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

— George Santayana, *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' (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)

4 Dynamic Programming

4.1 Fibonacci Numbers

The Fibonacci numbers F_n, named after Leonardo Fibonacci Pisano[1], the mathematician 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! The annihilator method gives us an asymptotic bound of $\Theta(\phi^n)$, where $\phi = (\sqrt{5}+1)/2 \approx 1.61803398875$, the so-called *golden ratio*, is the largest root of the polynomial $r^2 - r - 1$. But it's fairly easy to prove (hint, hint) the exact solution $\boldsymbol{T(n) = 2F_{n+1} - 1}$. In other words, computing F_n using this algorithm takes more than twice as many steps as just counting to F_n!

Another way to see this is that the RecFibo is building a big binary tree of additions, with nothing but zeros and ones at the leaves. Since the eventual output is F_n, our algorithm must call RecRibo(1) (which returns 1) exactly F_n times. A quick inductive argument implies that RecFibo(0) is called exactly F_{n-1} times. Thus, the recursion tree has $F_n + F_{n-1} = F_{n+1}$ leaves, and therefore, because it's a full binary tree, it must have $2F_{n+1} - 1$ nodes.

[1] literally, "Leonardo, son of Bonacci, of Pisa"

4.2 Memo(r)ization and Dynamic Programming

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 RECURSIVEFIBO(n) results in one recursive call to RECURSIVEFIBO($n-1$), two recursive calls to RECURSIVEFIBO($n-2$), three recursive calls to RECURSIVEFIBO($n-3$), five recursive calls to RECURSIVEFIBO($n-4$), and in general, F_{k-1} recursive calls to RECURSIVEFIBO($n-k$), for any $0 \le k < n$. For each call, we're recomputing some Fibonacci number from scratch.

We can speed up the algorithm considerably just by writing down the results of our recursive calls and looking them up again if we need them later. This process is called *memoization*.[2]

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

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 $F[n]$. Once we see this pattern, we can replace the recursion with a simple for-loop that fills the array in order, instead of relying on the complicated recursion to do it for us. This gives us our first explicit *dynamic programming* algorithm.

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

ITERFIBO clearly takes only $O(n)$ time and $O(n)$ space to compute F_n, an exponential speedup over our original recursive algorithm.

We can reduce the space to $O(1)$ by noticing that we never need more than the last two elements of the array:

```
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.)

Even this algorithm can be improved further. There's a faster algorithm defined in terms of matrix multiplication, using the following wonderful fact:

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

[2] "My name is Elmer J. Fudd, millionaire. I own a mansion and a yacht."

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?

4.3 Uh... wait a minute.

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!

I've been cheating by assuming we can do arbitrary-precision arithmetic in constant time. If we use fast Fourier transforms, multiplying two n-digit numbers takes $O(n \log n)$ time. Thus, the matrix-based algorithm's actual running time is given by the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + O(n \log n),$$

which solves to $T(n) = O(n \log n)$ by the Master Theorem.

Is this slower than our "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 $O(n^2)$. (Do you see why?) Our matrix algorithm really is faster than our iterative 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

$$T(n) = T(n-1) + T(n-2) + O(n),$$

which still has the solution $O(\phi^n)$, by the annihilator method.

4.4 The Pattern: Smart Recursion

In a nutshell, dynamic programming is *recursion without repetition*. Developing a dynamic programming algorithm almost always requires 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.

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 data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

(c) **Analyze running time and space.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2}, we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

(d) **Identify dependencies between subproblems.** Except for the base cases, every recursive 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.

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

(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!

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.

> ***Dynamic programming is not about filling in tables; it's about smart recursion.***

As long as we memoize the correct recurrence, an explicit table isn't really necessary, but if the recursion is incorrect, nothing works.

4.5 Edit Distance

The *edit distance* between two words—sometimes also called the *Levenshtein distance*—is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word into another. For example, the edit distance between `FOOD` and `MONEY` is at most four:

$$\underline{\texttt{F}}\texttt{OOD} \rightarrow \texttt{MO}\underline{\texttt{O}}\texttt{D} \rightarrow \texttt{MON}\underset{\wedge}{\underline{\texttt{D}}} \rightarrow \texttt{MONE}\underline{\texttt{D}} \rightarrow \texttt{MONEY}$$

A better way to display this editing process is to place the words one above the other, with a gap in the first word for every insertion, and a gap in the second word for every deletion. Columns with two *different* characters correspond to substitutions. Thus, the number of editing steps is just the number of columns that don't contain the same character twice.

```
F  O  O     D
M  O  N  E  Y
```

It's fairly obvious that you can't get from FOOD to MONEY in three steps, so their edit distance is exactly four. Unfortunately, this is not so easy in general. Here's a longer example, showing that the distance between ALGORITHM and ALTRUISTIC is at most six. Is this optimal?

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

To develop a dynamic programming algorithm to compute the edit distance between two strings, we first need to develop a recursive definition. 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 substrings.** We can easily prove this by contradiction. If the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings. Once we figure out what should go in the last column, the Recursion Fairy will magically give us 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.
- **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.
- **Substitution:** Both rows have characters in the last column. If the characters are the same, 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 $Edit(A[1..m-1], B[1..n-1]) + 1$.

The edit distance between A and B is the smallest of these three possibilities:[3]

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

This recurrence has two easy base cases. The only way to convert the empty string into a string of n characters is by performing n insertions. Similarly, the only way to convert a string of m characters into the empty string is with m deletions, Thus, if ε denotes the empty string, we have

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

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

[3]Once again, I'm using Iverson's bracket notation $\big[P\big]$ to denote the *indicator variable* for the logical proposition P, which has value 1 if P is true and 0 if P is false.

Now notice that the arguments to our recursive subproblems are always *prefixes* of the original strings A and B. Thus, we can simplify our notation considerably by using the lengths of the prefixes, instead of the prefixes themselves, as the arguments to our recursive function. So let's write $Edit(i, j)$ as shorthand for $Edit(A[1..i], B[1..j])$. This function satisfies the following recurrence:

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

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

This recurrence translates directly into a recursive algorithm. Just out of curiosity, we can analyze the running time of this algorithm by solving the following recurrence:[4]

$$T(m,n) = \begin{cases} O(1) & \text{if } n = 0 \text{ or } m = 0, \\ T(m,n-1) + T(m-1,n) + T(n-1,m-1) + O(1) & \text{otherwise.} \end{cases}$$

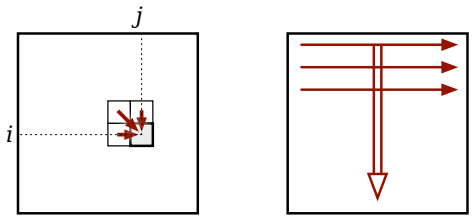
I don't know of a general closed-form solution for this mess, but we can derive an upper bound by defining a new function

$$T'(N) = \max_{n+m=N} T(n,m) = \begin{cases} O(1) & \text{if } N = 0, \\ 2T(N-1) + T(N-2) + O(1) & \text{otherwise.} \end{cases}$$

The annihilator method implies that $T'(N) = O((1+\sqrt{2})^N)$. Thus, the running time of our recursive edit-distance algorithm is *at most* $T'(n+m) = O((1+\sqrt{2})^{n+m})$.

We can dramatically reduce the running time of this algorithm down by memoization. Because each recursive subproblem can be identified by two indices i and j, we can store the intermediate values in a two-dimensional array $\boldsymbol{Edit[0..m, 0..n]}$. Note that the index ranges start at zero to accommodate the base cases. Since there are $\Theta(mn)$ entries in the table, our memoized algorithm uses $\boldsymbol{\Theta(mn)}$ ***space***. Since each entry in the table can be computed in $\Theta(1)$ time once we know its predecessors, our memoized algorithm runs in $\boldsymbol{\Theta(mn)}$ ***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 $Edit[i, j]$ is filled in *after* the neighboring entries directly above, directly to the left, and both above and to the left. But just this partial information is enough to give us an explicit evaluation order. 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.



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

[4]You can skip this step. Since we're going to memoize this algorithm, the running time of the non-memoized algorithm is not particularly important. In particular, we won't ask you do do this on homeworks or exams.

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

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

Here's the resulting table for `ALGORITHM` → `ALTRUISTIC`. Bold numbers indicate places where characters in the two strings are equal. The arrows represent the predecessor(s) that actually define each entry. Each direction of arrow corresponds to a different edit operation: horizontal=deletion, vertical=insertion, and diagonal=substitution. Bold 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.) Moreover, since we can compute these arrows in a post-processing phase from the values stored in the table, we can reconstruct the actual optimal editing sequence in $O(n+m)$ additional time.

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

The edit distance between `ALGORITHM` and `ALTRUISTIC` is indeed six. There are three paths through this table from the top left to the bottom right, so there are three optimal edit sequences:

```
 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
```

4.6 Warning: Greed is Stupid

If we're very very very very 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 look for the best first step, take it, and then continue. 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 edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions don't cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string. 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 $_{\text{almost}}$ **never work!**
Use dynamic programming instead!

What, never? No, never! What, *never*? Well... hardly ever.[5]

A different lecture note describes the effort required to prove that greedy algorithms are correct, in the rare instances when they are. **You will not receive *any* credit for *any* greedy algorithm for *any* problem in this class without a *formal* proof of correctness.** We'll push through the formal proofs for several greedy algorithms later in semester.

4.7 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.

A ***independent set*** in a graph is a subset of the vertices that have 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 described in another lecture note. But from some special cases of graphs, we can find the largest independent set efficiently. In particular, when the input graph is a tree (a connected and acyclic graph) with n vertices, we can compute the largest independent set in $O(n)$ time.

In the recursion notes, we saw a recursive algorithm for computing the size of the largest independent set in an arbitrary graph:

```
MaximumIndSetSize(G):
  if G = ∅
      return 0

  v ← any node in G
  withv ← 1 + MaximumIndSetSize(G \ N(v))
  withoutv ← MaximumIndSetSize(G \ {v})
  return max{withv, withoutv}.
```

[5]Greedy hardly ever ever works! Then give three cheers, and one cheer more, for the hardy Captain of the *Pinafore*! Then give three cheers, and one cheer more, for the Captain of the *Pinafore*!

Here, $N(v)$ denotes the *neighborhood* of v: the set containing v and all of its neighbors. As we observed in the other lecture notes, this algorithm has a worst-case running time of $O(2^n \text{ poly}(n))$, where n is the number of vertices in the input graph.

Now suppose we require that the input graph is a tree; we will call this tree T instead of G from now on. We need to make a slight change to the algorithm to make it truly recursive. The subgraphs $T \setminus \{v\}$ and $T \setminus N(v)$ are forests, which may have more than one component. But the largest independent set in a disconnected graph is just the union of the largest independent sets in its components, so we can separately consider each tree in these forests. Fortunately, this has the added benefit of making the recursive algorithm more efficient, especially if we can choose the node v such that the trees are all significantly smaller than T. Here is the modified algorithm:

```
MaximumIndSetSize(T):
  if T = ∅
      return 0
  v ← any node in T
  withv ← 1
  for each tree T' in T \ N(v)
      withv ← withv + MaximumIndSetSize(T')
  withoutv ← 0
  for each tree T' in T \ {v}
      withoutv ← withoutv + MaximumIndSetSize(T')
  return max{withv, withoutv}.
```

Now let's try to memoize this algorithm. Each recursive subproblem considers a subtree (that is, a connected subgraph) of the original tree T. Unfortunately, a single tree T can have exponentially many subtrees, so we seem to be doomed from the start!

Fortunately, there's a degree of freedom that we have not yet exploited: *We get to choose the vertex v.* We need a recipe—an algorithm!—for choosing v in each subproblem that limits the number of different subproblems the algorithm considers. To make this work, we impose some additional structure on the original input tree. Specifically, we declare one of the vertices of T to be the *root*, and we orient all the edges of T away from that root. Then we let v be the root of the input tree; this choice guarantees that each recursive subproblem considers a *rooted* subtree of T. Each vertex in T is the root of exactly one subtree, so now the number of distinct subproblems is exactly n. We can further simplify the algorithm by only passing a single node instead of the entire subtree:

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

What data structure should we use to store intermediate results? The most natural choice is the tree itself! Specifically, for each node v, we store the result of MaximumIndSetSize(v) in a new field $v.MIS$. (We *could* use an array, but then we'd have to add a new field to each node anyway, pointing to the corresponding array entry. Why bother?)

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! Since each vertex has at most one parent and at most one grandparent, the total running time is $O(n)$.

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 all children are visited before their parent. In particular, we can use a straightforward post-order traversal.

Here is the resulting dynamic programming algorithm. Yes, it's still recursive. I've swapped the evaluation of the with-v and without-v cases; we need to visit the kids first anyway, so why not consider the subproblem that depends directly on the kids first?

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

Another option is to store *two* values for each rooted subtree: the size of the largest independent set *that includes the root*, and the size of the largest independent set *that excludes the root*. This gives us an even simpler algorithm, with the same $O(n)$ running time.

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

4.8 Optimal Binary Search Trees

In an earlier lecture, 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 searches to $A[i]$. 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. We developed the following recurrence for this problem:

$$OptCost(f[1..n]) = \min_{1 \le r \le n} \left\{ OptCost(f[1..r-1]) \; + \; \sum_{i=1}^{n} f[i] \; + \; OptCost(f[r+1..n]) \right\}$$

To put this recurrence in more standard form, fix the frequency array f, and let $OptCost(i, j)$ denote the total search time in the optimal search tree for the subarray $A[i..j]$. To simplify notation a bit, let $F(i, j)$ denote the total frequency count for all the keys in the interval $A[i..j]$:

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

We can now write

$$\boxed{OptCost(i,j) = \begin{cases} 0 & \text{if } j < i \\ F(i,j) + \min\limits_{i \le r \le j} \big(OptCost(i,r-1) + OptCost(r+1,j)\big) & \text{otherwise} \end{cases}}$$

The base case might look a little weird, but all it means is that the total cost for searching an empty set of keys is zero.

The algorithm will be somewhat simpler and more efficient if we precompute all possible values of $F(i,j)$ and store them in an array. Computing each value $F(i,j)$ using a separate for-loop would $O(n^3)$ time. A better approach is to turn the recurrence

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

into the following $O(n^2)$-time dynamic programming algorithm:

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

This will be used as an initialization subroutine in our final algorithm.

So now let's compute the optimal search tree cost $OptCost(1,n)$ from the bottom up. We can store all intermediate results in a table $OptCost[1..n, 0..n]$. Only the entries $OptCost[i,j]$ with $j \ge i-1$ will actually be used. The base case of the recurrence tells us that any entry of the form $OptCost[i, i-1]$ can immediately be set to 0. For any other entry $OptCost[i,j]$, we can use the following algorithm fragment, which comes directly from the recurrence:

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

The only question left is what order to fill in the table.

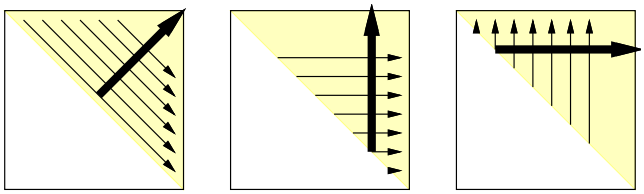
Each entry $OptCost[i,j]$ depends on all entries $OptCost[i, r-1]$ and $OptCost[r+1, j]$ with $i \le k \le j$. In other words, every entry in the table depends on all the entries directly to the left or directly below. In order to fill the table efficiently, we must choose an order that computes all those entries before $OptCost[i,j]$. There are at least three different orders that satisfy this constraint. The one that occurs to most people first is to scan through the table one diagonal at a time, starting with the trivial base cases $OptCost[i, i-1]$. The complete algorithm looks like this:

```
OptimalSearchTree(f[1..n]):
  InitF(f[1..n])
  for i ← 1 to n
      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. These two orders give us the following algorithms:

```
OPTIMALSEARCHTREE2(f[1..n]):
  INITF(f[1..n])
  for i ← n downto 1
      OptCost[i, i−1] ← 0
      for j ← i to n
          COMPUTEOPTCOST(i, j)
  return OptCost[1, n]
```

```
OPTIMALSEARCHTREE3(f[1..n]):
  INITF(f[1..n])
  for j ← 0 to n
      OptCost[j+1, j] ← 0
      for i ← j downto 1
          COMPUTEOPTCOST(i, j)
  return OptCost[1, n]
```



Three different evaluation orders for the table $OptCost[i, j]$.

No matter which of these orders we actually use, the resulting algorithm runs in $\boldsymbol{\Theta(n^3)}$ ***time*** and uses $\boldsymbol{\Theta(n^2)}$ ***space***.

We could have predicted these space and time bounds directly from the original recurrence.

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

First, the function has two arguments, each of which can take on any value between 1 and n, so we probably need a table of size $O(n^2)$. Next, there are *three* variables in the recurrence (i, j, and r), each of which can take any value between 1 and n, so it should take us $O(n^3)$ time to fill the table.

4.9 More Examples

We've already seen two other examples of recursive algorithms that we can significantly speed up via dynamic programming.

4.9.1 Subset Sum

Recall from the very first lecture that the *Subset Sum* problem asks, given a set X of positive integers (represented as an array $X[1..n]$ and an integer T, whether any subset of X sums to T. In that lecture, 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

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

Our goal is to compute $S(1,T)$, using the recurrence

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

Observe that there are only nT possible values for the input parameters that lead to the interesting case of this recurrence, so storing the results of all such subproblems requires ***O(mn) 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 memoizing this recurrence gives us and algorithm that runs in ***O(nT) time***.[6] To turn this into an explicit dynamic programming algorithm, we only need to consider the subproblems $S(i,t)$ in the proper order:

```
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]
```

This direct algorithm clearly always uses ***O(nT) time and space***. In particular, if T is significantly larger than 2^n, this algorithm is actually slower than our naïve recursive algorithm. Dynamic programming isn't *always* an improvement!

4.9.2 Longest Increasing Subsequence

We also developed a recurrence for the longest increasing subsequence problem. Fix the original input array $A[1..n]$ with a sentinel value $A[0] = -\infty$. Let $L(i,j)$ denote the length of the longest increasing subsequence of $A[j..n]$ with all elements larger than $A[i]$. Our goal is to compute $L(0,1)-1$. (The -1 removes the sentinel $-\infty$.) For any $i < j$, our recurrence can be stated as follows:

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

The recurrence suggests that our algorithm should use $O(n^2)$ time and space, since the input parameters i and j each can take n different values. To get an explicit dynamic programming algorithm, we only need to ensure that both $L(i,j+1)$ and $L(j,j+1)$ are considered before $L(i,j)$, for all i and j.

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

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

  return L[0, 1] − 1           ⟨⟨Don't count the sentinel⟩⟩
```

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

As predicted, this algorithm clearly uses ***O(n²) time and space***. We can reduce the space to $O(n)$ by only maintaining the two most recent columns of the table, $L[\cdot, j]$ and $L[\cdot, j+1]$.

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

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

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 $O(n)$ space, and the resulting algorithm runs in $O(n^2)$ time. To transform this into a dynamic programming algorithm, we only need to guarantee that $L'(j)$ is computed before $L'(i)$ whenever $i < j$.

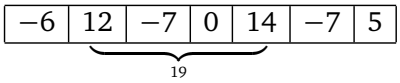
```
LIS2(A[1..n]):
  A[0] = -∞                          ⟨⟨Add a sentinel⟩⟩

  for i ← n downto 0
      L'[i] ← 1
      for j ← i + 1 to n
          if A[j] > A[i] and 1 + L'[j] > L'[i]
              L'[i] ← 1 + L'[j]

  return L'[0] - 1                   ⟨⟨Don't count the sentinel⟩⟩
```

Exercises

1. Suppose you are given an array $A[1..n]$ of integers. Describe and analyze an algorithm that finds the largest sum of of elements in a contiguous subarray $A[i..j]$. For example, if the array contains the numbers $(-6, 12, -7, 0, 14, -7, 5)$, then the largest sum of any contiguous subarray is $19 = 12 - 7 + 0 + 14$.



2. (a) 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.

   (b) Call a sequence $X[1..n]$ *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.

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

   (d) Call a sequence $X[1..n]$ *accelerating* 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 accelerating subsequence of an arbitrary array A of integers.

3. 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.

4. A palindrome is any string that is exactly the same as its reversal, such as DEED, RACECAR, or SAIPPUAKAUPPIAS.[7] 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 ILLINOISURBANACHAMPAIGN is NIAACAAIN (or NIAAHAAIN), and the longest palindrome subsequence of HYAKUGOJYUUICHI[8] is HUUUH[9] (or HUGUH or HUYUH or...).

[7] Finnish for 'soap dealer'.
[8] Japanese for 'one hundred fifty-one'.
[9] English for 'What the heck are you talking about?'

5. A *palindrome* is a string that reads the same forwards and backwards, like `x`, `pop`, `noon`, `redivider`, or `amanaplanacatahamayakayamahatacanalpanama`. Any string can be broken into sequence of palindromes. For example, the string `bubbaseesabanana` ('Bubba sees a banana.') can be broken into palindromes in several different ways; for example:

   ```
                    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 + a + n + a
   ```

   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.

6. You are given an $m \times n$ matrix M in which each entry is either 0 or 1. A *solid block* is a rectangular submatrix of M in which each entry is 1. Describe and analyze an efficient algorithm to find a solid block in M with maximum area.

7. You are a bus driver with a soda fountain machine in the back and a bus full of very hyper students, who are drinking more soda as they ride along the highway. Your goal is to drop the students off as quickly as possible. More specifically, each minute that a student is on your bus, he drinks another ounce of soda. Your goal is to drop the students off quickly, so that in total they drink as little soda as possible.

   You know how many students will get off of the bus at each exit. Your bus begins partway along the highway (probably not at either end), and moves 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 other 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. The input to the algorithm should be: 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).

8. 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.[10]

   (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.

[10] For more details on the history and culture of Nadira, including images of the various denominations of Dream Dollars, see http://www.dream-dollars.com.

(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.)

(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.)

9. What excitement! The Champaign Spinners and the Urbana Dreamweavers have advanced to meet each other in the World Series of Basket-weaving! The World Champions will be decided by a best-of- $2n-1$ series of head-to-head weaving matches, and the first to win n matches will take home the coveted Golden Basket (for example, a best-of-7 series requiring four match wins, but we will keep the generalized case). We know that for any given match there is a constant probability p that Champaign will win, and a subsequent probability $q = 1 - p$ that Urbana will win.

   Let $P(i, j)$ be the probability that Champaign will win the series given that they still need i more victories, whereas Urbana needs j more victories for the championship. $P(0, j) = 1$ for any j, because Champaign needs no more victories to win. Similarly, $P(i, 0) = 0$ for any i, as Champaign cannot possibly win if Urbana already has. $P(0, 0)$ is meaningless. Champaign wins any particular match with probability p and loses with probability q, so

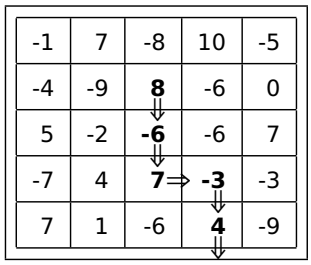
$$P(i, j) = p \cdot P(i-1, j) + q \cdot P(i, j-1)$$

   for any $i \geq 1$ and $j \geq 1$.

   Describe and analyze an efficient algorithm that computes the probability that Champaign will win the series (that is, calculate $P(n, n)$), given the parameters n, p, and q as input.

10. *Vankin's Mile* is a 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.)



    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.

11. 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, 'bananaananas' is a shuffle of 'banana' and 'ananas' in several different ways.

$$\mathsf{banana}_{\mathsf{ananas}} \qquad \mathsf{ban}_{\mathsf{ana}}\mathsf{ana}_{\mathsf{nas}} \qquad \mathsf{b}_{\mathsf{an}}\mathsf{an}_{\mathsf{a}}\mathsf{a}_{\mathsf{na}}\mathsf{na}_{\mathsf{s}}$$

The strings 'prodgyrnamammiincg' and 'dyprongarmammicing' are both shuffles of 'dynamic' and 'programming':

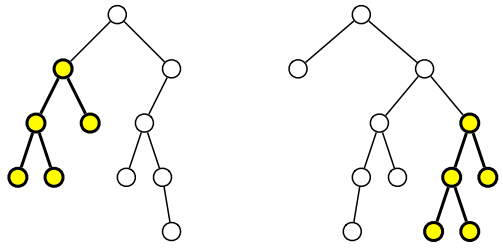
$$\mathsf{pro}^{\mathsf{d}}\mathsf{g}^{\mathsf{y}}\mathsf{r}^{\mathsf{nam}}\mathsf{ammi}^{\mathsf{i}}\mathsf{n}^{\mathsf{c}}\mathsf{g} \qquad\qquad {}^{\mathsf{dy}}\mathsf{pro}^{\mathsf{n}}\mathsf{g}^{\mathsf{a}}\mathsf{r}^{\mathsf{m}}\mathsf{amm}^{\mathsf{ic}}\mathsf{ing}$$

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.

12. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the ith word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we will need to insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel 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 number of extra white-space pixels in each line (not counting the one pixel required between every adjacent pair of words). Specifically, if a line contains words i through j, then the amount of extra white space on that line is $P - j + i - \sum_{k=i}^{j} p_k$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

13. A *subtree* of a (rooted, ordered) binary tree T consists of a node and all its descendants. Design and analyze an efficient algorithm to compute the ***largest common subtree*** of two given binary trees T_1 and T_2; this is the largest subtree of T_1 that is isomorphic to a subtree in T_2. The contents of the nodes are irrelevant; we are only interested in matching the underlying combinatorial structure.



Two binary trees, with their largest common subtree emphasized

14. A company is planning a party for its employees. The employees in the company are organized into a strict hierarchy, that is, a tree with the company president at the root. The organizers of the party 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 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.

*15. Scientists have branched out from the bizarre planet of Yggdrasil to study the vodes which have settled on Ygdrasil's moon, Xryltcon. All vodes on Xryltcon are descended from the first vode to arrive there, named George. Each vode has a color, either cyan, magenta, or yellow, but breeding patterns are *not* the same as on Yggdrasil; every vode, regardless of color, has either two children (with arbitrary colors) or no children.

George and all his descendants are alive and well, and they are quite excited to meet the scientists who wish to study them. Unsurprisingly, these vodes have had some strange mutations in their isolation on Xryltcon. Each vode has a *weirdness* rating; weirder vodes are more interesting to the visiting scientists. (Some vodes even have negative weirdness ratings; they make other vodes more boring just by standing next to them.)

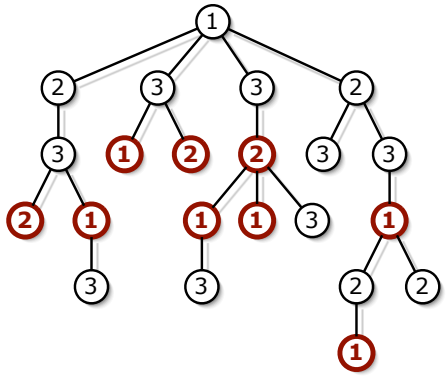
Also, Xryltconian society is strictly governed by a number of sacred cultural traditions.

- No cyan vode may be in the same room as its non-cyan children (if it has any).
- No magenta vode may be in the same room as its parent (if it has one).
- Each yellow vode must be attended at all times by its grandchildren (if it has any).
- George must be present at any gathering of more than fifty vodes.

The scientists have exactly one chance to study a group of vodes in a single room. You are given the family tree of all the vodes on Xryltcon, along with the weirdness value of each vode. Design and analyze an efficient algorithm to decide which vodes the scientists should invite to maximize the sum of the weirdness values of the vodes in the room. Be careful to respect all of the vodes' cultural taboos.

16. Oh, no! You have been appointed as the gift czar for Giggle, Inc.'s annual mandatory holiday party! The president of the company, who is certifiably insane, has declared that *every* Giggle employee must receive one of 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 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. How do you decide what gifts everyone gets if you want to minimize the number of people that get fired?

More formally, suppose you are given a rooted tree T, representing the company hierarchy. You want to label each node in T with an integer 1, 2, or 3, such 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. (Your algorithm does *not* have to compute the actual best labeling—just its cost.)



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

17. 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 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 $P[r]$.

18. 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 Hamiltonian cycle in G, or ∞ if G has no Hamiltonian cycles. *[Hint: The obvious recursive algorithm takes $O(n!)$ time.]*

19. 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 $b[i] \in \{A, C, G, U\}$ 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.

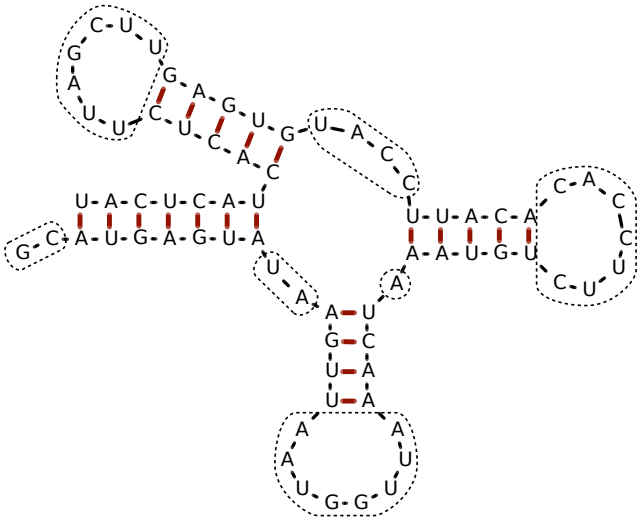
    We say that two base pairs (i, j) and (i', j') with $i < j$ and $i' < j'$ ***overlap*** if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most base pairs are non-overlapping. Overlapping base pairs create so-called *pseudoknots* in the secondary structure, which are essential for some RNA functions, 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 be paired 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.
- Overlapping base pairs cannot bond.

The last restriction allows us to visualize RNA secondary structure as a sort of fat tree.



Example RNA secondary structure with 21 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.

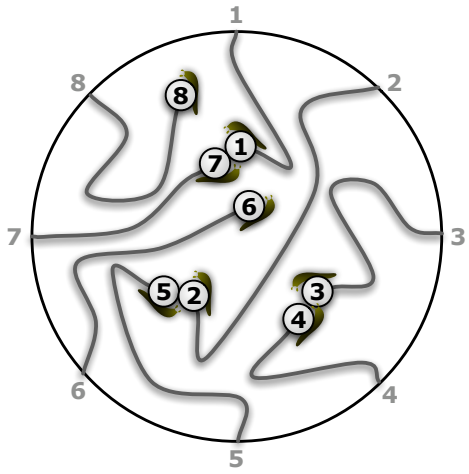
(b) A *gap* in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the *score* of a secondary structure to be the sum of the *squares* of the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

(The sum-of-squared-gap-lengths score function has absolutely no connection to reality; I just made it up. Real RNA structure prediction requires much more complicated scoring functions.)

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.

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.



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

20. Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. An *digital subsequence* of D is an 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 an increasing digital subsequence of the first several digits of π:

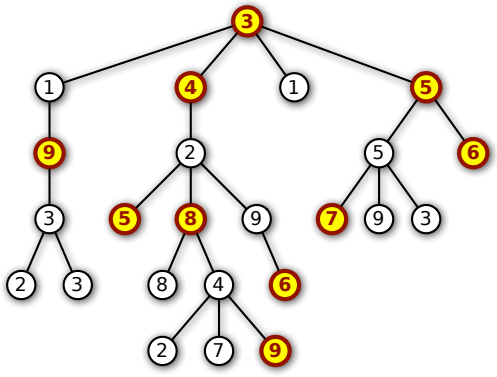
$$\underline{3},1,\underline{4},1,\underline{5},9,2,\underline{6},5,3,5,\underline{8},\underline{9},7,9,\underline{3,2},\underline{3,8},\underline{4,6},2,\underline{6,4},3,3,\underline{8,3},\underline{2,7,9}$$

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the preceding example has length 12.

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?]*

21. Let T be an arbitrary rooted tree, where each vertex is labeled with a positive integer. A subset S of the nodes of T is *heap-ordered* if it satisfies two properties:

- S contains a node that is an ancestor of every other node in S.
- For any node v in S, the label of v is larger than the labels of any ancestor of v in S.



A heap-ordered subset of nodes in a tree.

(a) Describe an algorithm to find the largest heap-ordered subset S of nodes in T that has the heap property in $O(n^2)$ time.

(b) Modify your algorithm from part (a) so that it runs in $O(n \log n)$ time when T is either a linked list. *[Hint: This special case is equivalent to a problem you've seen before.]*

*(c) Describe an algorithm to find the largest subset S of nodes in T that has the heap property, in $O(n \log n)$ time. *[Hint: Find an algorithm to merge two sorted lists of lengths k and ℓ in $O(\log \binom{k+\ell}{k})$ time.]*