left-to-right order of Figure 6.1, we can always be sure that by the time we get to a node $v$, we already have all the information we need to compute $\texttt{dist}(v)$. We are therefore able to compute all distances in a single pass:

```
initialize all dist(·) values to ∞
dist(s) = 0
for each v ∈ V\{s}, in linearized order:
    dist(v) = min_{(u,v)∈E}{dist(u) + l(u, v)}
```

Notice that this algorithm is solving a collection of *subproblems*, $\{\texttt{dist}(u) : u \in V\}$. We start with the smallest of them, $\texttt{dist}(s)$, since we immediately know its answer to be $0$. We then proceed with progressively "larger" subproblems—distances to vertices that are further and further along in the linearization—where we are thinking of a subproblem as large if we need to have solved a lot of other subproblems before we can get to it.

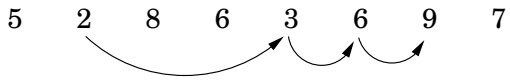
This is a very general technique. At each node, we compute some function of the values of the node's predecessors. It so happens that our particular function is a minimum of sums, but we could just as well make it a *maximum*, in which case we would get *longest* paths in the dag. Or we could use a product instead of a sum inside the brackets, in which case we would end up computing the path with the smallest product of edge lengths.

*Dynamic programming* is a very powerful algorithmic paradigm in which a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to small problems to help figure out larger ones, until the whole lot of them is solved. In dynamic programming we are not given a dag; the dag is *implicit*. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems: if to solve subproblem $B$ we need the answer to subproblem $A$, then there is a (conceptual) edge from $A$ to $B$. In this case, $A$ is thought of as a smaller subproblem than $B$—and it will always be smaller, in an obvious sense.

But it's time we saw an example.

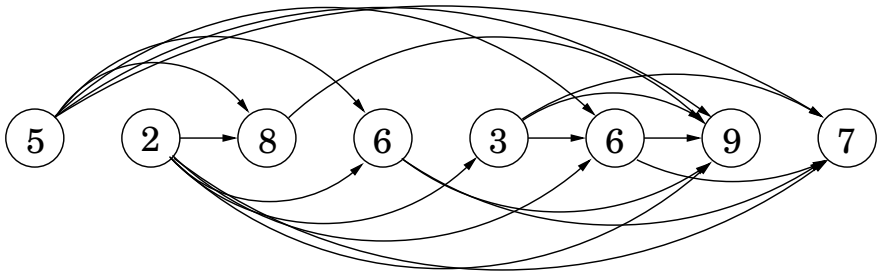
## 6.2 Longest increasing subsequences

In the *longest increasing subsequence* problem, the input is a sequence of numbers $a_1, \ldots, a_n$. A *subsequence* is any subset of these numbers taken in order, of the form $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, and an *increasing* subsequence is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length. For instance, the longest increasing subsequence of $5, 2, 8, 6, 3, 6, 9, 7$ is $2, 3, 6, 9$:




In this example, the arrows denote transitions between consecutive elements of the optimal solution. More generally, to better understand the solution space, let's create a graph of *all* permissible transitions: establish a node $i$ for each element $a_i$, and add directed edges $(i, j)$ whenever it is possible for $a_i$ and $a_j$ to be consecutive elements in an increasing subsequence, that is, whenever $i < j$ and $a_i < a_j$ (Figure 6.2).

**Figure 6.2** The dag of increasing subsequences.




Notice that (1) this graph $G = (V, E)$ is a dag, since all edges $(i, j)$ have $i < j$, and (2) there is a one-to-one correspondence between increasing subsequences and paths in this dag. Therefore, our goal is simply to find the longest path in the dag!

Here is the algorithm:

```
for j = 1, 2, ..., n:
    L(j) = 1 + max{L(i) : (i, j) ∈ E}
return max_j L(j)
```

$L(j)$ is the length of the longest path—the longest increasing subsequence—ending at $j$ (plus 1, since strictly speaking we need to count nodes on the path, not edges). By reasoning in the same way as we did for shortest paths, we see that any path to node $j$ must pass through one of its predecessors, and therefore $L(j)$ is 1 plus the maximum $L(\cdot)$ value of these predecessors. If there are no edges into $j$, we take the maximum over the empty set, zero. And the final answer is the *largest* $L(j)$, since any ending position is allowed.

This is dynamic programming. In order to solve our original problem, we have defined a collection of subproblems $\{L(j) : 1 \leq j \leq n\}$ with the following key property that allows them to be solved in a single pass:

> (*) There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems, that is, subproblems that appear earlier in the ordering.

In our case, each subproblem is solved using the relation

$$L(j) \;=\; 1 + \max\{L(i) : (i, j) \in E\},$$

an expression which involves only smaller subproblems. How long does this step take? It requires the predecessors of $j$ to be known; for this the adjacency list of the reverse graph $G^R$, constructible in linear time (recall Exercise 3.5), is handy. The computation of $L(j)$ then takes time proportional to the indegree of $j$, giving an overall running time linear in $|E|$. This is at most $O(n^2)$, the maximum being when the input array is sorted in increasing order. Thus the dynamic programming solution is both simple and efficient.

There is one last issue to be cleared up: the $L$-values only tell us the *length* of the optimal subsequence, so how do we recover the subsequence itself? This is easily managed with the

same bookkeeping device we used for shortest paths in Chapter 4. While computing $L(j)$, we should also note down $\texttt{prev}(j)$, the next-to-last node on the longest path to $j$. The optimal subsequence can then be reconstructed by following these backpointers.

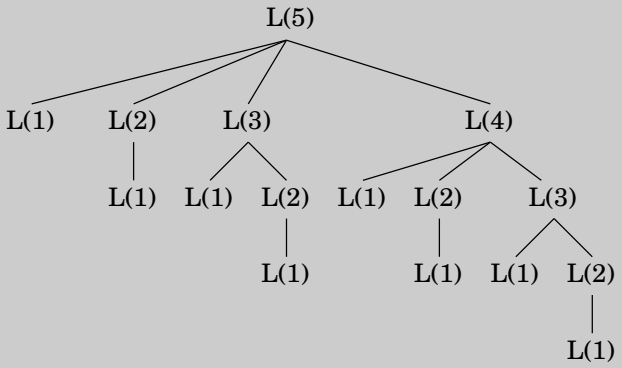
### Recursion? No, thanks.

Returning to our discussion of longest increasing subsequences: the formula for $L(j)$ also suggests an alternative, recursive algorithm. Wouldn't that be even simpler?

Actually, recursion is a very bad idea: the resulting procedure would require exponential time! To see why, suppose that the dag contains edges $(i, j)$ for *all* $i < j$—that is, the given sequence of numbers $a_1, a_2, \ldots, a_n$ is sorted. In that case, the formula for subproblem $L(j)$ becomes

$$L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}.$$

The following figure unravels the recursion for $L(5)$. Notice that the same subproblems get solved over and over again!




For $L(n)$ this tree has exponentially many nodes (can you bound it?), and so a recursive solution is disastrous.

Then why did recursion work so well with divide-and-conquer? The key point is that in divide-and-conquer, a problem is expressed in terms of subproblems that are *substantially smaller*, say half the size. For instance, `mergesort` sorts an array of size $n$ by recursively sorting two subarrays of size $n/2$. Because of this sharp drop in problem size, the full recursion tree has only logarithmic depth and a polynomial number of nodes.

In contrast, in a typical dynamic programming formulation, a problem is reduced to subproblems that are only slightly smaller—for instance, $L(j)$ relies on $L(j-1)$. Thus the full recursion tree generally has polynomial depth and an exponential number of nodes. However, it turns out that most of these nodes are repeats, that there are not too many *distinct* subproblems among them. Efficiency is therefore obtained by explicitly enumerating the distinct subproblems and solving them in the right order.

**Programming?**

The origin of the term *dynamic programming* has very little to do with writing code. It was first coined by Richard Bellman in the 1950s, a time when computer programming was an esoteric activity practiced by so few people as to not even merit a name. Back then programming meant "planning," and "dynamic programming" was conceived to optimally plan multistage processes. The dag of Figure 6.2 can be thought of as describing the possible ways in which such a process can evolve: each node denotes a state, the leftmost node is the starting point, and the edges leaving a state represent possible actions, leading to different states in the next unit of time.

The etymology of *linear programming*, the subject of Chapter 7, is similar.

## 6.3 Edit distance

When a spell checker encounters a possible misspelling, it looks in its dictionary for other words that are close by. What is the appropriate notion of closeness in this case?

A natural measure of the distance between two strings is the extent to which they can be *aligned*, or matched up. Technically, an alignment is simply a way of writing the strings one above the other. For instance, here are two possible alignments of SNOWY and SUNNY:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S | – | N | O | W | Y | | – | S | N | O | W | – | Y |
| S | U | N | N | – | Y | | S | U | N | – | – | N | Y |
| Cost: 3 | | | | | | | Cost: 5 | | | | | | |

The "–" indicates a "gap"; any number of these can be placed in either string. The *cost* of an alignment is the number of columns in which the letters differ. And the *edit distance* between two strings is the cost of their best possible alignment. Do you see that there is no better alignment of SNOWY and SUNNY than the one shown here with a cost of 3?

Edit distance is so named because it can also be thought of as the minimum number of *edits*—insertions, deletions, and substitutions of characters—needed to transform the first string into the second. For instance, the alignment shown on the left corresponds to three edits: insert U, substitute O $\rightarrow$ N, and delete W.

In general, there are so many possible alignments between two strings that it would be terribly inefficient to search through all of them for the best one. Instead we turn to dynamic programming.

### A dynamic programming solution

When solving a problem by dynamic programming, the most crucial question is, *What are the subproblems?* As long as they are chosen so as to have the property (*) from page 163. it is an easy matter to write down the algorithm: iteratively solve one subproblem after the other, in order of increasing size.

Our goal is to find the edit distance between two strings $x[1 \cdots m]$ and $y[1 \cdots n]$. What is a good subproblem? Well, it should go part of the way toward solving the whole problem; so how about looking at the edit distance between some *prefix* of the first string, $x[1 \cdots i]$, and some *prefix* of the second, $y[1 \cdots j]$? Call this subproblem $E(i, j)$ (see Figure 6.3). Our final objective, then, is to compute $E(m, n)$.