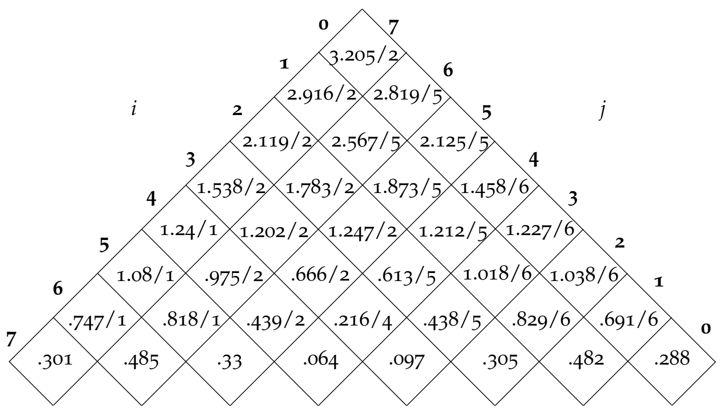
result. Any one of the given recursive formulas will do. The important thing is to store the subproblems' total probabilities in a table so that total probabilities for later subproblems can be computed from those for earlier subproblems instead of being computed from scratch.

Exercises

6.46 State a claim of optimal substructure for this problem formally and prove it. Use the proof of Lemma 2 and your answer to Exercise 6.25 as examples of how to structure the proof.

Building the tables for the optimal BST problem is similar to the algorithm for optimal matrix multiplication. Since the base cases correspond to the major diagonal of the table and the cells only in the "upper left triangle" of the table correspond to relevant subproblems, we visualize this table as a pyramid. The top-level problem is at the pinnacle, position $C[0][n-1]$. Here we show the combined tables for the best total weighted depths and the keys for the best roots using our sample instance. Our algorithm requires a third table for the total probabilities, which, for simplicity, we don't show.



To populate these tables, we iterate over the diagonals from the major diagonal at the base of the pyramid to the pinnacle cell at position $(0, n-1)$. As in the table-populating algorithm for optimal matrix multiplication, we use a variable d to identify a diagonal by the difference between i and j for the cells in that diagonal. For each cell, we evaluate options for $r \in (i, j)$ plus special cases for $r = i$ and $r = j$.

```
def opt_bst(key_probs, miss_probs):
    assert len(key_probs) + 1 == len(miss_probs)
    n = len(key_probs)

    # total_probs[i][j] is the sum of all the key probabilities in range [i,j]
    # and all the miss probabilities in range [i,j+1]
    total_probs = [[None for j in range(n)] for i in range(n)]

    # total_weighted_depths[i][j] indicates the minimum total weighted depth
    # for any try for keys in range [i,j]
    total_weighted_depths = [[None for j in range(n)] for i in range(n)]

    # decisions[i][j] indicates the key at the root of the best tree for range [i,j]
    decisions = [[None for j in range(n)] for i in range(n)]

    # Base cases for trees with only one key
    for i in range(n):
        total_probs[i][i] = miss_probs[i] + key_probs[i] + miss_probs[i+1]
        total_weighted_depths[i][i] = (2 * miss_probs[i] + key_probs[i] +
                                       2 * miss_probs[i+1])
        decisions[i][i] = i

    # For each diagonal, identified by the difference between indices
    for d in range(1, n) :
        # For each cell or subproblem in that diagonal, i, j
        for i in range(n - d):
            j = i + d
            # The total probability for range [i, j]
            total_probs[i][j] = miss_probs[i] + key_probs[i] + total_probs[i+1][j]
            # The cost of making key i the root, which is our initial
            # best-so-far
            least_subtree_cost = miss_probs[i] + total_weighted_depths[i+1][j]
            best_root = i
            # For each candiate root r between i and j exclusive
            for r in range(i+1,j):
                # The cost of making key r the root
                current_subtree_cost = (total_weighted_depths[i][r-1] +
                                        total_weighted_depths[r+1][j])
                # If its cost is better than best so far, it's the new best so far
                if current_subtree_cost < least_subtree_cost :
                    least_subtree_cost = current_subtree_cost
                    best_root = r

            # The cost of making key j the root
            current_subtree_cost = total_weighted_depths[i][j-1] + miss_probs[j+1]
            # If its cost is better than best-so-far, it's the new best-so-far
            if current_subtree_cost < least_subtree_cost :
                least_subtree_cost = current_subtree_cost
                best_root = j
            # Record the best option and corresponding cost in the tables
            total_weighted_depths[i][j] = total_probs[i][j] + least_subtree_cost
            decisions[i][j] = best_root
```

From its similarity to the algorithm for optimal matrix multiplication, we recognize the running time for building the tables as $\Theta(n^3)$. See Exercise 6.47 for details.

The value $C[0][n-1]$ in `total_weighted_depths[0][n-1]` gives us the cost of the best tree for the given keys with their probabilities. As with other dynamic programming problems, a more useful result is the tree itself. Exercise 6.48 asks you to write a function that reconstructs the optimal binary search tree using a populated decision table, but for Project 6.2 we have an alternate strategy. Instead of reconstructing the tree after building the table, we build the actual optimal subtrees along with the table. Instead of a table of decisions as in the algorithm above, we maintain a table such that in position (i, j) we store the root of the best subtree for keys k_i through k_j.

Suppose r is the index of the key that is the best root for a subtree containing keys k_i through k_j. Instead of storing r in the decision table at position (i, j), we allocate a new node for the binary tree with k_r as the key. Its children are the roots of the best subtrees for keys k_i through k_{r-1} and keys

k_{r+1} through k_j, but those nodes have already been allocated and stored in the table of nodes at positions $(i, r-1)$ and $(r+1, j)$. We store this newly allocated node in position (i, j). When the table-building algorithm terminates, we return the node in position $(0, n-1)$ of the node table as the root of the entire tree. No reconstruction is necessary because the tree is already built.

The last thing to consider about this algorithm is the sense in which the tree produced is *optimal*. Remember that the cost model is the expected number of nodes to be visited in a lookup. To optimize the expected case under that model, we sacrifice optimality for the *worst case*. More to the point, though, is how well this cost model predicts real performance.

As we observed at the end of Section 3.3 and again in Section 5.6, when the binary tree abstraction is implemented with linked memory, it has the inherent weakness of poor data locality. Optimizing the number of nodes that need to be visited on average has meager pay-off in real performance if each node is allocated in a different page of virtual memory. Optimizing a data structure to reduce page faults, as is done with a B-tree, is likely to have a bigger, positive effect on performance than optimizing the tree structure itself. The optimal binary search tree algorithm is a beautiful solution to a specific problem, but before deploying it, one must ask whether it solves the right problem.

Exercises

6.47 Demonstrate the $\Theta(n^3)$ complexity for `opt_bst()` by counting the number of iterations of the innermost loop, which is also the sum of all the options to be considered for all the subproblems. Notice that for the major diagonal there are n cells with 1 option each (although that corresponds to the loop of base cases before the big loop), the diagonal above that has $n-1$ cells with two options each, and the pinnacle has 1 cell with n options. Thus the total number of options considered for all subproblems is $\sum_{i=1}^{n} i(n-i+1)$. Solve this summation.

6.48 Write a function `build_tree()` that takes the list of keys, the decisions table, and indices i and j and returns an optimal tree for keys k_i to k_j by recursively descending into the `decisions` table.

☙ ☙ ☙

Project 6.2: Optimal binary search trees ✄

The goal of this project is to practice dynamic programming by implementing the algorithm for building optimal binary search trees.

The code base for this project has packages `adt`, `impl`, and `test`, as usual, but also package `optbstutil`. The class `optbstutil.OptBSTUtil` has methods for reading data from disk and computing weighted depths. The class `impl.OptimalBSTData` represents a problem instance, encapsulating the keys, values, key probabilities, and miss probabilities.

The class `impl.OptimalBSTMap` is similar to the BST map classes in Chapter 5 but simpler since it does not support the `remove()` method, and the `put()` method is supported to only overwrite values for keys already in the map—and, hence, there is no code for tree rotations. Moreover, the class has no code for building the tree itself. Instead, its constructor takes a node which is the root of an externally-built tree. The node types are nested in `impl.OptimalBSTMap`, but they are `protected`, not `private`, so they are visible to other classes in the same package.

Your task is to write the algorithm for building the tree by finishing the method `buildOptimalBST()` in class `impl.OptimalBSTMapFactory`. The method takes the keys, values, key probabilities, and miss probabilities and returns an `OptimalBSTMap` for that problem instance. Your solution should allocate and populate tables for total weighted depths, probabilities, and roots of optimal subtrees. The last thing the method does is pass the node at the pinnacle of the tree table to the `OptimalBSTMap` constructor.

Use `test.OBSTTest` to test.

☙ ☙ ☙