algorithm really is faster than the iterative addition algorithm, but not exponentially faster.

In the original recursive algorithm, the extra cost of arbitrary-precision arithmetic is overwhelmed by the huge number of recursive calls. The correct recurrence is $T(n) = T(n-1) + T(n-2) + O(n)$, for which the annihilator method still implies the solution $T(n) = O(\phi^n)$.

3.2 Longest Increasing Subsequence

In the previous chapter, we developed a recursive algorithm to find the length of the longest increasing subsequence of a given sequence of numbers. Given an array $A[1..n]$, the length of the longest increasing subsequence is computed by the function call LISBIGGER($-\infty, A[1..n]$), where LISBIGGER is the following recursive algorithm:

```
LISBigger(prev, A[1..n]):
  if n = 0
      return 0
  else
      max ← LISBigger(prev, A[2..n])
      if A[1] > prev
          L ← 1 + LISBigger(A[1], A[2..n])
          if L > max
              max ← L
      return max
```

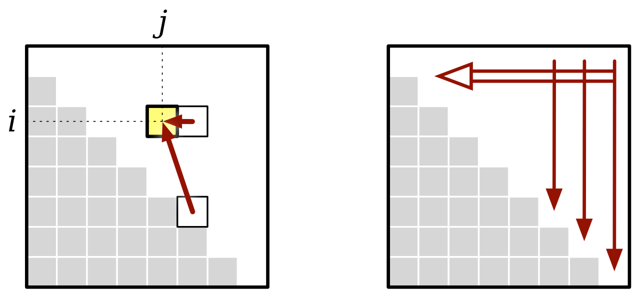
We can simplify our notation slightly with two simple observations. First, the input variable *prev* is always either $-\infty$ or an element of the input array. Second, the second argument of LISBIGGER is always a *suffix* of the original input array. If we add a new sentinel value $A[0] = -\infty$ to the input array, we can identify any recursive subproblem with two array indices.

Thus, we can rewrite the recursive algorithm as follows. Add the sentinel value $A[0] = -\infty$. Let $LIS(i, j)$ denote the length of the longest increasing subsequence of $A[j..n]$ with all elements larger than $A[i]$. Our goal is to compute $LIS(0, 1)$. For all $i < j$, we have

$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i, j+1),\ 1 + LIS(j, j+1)\} & \text{otherwise} \end{cases}$$

Because each recursive subproblem can be identified by two indices i and j, we can store the intermediate values in a two-dimensional array $\boldsymbol{LIS[0..n, 1..n]}$.[4] Since there are $O(n^2)$ entries in the table, our memoized algorithm uses $\boldsymbol{O(n^2)}$ ***space***. Each entry in the table can be computed in $O(1)$ time once we know its predecessors, so our memoized algorithm runs in $\boldsymbol{O(n^2)}$ ***time***.

It's not immediately clear what order the recursive algorithm fills the rest of the table; all we can tell from the recurrence is that each entry $LIS[i, j]$ is filled in *after* the entries $LIS[i, j+1]$ and $LIS[j, j+1]$ in the next columns. But just this partial information is enough to give us an explicit evaluation order. If we fill in our table one column at a time, from right to left, then whenever we reach an entry in the table, the entries it depends on are already available.



Dependencies in the memoization table for longest increasing subsequence, and a legal evaluation order

Putting everything together, we have the following dynamic programming algorithm:

```
LIS(A[1..n]):
  A[0] ← −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                     ⟨⟨Base cases⟩⟩
        LIS[i, n+1] ← 0

  for j ← n downto 1
        for i ← 0 to j−1
              if A[i] ≥ A[j]
                    LIS[i, j] ← LIS[i, j+1]
              else
                    LIS[i, j] ← max{LIS[i, j+1], 1 + LIS[j, j+1]}

  return LIS[0, 1]
```

As expected, the algorithm clearly uses $\boldsymbol{O(n^2)}$ ***time and space***. However, we can reduce the space to $O(n)$ by only maintaining the two most recent columns of the table, $LIS[\cdot, j]$ and $LIS[\cdot, j+1]$.[5]

[4]In fact, we only need half of this array, because we always have $i < j$. But even if we cared about constant factors in this class (we don't), this would be the wrong time to worry about them. The first order of business is to find an algorithm that actually *works*; once we have that, *then* we can think about optimizing it.

[5]See, I told you not to worry about constant factors yet!

This is not the only recursive strategy we could use for computing longest increasing subsequences efficiently. Here is another recurrence that gives us the $O(n)$ space bound for free. Let $LIS'(i)$ denote the length of the longest increasing subsequence of $A[i\,..\,n]$ that starts with $A[i]$. Our goal is to compute $LIS'(0)-1$; we subtract 1 to ignore the sentinel value $-\infty$. To define $LIS'(i)$ recursively, we only need to specify the *second* element in subsequence; the Recursion Fairy will do the rest.

$$LIS'(i) = 1 + \max\left\{LIS'(j) \mid j > i \text{ and } A[j] > A[i]\right\}$$

Here, I'm assuming that $\max \varnothing = 0$, so that the base case is $L'(n) = 1$ falls out of the recurrence automatically. Memoizing this recurrence requires only $\boldsymbol{O(n)}$ ***space***, and the resulting algorithm runs in $\boldsymbol{O(n^2)}$ ***time***. To transform this memoized recurrence into a dynamic programming algorithm, we only need to guarantee that $LIS'(j)$ is computed before $LIS'(i)$ whenever $i < j$.

```
LIS2(A[1..n]):
  A[0] = −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← n downto 0
      LIS′[i] ← 1
      for j ← i + 1 to n
          if A[j] > A[i] and 1 + LIS′[j] > LIS′[i]
              LIS′[i] ← 1 + LIS′[j]
  return LIS′[0] − 1                 ⟨⟨Don’t count the sentinel⟩⟩
```

3.3 The Pattern: Smart Recursion

In a nutshell, dynamic programming is *recursion without repetition*. Dynamic programming algorithms store the solutions of intermediate subproblems, often *but not always* in some kind of array or table. Many algorithms students make the mistake of focusing on the table (because tables are easy and familiar) instead of the *much* more important (and difficult) task of finding a correct recurrence. As long as we memoize the correct recurrence, an explicit table isn't really necessary, but if the recursion is incorrect, nothing works.

Dynamic programming is *not* about filling in tables.
It's about smart recursion!

Dynamic programming algorithms are almost always developed in two distinct stages.

1. **Formulate the problem recursively.** Write down a recursive formula or algorithm for the whole problem in terms of the answers to smaller subproblems. This is the hard part. A complete recursive formulation has two parts:

(a) **Specification.** Describe the problem that you want to solve recursively, in coherent and precise English—not *how* to solve that problem, but *what* problem you're trying to solve. Without this specification, it is impossible, even in principle, to determine whether your solution is correct.[6]

(b) **Solution.** Give a clear recursive formula or algorithm for the whole problem in terms of the answers to smaller instances of *exactly* the same problem.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

(a) **Identify the subproblems.** What are all the different ways can your recursive algorithm call itself, starting with some initial input? For example, the argument to RecFibo is always an integer between 0 and n.

(b) **Choose a memoization data structure.** Find a data structure that can store the solution to *every* subproblem you identified in step (a). This is usually *but not always* a multidimensional array.

(c) **Identify dependencies.** Except for the base cases, every subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

(d) **Find a good evaluation order.** Order the subproblems so that each one comes *after* the subproblems it depends on. Typically, you should consider the base cases first, then the subproblems that depends only on base cases, and so on. The dependencies you identified in the previous step define a partial order over the subproblems; now you need to find a linear extension of that partial order. ***Be careful!***

(e) **Analyze space and running time.** The number of distinct subproblems determines the space complexity of your memoized algorithm. To compute the total running time, add up the running times of all possible subproblems, *assuming deeper recursive calls are already memoized*. You can actually do this immediately after step (a).

(f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence.

Of course, you have to prove that each of these steps is correct. If your recurrence is wrong, or if you try to build up answers in the wrong order, your algorithm won't work!

[6] In my algorithms classes, as official course policy, omitting this specification is an automatic zero, even if the rest of the algorithm is correct.

3.4 Warning: Greed is Stupid

If we're incredibly lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is find the best possible initial *directly, without looking at any recursive subproblems*, and then let the Recursion Fairy make all the other decisions. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are *very* rare. Nevertheless, for many problems that should be solved by dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the longest increasing subsequence problem might look for the smallest element of the input array, accept that element as the start of the target subsequence, and then recursively look for the longest increasing subsequence to the right of that element. If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution.

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

What, never?
No, never!
What, *never*?
Well. . . hardly ever.[7]

Because the greedy approach is so incredibly tempting, but so rarely correct, I strongly advocate the following policy in any algorithms course, even (or perhaps *especially*) for courses that do not normally ask for proofs of correctness.

**You will not receive *any* credit for *any* greedy algorithm for *any* problem,
on *any* homework or exam, even if the algorithm is correct,
without a *formal* proof of correctness.**

Moreover, the vast majority of problems for which students are tempted to submit a greedy algorithm are actually best solved using dynamic programming. So I always offer the following advice to my algorithms students.

**Whenever you write—or even *think*—the word "greedy",
your subconscious mind is telling you to use dynamic programming.
Listen to it.**

[7]Greedy methods hardly ever work! So give three cheers, and one cheer more, for the careful Captain of the *Pinafore*! Then give three cheers, and one cheer more, for the Captain of the *Pinafore*!

We will see techniques for proving greedy algorithms correct in a later chapter.

3.5 Edit Distance

The ***edit distance*** between two strings is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one string into another. For example, the edit distance between FOOD and MONEY is at most four:

FOOD → MOOD → MON D → MONED → MONEY

This distance function was independently proposed by Vladimir Levenshtein in 1964 (working on coding theory), Taras Vintsyuk in 1968 (working on speech recognition), and Stanislaw Ulam in 1972 (working with biological sequences). For this reason, edit distance is sometimes called ***Levenshtein distance*** or ***Ulam distance*** (but strangely, never Vintsyuk distance).

We can visualize this editing process by placing the strings one above the other, with a gap in the first word for each insertion and a gap in the second word for each deletion. Columns with two *different* characters correspond to substitutions. In this representation, the number of editing steps is just the number of columns that do not contain the same character twice.

F	O	O		D
M	O	N	E	Y

It's fairly obvious that we can't transform FOOD into MONEY in three steps, so the edit distance between FOOD and MONEY is exactly four. Unfortunately, it's not so easy in general to tell when a sequence of edits is as short as possible. For example, the following table shows that the distance between the strings ALGORITHM and ALTRUISTIC is at most 6. Is that optimal, or can we do better?

A	L	G	O	R		I		T	H	M
A	L		T	R	U	I	S	T	I	C

Recurrence

To develop a dynamic programming algorithm to compute edit distance, we first need to develop a recurrence. Our gap representation for edit sequences has a crucial "optimal substructure" property. Suppose we have the gap representation for the shortest edit sequence for two strings. **If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining prefixes.** We can easily prove this observation by contradiction: If the prefixes had a shorter edit sequence, gluing the last column back on would gives us a shorter edit sequence for the original strings.

So once we figure out what should happen in the last column, the Recursion Fairy can figure out the rest of the optimal gap representation.

So let's recursively define the edit distance between two strings $A[1..m]$ and $B[1..n]$, which we denote by $Edit(A[1..m], B[1..n])$. If neither string is empty, there are three possibilities for the last column in the shortest edit sequence:

- **Insertion:** The last entry in the bottom row is empty. In this case, the edit distance is equal to $Edit(A[1..m-1], B[1..n]) + 1$. The $+1$ is the cost of the final insertion, and the recursive expression gives the minimum cost for the other columns.

```
 ALGORITH  | M
ALTRUISTIC |
```

- **Deletion:** The last entry in the top row is empty. In this case, the edit distance is equal to $Edit(A[1..m], B[1..n-1]) + 1$. The $+1$ is the cost of the final deletion, and the recursive expression gives the minimum cost for the other columns.

```
ALGORITHM |
ALTRUISTI | C
```

- **Substitution:** Both rows have characters in the last column. If the characters are equal, the substitution is free, so the edit distance is equal to $Edit(A[1..m-1], B[1..n-1])$. If the characters are different, then the edit distance is equal to $Edit(A[1..m-1], B[1..n-1]) + 1$.

```
 ALGORITH | M
ALTRUISTI | C
```

The edit distance between A and B is the smallest of these three possibilities.

$$Edit(A[1..m],\ B[1..n]) = \min \left\{ \begin{array}{c} Edit(A[1..m-1],\ B[1..n]) + 1 \\ Edit(A[1..m],\ B[1..n-1]) + 1 \\ Edit(A[1..m-1],\ B[1..n-1]) + \big[A[m] \neq B[n]\big] \end{array} \right\}$$

(Here I'm using the enormously useful *Iverson bracket* notation: For any proposition P, we define $[P] = 1$ if P is true and $[P] = 0$ if P is false.)

This recurrence has two easy base cases: Converting the empty string into a string of length n requires n insertions, and converting a string of length m into the empty string requires m deletions.

$$Edit(A[1..m], \varepsilon) = m \qquad \text{and} \qquad Edit(\varepsilon, B[1..n]) = n.$$

Both of these expressions imply the trivial base case $Edit(\varepsilon, \varepsilon) = 0$.

Index Formulation

Because the arguments to our recursive subproblems are always prefixes of the original strings A and B, we can use the lengths of the prefixes instead of the prefixes themselves as the arguments to our recursive function. So let $Edit(i, j)$ denote the edit distance between the prefixes $A[1\,..\,i]$ and $B[1\,..\,j]$. This function satisfies the following recurrence:

$$
\boxed{Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{l} Edit(i-1,j)+1, \\ Edit(i,j-1)+1, \\ Edit(i-1,j-1)+\big[A[i] \neq B[j]\big] \end{array} \right\} & \text{otherwise} \end{cases}}
$$

The edit distance between the original strings A and B is just $Edit(m, n)$.

As usual, this recurrence translates directly into a recursive algorithm. The precise running time is not obvious, but it's clearly exponential in m and n. ***Fortunately, we don't care about the precise running time of the recursive algorithm.*** The recursive running time wouldn't tell us anything about our eventual dynamic programming algorithm, so we won't bother figuring it out.[8]

Dynamic Programming

Now that we have a recurrence, we can transform it into a dynamic programming algorithm following the usual mechanical boilerplate.

- **Subproblems:** Each recursive subproblem is identified by two indices $0 \le i \le m$ and $0 \le j \le n$.
- **Memoization structure:** So we can memoize all possible values of $Edit(i, j)$ in a two-dimensional array $Edit[0\,..\,m, 0\,..\,n]$.
- **Dependencies:** Each entry $Edit[i, j]$ depends only on its three neighboring entries $Edit[i-1, j]$, $Edit[i, j-1]$, and $Edit[i-1, j-1]$.
- **Evaluation order:** So if we fill in our table in the standard row-major order—row by row from top down, each row from left to right—then whenever we reach an entry in the table, the entries it depends on are already available.
- **Space and time:** The memoization structure uses ***O(mn) space***, and we can compute each entry $Edit[i, j]$ in $O(1)$ time one we know its predecessors, so the overall algorithm runs in ***O(mn) time.***

The resulting dynamic programming algorithm is shown in Figure **??**.

This algorithm is most commonly attributed to Robert Wagner and Michael Fischer, who described the algorithm in 1974. However, in full compliance with Stigler's Law of Eponymy, either identical or more general algorithms were independently discovered by Taras Vintsyuk in 1968, V. M. Velichko and N. G. Zagoruyko in 1970, David Sankoff in

[8]At least, *you* shouldn't bother. The best upper bound I can prove is $O((1+\sqrt{2})^{n+m}) = O(2.41422)^n$.

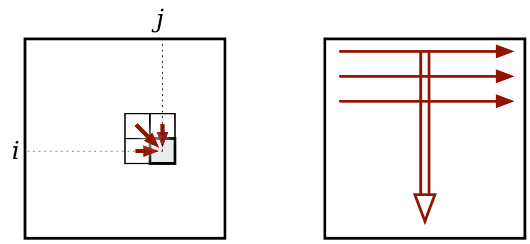


Figure 3.3. Dependencies in the memoization table for edit distance, and a legal evaluation order

```
EditDistance(A[1..m], B[1..n]):
  for j ← 0 to n
      Edit[0, j] ← j

  for i ← 1 to m
      Edit[i, 0] ← i
      for j ← 1 to n
          if A[i] = B[j]
              Edit[i, j] ← min {Edit[i − 1, j] + 1, Edit[i, j − 1] + 1, Edit[i − 1, j − 1]}
          else
              Edit[i, j] ← min {Edit[i − 1, j] + 1, Edit[i, j − 1] + 1, Edit[i − 1, j − 1] + 1}
  return Edit[m, n]
```

Figure 3.4. The (Vintsyuk–Velichko-Zagoruyko–Sankoff–Sellers–)Wagner-Fischer edit-distance algorithm.

1972, Peter Sellers[9] in 1974, and almost certainly several others.[10] Interestingly, *none* of these papers cite either Levenshtein or Ulam.

The resulting array for ALGORITHM → ALTRUISTIC is shown in Figure **??**. Bold numbers indicate places where characters in the two strings are equal. The edit distance between ALGORITHM and ALTRUISTIC is indeed six!

The arrows in Figure **??** indicate which predecessor(s) actually define each entry. Each direction of arrow corresponds to a different edit operation: horizontal=deletion, vertical=insertion, and diagonal=substitution. Bold red diagonal arrows indicate "free" substitutions of a letter for itself. Any path of arrows from the top left corner to the bottom right corner of this table represents an optimal edit sequence between the two strings. (There can be many such paths.) The edit distance algorithm does not actually

[9]"Gentlemen! You can't fight in here! This is the War Room!" Okay, not *that* Peter Sellers.

[10]The Vintsyuk–Velichko-Zagoruyko–Sankoff–Sellers–Wagner-Fischer edit-distance algorithm is occasionally also attributed to Saul Needleman and Christian Wunsch in 1970, but this attribution is incorrect. "The Needleman-Wunsch algorithm" more commonly refers to the standard dynamic programming algorithm for computing the longest common subsequence of two strings (or equivalently, the edit distance where only insertions and deletions are permitted) in $O(mn)$ time, but that attribution is *also* incorrect! In fact, Needleman and Wunsch's algorithm computes (weighted) longest common subsequences (possibly with gap costs) in $O(m^2n^2)$ time, using a different recurrence. Sankoff explicitly describes his $O(mn)$-time algorithm as an improvement of Needleman and Wunsch's algorithm.

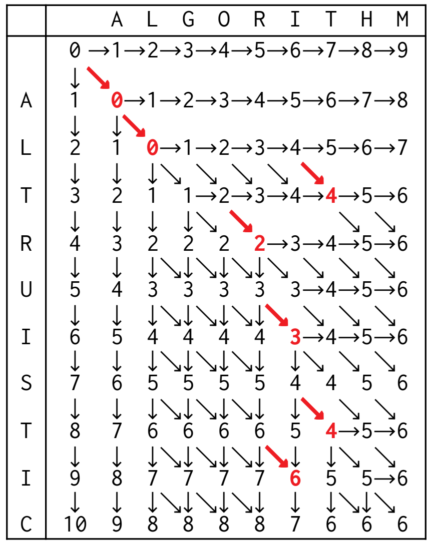


Figure 3.5. The memoization array for *Edit*(ALGORITHM, ALTRUISTIC)

compute or store these arrows, but the arrow(s) leading into any entry in the table can be reconstructed on the fly in $O(1)$ time. Thus, once we've filled in the table, we can reconstruct the actual optimal edit sequence in $O(n+m)$ additional time.

The table in Figure **??** shows exactly three paths from the top left to the bottom right, each indicating a different sequence of six edits transforming ALGORITHM into ALTRUISTIC.

```
A L G O R I   T H M
A L T R U I S T I C
```

```
A L G O R   I   T H M
A L   T R U I S T I C
```

```
A L G O R   I   T H M
A L T   R U I S T I C
```

3.6 Subset Sum

Recall that the *Subset Sum* problem asks, given an array $X[1..n]$ of positive integers and an integer T, whether any subset of X sums to T. In the previous chapter, we developed a recursive algorithm which can be reformulated as follows. Fix the original input array $X[1..n]$ and the original target sum T, and define the boolean function

$$SS(i,t) = \text{some subset of } X[i..n] \text{ sums to } t.$$

This function satisfies the following recurrence:

$$SS(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0 \\ \textsc{False} & \text{if } t < 0 \text{ or } i > n \\ SS(i+1,t) \ \vee \ SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

We can transform this recurrence into a dynamic programming algorithm as follows.

- **Subproblems:** Each subproblem is described by an integer i such that $1 \le i \le n+1$, and an integer $t \le T$. However, subproblems with $t < 0$ are trivial, so it seems rather silly to memoize them.[11] Indeed, we can modify the recurrence so that those subproblems never arise:

$$SS(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0 \\ \textsc{False} & \text{if } i > n \\ SS(i+1,t) & \text{if } t > X[i] \\ SS(i+1,t) \ \vee \ SS(i+1,t-X[i]) & \text{otherwise} \end{cases}$$

- **Data structure:** We can memoize all results into a two-dimensional array $S[1\,..\,n+1, 0\,..\,T]$, where $S[i,t]$ stores the value of $SS(i,t)$.

- **Evaluation order:** Each entry $S[i,t]$ depends on at most two other entries, both of the form $SS[i+1,\cdot]$. So we can fill the array by considering rows from bottom to top in the outer loop, and considering the elements in each row in arbitrary order in the inner loop.

- **Space and time:** The memoization structure uses ***$O(nT)$ space***. If $S[i+1,t)$ and $S[i+1,t-X[i]]$ are already known, we can compute $S[i,t]$ in constant time, so the algorithm runs in ***$O(nT)$ time***.

Here is the resulting dynamic programming algorithm:

```
SUBSETSUM(X[1..n], T):
  S[n+1, 0] ← TRUE
  for t ← 1 to T
      S[n+1, t] ← FALSE
  for i ← n downto 1
      S[i, 0] = TRUE
      for t ← 1 to X[i] − 1
          S[i, t] ← S[i+1, t]        ⟨⟨Avoid the case t < 0⟩⟩
      for t ← X[i] to T
          S[i, t] ← S[i+1, t] ∨ S[i+1, t − X[i]]
  return S[1, T]
```

[11]Yes, I'm breaking my own rule against premature optimization.

The running time $O(nT)$ for this algorithm is a significant improvement over the $O(2^n)$-time recursive backtracking algorithm when T is small.[12] However, if the target sum T is significantly larger than 2^n, this dynamic programming algorithm is actually slower than the naïve recursive algorithm, because it's wasting time solving subproblems that the recursive algorithm never considers. Dynamic programming isn't *always* an improvement!

3.7 Optimal Binary Search Trees

In the previous chapter, we developed a recursive algorithm for the optimal binary search tree problem. We are given a sorted array $A[1 .. n]$ of search keys and an array $f[1 .. n]$ of frequency counts, where $f[i]$ is the number of times we will search for $A[i]$. Our task is to construct a binary search tree for that set such that the total cost of all the searches is as small as possible.

Fix the frequency array f, and let $OptCost(i,k)$ denote the total search time in the optimal search tree for the subarray $A[i .. k]$. We derived the following recurrence for the function *OptCost*:

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ \displaystyle\sum_{j=i}^{k} f[i] + \min_{i \le r \le k} \big\{ OptCost(i, r-1) \ + \ OptCost(r+1, k) \big\} & \text{otherwise} \end{cases}$$

Let's follow our standard outline for transforming this recurrence into a dynamic programming algorithm.

Our algorithm will be somewhat simpler (and in practice, faster) if we can get that summation out of the recurrence. For any pair of indices $i \le k$, let $F(i,k)$ denote the total frequency count for all the keys in the interval $A[i .. k]$:

$$F(i,k) := \sum_{j=i}^{k} f[j]$$

This function satisfies the following simple recurrence:

$$F(i,k) = \begin{cases} f[i] & \text{if } i = k \\ F(i,k-1) + f[k] & \text{otherwise} \end{cases}$$

We can compute all possible values of $F(i,k)$ in $O(n^2)$ time using—you guessed it!—dynamic programming.

[12]Even though *SubsetSum* is NP-complete, this time bound does *not* imply that P=NP, because T is not necessary bounded by a polynomial function of the input size.

```
InitF(f[1..n]):
  for i ← 1 to n
      F[i, i−1] ← 0
      for k ← i to n
          F[i, k] ← F[i, k−1] + f[k]
```

Our final algorithm will use InitF as an initialization subroutine.

We can now simplify the original recurrence as follows:

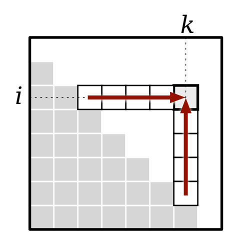
$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F(i,k) + \min\limits_{i \le r \le k} \left\{ OptCost(i,r-1) \ + \ OptCost(r+1,k) \right\} & \text{otherwise} \end{cases}$$

We can derive a dynamic programming algorithm for this recurrence using our standard strategy.

Subproblems: Each recursive subproblem is specified by two integers: $1 \le i \le n+1$ and $0 \le k \le n$.

Memoization: We can store all possible values of *OptCost* in a two-dimensional array $OptCost[1..n+1, 0..n]$. (Only the entries $OptCost[i, j]$ with $j \ge i-1$ will actually be used, but whatever.)

Dependencies: Each entry $OptCost[i,k]$ depends on the entries $OptCost[i, j-1]$ and $OptCost[j+1,k]$ for all j such that $i \le j \le k$. In other words, every entry in the table depends on all the entries either directly to the left or directly below.



Subproblem dependencies in the optimal binary search tree problem.

The following subroutine fills the entry $OptCost[i,k]$, assuming all the entries it depends on have already been computed.

```
ComputeOptCost(i, k):
  OptCost[i, k] ← ∞
  for r ← i to k
      tmp ← OptCost[i, r−1] + OptCost[r+1, k]
      if OptCost[i, k] > tmp
          OptCost[i, k] ← tmp
  OptCost[i, k] ← OptCost[i, k] + F[i, k]
```

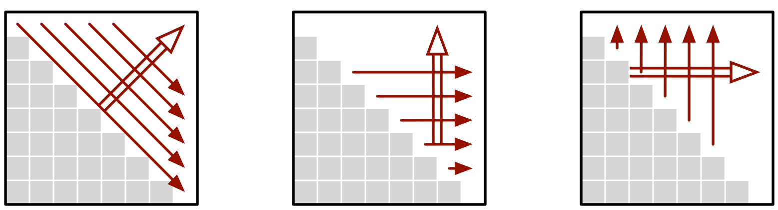
Evaluation order: There are at least three different orders that can be used to fill the array. The first one that occurs to most students is to scan through the table one diagonal at a time, starting with the trivial base cases $OptCost[i, i-1]$ and working toward the final answer $OptCost[1, n]$, like so:

```
OPTIMALBST(f[1..n]):
  INITF(f[1..n])
  for i ← 1 to n+1
      OptCost[i, i−1] ← 0
  for d ← 0 to n−1
      for i ← 1 to n−d
          COMPUTEOPTCOST(i, i+d)
  return OptCost[1, n]
```

We could also traverse the array row by row from the bottom up, traversing each row from left to right, or column by column from left to right, traversing each columns from the bottom up.

```
OPTIMALBST2(f[1..n]):
  INITF(f[1..n])
  for i ← n+1 downto 1
      OptCost[i, i−1] ← 0
      for j ← i to n
          COMPUTEOPTCOST(i, j)
  return OptCost[1, n]
```

```
OPTIMALBST3(f[1..n]):
  INITF(f[1..n])
  for j ← 0 to n+1
      OptCost[j+1, j] ← 0
      for i ← j downto 1
          COMPUTEOPTCOST(i, j)
  return OptCost[1, n]
```



Three different evaluation orders for the optimal binary search tree problem.

Space and time: No matter which of these orders we actually use, the resulting algorithm runs in ***$O(n^3)$ time*** and uses ***$O(n^2)$ space***.

In fact, we could have predicted these space and time bounds directly from the original recurrence.

$$OptCost(i,k) = \begin{cases} 0 & \text{if } i > k \\ F(i,k) + \min\limits_{i \le r \le k} \left\{ OptCost(i, r-1) \; + \; OptCost(r+1, k) \right\} & \text{otherwise} \end{cases}$$

The function has two arguments, each of which can take on roughly n different values, so we probably need a data structure of size $O(n^2)$. There are *three* variables in the body of the recurrence (i, k, and r), each of which can take roughly n different values, so it should take $O(n^3)$ time to compute everything.

3.8 Dynamic Programming on Trees

So far, all of our dynamic programming example use a multidimensional array to store the results of recursive subproblems. However, as the next example shows, this is not always the most appropriate date structure to use.

An ***independent set*** in a graph is a subset of the vertices with no edges between them. Finding the largest independent set in an arbitrary graph is extremely hard; in fact, this is one of the canonical NP-hard problems we will study in a later chapter. But for some special cases of graphs, we can find the largest independent set efficiently. In particular, when the input graph is a *tree* with n vertices, we can compute the largest independent set in $O(n)$ time.

So suppose we are given a tree T. Without loss of generality, suppose T is a rooted tree; that is, there is a special node in T called the *root*, and all edges are implicitly directed away from this vertex. (If T is an unrooted tree—a connected acyclic undirected graph—we can choose an arbitrary vertex as the root.) We call vertex w a *descendant* of vertex v if the unique path from w to the root includes v; equivalently, the descendants of v are v itself and the descendants of the children of v. The *subtree rooted at* v consists of all the descendants of v and the edges between them.

For any node v in T, let $MIS(v)$ denote the size of the largest independent set in the subtree rooted at v. Any independent set in this subtree that excludes v itself is the union of independent sets in the subtrees rooted at the children of v. On the other hand, any independent set that *includes* v necessarily excludes all of v's children, and therefore includes independent sets in the subtrees rooted at v's grandchildren. Thus, the function MIS obeys the following recurrence, where the notation $w \downarrow v$ means "w is a child of v":

$$MIS(v) = \max\left\{\sum_{w\downarrow v} MIS(v),\ 1 + \sum_{w\downarrow v}\sum_{x\downarrow w} MIS(x)\right\}$$

We need to compute $MIS(r)$, where r is the root of T.

What data structure should we use to memoize this recurrence? The most natural choice is ***the tree T itself!*** Specifically, for each vertex v in T, we store the result of $MIS(v)$ in a new field $v.MIS$. (In principle, we *could* use an array instead, but then we'd have to pointers back and forth between each node and its corresponding array entry, so why bother?)

What's a good order to consider the subproblems? The subproblem associated with any node v depends on the subproblems associated with the children and grandchildren of v. So we can visit the nodes in any order, provided that every vertex is visited before its parent; in particular, we can use a standard ***post-order*** traversal.

What's the running time of the algorithm? The non-recursive time associated with each node v is proportional to the number of children and grandchildren of v; this number can be very different from one vertex to the next. But we can turn the analysis around: Each vertex contributes a constant amount of time to its parent and its grandparent! Because each vertex has at most one parent and at most one grandparent, the algorithm runs in ***O*(*n*) time**.

Here is the resulting dynamic programming algorithm. Yes, it's still recursive, because that's the most natural way to implement a post-order tree traversal.

```
MIS(v):
  withoutv ← 0
  for each child w of v
        withoutv ← withoutv + MIS(w)
  withv ← 1
  for each grandchild x of v
        withv ← withv + x.MIS
  v.MIS ← max{withv, withoutv}
  return v.MIS
```

We can derive an even simpler linear-time algorithm by defining two separate functions over the nodes of T:

- Let $\mathit{MISyes}(v)$ denote the size of the largest independent set of the subtree rooted at v that *includes* v.
- Let $\mathit{MISno}(v)$ denote the size of the largest independent set of the subtree rooted at v that *excludes* v.

Again, we need to compute $\max\{\mathit{MISyes}(r), \mathit{MISno}(r)\}$, where r is the root of T. The first two functions satisfy the following mutual recurrence:

$$\mathit{MISyes}(v) = 1 + \sum_{w\downarrow v} \mathit{MISno}(w)$$
$$\mathit{MISno}(v) = \sum_{w\downarrow v} \max\{\mathit{MISyes}(w), \mathit{MISno}(w)\}$$

Again, we can memoize these functions into the tree itself, by defining two new fields for each vertex. A straightforward post-order traversal evaluates both functions at every node in ***O*(*n*) time**. The following function not only memoizes the separate function values at v, it also returns the larger of the two functions.

```
MIS(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
        v.MISno ← v.MISno + MIS(w)
        v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}
```

Exercises

Sequences/Arrays

1. In a previous life, you worked as a cashier in the lost Antarctican colony of Nadira, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource in Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadira, called Dream Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365.[13]

   Homework

   (a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream Dollar bills than the minimum possible. *[Hint: It may be easier to write a small program than to work this out by hand.]*

   Exam

   (b) Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream Dollars. (Don't worry about making your algorithm fast; just make sure it's correct.)

   Exam

   (c) Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream Dollars. (This one needs to be fast.)

2. Suppose you are given an array $A[1..n]$ of numbers, which may be positive, negative, or zero, and which are ***not*** necessarily integers.

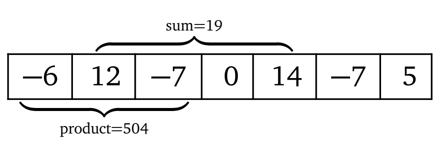
   Exam, Wikipedia

   (a) Describe and analyze an algorithm that finds the largest sum of of elements in a contiguous subarray $A[i..j]$.

   Exam

   (b) Describe and analyze an algorithm that finds the largest *product* of of elements in a contiguous subarray $A[i..j]$.

   For example, given the array $[-6, 12, -7, 0, 14, -7, 5]$ as input, your first algorithm should return the integer 19, and your second algorithm should return the integer 504.



   For the sake of analysis, assume that comparing, adding, or multiplying any pair of numbers takes $O(1)$ time.

[13]For more details on the history and culture of Nadira, including images of the various denominations of Dream Dollars, see http://moneyart.biz/dd/.

[Hint: Problem (a) has been a standard computer science interview question since at least the mid-1980s. You can find many correct solutions on the web; the problem even has its own Wikipedia page! But at least in 2016, a significant fraction of the solutions I found on the web for problem (b) were either slower than necessary or actually incorrect. Remember that the product of two negative numbers is positive.]

3. This series of exercises asks you to develop efficient algorithms to find optimal *subsequences* of various kinds. A subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings `C`, `DAMN`, `YAIOAI`, and `DYNAMICPROGRAMMING` are all subsequences of the string `DYNAMICPROGRAMMING`.

   (a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A ***common subsequence*** of A and B is another sequence that is a subsequence of both A and B. Describe an efficient algorithm to compute the length of the *longest* common subsequence of A and B. Exam

   (b) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A ***common supersequence*** of A and B is another sequence that contains both A and B as subsequences. Describe an efficient algorithm to compute the length of the *shortest* common supersequence of A and B. Exam

   (c) Call a sequence $X[1..n]$ of numbers ***bitonic*** if there is an index i with $1 < i < n$, such that the prefix $X[1..i]$ is increasing and the suffix $X[i..n]$ is decreasing. Describe an efficient algorithm to compute the length of the longest bitonic subsequence of an arbitrary array A of integers. Exam

   (d) Call a sequence $X[1..n]$ of numbers ***oscillating*** if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Describe an efficient algorithm to compute the length of the longest oscillating subsequence of an arbitrary array A of integers. Exam

   (e) Describe an efficient algorithm to compute the length of the shortest oscillating supersequence of an arbitrary array A of integers. Exam

   (f) Call a sequence $X[1..n]$ of numbers ***convex*** if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Describe an efficient algorithm to compute the length of the longest convex subsequence of an arbitrary array A of integers. Exam

   (g) Call a sequence $X[1..n]$ of numbers ***weakly increasing*** if each element is larger than the average of the two previous elements; that is, $2 \cdot X[i] > X[i-1] + X[i-2]$ for all $i > 2$. Describe an efficient algorithm to compute the length of the longest weakly increasing subsequence of an arbitrary array A of integers. Exam

   (h) Call a sequence $X[1..n]$ of numbers ***double-increasing*** if $X[i] > X[i-2]$ for all $i > 2$. (In other words, a double-increasing sequence is a perfect shuffle of two increasing sequences.) Describe an efficient algorithm to compute the length of the longest double-increasing subsequence of an arbitrary array A of integers. Exam

(i) Recall that a sequence $X[1..n]$ of numbers is *increasing* if $X[i] < X[i+1]$ for all i. Describe an efficient algorithm to compute the length of the *longest common increasing subsequence* of two given arrays of integers. For example, $\langle 1,4,5,6,7,9\rangle$ is the longest common increasing subsequence of the sequences $\langle 3,1,4,1,5,9,2,6,5,3,5,8,9,7,9,3\rangle$ and $\langle 1,4,1,4,2,1,3,5,6,2,3,7,3,0,9,5\rangle$. Homework

Exam

4. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

BANANA ANANAS   BAN ANA ANA NAS   B AN AN A NA NA S

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:

PRO D G Y R NAM AMMI I N C G   DY PRO N G A R M AMM IC ING

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

5. For each of the following problems, the input consists of two arrays $X[1..k]$ and $Y[1..n]$ where $k \le n$.

Exam

(a) Describe and analyze an algorithm to determine whether X is a subsequence of Y. For example, the string PPAP is a subsequence of the string PENPINEAPPLEAPPLEPEN.

Exam

(b) Describe and analyze an algorithm to determine whether X occurs as two *disjoint* subsequences of Y. For example, the string PPAP appears as two disjoint subsequences in the string PENPINEAPPLEAPPLEPEN.

Homework

(c) Suppose the input also includes a third array $C[1..n]$ of numbers, which may be positive, negative, or zero, where $C[i]$ is the *cost* of $Y[i]$. Describe and analyze an algorithm to compute the minimum-cost occurrence of X as a subsequence of Y. That is, we want to find an array $I[1..k]$ such that $I[j] < I[j+1]$ and $X[I[j]] = Y[j]$ for every index j, and the total cost $\sum_{j=1}^{k} C[j]$ is as small as possible.

Homework

(d) Describe and analyze an algorithm to compute the total number of (possibly overlapping) occurrences of X as a subsequence of Y. For purposes of analysis, assume that we can add two arbitrary integers in $O(1)$ time.

For example, the string PPAP appears exactly 23 times as a subsequence of the string PENPINEAPPLEAPPLEPEN. If all characters in X and Y are equal, your algorithm should return $\binom{n}{k}$.

Homework

(e) What is the running time of your algorithm for part (d) if adding two ℓ-bit integers requires $O(\ell)$ time?

6. A palindrome is any string that is exactly the same as its reversal, like I, or DEED, or RACECAR, or AMANAPLANACATACANALPANAMA.

   (a) Describe and analyze an algorithm to find the length of the *longest subsequence* of a given string that is also a palindrome. For example, the longest palindrome subsequence of MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM is MHYMRORMYHM, so given that string as input, your algorithm should output the number 11. Exam

   (b) Describe and analyze an algorithm to find the length of the *shortest supersequence* of a given string that is also a palindrome. For example, the shortest palindrome supersequence of TWENTYONE is TWENTOYOTNEWT, so given the string TWENTYONE as input, your algorithm should output the number 13. Exam

   (c) Any string can be decomposed into a sequence of palindromes. For example, the string BUBBASEESABANANA ("Bubba sees a banana.") can be broken into palindromes in the following ways (and many others): Exam

   BUB • BASEESAB • ANANA

   B • U • BB • A • SEES • ABA • NAN • A

   B • U • BB • A • SEES • A • B • ANANA

   B • U • B • B • A • S • E • E • S • A • B • A • N • ANA

   Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string BUBBASEESABANANA, your algorithm would return the integer 3.

7. Suppose you have a black-box subroutine Quality that can compute the "quality" of any given string $A[1..k]$ in $O(k)$ time. For example, the quality of a string might be 1 if the string is a Québecois curse word, and 0 otherwise. Exam

   Given an arbitrary input string $T[1..n]$, we would like to break it into contiguous substrings, such that the total quality of all the substrings is as large as possible. For example, the string SAINTCIBOIREDESACRAMENTDECRISSE can be decomposed into the substrings SAINT • CIBOIRE • DE • SACRAMENT • DE • CRISSE, of which three (or possibly four) are *sacres*.

   Describe an algorithm that breaks a string into substrings of maximum total quality, using the Quality subroutine.

8. Describe and analyze an efficient algorithm to find the length of the longest contiguous substring that appears both forward and backward in an input string $T[1..n]$. The forward and backward substrings must not overlap. Here are several examples: Exam

   - Given the input string ALGORITHM, your algorithm should return 0.
   - Given the input string RECURSION, your algorithm should return 1, for the substring R.

- Given the input string REDIVIDE, your algorithm should return 3, for the substring EDI. (The forward and backward substrings must not overlap!)
- Given the input string DYNAMICPROGRAMMINGMANYTIMES, your algorithm should return 4, for the substring YNAM. (In particular, it should *not* return 6, for the subsequence YNAMIR).

Homework, google

9. Suppose we want to display a paragraph of text on a computer screen. The text consists of a sequence of n words, where the ith word has length $\ell[i]$. We want to break the paragraph into several lines of total length exactly L. For example, according to TeX, the program used to typeset these notes, *the paragraph you are reading right now* is approximately 13.40944 cm ≈ 5.28108 inches wide.

   Depending on how the paragraph is broken into lines of text, we must insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first character on each line starts at the left margin, and *except for the last line*, the last character on each line ends at the right margin. There must be at least one unit of white space between any two words on the same line.

   Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the amount of extra white-space in each line, not counting the one unit of required space between each adjacent pair of words. Specifically, if a line contains words i through j, then the slop of that line is defined to be $\left(L - j + i - \sum_{k=i}^{j} \ell[k]\right)^3$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

Exam: (a)(b)(c) or (a)(b)(d)

10. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

    Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

    (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.

    (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

Assumes familiarity with probability

    (c) When Elmo was four, he used an even simple strategy—on his turn, he always chose his next card uniformly at random. That is, if there was more than one card left on his turn, he would take the leftmost card with probability $1/2$, and the rightmost card with probability $1/2$. Describe an algorithm to determine, given the initial sequence of cards, the maximum *expected* number of points you can collect playing against four-year-old-Elmo.

(d) Five years later, thirteen-year-old Elmo has become a *much* stronger player. Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a *perfect* opponent.

11. It's almost time to show off your flippin' sweet dancing skills! Tomorrow is the big dance contest you've been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You've obtained an advance copy of the the list of n songs that the judges will play during the contest, in chronological order. Exam

You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly $\mathit{Score}[k]$ points, but then you will be physically unable to dance for the next $\mathit{Wait}[k]$ songs (that is, you cannot dance to songs $k+1$ through $k+\mathit{Wait}[k]$). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $\mathit{Score}[1..n]$ and $\mathit{Wait}[1..n]$.

12. Lenny Rutenbar, the founding dean of the new Maximilian Q. Levchin College of Computer Science, has commissioned a series of snow ramps on the south slope of the Orchard Downs sledding hill[14] and challenged Bill Kudeki, head of the Department of Electrical and Computer Engineering, to a sledding contest. Bill and Lenny will both sled down the hill, each trying to maximize their air time. The winner gets to expand their department/college into both Siebel Center and the new ECE Building; the loser has to move their entire department/college under the Boneyard bridge next to Everitt Lab.

Whenever Lenny or Bill reaches a ramp *while on the ground*, they can either use that ramp to jump through the air, possibly flying over one or more ramps, or sled past that ramp and stay on the ground. Obviously, if someone flies over a ramp, they cannot use that ramp to extend their jump.

(a) Suppose you are given a pair of arrays $\mathit{Ramp}[1..n]$ and $\mathit{Length}[1..n]$, where $\mathit{Ramp}[i]$ is the distance from the top of the hill to the ith ramp, and $\mathit{Length}[i]$ is Exam

[14] The north slope is faster, but too short for an interesting contest.

the distance that any sledder who takes the ith ramp will travel through the air. Describe and analyze an algorithm to determine the maximum total distance that Lenny or Bill can spend in the air.

Homework

(b) Uh-oh. The university lawyers heard about Lenny and Bill's little bet and immediately objected. To protect the university from either lawsuits or sky-rocketing insurance rates, they impose an upper bound on the number of jumps that either sledder can take. Describe and analyze an algorithm to determine the maximum total distance that Lenny or Bill can spend in the air *with at most k jumps*, given the original arrays $Ramp[1..n]$ and $Length[1..n]$ and the integer k as input.

Homework

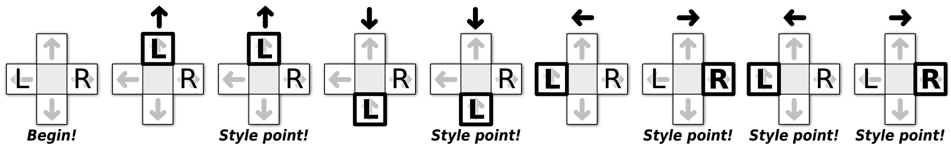
(c) When the lawyers realized that imposing their restriction didn't immediately shut down the contest, they added a new restriction: No ramp can be used more than once! Disgusted by the legal interference, Lenny and Bill give up on their bet and decide to cooperate to put on a good show for the spectators. Describe and analyze an algorithm to determine the maximum total distance that Lenny and Bill can spend in the air, each taking at most k jumps (so at most $2k$ jumps total), and with each ramp used at most once.

Exam

13. ***Dance Dance Revolution*** is a dance video game, first introduced in Japan by Konami in 1998. Players stand on a platform marked with four arrows, pointing forward, back, left, and right, arranged in a cross pattern. During play, the game plays a song and scrolls a sequence of n arrows (🡄, 🡅, 🡇, or 🡆) from the bottom to the top of the screen. At the precise moment each arrow reaches the top of the screen, the player must step on the corresponding arrow on the dance platform. (The arrows are timed so that you'll step with the beat of the song.)

You are playing a variant of this game called "Vogue Vogue Revolution", where the goal is to play perfectly but move as little as possible. When an arrow reaches the top of the screen, if one of your feet is already on the correct arrow, you are awarded one style point for maintaining your current pose. If neither foot is on the right arrow, you must move one (and *only* one) of your feet from its current location to the correct arrow on the platform. If you ever step on the wrong arrow, or fail to step on the correct arrow, or move more than one foot at a time, or move either foot when you are already standing on the correct arrow, all your style points are taken away and you lose the game.

How should you move your feet to maximize your total number of style points? For purposes of this problem, assume you always start with your left foot on 🡄 and your right foot on 🡆, and that you've memorized the entire sequence of arrows. For example, if the sequence is 🡅🡅🡇🡇🡄🡆🡄🡆, you can earn 5 style points by moving your feet as shown below:



(a) ***Prove*** that for *any* sequence of n arrows, it is possible to earn at least $n/4 - 1$ style points.

(b) Describe an efficient algorithm to find the maximum number of style points you can earn during a given VVR routine. The input to your algorithm is an array $\mathit{Arrow}[1..n]$ containing the sequence of arrows.

14. Consider the following solitaire form of Scrabble. We begin with a fixed, finite sequence of tiles; each tile contains a letter and a numerical value. At the start of the game, we draw the seven tiles from the sequence and put them into our hand. In each turn, we form an English word from some or all of the tiles in our hand, place those tiles on the table, and receive the total value of those tiles as points. If no English word can be formed from the tiles in our hand, the game immediately ends. Then we repeatedly draw the next tile from the start of the sequence until either (a) we have seven tiles in our hand, or (b) the sequence is empty. (Sorry, no double/triple word/letter scores, bingos, blanks, or passing.) Our goal is to obtain as many points as possible.

Homework: (a)(b)
Exam: (a)

For example, suppose we are given the following sequence of 20 tiles:

$\boxed{I_2}\,\boxed{N_2}\,\boxed{X_8}\,\boxed{A_1}\,\boxed{N_2}\,\boxed{A_1}\,\boxed{D_3}\,\boxed{U_5}\,\boxed{D_3}\,\boxed{I_2}\,\boxed{D_3}\,\boxed{K_8}\,\boxed{U_5}\,\boxed{B_4}\,\boxed{L_2}\,\boxed{A_1}\,\boxed{K_8}\,\boxed{H_5}\,\boxed{A_1}\,\boxed{N_2}$

Then we can earn 68 points as follows:

- We initially draw $\boxed{I_2}\,\boxed{N_2}\,\boxed{X_8}\,\boxed{A_1}\,\boxed{N_2}\,\boxed{A_1}\,\boxed{D_3}$.
- Play the word $\boxed{N_2}\,\boxed{A_1}\,\boxed{I_2}\,\boxed{A_1}\,\boxed{D_3}$ for 9 points, leaving $\boxed{N_2}\,\boxed{X_8}$ in our hand.
- Draw the next five tiles $\boxed{U_5}\,\boxed{D_3}\,\boxed{I_2}\,\boxed{D_3}\,\boxed{K_8}$.
- Play the word $\boxed{U_5}\,\boxed{N_2}\,\boxed{D_3}\,\boxed{I_2}\,\boxed{D_3}$ for 15 points, leaving $\boxed{K_8}\,\boxed{X_8}$ in our hand.
- Draw the next five tiles $\boxed{U_5}\,\boxed{B_4}\,\boxed{L_2}\,\boxed{A_1}\,\boxed{K_8}$.
- Play the word $\boxed{B_4}\,\boxed{U_5}\,\boxed{L_2}\,\boxed{K_8}$ for 19 points, leaving $\boxed{K_8}\,\boxed{X_8}\,\boxed{A_1}$ in our hand.
- Draw the next three tiles $\boxed{H_5}\,\boxed{A_1}\,\boxed{N_2}$, emptying the list.
- Play the word $\boxed{A_1}\,\boxed{N_2}\,\boxed{K_8}\,\boxed{H_5}$ for 16 points, leaving $\boxed{X_8}\,\boxed{A_1}$ in our hand.
- Play the word $\boxed{A_1}\,\boxed{X_8}$ for 9 points, emptying our hand and ending the game.

(a) Suppose you are given as input two arrays $\mathit{Letter}[1..n]$, containing a sequence of letters between A and Z, and $\mathit{Value}[\mathsf{A}..\mathsf{Z}]$, where $\mathit{Value}[\ell]$ is the value of letter ℓ. Design and analyze an efficient algorithm to compute the maximum number of points that can be earned from the given sequence of tiles.

(b) Now suppose two tiles with the same letter can have different values; you are given two arrays *Letter*[1 .. n] and *Value*[1 .. n]. Design and analyze an efficient algorithm to compute the maximum number of points that can be earned from the given sequence of tiles.

In both problems, the output is a single number: the maximum possible score. Assume that you can find all English words that can be made from any set of at most seven tiles, along with the point values of those words, in $O(1)$ time.

Homework: (a)(b) or (a)(c) or (b)(d) or (c)(d)

15. (a) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

    (b) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

    (c) Suppose we are given a set L of n line segments in the plane, where the endpoints of each segment lie on the unit circle $x^2 + y^2 = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

    (d) Suppose we are given a set L of n line segments in the plane, where the endpoints of each segment lie on the unit circle $x^2 + y^2 = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

Homework: (a)(b) or (a)(c) or (b)(d) or (c)(d)

16. Let P be a set of n points evenly distributed on the unit circle, and let S be a set of m line segments with endpoints in P. The endpoints of the m segments are *not* necessarily distinct; n could be significantly smaller than $2m$.

    (a) Describe an algorithm to find the size of the largest subset of segments in S such that every pair is disjoint. Two segments are disjoint if they do not intersect even at their endpoints.

    (b) Describe an algorithm to find the size of the largest subset of segments in S such that every pair is interior-disjoint. Two segments are interior-disjoint if their intersection is either empty or an endpoint of both segments.

    (c) Describe an algorithm to find the size of the largest subset of segments in S such that every pair intersects.

    (d) Describe an algorithm to find the size of the largest subset of segments in S such that every pair crosses. Two segments cross if they intersect but not at their endpoints.

    For full credit, all four algorithms should run in $O(mn)$ time.

17. You are driving a bus along a highway, full of rowdy, hyper, thirsty students and a soda fountain machine. Each minute that a student is on your bus, that student drinks one ounce of soda. Your goal is to drop the students off quickly, so that the total amount of soda consumed by all students is as small as possible. Homework

You know how many students will get off of the bus at each exit. Your bus begins somewhere along the highway (probably not at either end) and move s at a constant speed of 37.4 miles per hour. You must drive the bus along the highway; however, you may drive forward to one exit then backward to an exit in the opposite direction, switching as often as you like. (You can stop the bus, drop off students, and turn around instantaneously.)

Describe an efficient algorithm to drop the students off so that they drink as little soda as possible. Your input consists of the bus route (a list of the exits, together with the travel time between successive exits), the number of students you will drop off at each exit, and the current location of your bus (which you may assume is an exit).

18. Let's define a *summary* of two strings A and B to be a concatenation of substrings of the following form: Homework

- ▲SNA indicates a substring SNA of only the first string A.
- ♦FOO indicates a common substring FOO of both strings.
- ▼BAR indicates a substring BAR of only the second string B.

A summary is *valid* if we can recover the original strings A and B by concatenating the appropriate substrings of the summary in order and discarding the delimiters ▲, ♦, and ▼. Each regular character has length 1, and each delimiter ▲, ♦, or ▼ has some fixed non-negative length Δ. The *length* of a summary is the sum of the lengths of its symbols.

For example, each of the following strings is a valid summary of the strings KITTEN and KNITTING:

- ♦K▼N♦ITT▲E▼I♦N▼G has length $9 + 7\Delta$.
- ♦K▼N♦ITT▲EN▼ING has length $10 + 5\Delta$.
- ♦K▲ITTEN▼NITTING has length $13 + 3\Delta$.
- ▲KITTEN▼KNITTING has length $14 + 2\Delta$.

Describe and analyze an algorithm that computes the length of the shortest summary of two given strings $A[1..m]$ and $B[1..n]$. The delimiter length Δ is also part of the input to your algorithm. For example:

- Given strings KITTEN and KNITTING and $\Delta = 0$, your algorithm should return 9.
- Given strings KITTEN and KNITTING and $\Delta = 1$, your algorithm should return 15.
- Given strings KITTEN and KNITTING and $\Delta = 2$, your algorithm should return 18.

19. *Vankin's Mile* is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

    For example, given the grid below, the player can score $8-6+7-3+4=10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is *not* the best possible score for these values.)

−1	7	−8	10	−5
−4	−9	8	−6	0
5	−2	−6	−6	7
−7	4	7	−3	−3
7	1	−6	4	−9

Exam

(a) Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Mile, given the $n \times n$ array of values as input.

Homework

(b) In the European version of this game, appropriately called *Vankin's Kilometer*, the player can move the token either one square down, one square right, *or one square left* in each turn. However, to prevent infinite scores, the token cannot land on the same square more than once. Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Kilometer, given the $n \times n$ array of values as input.[15]

Homework
Exam: (a)

20. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..n, 1..n]$ of 0s and 1s. A *solid block* in M is a subarray of the form $M[i..i', j..j']$ in which all bits are equal. A solid block is square if it has the same number of rows and columns.

    (a) Describe an algorithm to find the maximum area of a solid *square* block in M in $O(n^2)$ time.

    (b) Describe an algorithm to find the maximum area of a solid block in M in $O(n^3)$ time.

    (c) Describe an algorithm to find the maximum area of a solid block in M in $O(n^2)$ time.

Homework

21. Let P be a set of points in the plane in *convex position*. Intuitively, if a rubber band were wrapped around the points, then every point would touch the rubber band. More

[15]If we also allowed upward movement, the resulting game (Vankin's Fathom?) would be NP-hard.

formally, for any point p in P, there is a line that separates p from the other points in P. Moreover, suppose the points are indexed $P[1], P[2], \ldots, P[n]$ in counterclockwise order around the "rubber band", starting with the leftmost point $P[1]$.

This problem asks you to solve a special case of the traveling salesman problem, where the salesman must visit every point in P, and the cost of moving from one point $p \in P$ to another point $q \in P$ is the Euclidean distance $|pq|$.

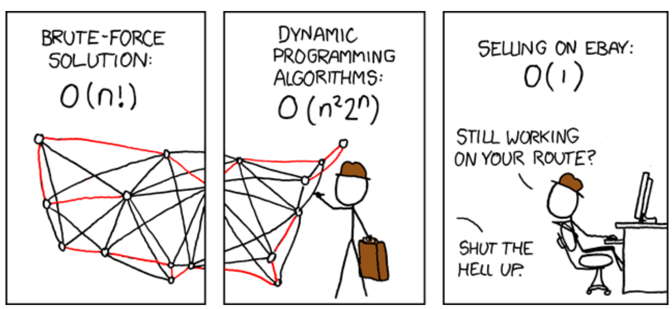
(a) Describe a simple algorithm to compute the shortest *cyclic* tour of P.

(b) A *simple* tour is one that never crosses itself. Prove that the shortest tour of P must be simple.

(c) Describe and analyze an efficient algorithm to compute the shortest tour of P that starts at the leftmost point $P[1]$ and ends at the rightmost point $P[r]$.

(d) Describe and analyze an efficient algorithm to compute the shortest tour of P, with no restrictions on the endpoints.

22. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \operatorname{poly}(n))$ time. Given an undirected n-vertex graph G with weighted edges, your algorithm should return the weight of the lightest cycle in G that visits every vertex exactly once, or ∞ if G has no such cycles. *[Hint: The obvious recursive algorithm takes $O(n!)$ time.]* Homework



— Randall Munroe, *xkcd* (http://xkcd.com/399/)

23. Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be a finite set of strings over some fixed alphabet Σ. An *edit center* for $\mathcal{A}$ is a string $C \in \Sigma^*$ such that the maximum edit distance from C to any string in $\mathcal{A}$ is as small as possible. The *edit radius* of $\mathcal{A}$ is the maximum edit distance from an edit center to a string in $\mathcal{A}$. A set of strings may have several edit centers, but its edit radius is unique. Homework

$$EditRadius(\mathcal{A}) = \min_{C \in \Sigma^*} \max_{A \in \mathcal{A}} Edit(A, C) \qquad EditCenter(\mathcal{A}) = \arg\min_{C \in \Sigma^*} \max_{A \in \mathcal{A}} Edit(A, C)$$

(a) Describe and analyze an efficient algorithm to compute the edit radius of three given strings.

(b) Describe and analyze an efficient algorithm to approximate the edit radius of an arbitrary set of strings within a factor of 2. (Computing the *exact* edit radius is NP-hard unless the number of strings is fixed.)

Fun Homework

24. Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. A ***digital subsequence*** of D is a sequence of positive integers composed in the usual way from disjoint substrings of D. For example, $3, 4, 5, 6, 8, 9, 32, 38, 46, 64, 83, 279$ is a digital subsequence of the first several digits of π:

$$\underbrace{3}, 1, \underbrace{4}, 1, \underbrace{5}, 9, 2, \underbrace{6}, 5, 3, 5, \underbrace{8}, \underbrace{9}, 7, 9, \underbrace{3, 2}, \underbrace{3, 8}, \underbrace{4, 6}, 2, \underbrace{6, 4}, 3, 3, \underbrace{8, 3}, \underbrace{2, 7, 9}$$

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the preceding example has length 12. As usual, a digital subsequence is ***increasing*** if each number is larger than its predecessor.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D. *[Hint: Be careful about your computational assumptions. How long does it take to compare two k-digit numbers?]*

For full credit, your algorithm should run in $O(n^4)$ time; faster algorithms are worth extra credit. The fastest algorithm I know for this problem runs in $O(n^2 \log n)$ time; achieving this bound requires several tricks, both in the algorithm and in its analysis.

Splitting Sequences/Arrays

Exam

25. A ***basic arithmetic expression*** is composed of characters from the set $\{1, +, \times\}$ and parentheses. Almost every integer can be represented by more than one basic arithmetic expression. For example, all of the following basic arithmetic expression represent the integer 14:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1+1)\times(1+1+1+1+1))+((1+1)\times(1+1))$$
$$(1+1)\times(1+1+1+1+1+1+1)$$
$$(1+1)\times(((1+1+1)\times(1+1))+1)$$

Describe and analyze an algorithm to compute, given an integer n as input, the minimum number of 1's in a basic arithmetic expression whose value is equal to n. The number of parentheses doesn't matter, just the number of 1's. For example, when $n = 14$, your algorithm should return 8, for the final expression above. The running time of your algorithm should be bounded by a small polynomial function of n.

Homework

26. Suppose you are given a sequence of integers separated by + and − signs; for example:

$$1+3-2-5+1-6+7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$1+3-2-5+1-6+7=-1$$
$$(1+3-(2-5))+(1-6)+7=9$$
$$(1+(3-2))-(5+1)-(6+7)=-17$$

Describe and analyze an algorithm to compute, given a list of integers separated by + and − signs, the maximum possible value the expression can take by adding parentheses. Parentheses must be used only to group additions and subtractions; in particular, do not use them to create implicit multiplication as in $1+3(-2)(-5)+1-6+7=33$.

27. Suppose you are given a sequence of integers separated by + and × signs; for example:

$$1+3\times 2\times 0+1\times 6+7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$(1+(3\times 2))\times 0+(1\times 6)+7=13$$
$$((1+(3\times 2\times 0)+1)\times 6)+7=19$$
$$(1+3)\times 2\times(0+1)\times(6+7)=208$$

(a) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, assuming all integers in the input are *positive*. *[Hint: This is easy.]* Exam

(b) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, assuming all integers in the input are *non-negative*. Exam

(c) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, with no restrictions on the input. Homework

Assume any arithmetic operation takes $O(1)$ time.

28. After graduating from Illinois, you decide to interview for a position at the Wall Street bank ***Long Live Boole***. The managing director of the bank, Eloob Egroeg, poses a 'solve-or-die' problems to each new employee, which they must solve within 24 hours. Those who fail to solve the problem are fired immediately! Homework

Entering the bank for the first time, you notice that the employee offices are organized in a straight row, with a large T or F printed on the door of each office.

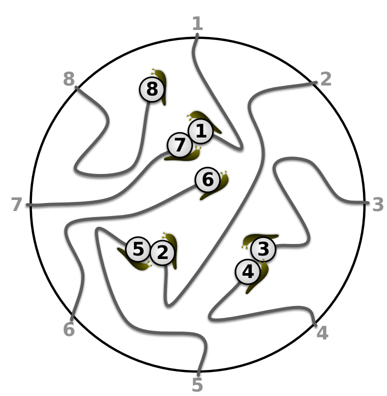
Furthermore, between each adjacent pair of offices, there is a board marked by one of the symbols $\wedge$, $\vee$, or $\oplus$. When you ask about these arcane symbols, Eloob confirms that T and F represent the boolean values TRUE and FALSE, and the symbols on the boards represent the standard boolean operators AND, OR, and XOR. He also explains that these letters and symbols describe whether certain combinations of employees can work together successfully. At the start of any new project, Eloob hierarchically clusters his employees by adding parentheses to the sequence of symbols, to obtain an unambiguous boolean expression. The project is successful if this parenthesized boolean expression evaluates to T.

For example, if the bank has three employees, and the sequence of symbols on and between their doors is $T \wedge F \oplus T$, there is exactly one successful parenthesization scheme: $(T \wedge (F \oplus T))$. However, if the list of door symbols is $F \wedge T \oplus F$, there is no way to add parentheses to make the project successful.

Eloob finally poses your solve-or-die interview question: Describe an algorithm to decide whether a given sequence of symbols can be parenthesized so that the resulting boolean expression evaluates to T. Your input is an array $S[0..2n]$, where $S[i] \in \{T, F\}$ when i is even, and $S[i] \in \{\vee, \wedge, \oplus\}$ when i is odd.

Homework

29. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.



The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates.
The organizers must pay $M[3,4] + M[2,5] + M[1,7]$.

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the

Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i, j] = M[j, i]$ is the reward to be paid if snails i and j meet.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.

30. ***AVL trees*** were the very first self-balancing balanced binary search trees, discovered in 1962 by Georgy Adelson-Velsky and Evgenii Landis. An AVL tree is a binary search tree where for every node v, the height of the left subtree of v and the height of the right subtree of v differ by at most one. Homework

    Describe and analyze an algorithm to construct an optimal AVL tree for a given set of keys and frequencies. Your input consists of a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches for $A[i]$. Your task is to construct an AVL tree for the given keys such that the total cost of all searches is as small as possible. This is exactly the same problem described in Section **??**, except that here the output tree must satisfy the AVL balance constraint.

31. You have mined a large slab of marble from a quarry. For simplicity, suppose the marble slab is a rectangle measuring n inches in height and m inches in width. You want to cut the slab into smaller rectangles of various sizes—some for kitchen counter tops, some for large sculpture projects, others for memorial headstones. You have a marble saw that can make either horizontal or vertical cuts across any rectangular slab. At any time, you can query the spot price $P[x, y]$ of an x-inch by y-inch marble rectangle, for any positive integers x and y. These prices depend on customer demand, and people who buy marble counter tops are weird, so don't make any assumptions about them; in particular, larger rectangles may have significantly smaller spot prices. Given the array of spot prices and the integers m and n as input, describe an algorithm to compute how to subdivide an $n \times m$ marble slab to maximize your profit. Homework

32. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..n, 1..n]$ of 0s and 1s. A *solid block* in M is a subarray of the form $M[i..i', j..i']$ in which all bits are equal. Suppose you want to decompose M into as few disjoint blocks as possible. Homework

    One natural recursive partitioning strategy is called a *guillotine subdivision*. If the entire bitmap M is a solid block, there is nothing to do. Otherwise, we cut M into two smaller bitmaps along a horizontal or vertical line, and then decompose the two smaller bitmaps recursively.

    (a) Describe and analyze an algorithm to compute a guillotine subdivision of M with as few cuts as possible.

    (b) Show that a guillotine subdivision does *not* always yield a partition into the smallest number of solid blocks.

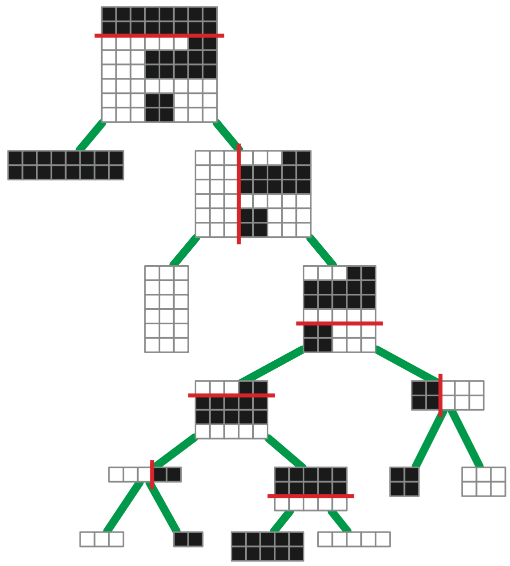


Figure 3.6. A guillotine subdivision of a bitmap with seven cuts and depth 5.

(c) Any guillotine subdivision can be represented as a binary tree, where each internal node stores the position and orientation of a cut, and each leaf stores a single but 0 or 1 indicting the contents of the corresponding block. Describe and analyze an algorithm to determine $M[i, j]$, given a guillotine decomposition for M and two indices i and j.

(d) Describe and analyze an algorithm to compute a guillotine subdivision for M with the smallest possible depth.

Homework

33. Congratulations! You've been hired by the giant online bookstore DeNile ("Not just a river in Egypt!") to optimize their warehouse robots. Each book that DeNile sells has a unique ISBN (International Standard Book Number), which is just a numerical value. Each of DeNile's warehouses contains a long row of bins, each containing multiple copies of a single book. These bins are arranged in sorted order by ISBN; each bin's ISBN is printed on the front of the bin in machine-readable form. Books are retrieved from these bins by robots, which run along rails parallel to the row of bins.

DeNile does not maintain a list of which bins contain which ISBN numbers; that would be too simple! Instead, to retrieve a desired book, the robot must first find that book's bin using a binary search. Because the search requires physical motion by the robot, we can no longer assume that each step of the binary search requires $O(1)$ time. Specifically:

- The robot always starts at the "0th bin" (where the books are loaded into boxes to ship to customers).

- Moving the robot from the ith bin to the jth bin requires $\alpha|i-j|$ seconds for some constant α.
- The robot must be directly in front of a bin in order to read that bin's ISBN. Reading an ISBN requires β seconds, for some constant β.
- Reversing the robot's direction of motion (from increasing to decreasing or vice versa) requires γ additional seconds, for some constant γ.
- When the robot finds the target bin, it extracts one book from that bin and returns to "the 0th bin".

Design and analyze an algorithm to compute a binary search tree over the bins that minimizes the total time the robot spends searching for books. Your input is an array $f[1..n]$ of integers, where $f[i]$ is the number of times that the robot will be asked to retrieve a book from the ith bin, along with the time parameters α, β, and γ.

34. A standard method to improve the cache performance of search trees is to pack more search keys and subtrees into each node. A ***B-tree*** is a rooted tree in which each internal node stores up to B keys and pointers to up to $B+1$ children, each the root of a smaller B-tree. Specifically, each node v stores three fields: Homework

- a positive integer $v.d \le B$,
- a *sorted* array $v.key[1..v.d]$, and
- an array $v.child[0..v.d]$ of child pointers.

In particular, the number of child pointers is always exactly one more than the number of keys.[16]

Each pointer $v.child[i]$ is either Null or a pointer to the root of a B-tree whose keys are all larger than $v.key[i]$ and smaller than $v.key[i+1]$. In particular, all keys in the leftmost subtree $v.child[0]$ are smaller than $v.key[1]$, and all keys in the rightmost subtree $v.child[v.d]$ are larger than $v.key[v.d]$.

Intuitively, you should have the following picture in mind:

$$
\begin{array}{ccccccccccc}
[\ \bullet & < key[1] < & \bullet & < key[2] < & \bullet & \cdots & \bullet & < key[d] < & \bullet\ ] \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow \\
T_0 & & T_1 & & T_2 & \cdots & T_{d-1} & & T_d
\end{array}
$$

Here T_i is the subtree pointed to by $child[i]$.

[16]Normally, B-trees are required to satisfy two additional constraints, which guarantee a worst-case search cost of $O(\log_B n)$: Every leaf must have exactly the same depth, and every node except possibly the root must contain at least $B/2$ keys. However, in this problem, we are not interested in optimizing the *worst-case* search cost, but rather the *total* cost of a sequence of searches, so we will not impose these additional constraints.

The ***cost*** of searching for a key x in a B-tree is the number of nodes in the path from the root to the node containing x as one of its keys. A 1-tree is just a standard binary search tree.

Fix an arbitrary positive integer $B > 0$. (I suggest $B = 8$.) Suppose your are given a sorted array $A[1, \dots, n]$ of search keys and a corresponding array $F[1, \dots, n]$ of frequency counts, where $F[i]$ is the number of times that we will search for $A[i]$. Your task is to describe and analyze an efficient algorithm to find a B-tree that minimizes the total cost of searching for the given keys with the given frequencies.

(a) Describe a polynomial-time algorithm for the special case $B = 2$.

(b) Describe an algorithm for arbitrary B that runs in $O(n^{B+c})$ time for some fixed integer c.

(c) Describe an algorithm for arbitrary B that runs in $O(n^c)$ time for some fixed integer c that does *not* depend on B.

Homework: (a)(b) or (a)(c) or (a)(d)

35. A string w of parentheses **(** and **)** and brackets **[** and **]** is ***balanced*** if it satisfies one of the following conditions:

- w is the empty string.
- $w = \mathtt{(}x\mathtt{)}$ for some balanced string x
- $w = \mathtt{[}x\mathtt{]}$ for some balanced string x
- $w = xy$ for some balanced strings x and y

For example, the string

$$w = \mathtt{([()][]())[()()]()}$$

is balanced, because $w = xy$, where

$$x = \mathtt{([()][]())} \qquad \text{and} \qquad y = \mathtt{[()()]()}.$$

(a) Describe and analyze an algorithm to determine whether a given string of parentheses and brackets is balanced.

(b) Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets.

(c) Describe and analyze an algorithm to compute the length of a shortest balanced supersequence of a given string of parentheses and brackets.

(d) Describe and analyze an algorithm to compute the minimum edit distance from a given string of parentheses and brackets to a balanced string of parentheses and brackets.

For each problem, your input is an array $w[1\,..\,n]$, where $w[i] \in \{\mathtt{(}, \mathtt{)}, \mathtt{[}, \mathtt{]}\}$ for every index i. (You may prefer to use different symbols instead of parentheses and brackets—for example, $\mathtt{L}, \mathtt{R}, \mathtt{l}, \mathtt{r}$ or $\triangleleft, \triangleright, \blacktriangleleft, \blacktriangleright$—but please tell your grader what symbols you're using!)

36. Congratulations! Your research team has just been awarded a \$50M multi-year project, jointly funded by DARPA, Google, and McDonald's, to produce DWIM: The first compiler to read programmers' minds! Your proposal and your numerous press releases all promise that DWIM will automatically correct errors in any given piece of code, while modifying that code as little as possible. Unfortunately, now it's time to start actually making the damn thing work. Homework

    As a warmup exercise, you decide to tackle the following necessary subproblem. Recall that the *edit distance* between two strings is the minimum number of single-character insertions, deletions, and replacements required to transform one string into the other. An *arithmetic expression* is a string w such that

    - w is a string of one or more decimal digits,
    - $w = (x)$ for some arithmetic expression x, or
    - $w = x \diamond y$ for some arithmetic expressions x and y and some binary operator $\diamond$.

    Suppose you are given a string of tokens from the alphabet $\{\#, \diamond, (,)\}$, where # represents a decimal digit and $\diamond$ represents a binary operator. Describe an algorithm to compute the minimum edit distance from the given string to an arithmetic expression.

37. (a) Describe and analyze an efficient algorithm to determine, given a string w and a regular expression R, whether $w \in L(R)$. Homework, assumes familiarity with regular expressions

    (b) *Generalized* regular expressions allow the binary operator $\cap$ (intersection) and the unary operator $\neg$ (complement), in addition to the usual $\bullet$ (concatenation), $+$ (or), and * (Kleene closure) operators. NFA constructions and Kleene's theorem imply that any generalized regular expression E represents a regular language $L(E)$.

    Describe and analyze an efficient algorithm to determine, given a string w and a generalized regular expression E, whether $w \in L(E)$.

    In both problems, assume that you are actually given a parse tree for the (generalized) regular expression, not just a string.

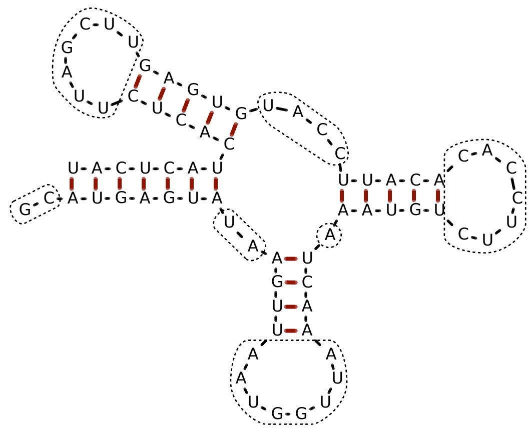
38. Ribonucleic acid (RNA) molecules are long chains of millions of nucleotides or *bases* of four different types: adenine (A), cytosine (C), guanine (G), and uracil (U). The *sequence* of an RNA molecule is a string $b[1..n]$, where each character $b[i] \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$ corresponds to a base. In addition to the chemical bonds between adjacent bases in the sequence, hydrogen bonds can form between certain pairs of bases. The set of bonded base pairs is called the *secondary structure* of the RNA molecule. Homework

    We say that two base pairs (i, j) and (i', j') with $i < j$ and $i' < j'$ ***overlap*** if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most base pairs are non-overlapping. Overlapping base pairs create so-called *pseudoknots* in the secondary structure, which are essential for some RNA functions, but are more difficult to predict.

    Suppose we want to predict the best possible secondary structure for a given RNA sequence. We will adopt a drastically simplified model of secondary structure:

- Each base can bond with at most one other base.
- Only A-U pairs and C-G pairs can bond.
- Pairs of the form $(i, i+1)$ and $(i, i+2)$ cannot bond.
- Bonded base pairs cannot overlap.

The last (and least realistic) restriction allows us to visualize RNA secondary structure as a sort of fat tree, as shown below.



Example RNA secondary structure with 21 bonded base pairs, indicated by heavy red lines.
Gaps are indicated by dotted curves. This structure has score $2^2 + 2^2 + 8^2 + 1^2 + 7^2 + 4^2 + 7^2 = 187$.

(a) Describe and analyze an algorithm that computes the maximum possible *number* of bonded base pairs in a secondary structure for a given RNA sequence.

(b) A *gap* in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the *score* of a secondary structure to be the sum of the *squares* of the gap lengths. (This score function is utterly fictional; real RNA structure prediction requires *much* more complicated scoring functions.) Describe and analyze an algorithm that computes the minimum possible score of a secondary structure for a given RNA sequence.

Trees and Subtrees

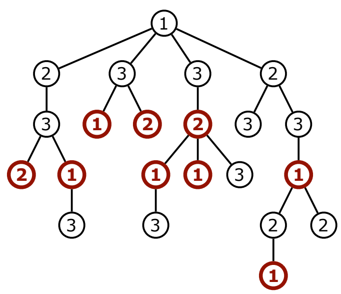
Exam

39. Oh, no! You've just been appointed as the new organizer of Giggle, Inc.'s annual mandatory holiday party! The employees at Giggle are organized into a strict hierarchy, that is, a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how "fun" the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all. Give an

algorithm that makes a guest list for the party that maximizes the sum of the "fun" ratings of the guests.

40. Since so few people came to last year's holiday party, the president of Giggle, Inc. decides to give each employee a present instead this year. Specifically, each employee must receive on the three gifts: (1) an all-expenses-paid six-week vacation anywhere in the world, (2) an all-the-pancakes-you-can-eat breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. Corporate regulations prohibit any employee from receiving exactly the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy. Exam

As Giggle, Inc.'s official party czar, it's *your* job to decide which gift each employee receives. Describe an algorithm to distribute gifts so that the minimum number of people are fired. Yes, you may send the president a flaming bag of dog poop.



A tree labeling with cost 9. The nine bold nodes have smaller labels than their parents.
This is *not* the optimal labeling for this tree.

More formally, you are given a rooted tree T, representing the company hierarchy, and you want to label each node in T with an integer 1, 2, or 3, so that every node has a different label from its parent. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T.

41. After the Flaming Dog Poop Holiday Debacle, you were strongly encouraged to seek other employment, and so you left Giggle for its competitor Yeehaw! Unfortunately, the new president of Yeehaw! just decided to imitate Giggle by throwing her own holiday party, and in light of your past experience, appointed you as the official party organizer. The president demands that you invite exactly k employees, including the president herself, and everyone who is invited is required to attend. Yeah, that'll be fun.

Just like at Giggle, employees at Yeehaw! are organized into a strict hierarchy: a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee indicating the *awkwardness* of inviting both that employee and their immediate supervisor; a negative value indicates that the employee and their supervisor actually like each other. Your goal is to choose a subset of exactly k employees to invite, so that the total awkwardness of the resulting party is as small as possible. For example, if the guest list does not include both an employee and their immediate supervisor, the total awkwardness is zero. The input to your algorithm is the tree T, the integer k, and the awkwardness of each node in T.

Exam

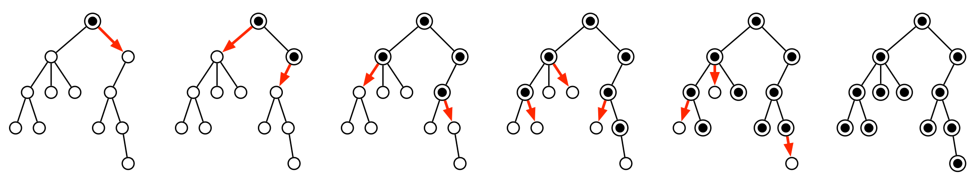
(a) Describe an algorithm that computes the total awkwardness of the least awkward subset of k employees, assuming the company hierarchy is described by a *binary* tree. That is, assume that each employee directly supervises at most two others.

Homework

(b) Describe an algorithm that computes the total awkwardness of the least awkward subset of k employees, with no restrictions on the company hierarchy.

Exam

42. Suppose we need to distribute a message to all the nodes in a rooted tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. Design an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in a given tree.



A message being distributed through a tree in five rounds.

Homework

43. One day, Alex got tired of climbing in a gym and decided to take a very large group of climber friends outside to climb. The climbing area where they went, had a huge wide boulder, not very tall, with various marked hand and foot holds. Alex quickly determined an “allowed” set of moves that her group of friends can perform to get from one hold to another.

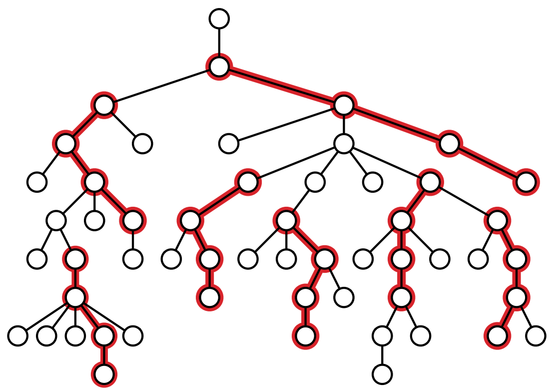
The overall system of holds can be described by a rooted tree T with n vertices, where each vertex corresponds to a hold and each edge corresponds to an allowed move between holds. The climbing paths converge as they go up the boulder, leading to a unique hold at the summit, represented by the root of T.[17]

Alex and her friends (who are all excellent climbers) decided to play a game, where as many climbers as possible are simultaneously on the boulder and each climber needs to perform a sequence of *exactly* k moves. Each climber can choose an arbitrary hold to start from, and all moves must move away from the ground. Thus, each climber traces out a path of k edges in the tree T, all directed toward the root.

[17]Q: Why do computer science professors think trees have their roots at the top?
A: Because they’ve never been outside!

However, no two climbers are allowed to touch the same hold; the paths followed by different climbers cannot intersect at all.

Describe and analyze an efficient algorithm to compute the maximum number of climbers that can play this game. More formally, you are given a rooted tree T and an integer k, and you want to find the largest possible number of disjoint paths in T, where each path has length k. Do ***not*** assume that T is a binary tree. For example, given the tree T below and $k = 3$ as input, your algorithm should return the integer 8.



Seven disjoint paths of length $k = 3$. This is *not* the largest such set of paths in this tree.

44. Let T be a rooted binary tree with n vertices, and let $k \leq n$ be a positive integer. We would like to mark k vertices in T so that every vertex has a nearby marked ancestor. More formally, we define the *clustering cost* of any subset K of vertices as Homework

$$cost(K) = \max_{v} cost(v, K),$$

where the maximum is taken over all vertices v in the tree, and $cost(v, K)$ is the distance from v to its nearest ancestor in K:

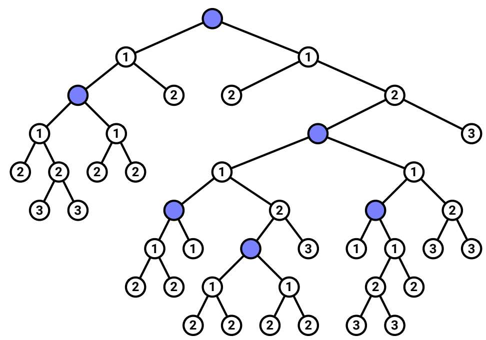
$$cost(v, K) = \begin{cases} 0 & \text{if } v \in K \\ \infty & \text{if } v \text{ is the root of } T \text{ and } v \notin K \\ 1 + cost(parent(v)) & \text{otherwise} \end{cases}$$

In particular, $cost(K) = \infty$ if K excludes the root of T.

(a) Describe a dynamic programming algorithm to compute, given the tree T and an integer k, the minimum clustering cost of any subset of k vertices in T. For full credit, your algorithm should run in $O(n^2k^2)$ time.

(b) Describe a dynamic programming algorithm to compute, given the tree T and an integer r, the size of the smallest subset of vertices whose clustering cost is at most r. For full credit, your algorithm should run in $O(nr)$ time.

(c) Show that your solution for part (b) implies an algorithm for part (a) that runs in $O(n^2 \log n)$ time.

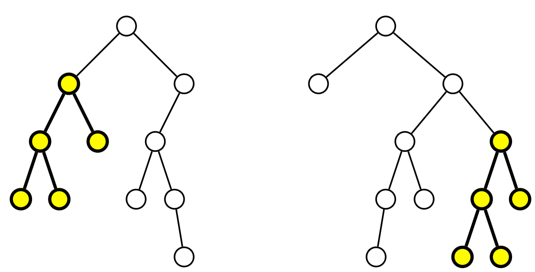


A subset of 5 vertices with clustering cost 3

45. This question asks you to find efficient algorithms to compute the ***largest common rooted subtree*** of two given rooted trees. Recall that a *rooted tree* is a connected acyclic graph with a designated node called the root. A rooted subtree of a rooted tree consists of an arbitrary node and all its descendants. The precise definition of “common” depends on which pairs of rooted trees we consider isomorphic.

Exam

(a) Recall that a *binary* tree is a rooted tree in which every node has a (possibly empty) *left* subtree and a (possibly empty) *right* subtree. Two binary trees are isomorphic if and only if they are both empty, or their left subtrees are isomorphic and their right subtrees are isomorphic. Describe an algorithm to find the largest common *binary* subtree of two given *binary* trees.



Two binary trees, with their largest common (rooted) subtree emphasized

Homework

(b) In an *ordered* rooted tree, each node has a *sequence* of children, which are the roots of ordered rooted subtrees. Two ordered rooted trees are isomorphic if they are both empty, or if their ith subtrees are isomorphic for every index i. Describe an algorithm to find the largest common ordered subtree of two ordered trees T_1 and T_2.

Homework

(c) In an *unordered* rooted tree, each node has an unordered *set* of children, which are the roots of unordered rooted subtrees. Two unordered rooted trees are isomorphic if they are both empty, or the subtrees of each root *can be ordered so that* their ith subtrees are isomorphic for every index i. Describe an algorithm to find the largest common unordered subtree of two unordered trees T_1 and T_2.

46. This question asks you to find efficient algorithms to compute optimal subtrees in *unrooted* trees—connected acyclic undirected graphs. A *subtree* of an unrooted tree is any connected subgraph.

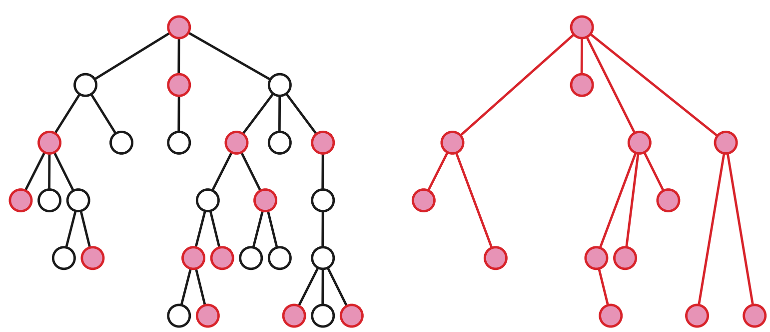
    (a) Suppose you are given an unrooted tree T with weights on its *edges*, which may be positive, negative, or zero. Describe an algorithm to find a *path* in T with maximum total weight. Exam

    (b) Suppose you are given an unrooted tree T with weights on its *vertices*, which may be positive, negative, or zero. Describe an algorithm to find a *subtree* of T with maximum total weight. *[This was a 2016 Google interview question.]* Homework

    (c) Let T_1 and T_2 be arbitrary *ordered* unrooted trees, meaning that the neighbors of every node have a well-defined cyclic order. Describe an algorithm to find the largest common *ordered* subtree of T_1 and T_2. Homework

    (d) Let T_1 and T_2 be arbitrary *unordered* unrooted trees. Describe an algorithm to find the largest common *unordered* subtree of T_1 and T_2. Homework, assumes familiarity with flows

47. ***Rooted minors*** of rooted trees are a natural generalization of subsequences. A rooted minor of a rooted tree T is any tree obtained by *contracting* one or more edges. When we contract an edge $u \to v$, where u is the parent of v, the children of v become new children of u and then v is deleted. In particular, the root of T is also the root of every rooted minor of T.



A rooted tree and one of its rooted minors.

    (a) Let T be a rooted tree with labeled nodes. We say that T is *boring* if, for each node x, all children of x have the same label; children of different nodes may have different labels. Describe an algorithm to find the largest boring rooted minor of a given labeled rooted tree. Homework

(b) Suppose we are given a rooted tree T whose nodes are labeled with numbers. Describe an algorithm to find the largest *heap-ordered rooted minor* of T. That is, your algorithm should return the largest rooted minor M such that every node in M has a smaller label than its children in M. Homework

Homework (c) Suppose we are given a *binary* tree T whose nodes are labeled with numbers. Describe an algorithm to find the largest *binary-search-ordered rooted minor* of T. That is, your algorithm should return a rooted minor M such that every node in M has at most two children, and an inorder traversal of M is an increasing subsequence of an inorder traversal of T.

Homework (d) Recall that a rooted tree is *ordered* if the children of each node have a well-defined left-to-right order. Describe an algorithm to find the largest binary-search-ordered minor of an *arbitrary* ordered tree T whose nodes are labeled with numbers. Again, the left-to-right order of nodes in M should be consistent with their order in T.

Homework (e) Describe an algorithm to find the largest common *ordered* rooted minor of two *ordered* labeled rooted trees.

Fun Homework, assumes familiarity with flows (f) Describe an algorithm to find the largest common *unordered* rooted minor of two *unordered* labeled rooted trees.