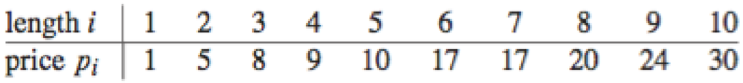
Dynamic programming is a problem solving method that is applicable to many different types of problems. I think it is best learned by example, so we will mostly do examples today.

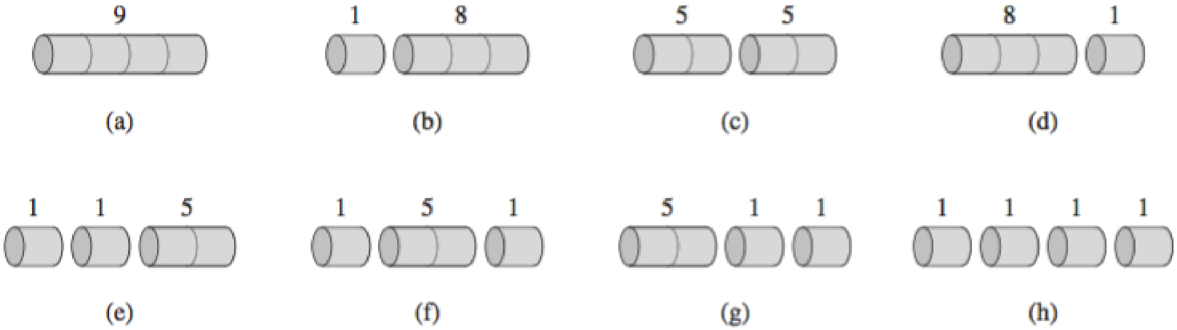
1 Rod cutting

Suppose you have a rod of length n, and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth p_i dollars.



length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

For example, if you have a rod of length 4, there are eight different ways to cut it, and the best strategy is cutting it into two pieces of length 2, which gives you 10 dollars.

Exercise: How many ways are there to cut up a rod of length n?

Answer: 2^{n-1}, because there are $n-1$ places where we can choose to make cuts, and at each place, we either make a cut or we do not make a cut.

Despite the exponentially large possibility space, we can use dynamic programming to write an algorithm that runs in $\Theta(n^2)$.

1.1 Basic approach

First we ask "what is the maximum amount of money we can get?" And later we can extend the algorithm to give us the actual rod decomposition that leads to that maximum value.

Let r_i be the maximum amount of money you can get with a rod of size i. We can view the problem recursively as follows:

- First, cut a piece off the left end of the rod, and sell it.
- Then, find the optimal way to cut the remainder of the rod.

Now we don't know how large a piece we should cut off. So we try all possible cases. First we try cutting a piece of length 1, and combining it with the optimal way to cut a rod of length $n-1$. Then we try cutting a piece of length 2, and combining it with the optimal way to cut a rod of length $n-2$. We try all the possible lengths and then pick the best one.

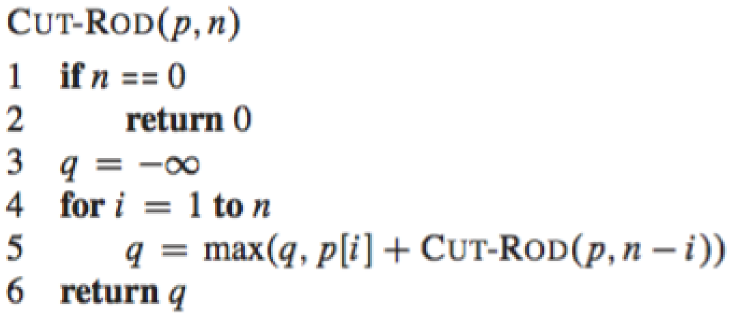
We end up with

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

(Note that by allowing i to be n, we handle the case where the rod is not cut at all.)

1.1.1 Naive algorithm

This formula immediately translates into a recursive algorithm.



```
CUT-ROD(p, n)
if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ to $n$
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return $q$
```

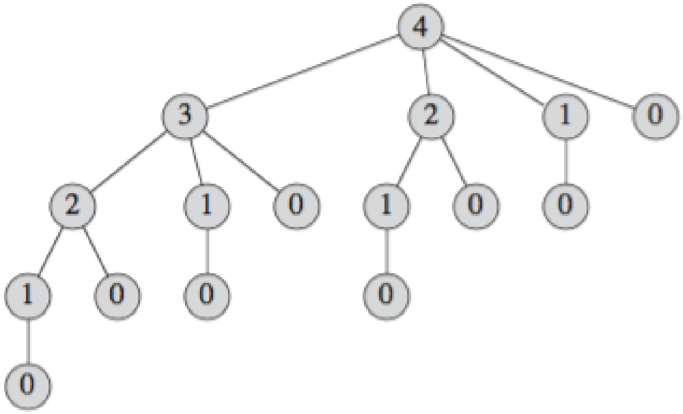


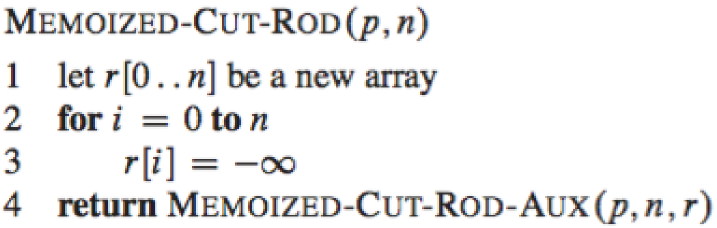
Figure 15.3 The recursion tree showing recursive calls resulting from a call CUT-ROD(p, n) for $n = 4$. Each node label gives the size n of the corresponding subproblem, so that an edge from a parent with label s to a child with label t corresponds to cutting off an initial piece of size $s - t$ and leaving a remaining subproblem of size t. A path from the root to a leaf corresponds to one of the 2^{n-1} ways of cutting up a rod of length n. In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

However, the computation time is ridiculous, because there are so many subproblems. If you draw the recursion tree, you will see that we are actually doing a lot of extra work, because we are computing the same things over and over again. For example, in the computation for $n = 4$, we compute the optimal solution for $n = 1$ four times!

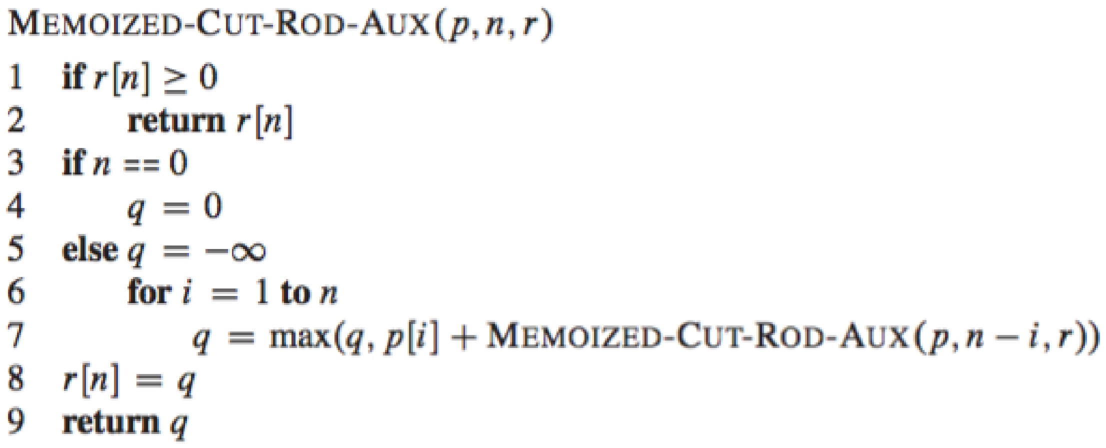
It is much better to compute it once, and then refer to it in future recursive calls.

1.1.2 Memoization (top down approach)

One way we can do this is by writing the recursion as normal, but store the result of the recursive calls, and if we need the result in a future recursive call, we can use the precomputed value. The answer will be stored in `r[n]`.



```
MEMOIZED-CUT-ROD$(p, n)$
let $r[0..n]$ be a new array
for $i = 0$ to $n$
    $r[i] = -\infty$
return MEMOIZED-CUT-ROD-AUX$(p, n, r)$
```

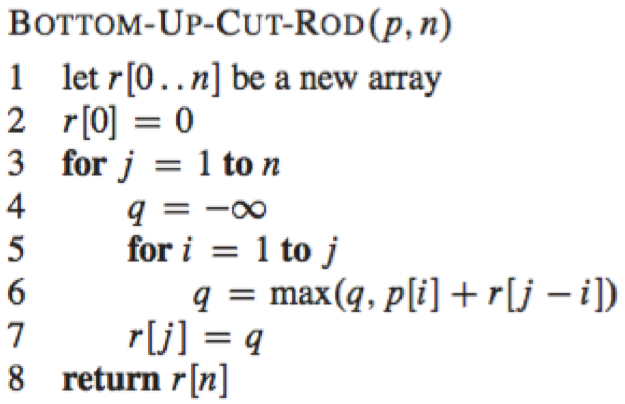


```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to $n$
        $q = \max(q, p[i] + $MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
$r[n] = q$
return $q$
```

Runtime: $\Theta(n^2)$. Each subproblem is solved exactly once, and to solve a subproblem of size i, we run through i iterations of the for loop. So the total number of iterations of the for loop, over all recursive calls, forms an arithmetic series, which produces $\Theta(n^2)$ iterations in total.

1.1.3 Bottom up approach

Here we proactively compute the solutions for smaller rods first, knowing that they will later be used to compute the solutions for larger rods. The answer will once again be stored in `r[n]`.



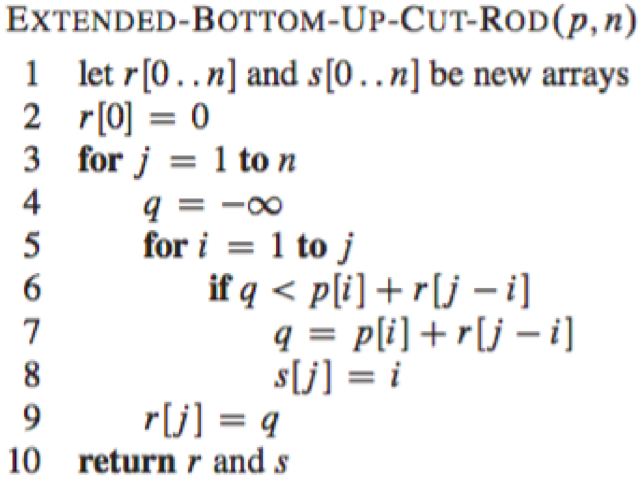
```
BOTTOM-UP-CUT-ROD$(p, n)$
let $r[0..n]$ be a new array
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        $q = \max(q, p[i] + r[j - i])$
    $r[j] = q$
return $r[n]$
```

Often the bottom up approach is simpler to write, and has less overhead, because you don't have to keep a recursive call stack. Most people will write the bottom up procedure when they implement a dynamic programming algorithm.

Runtime: $\Theta(n^2)$, because of the double for loop.

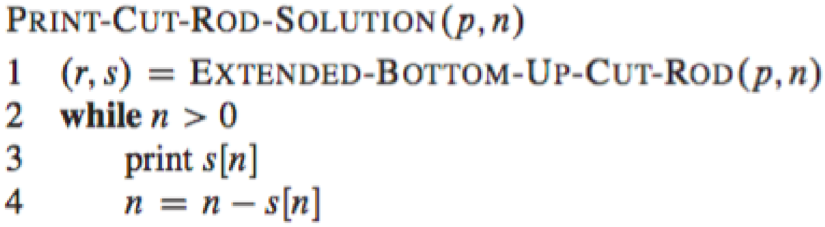
1.1.4 Reconstructing a solution

If we want to actually find the optimal way to split the rod, instead of just finding the maximum profit we can get, we can create another array s, and let $s[j] = i$ if we determine that the best thing to do when we have a rod of length j is to cut off a piece of length i.



```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let $r[0..n]$ and $s[0..n]$ be new arrays
$r[0] = 0$
for $j = 1$ to $n$
    $q = -\infty$
    for $i = 1$ to $j$
        if $q < p[i] + r[j - i]$
            $q = p[i] + r[j - i]$
            $s[j] = i$
    $r[j] = q$
return $r$ and $s$
```

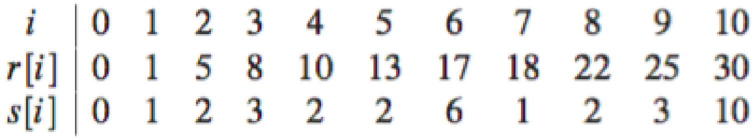
Using these values $s[j]$, we can reconstruct a rod decomposition as follows:



```
PRINT-CUT-ROD-SOLUTION(p, n)
$(r, s) =$ EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while $n > 0$
    print $s[n]$
    $n = n - s[n]$
```

1.1.5 Answer to example problem

In our example, the program produces this answer:



i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

2 General dynamic programming remarks

2.0.1 Optimal substructure

To solve a optimization problem using dynamic programming, we must first characterize the structure of an optimal solution. Specifically, we must prove that we can create an optimal solution to a problem using optimal solutions to subproblems. (Then we can store all the optimal solutions in an array and compute later elements in the array from earlier elements in the array.)

We can't really use dynamic programming if the optimal solution to a problem might not require subproblem solutions to be optimal. This often happens when the subproblems are not independent of each other.

2.0.2 Overlapping subproblems

For dynamic programming to be useful, the recursive algorithm should require us to compute optimal solutions to the same subproblems over and over again, because in this case, we benefit from just computing them once and then using the results later.

In total, there should be a small number of distinct subproblems (i.e. polynomial in the input size), even if there is an exponential number of total subproblems.

Note that in the divide-and-conquer algorithms we saw earlier in the class, the number of subproblems gets exponentially larger on each level of the recursion tree, but the size of the subproblems gets exponentially smaller. In these dynamic programming algorithms, the number of distinct subproblems should be polynomial, but the size of the subproblems might decrease by 1 every time.

3 Longest common subsequence

Suppose we have a sequence of letters `ACCGGTC`. Then a subsequence of this sequence would be like `ACCG` or `ACTC` or `CCC`. To get `ACCG`, we pick the first four letters. To get `ACTC`, we pick letters 1, 2, 6, and 7. To get `CCC`, we pick letters 2, 3, and 7, etc.

Formally, given a sequence $X = x_1, x_2, \ldots, x_m$, another sequence $Z = z_1, \ldots, z_k$ is a subsequence of X if there exists a **strictly increasing** sequence $i_1, i_2, \ldots, i_k$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.

In the **longest-common-subsequence** problem, we are given two sequences X and Y, and want to find the longest possible sequence that is a subsequence of both X and Y.

For example, if $X =$ `ABCBDAB` and $Y =$ `BDCABA`, the sequence `BCA` is a common sequence of both X and Y. However, it is not a longest common subsequence, because `BCBA` is a longer sequence that is also common to both X and Y. Both `BCBA` and `BDAB` are longest common subsequences, since there are no common sequences of length 5 or greater.

3.0.1 Optimal substructure

The first step to solving this using dynamic programming is to say that we can create an optimal solution to this problem using optimal solutions to subproblems. The hardest part is to decide what the subproblems are.

Here there are two possible cases:

1. The last elements of X and Y are equal. Then they must both be part of the longest common subsequence. So we can chop both elements off the ends of the subsequence (adding them to a common subsequence) and find the longest common subsequence of the smaller sequences.
2. The last elements are different. Then either the last element of X or the last element of Y cannot be part of the longest common subsequence. So now we can find the longest common subsequence of X and a smaller version of Y, or the longest common subsequence of Y and a smaller version of X.

Formally, let $X = x_1, \dots, x_m$ and $Y = y_1, \dots, y_n$ be sequences, and let $Z = z_1, \dots, z_k$ be any longest common subsequence of X and Y. Let X_i refer to the first i elements of X, and Y_i refer to the first i elements of Y, etc.

Then

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is a longest common subsequence of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is a longest common subsequence of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is a longest common subsequence of X and Y_{n-1}.

Using this theorem, we show that the longest common subsequence problem can always be solved by finding the longest common subsequence of smaller problems, and then combining the solutions.

Proof:

1. If $z_k \neq x_m$, then we can append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, which contradicts the fact that Z is the longest common subsequence. Now Z_{k-1} is a common subsequence of X_{m-1} and Y_{n-1} of length $k - 1$. It must be a longest common subsequence, because if W was a common subsequence of X_{m-1} and Y_{m-1} with length greater than $k - 1$, then appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.
2. If $z_k \neq x_m$, then Z is a longest common subsequence of X_{m-1} and Y. This is because if there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, so Z would not be a longest common subsequence. Contradiction.
3. Same as the proof for part (2).

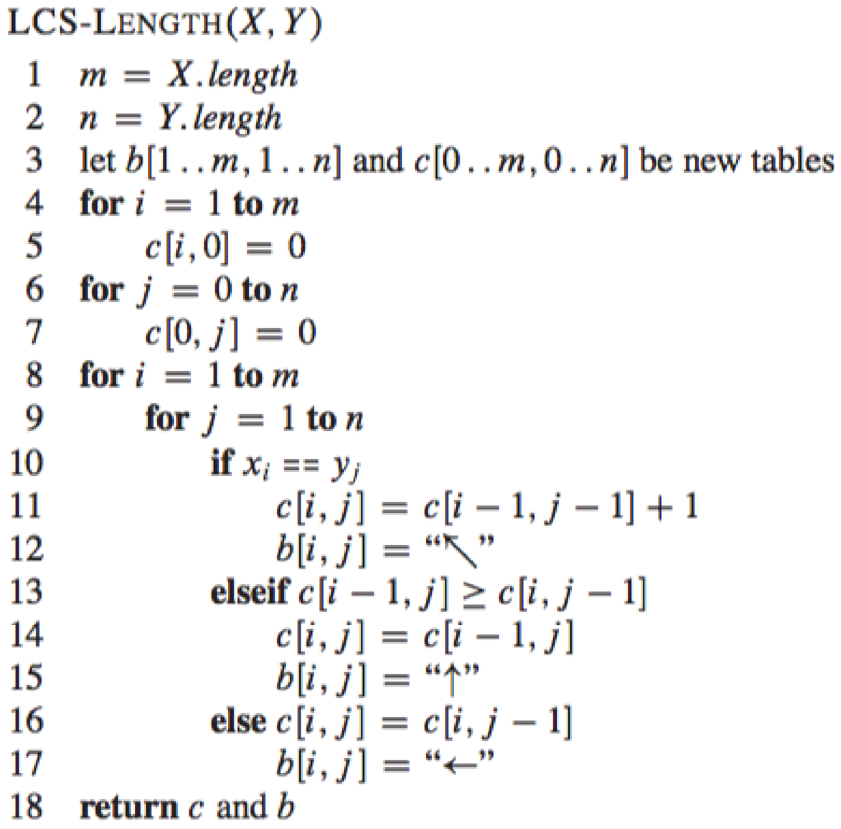
3.0.2 A recursive solution

To store the solutions to subproblems, this time we use a 2D matrix, instead of a 1D array. We want to compute all the values `c[i, j]`, which are the lengths of the longest common subsequences between the first i elements of X and the first j elements of Y. At the end, the answer will be stored in `c[m, n]`.

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j\,. \end{cases} \tag{15.9}$$

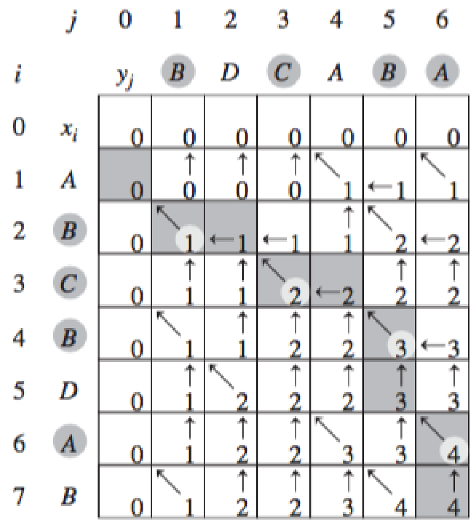
3.0.3 Dynamic programming algorithm

Using this recurrence, we can write the actual pseudocode. Observe that it is necessary to populate the table in a certain order, because some elements of the table depend on other elements of the table having already been computed.



```
LCS-LENGTH($X, Y$)
m = X.length
n = Y.length
let $b[1..m, 1..n]$ and $c[0..m, 0..n]$ be new tables
for $i = 1$ to $m$
    $c[i, 0] = 0$
for $j = 0$ to $n$
    $c[0, j] = 0$
for $i = 1$ to $m$
    for $j = 1$ to $n$
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j]$ = "↖"
        elseif $c[i-1, j] \geq c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j]$ = "↑"
        else $c[i, j] = c[i, j-1]$
            $b[i, j]$ = "←"
return $c$ and $b$
```

The algorithm fills in the elements of the table as follows:

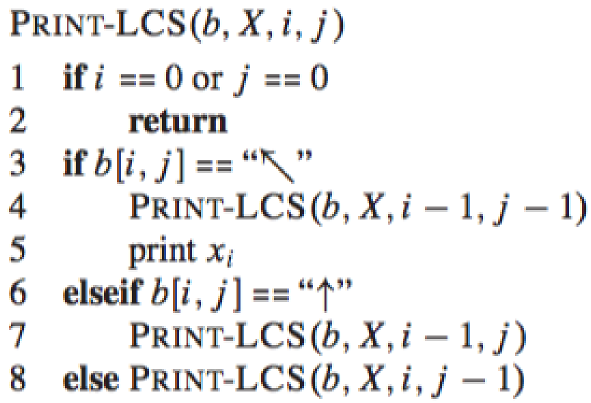


	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the sequence is shaded. Each "↖" on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

3.0.4 Reconstructing the solution

To reconstruct the solution, we just print the elements that we marked as part of the longest common subsequence.



```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == "↖"
    PRINT-LCS(b, X, i − 1, j − 1)
    print x_i
elseif b[i, j] == "↑"
    PRINT-LCS(b, X, i − 1, j)
else PRINT-LCS(b, X, i, j − 1)
```

4 Correctness proof for Dijkstra's algorithm (if there is time)

While breadth-first-search computes shortest paths in an unweighted graph, Dijkstra's algorithm is a way of computing shortest paths in a weighted graph. Specifically Dijkstra's computes the shortest paths from a source node s to every other node in the graph.

The idea is that we keep "distance estimates" for every node in the graph (which are always greater than the true distance from the start node). On each iteration of the algorithm we process the (unprocessed) vertex with the smallest distance estimate. (We can prove that by the time we get around to processing a vertex, its distance estimate reflects the true distance to that vertex. This is nontrivial and must be proven.)

Whenever we process a vertex, we update the distance estimates of its neighbors, to account for the possibility that we may be reaching those neighbors through that vertex. Specifically, if we are processing u, and there is an edge from $u \to v$ with weight w, we change v's distance estimate $v.d$ to be the minimum of its current value and $u.d+w$. (It's possible that $v.d$ doesn't change at all, for example, if the shortest path from s to v was through a different vertex.)

If we did lower the estimate $v.d$, we set v's parent to be u, to signify that (we think) the best way to reach v is through u. The parent may change multiple times through the course of the algorithm, and at the end, the parent-child relations form a shortest path tree, where the path (along tree edges) from s to any node in the tree is a shortest path to that node. Note that the shortest path tree is very much like the breadth first search tree.

Important: Dijkstra's algorithm does not handle graphs with negative edge weights! For that you would need to use a different algorithm, such as Bellman-Ford.

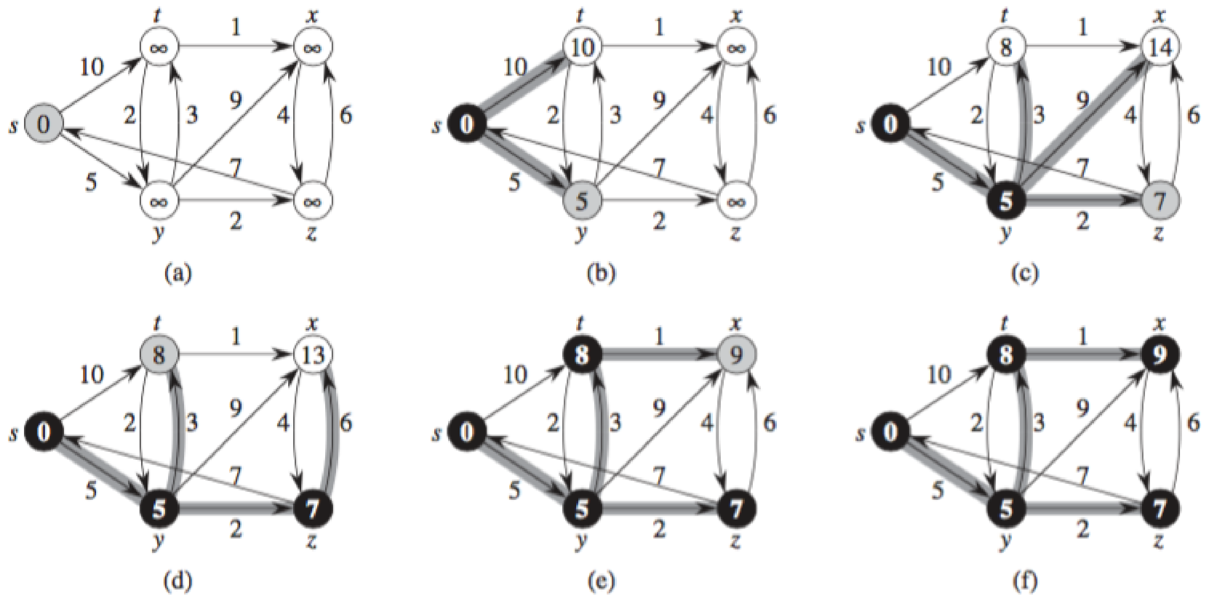
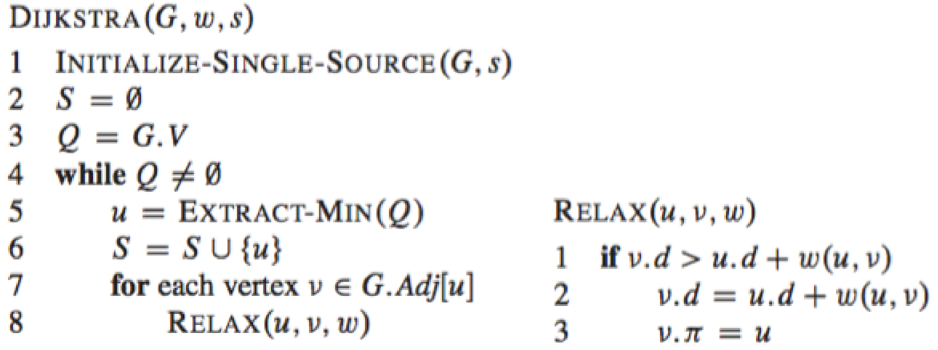


Figure 24.6 The execution of Dijkstra's algorithm. The source s is the leftmost vertex. The shortest-path estimates appear within the vertices, and shaded edges indicate predecessor values. Black vertices are in the set S, and white vertices are in the min-priority queue $Q = V - S$. **(a)** The situation just before the first iteration of the **while** loop of lines 4–8. The shaded vertex has the minimum d value and is chosen as vertex u in line 5. **(b)–(f)** The situation after each successive iteration of the **while** loop. The shaded vertex in each part is chosen as vertex u in line 5 of the next iteration. The d values and predecessors shown in part (f) are the final values.



```
DIJKSTRA($G, w, s$)
1  INITIALIZE-SINGLE-SOURCE($G, s$)
2  $S = \emptyset$
3  $Q = G.V$
4  while $Q \neq \emptyset$
5      $u$ = EXTRACT-MIN($Q$)
6      $S = S \cup \{u\}$
7      for each vertex $v \in G.Adj[u]$
8          RELAX($u, v, w$)
```

```
RELAX($u, v, w$)
1  if $v.d > u.d + w(u, v)$
2      $v.d = u.d + w(u, v)$
3      $v.\pi = u$
```

4.1 Correctness

Let s be the start node/source node, $v.d$ be the "distance estimate" of a vertex v, and $\delta(u, v)$ be the true distance from u to v. We want to prove two statements:

1. At any point in time, $v.d \geq \delta(s, v)$.
2. When v is extracted from the queue, $v.d = \delta(s, v)$. (Distance estimates never increase, so once $v.d = \delta(s, v)$, it stays that way.)

4.1.1 $v.d \geq \delta(s, v)$

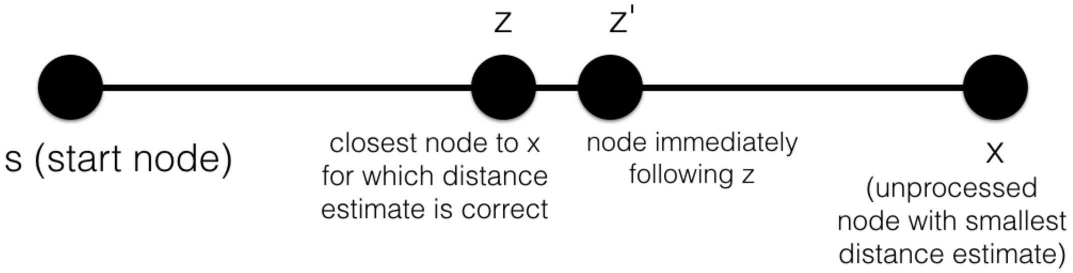
We want to show that at any point in time, if $v.d < \infty$ then $v.d$ is the weight of some path from s to v (not necessarily the shortest path). Since $\delta(s, v)$ is the weight of the shortest path from s to v, the conclusion will follow immediately.

We induct on the number of Relax operations. As our base case, we know that $s.d = 0 = \delta(s, s)$, and all other distance estimates are ∞, which is greater than or equal to their true distance.

As our inductive step, assume that at some point in time, every distance estimate corresponds to the weight of some path from s to that vertex. Now when a Relax operation is performed, the distance estimate of some neighbor $u.d$ may be changed to $x.d + w(x, u)$, for some vertex x. We know $x.d$ is the weight of some path from s to x. If we add the edge (x, u) at the end of that path, the weight of the resulting path is $x.d + w(x, u)$, which is $u.d$.

Alternatively, the distance estimate $u.d$ may not change at all during the Relax step. In that case we already know (from the inductive hypothesis) that $u.d$ is the weight of some path from s to u, so the inductive step is still satisfied.

4.1.2 $v.d = \delta(s, v)$ when v is extracted from the queue


z
z'
s (start node)
closest node to x for which distance estimate is correct
node immediately following z
x
(unprocessed node with smallest distance estimate)

We induct on the order in which we add nodes to S. For the base case, s is added to S when $s.d = \delta(s, s) = 0$, so the claim holds.

For the inductive step, assume that the claim holds for all nodes that are currently in S, and let x be the node in Q that currently has the minimum distance estimate. (This is the node that is about to be extracted from the queue.) We will show that $x.d = \delta(s, x)$.

Suppose p is a shortest path from s to x. Suppose z is the node on p closest to x for which $z.d = \delta(s, z)$. (We know z exists because there is at least one such node, namely s, where $s.d = \delta(s, s)$.) This means for every node y on the path p between z (not inclusive) and x (inclusive), we have $y.d > \delta(s, y)$.

If $z = x$, then $x.d = \delta(s, x)$, so we are done.

So suppose $z \neq x$. Then there is a node z' after z on p (which might equal x). We argue that $z.d = \delta(s, z) \leq \delta(s, x) \leq x.d$.

- $\delta(s, x) \leq x.d$ from the previous lemma, and $z.d = \delta(s, z)$ by assumption.
- $\delta(s, z) \leq \delta(s, x)$ because subpaths of shortest paths are also shortest paths. That is, if $s \to \cdots \to z \to \cdots \to x$ is a shortest path to x, then the subpath $s \to \cdots \to z$ is a shortest path to z. This is because, suppose there were an alternate path from s to z that was shorter. Then we could "glue that path into" the path from s to x, producing a shorter path and contradicting the fact that the s-to-x path was a shortest path.

Now we want to collapse these inequalities into equalities, by proving that $z.d = x.d$. Assume (by way of contradiction) that $z.d < x.d$. Because $x.d$ has the minimum distance estimate out of all the unprocessed vertices, it follows that z has already been added to S. This means that all of the edges coming out of z have already been relaxed by our algorithm, which means that $z'.d \leq \delta(s, z) + w(z, z') = \delta(s, z')$. (This last equality holds because z precedes z' on the shortest path to x, so z is on the shortest path to z'.)

However, this contradicts the assumption that z is the closest node on the path to x with a correct distance estimate. Thus, $z.d = x.d$.