with left endpoint $s$ containing elements from each array must be $[s, t]$, since the remaining two values are the minimum possible.

Now remove $s$ from the triple and bring the next smallest element from the array it belongs to into the triple. Let $s'$ and $t'$ be the next minimum and maximum values in the new triple. Observe $[s', t']$ must be the smallest interval whose left endpoint is $s'$: the other two values are the smallest values in the corresponding arrays that are greater than or equal to $s'$. By iteratively examining and removing the smallest element from the triple, we compute the minimum interval starting at that element. Since the minimum interval containing elements from each array must begin with the element of *some* array, we are guaranteed to encounter the minimum element.

For example, we begin with $(5, 3, 8)$. The smallest interval whose left endpoint is 3 has length $8 - 3 = 5$. The element after 3 is 6, so we continue with the triple $(5, 6, 8)$. The smallest interval whose left endpoint is 5 has length $8 - 5 = 3$. The element after 5 is 10, so we continue with the triple $(10, 6, 8)$. The smallest interval whose left endpoint is 6 has length $10 - 6 = 4$. The element after 6 is 9, so we continue with the triple $(10, 9, 8)$. Proceeding in this way, we obtain the triples $(10, 9, 16)$, $(10, 12, 16)$, $(15, 12, 16)$, $(15, 15, 16)$. Out of all these triples, the one contained in a minimum length interval is $(15, 15, 16)$.

In the following code, we implement a general purpose function which finds the closest entries in $k$ sorted arrays. Since we need to repeatedly insert, delete, find the minimum, and find the maximum amongst a collection of $k$ elements, a BST is the natural choice.

```
def find_closest_elements_in_sorted_arrays(sorted_arrays):
    min_distance_so_far = float('inf')
    # Stores array iterators in each entry.
    iters = bintrees.RBTree()
    for idx, sorted_array in enumerate(sorted_arrays):
        it = iter(sorted_array)
        first_min = next(it, None)
        if first_min is not None:
            iters.insert((first_min, idx), it)

    while True:
        min_value, min_idx = iters.min_key()
        max_value = iters.max_key()[0]
        min_distance_so_far = min(max_value - min_value, min_distance_so_far)
        it = iters.pop_min()[1]
        next_min = next(it, None)
        # Return if some array has no remaining elements.
        if next_min is None:
            return min_distance_so_far
        iters.insert((next_min, min_idx), it)
```

The time complexity is $O(n \log k)$, where $n$ is the total number of elements in the $k$ arrays. For the special case $k = 3$ specified in the problem statement, the time complexity is $O(n \log 3) = O(n)$.

### 14.7 Enumerate numbers of the form $a + b\sqrt{2}$

Numbers of the form $a + b\sqrt{q}$, where $a$ and $b$ are nonnegative integers, and $q$ is an integer which is not the square of another integer, have special properties, e.g., they are closed under addition and multiplication. Some of the first few numbers of this form are given in Figure 14.4 on the following page.

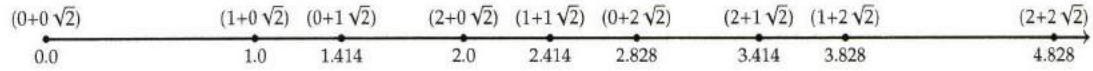



**Figure 14.4:** Some points of the form $a + b\sqrt{2}$. (For typographical reasons, this figure does not include all numbers of the form $a + b\sqrt{2}$ between 0 and $2 + 2\sqrt{2}$, e.g., $3 + 0\sqrt{2}, 4 + 0\sqrt{2}, 0 + 3\sqrt{2}, 3 + 1\sqrt{2}$ lie in the interval but are not included.)

Design an algorithm for efficiently computing the $k$ smallest numbers of the form $a + b\sqrt{2}$ for nonnegative integers $a$ and $b$.

*Hint:* Systematically enumerate points.

**Solution:** A key fact about $\sqrt{2}$ is that it is irrational, i.e., it cannot equal to $\frac{a}{b}$ for any integers $a, b$. This implies that if $x + y\sqrt{2} = x' + y'\sqrt{2}$, where $x$ and $y$ are integers, then $x = x'$ and $y = y'$ (since otherwise $\sqrt{2} = \frac{x-x'}{y-y'}$).

Here is a brute-force solution. Generate all numbers of the form $a + b\sqrt{2}$ where $a$ and $b$ are integers, $0 \le a, b \le k - 1$. This yields exactly $k^2$ numbers and the $k$ smallest numbers must lie in this collection. We can sort these numbers and return the $k$ smallest ones. The time complexity is $O(k^2 \log(k^2)) = O(k^2 \log k)$.

Intuitively, it is wasteful to generate $k^2$ numbers, since we only care about a small fraction of them.

We know the smallest number is $0 + 0\sqrt{2}$. The candidates for next smallest number are $1 + 0\sqrt{2}$ and $0 + 1\sqrt{2}$. From this, we can deduce the following algorithm. We want to maintain a collection of real numbers, initialized to $0 + 0\sqrt{2}$. We perform $k$ extractions of the smallest element, call it $a + b\sqrt{2}$, followed by insertion of $(a + 1) + b\sqrt{2}$ and $a + (b + 1)\sqrt{2}$ to the collection.

The operations on this collection are extract the minimum and insert. Since it is possible that the same number may be inserted more than once, we need to ensure the collection does not create duplicates when the same item is inserted twice. A BST satisfies these operations efficiently, and is used in the implementation below. It is initialized to contain $0 + 0\sqrt{2}$. We extract the minimum from the BST, which is $0 + 0\sqrt{2}$, and insert $1 + 0\sqrt{2}$ and $0 + 1\sqrt{2}$ to the BST. We extract the minimum from the BST, which is $1 + 0\sqrt{2}$, and insert $2 + 0\sqrt{2}$ and $1 + 1\sqrt{2}$ to the BST, which now consists of $0 + 1\sqrt{2} = 1.414, 2 + 0\sqrt{2} = 2, 1 + 1\sqrt{2} = 2.414$. We extract the minimum from the BST, which is $0 + 1\sqrt{2}$, and insert $1 + 1\sqrt{2}$ and $0 + 2\sqrt{2}$. The first value is already present, so the BST updates to $2 + 0\sqrt{2} = 2, 1 + 1\sqrt{2} = 2.414, 0 + 2\sqrt{2} = 2.828$. (Although it's not apparent from this small example, the values we add back to the BST may be smaller than values already present in it, so we really need the BST to hold values.)

```
class ABSqrt2:
    def __init__(self, a, b):
        self.a, self.b = a, b
        self.val = a + b * math.sqrt(2)

    def __lt__(self, other):
        return self.val < other.val

    def __eq__(self, other):
        return self.val == other.val
```

```
def generate_first_k_a_b_sqrt2(k):
    # Initial for 0 + 0 * sqrt(2).
    candidates = bintrees.RBTree([(ABSqrt2(0, 0), None)])

    result = []
    while len(result) < k:
        next_smallest = candidates.pop_min()[0]
        result.append(next_smallest.val)
        # Adds the next two numbers derived from next_smallest.
        candidates[ABSqrt2(next_smallest.a + 1, next_smallest.b)] = None
        candidates[ABSqrt2(next_smallest.a, next_smallest.b + 1)] = None
    return result
```

In each iteration we perform a deletion and two insertions. There are $k$ such insertions, so the time complexity is $O(k \log k)$. The space complexity is $O(k)$, since there are not more than $2k$ insertions.

Now we describe an $O(n)$ time solution. It is simple to implement, but is less easy to understand than the one based on BST. The idea is that the $(n+1)$th value will be the sum of 1 or $\sqrt{2}$ with a previous value. We could iterate through all the entries in the result and track the smallest such value which is greater than $n$th value. However, this takes time $O(n)$ to compute the $(n+1)$th element.

Intuitively, there is no need to examine all prior values entries when computing $(n+1)$th value. Let's say we are storing the result in an array $A$. Then we need to track just two entries—$i$, the smallest index such that $A[i] + 1 > A[n-1]$, and $j$, smallest index such that $A[j] + \sqrt{2} > A[n-1]$. Clearly, the $(n+1)$th entry will be the smaller of $A[i]+1$ and $A[j]+\sqrt{2}$. After obtaining the $(n+1)$th entry, if it is $A[i]+1$, we increment $i$. If it is $A[j]+\sqrt{2}$, we increment $j$. If $A[i]+1$ equals $A[j]+\sqrt{2}$, we increment both $i$ and $j$.

To illustrate, suppose $A$ is initialized to $\langle 0 \rangle$, and $i$ and $j$ are 0. The computation proceeds as follows:

(1.) Since $A[0] + 1 = 1 < A[0] + \sqrt{2} = 1.414$, we push 1 into $A$ and increment $i$. Now $A = \langle 0, 1 \rangle$, $i = 1, j = 0$.

(2.) Since $A[1] + 1 = 2 > A[0] + \sqrt{2} = 1.414$, we push 1.414 into $A$ and increment $j$. Now $A = \langle 0, 1, 1.414 \rangle$, $i = 1, j = 1$.

(3.) Since $A[1] + 1 = 2 < A[1] + \sqrt{2} = 2.414$, we push 2 into $A$ and increment $i$. Now $A = \langle 0, 1, 1.414, 2 \rangle$, $i = 2, j = 1$.

(4.) Since $A[2] + 1 = 2.414 = A[1] + \sqrt{2} = 2.414$, we push 2.414 into $A$ and increment both $i$ and $j$. Now $A = \langle 0, 1, 1.414, 2, 2.414 \rangle$, $i = 3, j = 2$.

(5.) Since $A[3] + 1 = 3 > A[2] + \sqrt{2} = 2.828$, we push 2.828 into $A$ and increment $j$. Now $A = \langle 0, 1, 1.414, 2, 2.828 \rangle$, $i = 3, j = 3$.

(6.) Since $A[3] + 1 = 3 < A[3] + \sqrt{2} = 3.414$, we push 3 into $A$ and increment $i$. Now $A = \langle 0, 1, 1.414, 2, 2.828, 3 \rangle$, $i = 4, j = 3$.

```
def generate_first_k_a_b_sqrt2(k):
    # Will store the first k numbers of the form a + b sqrt(2).
    cand = [ABSqrt2(0, 0)]
    i = j = 0
    for _ in range(1, k):
        cand_i_plus_1 = ABSqrt2(cand[i].a + 1, cand[i].b)
        cand_j_plus_sqrt2 = ABSqrt2(cand[j].a, cand[j].b + 1)
        cand.append(min(cand_i_plus_1, cand_j_plus_sqrt2))
        if cand_i_plus_1.val == cand[-1].val:
```

```
            i += 1
        if cand_j_plus_sqrt2.val == cand[-1].val:
            j += 1
    return [a.val for a in cand]
```

Each additional element takes $O(1)$ time to compute, implying an $O(n)$ time complexity to compute the first $n$ values of the form $a + b\sqrt{2}$.

### 14.8 Build a minimum height BST from a sorted array

Given a sorted array, the number of BSTs that can be built on the entries in the array grows enormously with its size. Some of these trees are skewed, and are closer to lists; others are more balanced. See Figure 14.3 on Page 205 for an example.

How would you build a BST of minimum possible height from a sorted array?

*Hint:* Which element should be the root?

**Solution:** Brute-force is not much help here—enumerating all possible BSTs for the given array in search of the minimum height one requires a nontrivial recursion, not to mention enormous time complexity.

Intuitively, to make a minimum height BST, we want the subtrees to be as balanced as possible—there's no point in one subtree being shorter than the other, since the height is determined by the taller one. More formally, balance can be achieved by keeping the number of nodes in both subtrees as close as possible.

Let $n$ be the length of the array. To achieve optimum balance we can make the element in the middle of the array, i.e., the $\lfloor \frac{n}{2} \rfloor$th entry, the root, and recursively compute minimum height BSTs for the subarrays on either side of this entry.

As a concrete example, if the array is $\langle 2, 3, 5, 7, 11, 13, 17, 19, 23 \rangle$, the root's key will be the middle element, i.e., 11. This implies the left subtree is to be built from $\langle 2, 3, 5, 7 \rangle$, and the right subtree is to be built from $\langle 13, 17, 19, 23 \rangle$. To make both of these minimum height, we call the procedure recursively.

```
def build_min_height_bst_from_sorted_array(A):
    def build_min_height_bst_from_sorted_subarray(start, end):
        if start >= end:
            return None
        mid = (start + end) // 2
        return BSTNode(A[mid],
                       build_min_height_bst_from_sorted_subarray(start, mid),
                       build_min_height_bst_from_sorted_subarray(mid + 1, end))

    return build_min_height_bst_from_sorted_subarray(0, len(A))
```

The time complexity $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + O(1)$, which solves to $T(n) = O(n)$. Another explanation for the time complexity is that we make exactly $n$ calls to the recursive function and spend $O(1)$ within each call.

Write a program which takes two nodes in a BST and a third node, the "middle" node, and determines if one of the two nodes is a proper ancestor and the other a proper descendant of the middle. (A proper ancestor of a node is an ancestor that is not equal to the node; a proper descendant is defined similarly.) For example, in Figure 14.1 on Page 198, if the middle is Node $J$, your function should return true if the two nodes are $\{A, K\}$ or $\{I, M\}$. It should return false if the two nodes are $\{I, P\}$ or $\{J, K\}$. You can assume that all keys are unique. Nodes do not have pointers to their parents

*Hint:* For what specific arrangements of the three nodes does the check pass?

**Solution:** A brute-force approach would be to check if the first node is a proper ancestor of the middle and the second node is a proper descendant of the middle. If this check returns true, we return true. Otherwise, we return the result of the same check, swapping the roles of the first and second nodes. For the BST in Figure 14.1 on Page 198, with the two nodes being $\{L, I\}$ and middle $K$, searching for $K$ from $L$ would be unsuccessful, but searching for $K$ from $I$ would succeed. We would then search for $L$ from $K$, which would succeed, so we would return true.

Searching has time complexity $O(h)$, where $h$ is the height of the tree, since we can use the BST property to prune one of the two children at each node. Since we perform a maximum of three searches, the total time complexity is $O(h)$.

One disadvantage of trying the two input nodes for being the middle's ancestor one-after-another is that even when the three nodes are very close, e.g., if the two nodes are $\{A, J\}$ and middle node is $I$ in Figure 14.1 on Page 198, if we begin the search for the middle from the lower of the two nodes, e.g., from $J$, we incur the full $O(h)$ time complexity.

We can prevent this by performing the searches for the middle from both alternatives in an interleaved fashion. If we encounter the middle from one node, we subsequently search for the second node from the middle. This way we avoid performing an unsuccessful search on a large subtree. For the example of $\{A, J\}$ and middle $I$ in Figure 14.1 on Page 198, we would search for $I$ from both $A$ and $J$, stopping as soon as we get to $I$ from $A$, thereby avoiding a wasteful search from $J$. (We would still have to search for $J$ from $I$ to complete the computation.)

```
def pair_includes_ancestor_and_descendant_of_m(possible_anc_or_desc_0,
                                               possible_anc_or_desc_1, middle):
    search_0, search_1 = possible_anc_or_desc_0, possible_anc_or_desc_1

    # Perform interleaved searching from possible_anc_or_desc_0 and
    # possible_anc_or_desc_1 for middle.
    while (search_0 is not possible_anc_or_desc_1 and search_0 is not middle
           and search_1 is not possible_anc_or_desc_0 and search_1 is not middle
           and (search_0 or search_1)):
        if search_0:
            search_0 = (search_0.left
                        if search_0.data > middle.data else search_0.right)
        if search_1:
            search_1 = (search_1.left
                        if search_1.data > middle.data else search_1.right)

    # If both searches were unsuccessful, or we got from
    # possible_anc_or_desc_0 to possible_anc_or_desc_1 without seeing middle,
    # or from possible_anc_or_desc_1 to possible_anc_or_desc_0 without seeing
    # middle, middle cannot lie between possible_anc_or_desc_0 and
```

```
    # possible_anc_or_desc_1.
    if ((search_0 is not middle and search_1 is not middle)
            or search_0 is possible_anc_or_desc_1
            or search_1 is possible_anc_or_desc_0):
        return False

    def search_target(source, target):
        while source and source is not target:
            source = source.left if source.data > target.data else source.right
        return source is target

    # If we get here, we already know one of possible_anc_or_desc_0 or
    # possible_anc_or_desc_1 has a path to middle. Check if middle has a path
    # to possible_anc_or_desc_1 or to possible_anc_or_desc_0.
    return search_target(middle, possible_anc_or_desc_1
                         if search_0 is middle else possible_anc_or_desc_0)
```

When the middle node does have an ancestor and descendant in the pair, the time complexity is $O(d)$, where $d$ is the difference between the depths of the ancestor and descendant. The reason is that the interleaved search will stop when the ancestor reaches the middle node, i.e., after $O(d)$ iterations. The search from the middle node to the descendant then takes $O(d)$ steps to succeed. When the middle node does not have an ancestor and descendant in the pair, the time complexity is $O(h)$, which corresponds to a worst-case search in a BST.

## 14.10 The range lookup problem

Consider the problem of developing a web-service that takes a geographical location, and returns the nearest restaurant. The service starts with a set of restaurant locations—each location includes $X$ and $Y$-coordinates. A query consists of a location, and should return the nearest restaurant (ties can be broken arbitrarily).

One approach is to build two BSTs on the restaurant locations: $T_X$ sorted on the $X$ coordinates, and $T_Y$ sorted on the $Y$ coordinates. A query on location $(p, q)$ can be performed by finding all the restaurants whose $X$ coordinate is in the interval $[p - D, p + D]$, and all the restaurants whose $Y$ coordinate is in the interval $[q - D, q + D]$, taking the intersection of these two sets, and finding the restaurant in the intersection which is closest to $(p, q)$. Heuristically, if $D$ is chosen correctly, the subsets are small and a brute-force search for the closest point is fast. One approach is to start with a small value for $D$ and keep doubling it until the final intersection is nonempty.

There are other data structures which are more robust, e.g., Quadtrees and $k$-d trees, but the approach outlined above works well in practice.

Write a program that takes as input a BST and an interval and returns the BST keys that lie in the interval. For example, for the tree in Figure 14.1 on Page 198, and interval $[16, 31]$, you should return $17, 19, 23, 29, 31$.

*Hint:* How many edges are traversed when the successor function is repeatedly called $m$ times?

**Solution:** A brute-force approach would be to perform a traversal (inorder, postorder, or preorder) of the BST and record the keys in the specified interval. The time complexity is that of the traversal, i.e., $O(n)$, where $n$ is the number of nodes in the tree.

The brute-force approach does not exploit the BST property—it would work unchanged for an arbitrary binary tree.

We can use the BST property to prune the traversal as follows:

- If the root of the tree holds a key that is less than the left endpoint of the interval, the left subtree cannot contain any node whose key lies in the interval.
- If the root of the tree holds a key that is greater than the right endpoint of the interval, the right subtree cannot contain any node whose key lies in the interval.
- Otherwise, the root of the tree holds a key that lies within the interval, and it is possible for both the left and right subtrees to contain nodes whose keys lie in the interval.

For example, for the tree in Figure 14.1 on Page 198, and interval [16, 42], we begin the traversal at $A$, which contains 19. Since 19 lies in [16, 42], we explore both of $A$'s children, namely $B$ and $I$. Continuing with $B$, we see $B$'s key 7 is less than 16, so no nodes in $B$'s left subtree can lie in the interval [16, 42]. Similarly, when we get to $I$, since $43 > 42$, we need not explore $I$'s right subtree.

```
Interval = collections.namedtuple('Interval', ('left', 'right'))


def range_lookup_in_bst(tree, interval):
    def range_lookup_in_bst_helper(tree):
        if tree is None:
            return

        if interval.left <= tree.data <= interval.right:
            # tree.data lies in the interval.
            range_lookup_in_bst_helper(tree.left)
            result.append(tree.data)
            range_lookup_in_bst_helper(tree.right)
        elif interval.left > tree.data:
            range_lookup_in_bst_helper(tree.right)
        else:  # interval.right > tree.data
            range_lookup_in_bst_helper(tree.left)

    result = []
    range_lookup_in_bst_helper(tree)
    return result
```

The time complexity is tricky to analyze. It makes sense to reason about time complexity in terms of the number of keys $m$ that lie in the specified interval. We partition the nodes into two categories—those that the program recurses on and those that it does not. For our working example, the program recurses on $A, B, F, G, H, I, J, K, L, M, N$. Not all of these have keys in the specified interval, but no nodes outside of this set can have keys in the interval. Looking more carefully at the nodes we recurse on, we see these nodes can be partitioned into three subsets—nodes on the search path to 16, nodes on the search path to 42, and the rest. All nodes in the third subset must lie in the result, but some of the nodes in the first two subsets may or may not lie in the result. The traversal spends $O(h)$ time visiting the first two subsets, and $O(m)$ time traversing the third subset—each edge is visited twice, once downwards, once upwards. Therefore the total time complexity is $O(m + h)$, which is much better than $O(n)$ brute-force approach when the tree is balanced, and very few keys lie in the specified range.

### *Augmented BSTs*

Thus far we have considered BSTs in which each node stores a key, a left child, a right child, and, possibly, the parent. Adding fields to the nodes can speed up certain queries. As an example, consider the following problem.

Suppose you needed a data structure that supports efficient insertion, deletion, lookup of integer keys, as well as range queries, i.e., determining the number of keys that lie in an interval.

We could use a BST, which has efficient insertion, deletion and lookup. To find the number of keys that lie in the interval $[U, V]$, we could search for the first node with a key greater than or equal to $U$, and then call the successor operation ( 9.10 on Page 123) until we reach a node whose key is greater than $V$ (or we run out of nodes). This has $O(h + m)$ time complexity, where $h$ is the height of the tree and $m$ is the number of nodes with keys within the interval. When $m$ is large, this become slow.

We can do much better by augmenting the BST. Specifically, we add a size field to each node, which is the number of nodes in the BST rooted at that node.

For simplicity, suppose we want to find the number of entries that are less than a specified value. As an example, say we want to count the number of keys less than 40 in the BST in Figure 14.1 on Page 198, and that each node has a size field. Since the root $A$'s key, 19, is less than 40, the BST property tells us that all keys in $A$'s left subtree are less than 40. Therefore we can add 7 (which we get from the left child's size field) and 1 (for $A$ itself) to the running count, and recurse with $A$'s right child.

Generalizing, let's say we want to count all entries less than $v$. We initialize count to 0. Since there can be duplicate keys in the tree, we search for the first occurrence of $v$ in an inorder traversal using Solution 14.2 on Page 201. (If $v$ is not present, we stop when we have determined this.) Each time we take a left child, we leave count unchanged; each time we take a right child, we add one plus the size of the corresponding left child. If $v$ is present, when we reach the first occurrence of $v$, we add the size of $v$'s left child, The same approach can be used to find the number of entries that are greater than $v$, less than or equal to $v$, and greater than or equal to $v$.

For example, to count the number of less than 40 in the BST in Figure 14.1 on Page 198 we would search for 40. Since $A$'s key, 19 is less than 40, we update count to $7 + 1 = 8$ and continue from $I$. Since $I$'s key, 43 is greater than 40, we move to $I$'s left child, $J$. Since $J$'s key, 23 is less than 40, update count to $8 + 1 = 9$, and continue from $K$. Since $K$'s key, 37 is less than 40, we update count to $9 + 2 + 1 = 12$, and continue from $N$. Since $N$'s key, 41, is greater than 40, we move to $N$'s left child, which is empty. No other keys can be less than 40, so we return count, i.e., 12. Note how we avoided exploring $A$ and $K$'s left subtrees.

The time bound for these computations is $O(h)$, since the search always descends the tree. When $m$ is large, e.g., comparable to the total number of nodes in the tree, this approach is much faster than repeated calling successor.

To compute the number of nodes with keys in the interval $[L, U]$, first compute the number of nodes with keys less than $L$ and the number of nodes with keys greater than $U$, and subtract that from the total number of nodes (which is the size stored at the root).

The size field can be updated on insert and delete without changing the $O(h)$ time complexity of both. Essentially, the only nodes whose size field change are those on the search path to the added/deleted node. Some conditional checks are needed for each such node, but these add constant time per node, leaving the $O(h)$ time complexity unchanged.

## 14.11 Add credits

Consider a server that a large number of clients connect to. Each client is identified by a string. Each client has a "credit", which is a nonnegative integer value. The server needs to maintain a data structure to which clients can be added, removed, queried, or updated. In addition, the server needs to be able to add a specified number of credits to all clients simultaneously.

Design a data structure that implements the following methods:

- Insert: add a client with specified credit, replacing any existing entry for the client.
- Remove: delete the specified client.
- Lookup: return the number of credits associated with the specified client.
- Add-to-all: increment the credit count for all current clients by the specified amount.
- Max: return a client with the highest number of credits.

*Hint:* Use additional global state.

**Solution:** A hash table is a natural data structure for this application. However, it does not support efficient max operations, nor is there an obvious way to perform the simultaneous increment, short traversing all entries. A BST does have efficient max operation, but it too does not natively support the global increment.

A general principle when adding behaviors to an object is to wrap the object, and add functions in the wrapper, which add behaviors before or after delegating to the object. In our context, this suggests storing the clients in a BST, and having the wrapper track the total increment amount.

For example, if we have clients $A, B, C$, with credits $1, 2, 3$, respectively, and want to add 5 credits to each, the wrapper sets the total increment amount to 5. A lookup on $B$ then is performed by looking up in the BST, which returns 2, and then adding 5 before returning. If we want to add 4 more credits to each, we simply update the total increment amount to 9.

One issue to watch out for is what happens to clients inserted after a call to the add-to-all function. Continuing with the given example, if we were to now add $D$ with a credit of 6, the lookup would return $6 + 9$, which is an error.

The solution is simple—subtract the increment from the credit, i.e., add $D$ with a credit of $6 - 9 = -3$ to the BST. Now a lookup for $D$ will return $-3 + 9$, which is the correct amount.

More specifically, the BST keys are credits, and the corresponding values are the clients with that credit. This makes for fast max-queries. However, to perform lookups and removes by client quickly, the BST by itself is not enough (since it is ordered by credit, not client id). We can solve this by maintaining an additional hash table in which keys are clients, and values are credits. Lookup is trivial. Removes entails a lookup in the hash to get the credit, and then a search into the BST to get the set of clients with that credit, and finally a delete on that set.

```
class ClientsCreditsInfo:
    def __init__(self):
        self._offset = 0
        self._client_to_credit = {}
        self._credit_to_clients = bintrees.RBTree()

    def insert(self, client_id, c):
        self.remove(client_id)
        self._client_to_credit[client_id] = c - self._offset
        self._credit_to_clients.setdefault(c - self._offset,
                                           set()).add(client_id)
```

```
def remove(self, client_id):
    credit = self._client_to_credit.get(client_id)
    if credit is not None:
        self._credit_to_clients[credit].remove(client_id)
        if not self._credit_to_clients[credit]:
            del self._credit_to_clients[credit]
        del self._client_to_credit[client_id]
        return True
    return False

def lookup(self, client_id):
    credit = self._client_to_credit.get(client_id)
    return -1 if credit is None else credit + self._offset

def add_all(self, C):
    self._offset += C

def max(self):
    if not self._credit_to_clients:
        return ''
    clients = self._credit_to_clients.max_item()[1]
    return '' if not clients else next(iter(clients))
```

The time complexity to insert and remove is dominated by the BST, i.e., $O(\log n)$, where $n$ is the number of clients in the data structure. Lookup and add-to-all operate only on the hash table, and have $O(1)$ time complexity. Library BST implementations uses caching to perform max in $O(1)$ time.

CHAPTER

15

# Recursion

*The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.*

— *"Algorithms + Data Structures = Programs,"*
N. E. WIRTH, 1976

Recursion is an approach to problem solving where the solution depends partially on solutions to smaller instances of related problems. It is often appropriate when the input is expressed using recursive rules, such as a computer grammar. More generally, searching, enumeration, divide-and-conquer, and decomposing a complex problem into a set of similar smaller instances are all scenarios where recursion may be suitable.

A recursive function consists of base cases and calls to the same function with different arguments. Two key ingredients to a successful use of recursion are identifying the base cases, which are to be solved directly, and ensuring progress, that is the recursion converges to the solution.

A divide-and-conquer algorithm works by repeatedly decomposing a problem into two or more smaller independent subproblems of the same kind, until it gets to instances that are simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem. Merge sort and quicksort are classical examples of divide-and-conquer.

Divide-and-conquer is not synonymous with recursion. In divide-and-conquer, the problem is divided into two or more independent smaller problems that are of the same type as the original problem. Recursion is more general—there may be a single subproblem, e.g., binary search, the subproblems may not be independent, e.g., dynamic programming, and they may not be of the same type as the original, e.g., regular expression matching. In addition, sometimes to improve runtime, and occasionally to reduce space complexity, a divide-and-conquer algorithm is implemented using iteration instead of recursion.

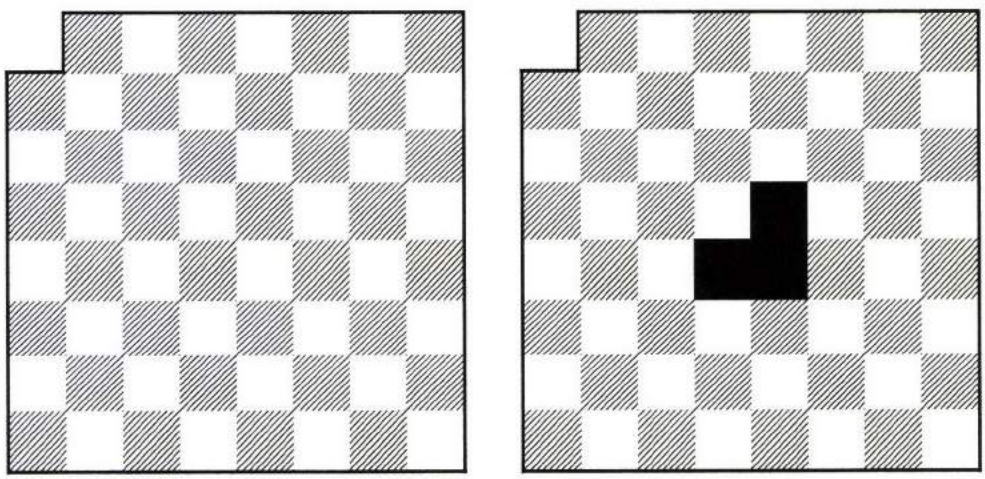
### *Recursion boot camp*

The Euclidean algorithm for calculating the greatest common divisor (GCD) of two numbers is a classic example of recursion. The central idea is that if $y > x$, the GCD of $x$ and $y$ is the GCD of $x$ and $y - x$. For example, GCD(156, 36) = GCD((156 − 36) = 120, 36). By extension, this implies that the GCD of $x$ and $y$ is the GCD of $x$ and $y$ mod $x$, i.e., GCD(156, 36) = GCD((156 mod 36) = 12, 36) = GCD(12, 36 mod 12 = 0) = 12.

```
def gcd(x, y):
    return x if y == 0 else gcd(y, x % y)
```

Since with each recursive step one of the arguments is at least halved, it means that the time complexity is $O(\log \max(x, y))$. Put another way, the time complexity is $O(n)$, where $n$ is the number of bits needed to represent the inputs. The space complexity is also $O(n)$, which is the maximum depth of the function call stack. (The program is easily converted to one which loops, thereby reducing the space complexity to $O(1)$.)

Now we describe a problem that can be elegantly solved using a recursive divide-and-conquer algorithm. A triomino is formed by joining three unit-sized squares in an L-shape. A mutilated chessboard (henceforth $8 \times 8$ Mboard) is made up of 64 unit-sized squares arranged in an $8 \times 8$ square, minus the top-left square, as depicted in Figure 15.1(a). Suppose you are asked to design an algorithm that computes a placement of 21 triominoes that covers the $8 \times 8$ Mboard. Since the $8 \times 8$ Mboard contains 63 squares, and we have 21 triominoes, a valid placement cannot have overlapping triominoes or triominoes which extend out of the $8 \times 8$ Mboard.



(a) An $8 \times 8$ Mboard.

(b) Four $4 \times 4$ Mboards.

**Figure 15.1:** Mutilated chessboards.

Divide-and-conquer is a good strategy for this problem. Instead of the $8 \times 8$ Mboard, let's consider an $n \times n$ Mboard. A $2 \times 2$ Mboard can be covered with one triomino since it is of the same exact shape. You may hypothesize that a triomino placement for an $n \times n$ Mboard with the top-left square missing can be used to compute a placement for an $(n + 1) \times (n + 1)$ Mboard. However, you will quickly see that this line of reasoning does not lead you anywhere.

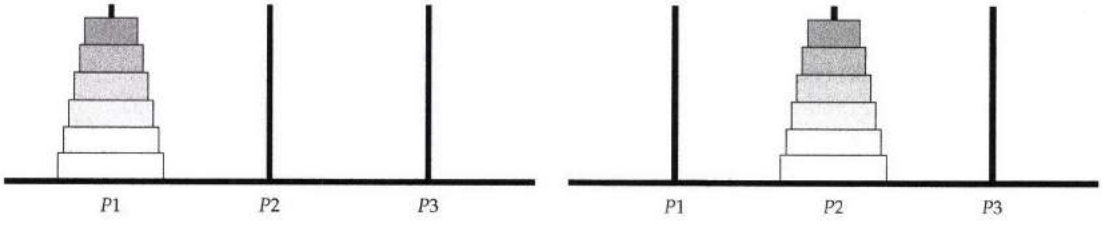
Another hypothesis is that if a placement exists for an $n \times n$ Mboard, then one also exists for a $2n \times 2n$ Mboard. Now we can apply divide-and-conquer. Take four $n \times n$ Mboards and arrange them to form a $2n \times 2n$ square in such a way that three of the Mboards have their missing square set towards the center and one Mboard has its missing square outward to coincide with the missing corner of a $2n \times 2n$ Mboard, as shown in Figure 15.1(b). The gap in the center can be covered with a triomino and, by hypothesis, we can cover the four $n \times n$ Mboards with triominoes as well. Hence, a placement exists for any $n$ that is a power of 2. In particular, a placement exists for the $2^3 \times 2^3$ Mboard; the recursion used in the proof directly yields the placement.

| |
|---|
| Recursion is especially suitable when the **input is expressed using recursive rules** such as a computer grammar. |
| Recursion is a good choice for **search, enumeration**, and **divide-and-conquer**. |
| Use recursion as **alternative to deeply nested iteration** loops. For example, recursion is much better when you have an undefined number of levels, such as the IP address problem generalized to $k$ substrings. |
| If you are asked to **remove recursion** from a program, consider mimicking call stack with the **stack data structure**. |
| Recursion can be easily removed from a **tail-recursive** program by using a while-loop—no stack is needed. (Optimizing compilers do this.) |
| If a recursive function may end up being called with the **same arguments** more than once, **cache** the results—this is the idea behind Dynamic Programming (Chapter 16). |

**Table 15.1:** Top Tips for Recursion

## 15.1 The Towers of Hanoi problem

A peg contains rings in sorted order, with the largest ring being the lowest. You are to transfer these rings to another peg, which is initially empty. This is illustrated in Figure 15.2.




(a) Initial configuration. (b) Desired configuration.

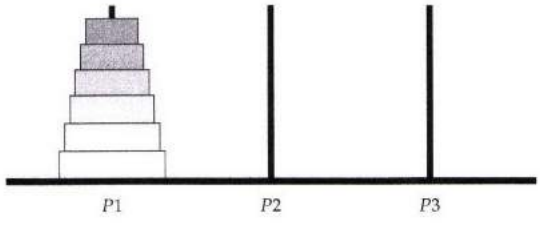
**Figure 15.2:** Tower of Hanoi with 6 pegs.

Write a program which prints a sequence of operations that transfers $n$ rings from one peg to another. You have a third peg, which is initially empty. The only operation you can perform is taking a single ring from the top of one peg and placing it on the top of another peg. You must never place a larger ring above a smaller ring.

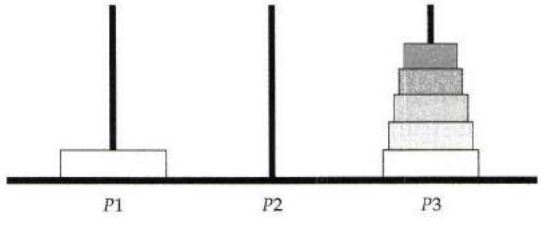
*Hint:* If you know how to transfer the top $n-1$ rings, how does that help move the $n$th ring?

**Solution:** The insight to solving this problem can be gained by trying examples. The three ring transfer can be achieved by moving the top two rings to the third peg, then moving the lowest ring (which is the largest) to the second peg, and then transferring the two rings on the third peg to the second peg, using the first peg as the intermediary. To transfer four rings, move the top three rings to the third peg, then moving the lowest ring (which is the largest) to the second peg, and then transfer the three rings on the third peg to the second peg, using the first peg as an intermediary. For both the three ring and four ring transfers, the first and third steps are instances of the same

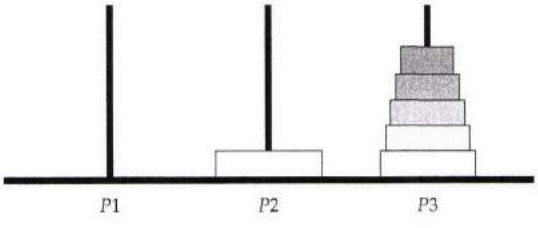
problem, which suggests the use of recursion. This approach is illustrated in Figure 15.3. Code implementing this idea is given below.



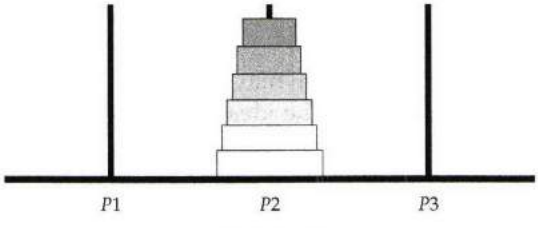

(a) Move all but the lowest ring from $P1$ to $P3$ using $P2$ as an intermediary.




(b) Move the lowest ring from $P1$ to $P2$.




(c) Move the rings to $P3$ to $P2$ using $P1$.




(d) Solved!

**Figure 15.3:** A recursive solution to the Tower of Hanoi for $n = 6$.

```
NUM_PEGS = 3


def compute_tower_hanoi(num_rings):
    def compute_tower_hanoi_steps(num_rings_to_move, from_peg, to_peg, use_peg):
        if num_rings_to_move > 0:
            compute_tower_hanoi_steps(num_rings_to_move - 1, from_peg, use_peg,
                                      to_peg)
            pegs[to_peg].append(pegs[from_peg].pop())
            result.append([from_peg, to_peg])
            compute_tower_hanoi_steps(num_rings_to_move - 1, use_peg, to_peg,
                                      from_peg)

    # Initialize pegs.
    result = []
    pegs = [list(reversed(range(1, num_rings + 1)))
            ] + [[] for _ in range(1, NUM_PEGS)]
    compute_tower_hanoi_steps(num_rings, 0, 1, 2)
    return result
```

The number of moves, $T(n)$, satisfies the following recurrence: $T(n) = T(n-1) + 1 + T(n-1) = 1 + 2T(n-1)$. The first $T(n-1)$ corresponds to the transfer of the top $n-1$ rings from $P1$ to $P3$, and the second $T(n-1)$ corresponds to the transfer from $P3$ to $P2$. This recurrence solves to $T(n) = 2^n - 1$. One way to see this is to "unwrap" the recurrence: $T(n) = 1 + 2 + 4 + \cdots + 2^k T(n-k)$. Printing a single move takes $O(1)$ time, so the time complexity is $O(2^n)$.

**Variant:** Solve the same problem without using recursion.

**Variant:** Find the minimum number of operations subject to the constraint that each operation must involve $P3$.

**Variant:** Find the minimum number of operations subject to the constraint that each transfer must be from $P1$ to $P2$, $P2$ to $P3$, or $P3$ to $P1$.

**Variant:** Find the minimum number of operations subject to the constraint that a ring can never be transferred directly from $P1$ to $P2$ (transfers from $P2$ to $P1$ are allowed).

**Variant:** Find the minimum number of operations when the stacking constraint is relaxed to the following—the largest ring on a peg must be the lowest ring on the peg. (The remaining rings on the peg can be in any order, e.g., it is fine to have the second-largest ring above the third-largest ring.)

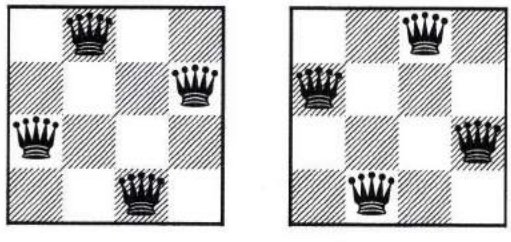
**Variant:** You have $2n$ disks of $n$ different sizes, two of each size. You cannot place a larger disk on a smaller disk, but can place a disk of equal size on top of the other. Compute the minimum number of moves to transfer the $2n$ disks from $P1$ to $P2$.

**Variant:** You have $2n$ disks which are colored black or white. You cannot place a white disk directly on top of a black disk. Compute the minimum number of moves to transfer the $2n$ disks from $P1$ to $P2$.

**Variant:** Find the minimum number of operations if you have a fourth peg, $P4$.

## 15.2 Generate all nonattacking placements of $n$-Queens

A nonattacking placement of queens is one in which no two queens are in the same row, column, or diagonal. See Figure 15.4 for an example.



(a) Solution 1. (b) Solution 2.

**Figure 15.4:** The only two ways in which four queens can be placed on a $4 \times 4$ chessboard.

Write a program which returns all distinct nonattacking placements of $n$ queens on an $n \times n$ chessboard, where $n$ is an input to the program.

*Hint:* If the first queen is placed at $(i, j)$, where can the remaining queens definitely not be placed?

**Solution:** A brute-force approach is to consider all possible placements of the $n$ queens—there are $\binom{n^2}{n}$ possible placements which grows very large with $n$.

Since we never would place two queens on the same row, a much faster solution is to enumerate placements that use distinct rows. Such a placement cannot lead to conflicts on rows, but it may

lead to conflicts on columns and diagonals. It can be represented by an array of length $n$, where the $i$th entry is the location of the queen on Row $i$.

As an example, if $n = 4$, begin by placing the first row's queen at Column 0. Now we enumerate all placements of the form $(0, \_, \_, \_)$. Placing the second row's queen at Column 0 leads to a column conflict, so we skip all placements of the form $(0, 0, \_, \_)$. Placing the second row's queen at Column 1 leads to a diagonal conflict, so we skip all placements of the form $(0, 1, \_, \_)$. Now we turn to placements of the form $(0, 2, 0, \_)$. Such placements are conflicting because of the conflict on Column 0. Now we turn to placements of the form $(0, 2, 1, \_)$ and $(0, 2, 2, \_)$. Such placements are conflicting because of the diagonal conflict between the queens at Row 1 and Column 2 and Row 2 and Column 1, and the column conflict between the queens at Row 1 and Column 2 and Row 2 and Column 2, respectively, so we move on to $(0, 2, 3, \_)$, which also violates a diagonal constraint. Now we advance to placements of the form $(0, 3, \_, \_)$. Both $(0, 3, 1, \_)$ and $(0, 3, 2, \_)$ lead to conflicts, implying there is no nonattacking placement possible with a queen placed at Row 0 and Column 0. The first nonattacking placement is $(1, 3, 0, 2)$; the only other nonattacking placement is $(2, 0, 3, 1)$.

```
def n_queens(n):
    def solve_n_queens(row):
        if row == n:
            # All queens are legally placed.
            result.append(list(col_placement))
            return
        for col in range(n):
            # Test if a newly placed queen will conflict any earlier queens
            # placed before.
            if all(
                    abs(c - col) not in (0, row - i)
                    for i, c in enumerate(col_placement[:row])):
                col_placement[row] = col
                solve_n_queens(row + 1)

    result, col_placement = [], [0] * n
    solve_n_queens(0)
    return result
```

The time complexity is lower bounded by the number of nonattacking placements. No exact form is known for this quantity as a function of $n$, but it is conjectured to tend to $n!/c^n$, where $c \approx 2.54$, which is super-exponential.

**Variant:** Compute the number of nonattacking placements of $n$ queens on an $n \times n$ chessboard.

**Variant:** Compute the smallest number of queens that can be placed to attack each uncovered square.

**Variant:** Compute a placement of 32 knights, or 14 bishops, 16 kings or 8 rooks on an 8×8 chessboard in which no two pieces attack each other.

### 15.3 Generate permutations

This problem is concerned with computing all permutations of an array. For example, if the array is $\langle 2, 3, 5, 7 \rangle$ one output could be $\langle 2, 3, 5, 7 \rangle$, $\langle 2, 3, 7, 5 \rangle$, $\langle 2, 5, 3, 7 \rangle$, $\langle 2, 5, 7, 3 \rangle$, $\langle 2, 7, 3, 5 \rangle$, $\langle 2, 7, 5, 3 \rangle$,

$\langle 3,2,5,7\rangle$, $\langle 3,2,7,5\rangle$, $\langle 3,5,2,7\rangle$, $\langle 3,5,7,2\rangle$, $\langle 3,7,2,5\rangle$, $\langle 3,7,5,2\rangle$, $\langle 5,2,3,7\rangle$, $\langle 5,2,7,3\rangle$, $\langle 5,3,2,7\rangle$, $\langle 5,3,7,2\rangle$, $\langle 5,7,2,3\rangle$, $\langle 5,7,2,3\rangle$, $\langle 7,2,3,5\rangle$, $\langle 7,2,5,3\rangle$, $\langle 7,3,2,5\rangle$, $\langle 7,3,5,2\rangle$, $\langle 7,5,2,3\rangle$, $\langle 7,5,3,2\rangle$. (Any other ordering is acceptable too.)

Write a program which takes as input an array of distinct integers and generates all permutations of that array. No permutation of the array may appear more than once.

*Hint:* How many possible values are there for the first element?

**Solution:** Let the input array be $A$. Suppose its length is $n$. A truly brute-force approach would be to enumerate all arrays of length $n$ whose entries are from $A$, and check each such array for being a permutation. This enumeration can be performed recursively, e.g., enumerate all arrays of length $n-1$ whose entries are from $A$, and then for each array, consider the $n$ arrays of length $n$ which is formed by adding a single entry to the end of that array. Since the number of possible arrays is $n^n$, the time and space complexity are staggering.

A better approach is to recognize that once a value has been chosen for an entry, we do not want to repeat it. Specifically, every permutation of $A$ begins with one of $A[0], A[1], \ldots, A[n-1]$. The idea is to generate all permutations that begin with $A[0]$, then all permutations that begin with $A[1]$, and so on. Computing all permutations beginning with $A[0]$ entails computing all permutations of $A[1, n-1]$, which suggests the use of recursion. To compute all permutations beginning with $A[1]$ we swap $A[0]$ with $A[1]$ and compute all permutations of the updated $A[1, n-1]$. We then restore the original state before embarking on computing all permutations beginning with $A[2]$, and so on.

For example, for the array $\langle 7,3,5\rangle$, we would first generate all permutations starting with 7. This entails generating all permutations of $\langle 3,5\rangle$, which we do by finding all permutations of $\langle 3,5\rangle$ beginning with 3. Since $\langle 5\rangle$ is an array of length 1, it has a single permutation. This implies $\langle 3,5\rangle$ has a single permutation beginning with 3. Next we look for permutations of $\langle 3,5\rangle$ beginning with 5. To do this, we swap 3 and 5, and find, as before, there is a single permutation of $\langle 3,5\rangle$ beginning with 5, namely, $\langle 5,3\rangle$. Hence, there are two permutations of $A$ beginning with 7, namely $\langle 7,3,5\rangle$ and $\langle 7,5,3\rangle$. We swap 7 with 3 to find all permutations beginning with 3, namely $\langle 3,7,5\rangle$ and $\langle 3,5,7\rangle$. The last two permutations we add are $\langle 5,3,7\rangle$ and $\langle 5,7,3\rangle$. In all there are six permutations.

```
def permutations(A):
    def directed_permutations(i):
        if i == len(A) - 1:
            result.append(A.copy())
            return

        # Try every possibility for A[i].
        for j in range(i, len(A)):
            A[i], A[j] = A[j], A[i]
            # Generate all permutations for A[i + 1:].
            directed_permutations(i + 1)
            A[i], A[j] = A[j], A[i]

    result = []
    directed_permutations(0)
    return result
```

The time complexity is determined by the number of recursive calls, since within each function the time spent is $O(1)$, not including the time in the subcalls. The number of function calls, $C(n)$

satisfies the recurrence $C(n) = 1 + nC(n-1)$ for $n \geq 1$, with $C(0) = 1$. Expanding this, we see $C(n) = 1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n! = n!(1/n! + 1/(n-1)! + 1/(n-2)! + \cdots + 1/1!)$. The sum $(1 + 1/1! + 1/2! + \cdots + 1/n!)$ tends to Euler's number $e$, so $C(n)$ tends to $(e-1)n!$, i.e., $O(n!)$. The time complexity $T(n)$ is $O(n \times n!)$, since we do $O(n)$ computation per call outside of the recursive calls.

Now we describe a qualitatively different algorithm for this problem. In Solution 5.11 on Page 53 we showed how to efficiently compute the permutation that follows a given permutation, e.g., $\langle 2,3,1,4 \rangle$ is followed by $\langle 2,3,4,1 \rangle$. We can extend that algorithm to solve the current problem. The idea is that the $n$ distinct entries in the array can be mapped to $1, 2, 3, \ldots$, with 1 corresponding to the smallest entry. For example, if the array is $\langle 7,3,5 \rangle$, we first sort it to obtain, $\langle 3,5,7 \rangle$. Using the approach of Solution 5.11 on Page 53, the next array will be $\langle 3,7,5 \rangle$, followed by $\langle 5,3,7 \rangle$, $\langle 5,7,3 \rangle$, $\langle 7,3,5 \rangle$, $\langle 7,5,3 \rangle$. This is the approach in the program below.

```
def permutations(A):
    result = []
    while True:
        result.append(A.copy())
        A = next_permutation(A)
        if not A:
            break
    return result
```

The time complexity is $O(n \times n!)$, since there are $n!$ permutations and we spend $O(n)$ time to store each one.

**Variant:** Solve Problem 15.3 on Page 222 when the input array may have duplicates. You should not repeat any permutations. For example, if $A = \langle 2,2,3,0 \rangle$ then the output should be $\langle 2,2,0,3 \rangle$, $\langle 2,2,3,0 \rangle$, $\langle 2,0,2,3 \rangle$, $\langle 2,0,3,2 \rangle$, $\langle 2,3,2,0 \rangle$, $\langle 2,3,0,2 \rangle$, $\langle 0,2,2,3 \rangle$, $\langle 0,2,3,2 \rangle$, $\langle 0,3,2,2 \rangle$, $\langle 3,2,2,0 \rangle$, $\langle 3,2,0,2 \rangle$, $\langle 3,0,2,2 \rangle$.

### 15.4 Generate the power set

The power set of a set $S$ is the set of all subsets of $S$, including both the empty set $\emptyset$ and $S$ itself. The power set of $\{0,1,2\}$ is graphically illustrated in Figure 15.5.

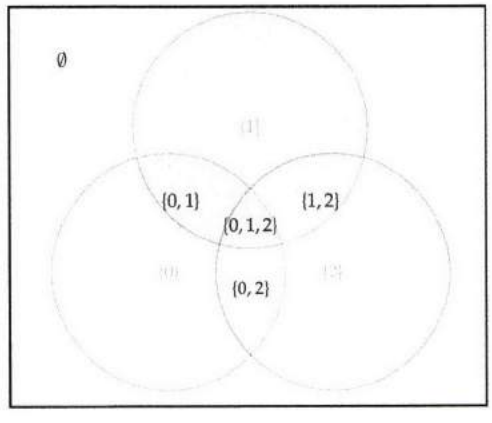



**Figure 15.5:** The power set of $\{0,1,2\}$ is $\{\emptyset, \{0\}, \{1\}, \{2\}, \{0,1\}, \{1,2\}, \{0,2\}, \{0,1,2\}\}$.

Write a function that takes as input a set and returns its power set.

*Hint:* There are $2^n$ subsets for a given set $S$ of size $n$. There are $2^k$ $k$-bit words.

**Solution:** A brute-force way is to compute all subsets $U$ that do not include a particular element (which could be any single element). Then we compute all subsets $V$ which do include that element. Each subset set must appear in $U$ or in $V$, so the final result is just $U \cup V$. The construction is recursive, and the base case is when the input set is empty, in which case we return $\{\{\}\}$.

As an example, let $S = \{0, 1, 2\}$. Pick any element, e.g., 0. First, we recursively compute all subsets of $\{1, 2\}$. To do this, we select 1. This leaves us with $\{2\}$. Now we pick 2, and get to a base case. So the set of subsets of $\{2\}$ is $\{\}$ union with $\{2\}$, i.e., $\{\{\}, \{2\}\}$. The set of subsets of $\{1, 2\}$ then is $\{\{\}, \{2\}\}$ union with $\{\{1\}, \{1, 2\}\}$, i.e., $\{\{\}, \{2\}, \{1\}, \{1, 2\}\}$. The set of subsets of $\{0, 1, 2\}$ then is $\{\{\}, \{2\}, \{1\}, \{1, 2\}\}$ union with $\{\{0\}, \{0, 2\}, \{0, 1\}, \{0, 1, 2\}\}$, i.e., $\{\{\}, \{2\}, \{1\}, \{1, 2\}, \{0\}, \{0, 2\}, \{0, 1\}, \{0, 1, 2\}\}$,

```
def generate_power_set(input_set):
    # Generate all subsets whose intersection with input_set[0], ...,
    # input_set[to_be_selected - 1] is exactly selected_so_far.
    def directed_power_set(to_be_selected, selected_so_far):
        if to_be_selected == len(input_set):
            power_set.append(list(selected_so_far))
            return

        directed_power_set(to_be_selected + 1, selected_so_far)
        # Generate all subsets that contain input_set[to_be_selected].
        directed_power_set(to_be_selected + 1,
                           selected_so_far + [input_set[to_be_selected]])

    power_set = []
    directed_power_set(0, [])
    return power_set
```

The number of recursive calls, $C(n)$ satisfies the recurrence $C(n) = 2C(n-1)$, which solves to $C(n) = O(2^n)$. Since we spend $O(n)$ time within a call, the time complexity is $O(n2^n)$. The space complexity is $O(n2^n)$, since there are $2^n$ subsets, and the average subset size is $n/2$. If we just want to print the subsets, rather than returning all of them, we simply perform a print instead of adding the subset to the result, which reduces the space complexity to $O(n)$—the time complexity remains the same.

For a given ordering of the elements of $S$, there exists a one-to-one correspondence between the $2^n$ bit arrays of length $n$ and the set of all subsets of $S$—the 1s in the $n$-length bit array $v$ indicate the elements of $S$ in the subset corresponding to $v$. For example, if $S = \{a, b, c, d\}$, the bit array $\langle 1, 0, 1, 1 \rangle$ denotes the subset $\{a, c, d\}$. This observation can be used to derive a nonrecursive algorithm for enumerating subsets.

In particular, when $n$ is less than or equal to the width of an integer on the architecture (or language) we are working on, we can enumerate bit arrays by enumerating integers in $[0, 2^n - 1]$ and examining the indices of bits set in these integers. These indices are determined by first isolating the lowest set bit by computing $y = x\&\sim(x-1)$, which is described on Page 23, and then getting the index by computing $\log y$.

```
def generate_power_set(S):
    power_set = []
```

```
    for int_for_subset in range(1 << len(S)):
        bit_array = int_for_subset
        subset = []
        while bit_array:
            subset.append(int(math.log2(bit_array & ~(bit_array - 1))))
            bit_array &= bit_array - 1
        power_set.append(subset)
    return power_set
```

Since each set takes $O(n)$ time to compute, the time complexity is $O(n2^n)$. In practice, this approach is very fast. Furthermore, its space complexity is $O(n)$ when we want to just enumerate subsets, e.g., to print them, rather that to return all the subsets.

**Variant:** Solve this problem when the input array may have duplicates, i.e., denotes a multiset. You should not repeat any multiset. For example, if $A = \langle 1,2,3,2\rangle$, then you should return $\big\langle\langle\rangle,\langle 1\rangle,\langle 2\rangle,\langle 3\rangle,\langle 1,2\rangle,\langle 1,3\rangle,\langle 2,2\rangle,\langle 2,3\rangle,\langle 1,2,2\rangle,\langle 1,2,3\rangle,\langle 2,2,3\rangle,\langle 1,2,2,3\rangle\big\rangle$.

### 15.5 Generate all subsets of size $k$

There are a number of testing applications in which it is required to compute all subsets of a given size for a specified set.

Write a program which computes all size $k$ subsets of $\{1,2,\ldots,n\}$, where $k$ and $n$ are program inputs. For example, if $k = 2$ and $n = 5$, then the result is the following: $\{\{1,2\},\{1,3\},\{1,4\},\{1,5\},\{2,3\},\{2,4\},\{2,5\},\{3,4\},\{3,5\},\{4,5\}\}$

*Hint:* Think of the right function signature.

**Solution:** One brute-force approach is to compute all subsets of $\{1,2,\ldots,n\}$, and then restrict the result to subsets of size $k$. A convenient aspect of this approach is that we can use Solution 15.4 on the preceding page to compute all subsets. The time complexity is $O(n2^n)$, regardless of $k$. When $k$ is much smaller than $n$, or nearly equal to $n$, it ends up computing many subsets which cannot possibly be of the right size.

To gain efficiency, we use a more focused approach. In particular, we can make nice use of case analysis. There are two possibilities for a subset—it does not contain 1, or it does contain 1. In the first case, we return all subsets of size $k$ of $\{2,3,\ldots,n\}$; in the second case, we compute all $k-1$ sized subsets of $\{2,3,\ldots,n\}$ and add 1 to each of them.

For example, if $n = 4$ and $k = 2$, then we compute all subsets of size 2 from $\{2,3,4\}$, and all subsets of size 1 from $\{2,3,4\}$. We add 1 to each of the latter, and the result is the union of the two sets of subsets, i.e., $\{\{2,3\},\{2,4\},\{3,4\}\} \cup \{\{1,2\},\{1,3\},\{1,4\}\}$.

```
def combinations(n, k):
    def directed_combinations(offset, partial_combination):
        if len(partial_combination) == k:
            result.append(list(partial_combination))
            return

        # Generate remaining combinations over {offset, ..., n - 1} of size
        # num_remaining.
        num_remaining = k - len(partial_combination)
```

```
        i = offset
        while i <= n and num_remaining <= n - i + 1:
            directed_combinations(i + 1, partial_combination + [i])
            i += 1

    result = []
    directed_combinations(1, [])
    return result
```

The time complexity is $O(n\binom{n}{k})$; the reasoning is analogous to that for the recursive solution enumerating the powerset (Page 225).

### 15.6 Generate strings of matched parens

Strings in which parens are matched are defined by the following three rules:

- The empty string, "", is a string in which parens are matched.
- The addition of a leading left parens and a trailing right parens to a string in which parens are matched results in a string in which parens are matched. For example, since "(())()" is a string with matched parens, so is "((())())".
- The concatenation of two strings in which parens are matched is itself a string in which parens are matched. For example, since "(())()" and "()" are strings with matched parens, so is "(())()()".

For example, the set of strings containing two pairs of matched parens is {(()), ()()}, and the set of strings with three pairs of matched parens is {((())), (()()), (())(), ()(()), ()()()}.

Write a program that takes as input a number and returns all the strings with that number of matched pairs of parens.

*Hint:* Think about what the prefix of a string of matched parens must look like.

**Solution:** A brute-force approach would be to enumerate all strings on $2k$ parentheses. To test if the parens in a string are matched, we use Solution 8.3 on Page 102, specialized to one type of parentheses. There are $2^{2k}$ possible strings, which is a lower bound on the time complexity. Even if we restrict the enumeration to strings with an equal number of left and right parens, there are $\binom{2k}{k}$ strings to consider.

We can greatly improve upon the time complexity by enumerating in a more directed fashion. For example, some strings can never be completed to a string with $k$ pairs of matched parens, e.g., if a string begins with ). Therefore, one way to be more directed is to build strings incrementally. We will ensure that as each additional character is added, the resulting string has the potential to be completed to a string with $k$ pairs of matched parens.

Suppose we have a string whose length is less than $2k$, and we know that string can be completed to a string with $k$ pairs of matched parens. How can we extend that string with an additional character so that the resulting string can still be completed to a string with $k$ pairs of matched parens?

There are two possibilities: we add a left parens, or we add a right parens.

- If we add a left parens, and still want to complete the string to a string with $k$ pairs of matched parens, it must be that the number of left parens we need is greater than 0.

- If we add a right parens, and still want to complete the string to a string with $k$ pairs of matched parens, it must be that the number of left parens we need is less than the number of right parens (i.e., there are unmatched left parens in the string).

As a concrete example, if $k = 2$, we would go through the following sequence of strings: "", "(", "((", "(()", "(())", "()", "()(", "()()". Of these, "(())" and "()()" are complete, and we would add them to the result.

```
def generate_balanced_parentheses(num_pairs):
    def directed_generate_balanced_parentheses(num_left_parens_needed,
                                               num_right_parens_needed,
                                               valid_prefix,
                                               result=[]):
        if num_left_parens_needed > 0:  # Able to insert '('.
            directed_generate_balanced_parentheses(num_left_parens_needed - 1,
                                                   num_right_parens_needed,
                                                   valid_prefix + '(')
        if num_left_parens_needed < num_right_parens_needed:
            # Able to insert ')'.
            directed_generate_balanced_parentheses(num_left_parens_needed,
                                                   num_right_parens_needed - 1,
                                                   valid_prefix + ')')
        if not num_right_parens_needed:
            result.append(valid_prefix)
        return result

    return directed_generate_balanced_parentheses(num_pairs, num_pairs, '')
```

The number $C(k)$ of strings with $k$ pairs of matched parens grows very rapidly with $k$. Specifically, it can be shown that $C(k+1) = \sum_{i=0}^{k} \binom{k}{i}/(k+1)$, which solves to $(2k)!/((k!(k+1)!)$.

## 15.7 Generate palindromic decompositions

A string is said to be palindromic if it reads the same backwards and forwards. A decomposition of a string is a set of strings whose concatenation is the string. For example, "611116" is palindromic, and "611", "11", "6" is one decomposition for it.

Compute all palindromic decompositions of a given string. For example, if the string is "0204451881", then the decomposition "020", "44", "5", "1881" is palindromic, as is "020", "44", "5", "1", "88", "1". However, "02044, "5", "1881" is not a palindromic decomposition.

*Hint:* Focus on the first palindromic string in a palindromic decomposition.

**Solution:** We can brute-force compute all palindromic decompositions by first computing all decompositions, and then checking which ones are palindromic. To compute all decompositions, we use prefixes of length $1, 2\ldots$ for the first string in the decomposition, and recursively compute the decomposition of the corresponding suffix. The number of such decompositions is $2^{n-1}$, where $n$ is the length of the string. (One way to understand this is from the fact that every $n$-bit vector corresponds to a unique decomposition—the 1s in the bit vector denote the starting point of a substring.)

Clearly, the brute-force approach is inefficient because it continues with decompositions that cannot possibly be palindromic, e.g., it will recursively compute decompositions that begin with

"02" for "0204451881". We need a more directed approach—specifically, we should enumerate decompositions that begin with a palindrome.

For the given example, "0204451881", we would recursively compute palindromic sequences for "204451881" (since "0" is a palindrome), and for "4451881" (since "020" is a palindrome). To compute palindromic decompositions for "204451881", we would recursively compute palindromic sequences for "04451881" (since "2" is the only prefix that is a palindrome). To compute palindromic decompositions for "04451881", we would recursively compute palindromic sequences for "4451991" (since "0" is the only prefix that is a palindrome). To compute palindromic decompositions for "4451991", we would recursively compute palindromic sequences for "451991" (since "4" is a palindrome) and for "51991" (since "44' is a palindrome).

```
def palindrome_decompositions(input):
    def directed_palindrome_decompositions(offset, partial_partition):
        if offset == len(input):
            result.append(list(partial_partition))
            return

        for i in range(offset + 1, len(input) + 1):
            prefix = input[offset:i]
            if prefix == prefix[::-1]:
                directed_palindrome_decompositions(i,
                                                   partial_partition + [prefix])

    result = []
    directed_palindrome_decompositions(0, [])
    return result


# Pythonic solution uses bottom-up construction.
def palindrome_decompositions_pythonic(text):
    return ([[text[:i]] + right for i in range(1, len(text) + 1)
             if text[:i] == text[:i][::-1]
             for right in palindrome_decompositions_pythonic(text[i:])] or [[]])
```

The worst-case time complexity is still $O(n \times 2^n)$, e.g., if the input string consists of $n$ repetitions of a single character. However, our program has much better best-case time complexity than the brute-force approach, e.g., when there are very few palindromic decompositions.

## 15.8 Generate binary trees

Write a program which returns all distinct binary trees with a specified number of nodes. For example, if the number of nodes is specified to be three, return the trees in Figure 15.6 on the following page.

*Hint:* Can two binary trees whose left subtrees differ in size be the same?

**Solution:** A brute-force approach to generate all binary trees on $n$ nodes would be to generate all binary trees on $n-1$ or fewer nodes. Afterwards, form all binary trees with a root and a left child with $n-1$ or fewer nodes, and a right child with $n-1$ or fewer nodes. The resulting trees would all

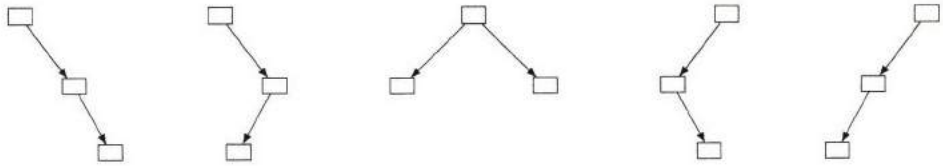


**Figure 15.6:** The five binary trees on three nodes.

be distinct, since no two would have the same left and right child. However, some will have fewer than $n-1$ nodes, and some will have more.

The key to efficiency is to direct the search. If the left child has $k$ nodes, we should only use right children with $n-1-k$ nodes, to get binary trees with $n$ nodes that have that left child. Specifically, we get all binary trees on $n$ nodes by getting all left subtrees on $i$ nodes, and right subtrees on $n-1-i$ nodes, for $i$ between 0 and $n-1$.

Looking carefully at Figure 15.6, you will see the first two trees correspond to the trees on three nodes which have a left subtree of size 0 and a right subtree of size 2. The third tree is the only tree on three nodes which has a left subtree of size 1 and a right subtree of size 1. The last two trees correspond to the trees on three nodes which have a left subtree of size 2 and a right subtree of size 0. The set of two trees on two nodes is itself computed recursively: there is a single binary tree on one node, and it may be on either side of the root.

```
def generate_all_binary_trees(num_nodes):
    if num_nodes == 0:  # Empty tree, add as a None.
        return [None]

    result = []
    for num_left_tree_nodes in range(num_nodes):
        num_right_tree_nodes = num_nodes - 1 - num_left_tree_nodes
        left_subtrees = generate_all_binary_trees(num_left_tree_nodes)
        right_subtrees = generate_all_binary_trees(num_right_tree_nodes)
        # Generates all combinations of left_subtrees and right_subtrees.
        result += [
            BinaryTreeNode(0, left, right)
            for left in left_subtrees for right in right_subtrees
        ]
    return result
```

The number of calls $C(n)$ to the recursive function satisfies the recurrence $C(n) = \sum_{i=1}^{n} C(n-i)C(i-1)$. The quantity $C(n)$ is called the $n$th Catalan number. It is known to be equal to $\frac{(2n)!}{n!(n+1)!}$. Comparing Solution 15.6 on Page 227 to this solution, you will see considerable similarity—the Catalan numbers appear in numerous types of combinatorial problems.

### 15.9 Implement a Sudoku solver

Implement a Sudoku solver. See Problem 5.17 on Page 60 for a definition of Sudoku.

*Hint:* Apply the constraints to speed up a brute-force algorithm.

**Solution:** A brute-force approach would be to try every possible assignment to empty entries, and then check if that assignment leads to a valid solution using Solution 5.17 on Page 60. This

is wasteful, since if setting a value early on leads to a constraint violation, there is no point in continuing. Therefore, we should apply the backtracking principle.

Specifically, we traverse the 2D array entries one at a time. If the entry is empty, we try each value for the entry, and see if the updated 2D array is still valid; if it is, we recurse. If all the entries have been filled, the search is successful. The naive approach to testing validity is calling Solution 5.17 on Page 60. However, we can reduce runtime considerably by making use of the fact that we are adding a value to an array that already satisfies the constraints. This means that we need to check just the row, column, and subgrid of the added entry.

For example, suppose we begin with the lower-left entry for the configuration in Figure 5.2(a) on Page 60. Adding a 1 to entry does not violate any row, column, or subgrid constraint, so we move on to the next entry in that row. We cannot put a 1, since that would now violate a row constraint; however, a 2 is acceptable.

```
def solve_sudoku(partial_assignment):
    def solve_partial_sudoku(i, j):
        if i == len(partial_assignment):
            i = 0  # Starts a row.
            j += 1
            if j == len(partial_assignment[i]):
                return True  # Entire matrix has been filled without conflict.

        # Skips nonempty entries.
        if partial_assignment[i][j] != EMPTY_ENTRY:
            return solve_partial_sudoku(i + 1, j)

        def valid_to_add(i, j, val):
            # Check row constraints.
            if any(val == partial_assignment[k][j]
                   for k in range(len(partial_assignment))):
                return False

            # Check column constraints.
            if val in partial_assignment[i]:
                return False

            # Check region constraints.
            region_size = int(math.sqrt(len(partial_assignment)))
            I = i // region_size
            J = j // region_size
            return not any(
                val == partial_assignment[region_size * I + a][region_size * J
                                                               + b]
                for a, b in itertools.product(range(region_size), repeat=2))

        for val in range(1, len(partial_assignment) + 1):
            # It's substantially quicker to check if entry val with any of the
            # constraints if we add it at (i,j) adding it, rather than adding it and
            # then checking all constraints. The reason is that we know we are
            # starting with a valid configuration, and the only entry which can
            # cause a problem is entry val at (i,j).
            if valid_to_add(i, j, val):
                partial_assignment[i][j] = val
```

```
            if solve_partial_sudoku(i + 1, j):
                return True
        partial_assignment[i][j] = EMPTY_ENTRY  # Undo assignment.
        return False

    EMPTY_ENTRY = 0
    return solve_partial_sudoku(0, 0)
```

Because the program is specialized to $9 \times 9$ grids, it does not make sense to speak of its time complexity, since there is no notion of scaling with a size parameter. However, since the problem of solving Sudoku generalized to $n \times n$ grids is NP-complete, it should not be difficult to prove that the generalization of this algorithm to $n \times n$ grids has exponential time complexity.

## 15.10 Compute a Gray code

An $n$-bit Gray code is a permutation of $\{0, 1, 2, \ldots, 2^n - 1\}$ such that the binary representations of successive integers in the sequence differ in only one place. (This is with wraparound, i.e., the last and first elements must also differ in only one place.) For example, both $\langle (000)_2, (100)_2, (101)_2, (111)_2, (110)_2, (010)_2, (011)_2, (001)_2 \rangle = \langle 0, 4, 5, 7, 6, 2, 3, 1 \rangle$ and $\langle 0, 1, 3, 2, 6, 7, 5, 4 \rangle$ are Gray codes for $n = 3$.

Write a program which takes $n$ as input and returns an $n$-bit Gray code.

*Hint:* Write out Gray codes for $n = 2, 3, 4$.

**Solution:** A brute-force approach would be to enumerate sequences of length $2^n$ whose entries are $n$ bit integers. We would test if the sequence is a Gray code, and stop as soon as we find one. The complexity is astronomical—there are $2^{n \times 2^n}$ sequences. We can improve complexity by enumerating permutations of $0, 1, 2, \ldots, 2^n - 1$, since we want distinct entries, but this is still a very high complexity.

We can do much better by directing the enumeration. Specifically, we build the sequence incrementally, adding a value only if it is distinct from all values currently in the sequence, and differs in exactly one place with the previous value. (For the last value, we have to check that it differs in one place from the first value.) This is the approach shown below. For $n = 4$, we begin with $(0000)_2$. Next we try changing bits in $(0000)_2$, one-at-a-time to get a value not currently present in the sequence, which yields $(0001)_2$, which we append to the sequence. Changing bits in $(0001)_2$, one-at-a-time, we get $(0000)_2$ (which is already present), and then $(0011)_2$, which is not, so we append it to the sequence, which is now $\langle (0000)_2, (0001)_2, (0011)_2 \rangle$. The next few values are $(0010)_2$, $(0011)_2$, $(0111)_2$.

```
def gray_code(num_bits):
    def directed_gray_code(history):
        def differs_by_one_bit(x, y):
            bit_difference = x ^ y
            return bit_difference and not (bit_difference &
                                           (bit_difference - 1))

        if len(result) == 1 << num_bits:
            # Check if the first and last codes differ by one bit.
            return differs_by_one_bit(result[0], result[-1])
```

```
        for i in range(num_bits):
            previous_code = result[-1]
            candidate_next_code = previous_code ^ (1 << i)
            if candidate_next_code not in history:
                history.add(candidate_next_code)
                result.append(candidate_next_code)
                if directed_gray_code(history):
                    return True
                history.remove(candidate_next_code)
                del result[-1]
        return False

    result = [0]
    directed_gray_code(set([0]))
    return result
```

Now we present a more analytical solution. The inspiration comes from small case analysis. The sequence $\langle (00)_2, (01)_2, (11)_2, (10)_2 \rangle$ is a 2-bit Gray code. To get to $n = 3$, we cannot just prepend 0 to each elements of $\langle (00)_2, (01)_2, (11)_2, (10)_2 \rangle$, 1 to $\langle (00)_2, (01)_2, (11)_2, (10)_2 \rangle$ and concatenate the two sequences—that leads to the Gray code property being violated from $(010)_2$ to $(100)_2$. However, it is preserved everywhere else.

Since Gray codes differ in one place on wrapping around, prepending 1 to the reverse of $\langle (00)_2, (01)_2, (11)_2, (10)_2 \rangle$ solves the problem when transitioning from a leading 0 to a leading 1. For $n = 3$ this leads to the sequence $\langle (000)_2, (001)_2, (011)_2, (010)_2, (110)_2, (111)_2, (101)_2, (100)_2 \rangle$. The general solution uses recursion in conjunction with this reversing, and is presented below.

```
def gray_code(num_bits):
    if num_bits == 0:
        return [0]

    # These implicitly begin with 0 at bit-index (num_bits - 1).
    gray_code_num_bits_minus_1 = gray_code(num_bits - 1)
    # Now, add a 1 at bit-index (num_bits - 1) to all entries in
    # gray_code_num_bits_minus_1.
    leading_bit_one = 1 << (num_bits - 1)
    # Process in reverse order to achieve reflection of
    # gray_code_num_bits_minus_1.
    return gray_code_num_bits_minus_1 + [
        leading_bit_one | i for i in reversed(gray_code_num_bits_minus_1)
    ]


# Pythonic solution uses list comprehension.
def gray_code_pythonic(num_bits):
    result = [0]
    for i in range(num_bits):
        result += [x + 2**i for x in reversed(result)]
    return result
```

Assuming we operate on integers that fit within the size of the integer word, the time complexity $T(n)$ satisfies $T(n) = T(n-1) + O(2^{n-1})$. The time complexity is $O(2^n)$.

CHAPTER

# 16 Dynamic Programming

*The important fact to observe is that we have attempted to solve a maximization problem involving a particular value of x and a particular value of N by first solving the general problem involving an arbitrary value of x and an arbitrary value of N.*

— *"Dynamic Programming,"*
R. E. BELLMAN, 1957

DP is a general technique for solving optimization, search, and counting problems that can be decomposed into subproblems. You should consider using DP whenever you have to make choices to arrive at the solution, specifically, when the solution relates to subproblems.

Like divide-and-conquer, DP solves the problem by combining the solutions of multiple smaller problems, but what makes DP different is that the same subproblem may reoccur. Therefore, a key to making DP efficient is caching the results of intermediate computations. Problems whose solutions use DP are a popular choice for hard interview questions.

To illustrate the idea underlying DP, consider the problem of computing Fibonacci numbers. The first two Fibonacci numbers are 0 and 1. Successive numbers are the sums of the two previous numbers. The first few Fibonacci numbers are $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$. The Fibonacci numbers arise in many diverse applications—biology, data structure analysis, and parallel computing are some examples.

Mathematically, the $n$th Fibonacci number $F(n)$ is given by the equation $F(n) = F(n-1) + F(n-2)$, with $F(0) = 0$ and $F(1) = 1$. A function to compute $F(n)$ that recursively invokes itself has a time complexity that is exponential in $n$. This is because the recursive function computes some $F(i)$s repeatedly. Figure 16.1 on the next page graphically illustrates how the function is repeatedly called with the same arguments.

Caching intermediate results makes the time complexity for computing the $n$th Fibonacci number linear in $n$, albeit at the expense of $O(n)$ storage.

```
def fibonacci(n, cache={}):
    if n <= 1:
        return n
    elif n not in cache:
        cache[n] = fibonacci(n - 1) + fibonacci(n - 2)
    return cache[n]
```

Minimizing cache space is a recurring theme in DP. Now we show a program that computes $F(n)$ in $O(n)$ time and $O(1)$ space. Conceptually, in contrast to the above program, this program iteratively fills in the cache in a bottom-up fashion, which allows it to reuse cache storage to reduce the space complexity of the cache.

```
def fibonacci(n):
```

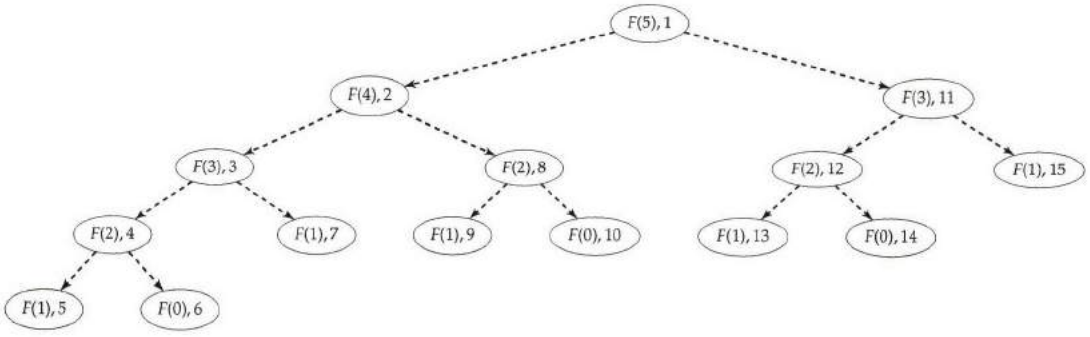



**Figure 16.1:** Tree of recursive calls when naively computing the 5th Fibonacci number, $F(5)$. Each node is a call: $F(x)$ indicates a call with argument $x$, and the italicized numbers on the right are the sequence in which calls take place. The children of a node are the subcalls made by that call. Note how there are 2 calls to $F(3)$, and 3 calls to each of $F(2)$, $F(1)$, and $F(0)$.

```
if n <= 1:
    return n

f_minus_2, f_minus_1 = 0, 1
for _ in range(1, n):
    f = f_minus_2 + f_minus_1
    f_minus_2, f_minus_1 = f_minus_1, f
return f_minus_1
```

The key to solving a DP problem efficiently is finding a way to break the problem into subproblems such that

- the original problem can be solved relatively easily once solutions to the subproblems are available, and
- these subproblem solutions are cached.

Usually, but not always, the subproblems are easy to identify.

Here is a more sophisticated application of DP. Consider the following problem: find the maximum sum over all subarrays of a given array of integer. As a concrete example, the maximum subarray for the array in Figure 16.2 starts at index 0 and ends at index 3.

| 904 | 40 | 523 | 12 | -335 | -385 | -124 | 481 | -31 |
|---|---|---|---|---|---|---|---|---|
| $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ | $A[4]$ | $A[5]$ | $A[6]$ | $A[7]$ | $A[8]$ |

**Figure 16.2:** An array with a maximum subarray sum of 1479.

The brute-force algorithm, which computes each subarray sum, has $O(n^3)$ time complexity—there are $\frac{n(n+1)}{2}$ subarrays, and each subarray sum takes $O(n)$ time to compute. The brute-force algorithm can be improved to $O(n^2)$, at the cost of $O(n)$ additional storage, by first computing $S[k] = \sum A[0,k]$ for all $k$. The sum for $A[i,j]$ is then $S[j] - S[i-1]$, where $S[-1]$ is taken to be 0.

Here is a natural divide-and-conquer algorithm. Take $m = \lfloor \frac{n}{2} \rfloor$ to be the middle index of $A$. Solve the problem for the subarrays $L = A[0,m]$ and $R = A[m+1, n-1]$. In addition to the answers for each, we also return the maximum subarray sum $l$ for a subarray ending at the last entry in $L$, and the maximum subarray sum $r$ for a subarray starting at the first entry of $R$. The maximum subarray

sum for $A$ is the maximum of $l + r$, the answer for $L$, and the answer for $R$. The time complexity analysis is similar to that for quicksort, and the time complexity is $O(n \log n)$. Because of off-by-one errors, it takes some time to get the program just right.

Now we will solve this problem by using DP. A natural thought is to assume we have the solution for the subarray $A[0, n-2]$. However, even if we knew the largest sum subarray for subarray $A[0, n-2]$, it does not help us solve the problem for $A[0, n-1]$. Instead we need to know the subarray amongst all subarrays $A[0, i]$, $i < n-1$, with the smallest subarray sum. The desired value is $S[n-1]$ minus this subarray's sum. We compute this value by iterating through the array. For each index $j$, the maximum subarray sum ending at $j$ is equal to $S[j] - \min_{k \leq j} S[k]$. During the iteration, we track the minimum $S[k]$ we have seen so far and compute the maximum subarray sum for each index. The time spent per index is constant, leading to an $O(n)$ time and $O(1)$ space solution. It is legal for all array entries to be negative, or the array to be empty. The algorithm handles these input cases correctly.

```
def find_maximum_subarray(A):
    min_sum = max_sum = 0
    for running_sum in itertools.accumulate(A):
        min_sum = min(min_sum, running_sum)
        max_sum = max(max_sum, running_sum - min_sum)
    return max_sum
```

## *Dynamic programming boot camp*

The programs presented in the introduction for computing the Fibonacci numbers and the maximum subarray sum are good illustrations of DP.

### 16.1 Count the number of score combinations

In an American football game, a play can lead to 2 points (safety), 3 points (field goal), or 7 points (touchdown, assuming the extra point). Many different combinations of 2, 3, and 7 point plays can make up a final score. For example, four combinations of plays yield a score of 12:

- 6 safeties ($2 \times 6 = 12$),
- 3 safeties and 2 field goals ($2 \times 3 + 3 \times 2 = 12$),
- 1 safety, 1 field goal and 1 touchdown ($2 \times 1 + 3 \times 1 + 7 \times 1 = 12$), and
- 4 field goals ($3 \times 4 = 12$).

Write a program that takes a final score and scores for individual plays, and returns the number of combinations of plays that result in the final score.

*Hint:* Count the number of combinations in which there are 0 $w_0$ plays, then 1 $w_0$ plays, etc.

**Solution:** We can gain some intuition by considering small scores. For example, a 9 point score can be achieved in the following ways:

- scoring 7 points, followed by a 2 point play,
- scoring 6 points, followed by a 3 point play, and
- scoring 2 points, followed by a 7 point play.

Generalizing, an $s$ point score can be achieved by an $s-2$ point score, followed by a 2 point play, an $s-3$ point score, followed by a 3 point play, or an $s-7$ point score, followed by a 7 point play. This gives us a mechanism for recursively enumerating all possible scoring sequences which lead to a given score. Note that different sequences may lead to the same score combination, e.g., a 2 point

Consider using DP whenever you have to **make choices** to arrive at the solution. Specifically, DP is applicable when you can construct a solution to the given instance from solutions to subinstances of smaller problems of the same kind.

In addition to optimization problems, DP is also **applicable** to **counting and decision problems**—any problem where you can express a solution recursively in terms of the same computation on smaller instances.

Although conceptually DP involves recursion, often for efficiency the cache is **built "bottom-up"**, i.e., iteratively. .

When DP is implemented recursively the cache is typically a dynamic data structure such as a hash table or a BST; when it is implemented iteratively the cache is usually a one- or multi-dimensional array.

To save space, **cache space** may be **recycled** once it is known that a set of entries will not be looked up again.

Sometimes, **recursion may out-perform a bottom-up DP** solution, e.g., when the solution is found early or subproblems can be **pruned** through bounding.
A common **mistake** in solving DP problems is trying to think of the recursive case by splitting the problem into **two equal halves**, *a la* quicksort, i.e., solve the subproblems for subarrays $A[0, \lfloor \frac{n}{2} \rfloor]$ and $A[\lfloor \frac{n}{2} \rfloor + 1, n]$ and combine the results. However, in most cases, these two subproblems are not sufficient to solve the original problem.

DP is based on **combining solutions** to subproblems to **yield a solution** to the original problem. However, for some problems DP will not work. For example, if you need to compute the longest path from City 1 to City 2 without repeating an intermediate city, and this longest path passes through City 3, then the subpaths from City 1 to City 3 and City 3 to City 2 may not be individually longest paths without repeated cities.

**Table 16.1:** Top Tips for Dynamic Programming

play followed by a 7 point play and a 7 point play followed by a 2 point play both lead to a final score of 9. A brute-force approach might be to enumerate these sequences, and count the distinct combinations within these sequences, e.g., by sorting each sequence and inserting into a hash table.

The time complexity is very high, since there may be a very large number of scoring sequences. Since all we care about are the combinations, a better approach is to focus on the number of combinations for each possible number of plays of a single type.

For example, if the final score is 12 and we are only allowed 2 point plays, there is exactly one way to get 12. Now suppose we are allowed both 2 and 3 point plays. Since all we care about are combinations, assume the 2 point plays come before the 3 point plays. We could have zero 2 point plays, for which there is one combination of 3 point plays, one 2 point play, for which there no combination of 3 point plays (since 3 does not evenly divide $12 - 2$, two 2 point plays, for which there no combination of 3 point plays (since 3 does not evenly $12 - 2 \times 2$), three 2 point plays, for which there is one combination of 3 point plays (since 3 evenly divides $12 - 2 \times 3$), etc. To count combinations when we add 7 point plays to the mix, we add the number of combinations of 2 and 3 that lead to 12 and to 5—these are the only scores from which 7 point plays can lead to 12.

Naively implemented, for the general case, i.e., individual play scores are $W[0], W[1], \ldots, W[n-1]$, and $s$ the final score, the approach outlined above has exponential complexity because it repeat-

edly solves the same problems. We can use DP to reduce its complexity. Let the 2D array $A[i][j]$ store the number of score combinations that result in a total of $j$, using individual plays of scores $W[0], W[1], \ldots, W[i-1]$. For example, $A[1][12]$ is the number of ways in which we can achieve a total of 12 points, using 2 and/or 3 point plays. Then $A[i+1][j]$ is simply $A[i][j]$ (no $W[i+1]$ point plays used to get to $j$), plus $A[i][j - W[i+1]]$ (one $W[i+1]$ point play), plus $A[i][j - 2W[i+1]]$ (two $W[i+1]$ point plays), etc.

The algorithm directly based on the above discussion consists of three nested loops. The first loop is over the total range of scores, and the second loop is over scores for individual plays. For a given $i$ and $j$, the third loop iterates over at most $j/W[i] + 1$ values. (For example, if $j = 10$, $i = 1$, and $W[i] = 3$, then the third loop examines $A[0][10], A[0][7], A[0][4], A[0][1]$.) Therefore number of iterations for the third loop is bounded by $s$. Hence, the overall time complexity is $O(sns) = O(s^2n)$ (first loop is to $n$, second is to $s$, third is bounded by $s$).

Looking more carefully at the computation for the row $A[i+1]$, it becomes apparent that it is not as efficient as it could be. As an example, suppose we are working with 2 and 3 point plays. Suppose we are done with 2 point plays. Let $A[0]$ be the row holding the result for just 2 point plays, i.e., $A[0][j]$ is the number of combinations of 2 point plays that result in a final score of $j$. The number of score combinations to get a final score of 12 when we include 3 point plays in addition to 2 point plays is $A[0][0] + A[0][3] + A[0][6] + A[0][9] + A[0][12]$. The number of score combinations to get a final score of 15 when we include 3 point plays in addition to 2 point plays is $A[0][0] + A[0][3] + A[0][6] + A[0][9] + A[0][12] + A[0][15]$. Clearly this repeats computation—$A[0][0] + A[0][3] + A[0][6] + A[0][9] + A[0][12]$ was computed when considering the final score of 12.

Note that $A[1][15] = A[0][15] + A[1][12]$. Therefore a better way to fill in $A[1]$ is as follows: $A[1][0] = A[0][0], A[1][1] = A[0][1], A[1][2] = A[0][2], A[1][3] = A[0][3] + A[1][0], A[1][4] = A[0][4] + A[1][1], A[1][5] = A[0][5] + A[1][2], \ldots$. Observe that $A[1][i]$ takes $O(1)$ time to compute—it's just $A[0][i] + A[1][i-3]$. See Figure 16.3 for an example table.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2** | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| **2,3** | 1 | 0 | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 2 | 2 | 2 | 3 |
| **2,3,7** | 1 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4 |

**Figure 16.3:** DP table for 2, 3, 7 point plays (rows) and final scores from 0 to 12 (columns). As an example, for 2, 3, 7 point plays and a total of 9, the entry is the sum of the entry for 2, 3 point plays and a total of 9 (no 7 point plays) and the entry for 2, 3, 7 point plays and a total of 2 (one additional 7 point play).

The code below implements the generalization of this approach.

```
def num_combinations_for_final_score(final_score, individual_play_scores):
    # One way to reach 0.
    num_combinations_for_score = [[1] + [0] * final_score
                                  for _ in individual_play_scores]
    for i in range(len(individual_play_scores)):
        for j in range(1, final_score + 1):
            without_this_play = (num_combinations_for_score[i - 1][j]
                                 if i >= 1 else 0)
            with_this_play = (
```

```
            num_combinations_for_score[i][j - individual_play_scores[i]]
            if j >= individual_play_scores[i] else 0)
        num_combinations_for_score[i][j] = (
            without_this_play + with_this_play)
    return num_combinations_for_score[-1][-1]
```

The time complexity is $O(sn)$ (two loops, one to $s$, the other to $n$) and the space complexity is $O(sn)$ (the size of the 2$D$ array).

**Variant:** Solve the same problem using $O(s)$ space.

**Variant:** Write a program that takes a final score and scores for individual plays, and returns the number of sequences of plays that result in the final score. For example, 18 sequences of plays yield a score of 12. Some examples are $\langle 2,2,2,3,3\rangle$, $\langle 2,3,2,2,3\rangle$, $\langle 2,3,7\rangle$, $\langle 7,3,2\rangle$.

**Variant:** Suppose the final score is given in the form $(s,s')$, i.e., Team 1 scored $s$ points and Team 2 scored $s'$ points. How would you compute the number of distinct scoring sequences which result in this score? For example, if the final score is (6,3) then Team 1 scores 3, Team 2 scores 3, Team 1 scores 3 is a scoring sequence which results in this score.

**Variant:** Suppose the final score is $(s,s')$. How would you compute the maximum number of times the team that lead could have changed? For example, if $s = 10$ and $s' = 6$, the lead could have changed 4 times: Team 1 scores 2, then Team 2 scores 3 (lead change), then Team 1 scores 2 (lead change), then Team 2 scores 3 (lead change), then Team 1 scores 3 (lead change) followed by 3.

**Variant:** You are climbing stairs. You can advance 1 to $k$ steps at a time. Your destination is exactly $n$ steps up. Write a program which takes as inputs $n$ and $k$ and returns the number of ways in which you can get to your destination. For example, if $n = 4$ and $k = 2$, there are five ways in which to get to the destination:

- four single stair advances,
- two single stair advances followed by a double stair advance,
- a single stair advance followed by a double stair advance followed by a single stair advance,
- a double stair advance followed by two single stairs advances, and
- two double stair advances.

## 16.2 Compute the Levenshtein distance

Spell checkers make suggestions for misspelled words. Given a misspelled string, a spell checker should return words in the dictionary which are close to the misspelled string.

In 1965, Vladimir Levenshtein defined the distance between two words as the minimum number of "edits" it would take to transform the misspelled word into a correct word, where a single edit is the *insertion*, *deletion*, or *substitution* of a single character. For example, the Levenshtein distance between "Saturday" and "Sundays" is 4—delete the first 'a' and 't', substitute 'r' by 'n' and insert the trailing 's'.

Write a program that takes two strings and computes the minimum number of edits needed to transform the first string into the second string.

*Hint:* Consider the same problem for prefixes of the two strings.

**Solution:** A brute-force approach would be to enumerate all strings that are distance $1,2,3,\ldots$ from the first string, stopping when we reach the second string. The number of strings may grow

enormously, e.g., if the first string is $n$ 0s and the second is $n$ 1s we will visit all of the $2^n$ possible bit strings from the first string before we reach the second string.

A better approach is to "prune" the search. For example, if the last character of the first string equals the last character of the second string, we can ignore this character. If they are different, we can focus on the initial portion of each string and perform a final edit step. (As we will soon see, this final edit step may be an insertion, deletion, or substitution.)

Let $a$ and $b$ be the length of strings $A$ and $B$, respectively. Let $E(A[0, a-1], B[0, b-1])$ be the Levenshtein distance between the strings $A$ and $B$. (Note that $A[0, a-1]$ is just $A$, but we prefer to write $A$ using the subarray notation for exposition; we do the same for $B$.)

We now make some observations:

- If the last character of $A$ equals the last character of $B$, then $E(A[0, a-1], B[0, b-1]) = E(A[0, a-2], B[0, b-2])$.
- If the last character of $A$ is not equal to the last character of $B$ then

$$E(A[0, a-1], B[0, b-1]) = 1 + \min \begin{pmatrix} E(A[0, a-2], B[0, b-2]), \\ E(A[0, a-1], B[0, b-2]), \\ E(A[0, a-2], B[0, b-1]) \end{pmatrix}$$

  The three terms correspond to transforming $A$ to $B$ by the following three ways:
  - Transforming $A[0, a-1]$ to $B[0, b-1]$ by transforming $A[0, a-2]$ to $B[0, b-2]$ and then substituting $A$'s last character with $B$'s last character.
  - Transforming $A[0, a-1]$ to $B[0, b-1]$ by transforming $A[0, a-1]$ to $B[0, b-2]$ and then adding $B$'s last character at the end.
  - Transforming $A[0, a-1]$ to $B[0, b-1]$ by transforming $A[0, a-2]$ to $B[0, b-1]$ and then deleting $A$'s last character.

These observations are quite intuitive. Their rigorous proof, which we do not give, is based on reordering steps in an arbitrary optimum solution for the original problem.

DP is a great way to solve this recurrence relation: cache intermediate results on the way to computing $E(A[0, a-1], B[0, b-1])$.

We illustrate the approach in Figure 16.4 on the facing page. This shows the $E$ values for "Carthorse" and "Orchestra". Uppercase and lowercase characters are treated as being different. The Levenshtein distance for these two strings is 8.

```
def levenshtein_distance(A, B):
    def compute_distance_between_prefixes(A_idx, B_idx):
        if A_idx < 0:
            # A is empty so add all of B's characters.
            return B_idx + 1
        elif B_idx < 0:
            # B is empty so delete all of A's characters.
            return A_idx + 1
        if distance_between_prefixes[A_idx][B_idx] == -1:
            if A[A_idx] == B[B_idx]:
                distance_between_prefixes[A_idx][B_idx] = (
                    compute_distance_between_prefixes(A_idx - 1, B_idx - 1))
            else:
                substitute_last = compute_distance_between_prefixes(
                    A_idx - 1, B_idx - 1)
                add_last = compute_distance_between_prefixes(A_idx - 1, B_idx)
                delete_last = compute_distance_between_prefixes(
```

```
                A_idx, B_idx - 1)
            distance_between_prefixes[A_idx][B_idx] = (
                1 + min(substitute_last, add_last, delete_last))
        return distance_between_prefixes[A_idx][B_idx]

    distance_between_prefixes = [[-1] * len(B) for _ in A]
    return compute_distance_between_prefixes(len(A) - 1, len(B) - 1)
```

The value $E(A[0, a-1], B[0, b-1])$ takes time $O(1)$ to compute once $E(A[0, k], B[0, l])$ is known for all $k < a$ and $l < b$. This implies $O(ab)$ time complexity for the algorithm. Our implementation uses $O(ab)$ space.

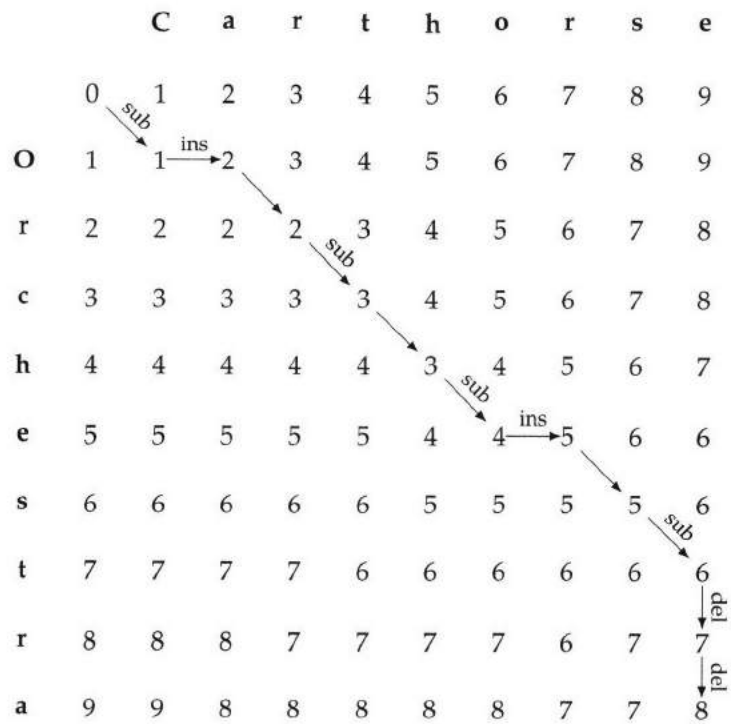



**Figure 16.4:** The $E$ table for "Carthorse" and "Orchestra".

**Variant:** Compute the Levenshtein distance using $O(\min(a, b))$ space and $O(ab)$ time.

**Variant:** Given $A$ and $B$ as above, compute a longest sequence of characters that is a subsequence of $A$ and of $B$. For example, the longest subsequence which is present in both strings in Figure 16.4 is $\langle r, h, s \rangle$.

**Variant:** Given a string $A$, compute the minimum number of characters you need to delete from $A$ to make the resulting string a palindrome.

**Variant:** Given a string $A$ and a regular expression $r$, what is the string in the language of the regular expression $r$ that is closest to $A$? The distance between strings is the Levenshtein distance specified above.

**Variant:** Define a string $t$ to be an interleaving of strings $s_1$ and $s_2$ if there is a way to interleave the characters of $s_1$ and $s_2$, keeping the left-to-right order of each, to obtain $t$. For example, if $s_1$

is "gtaa" and $s_2$ is "atc", then "gattaca" and "gtataac" can be formed by interleaving $s_1$ and $s_2$ but "gatacta" cannot. Design an algorithm that takes as input strings $s_1$, $s_2$ and $t$, and determines if $t$ is an interleaving of $s_1$ and $s_2$.

## 16.3 Count the number of ways to traverse a 2D array

In this problem you are to count the number of ways of starting at the top-left corner of a 2D array and getting to the bottom-right corner. All moves must either go right or down. For example, we show three ways in a $5 \times 5$ 2D array in Figure 16.5. (As we will see, there are a total of 70 possible ways for this example.)

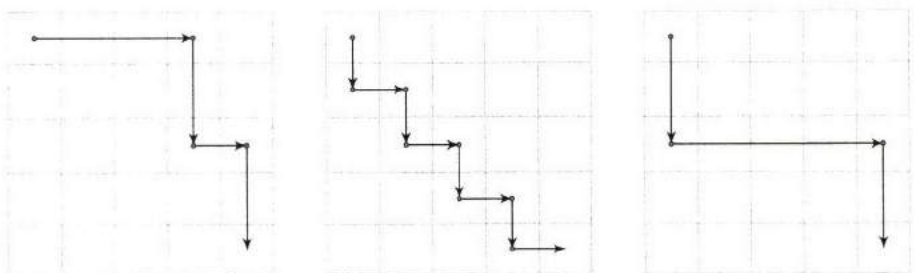


**Figure 16.5:** Paths through a 2D array.

Write a program that counts how many ways you can go from the top-left to the bottom-right in a 2D array.

*Hint:* If $i > 0$ and $j > 0$, you can get to $(i, j)$ from $(i-1, j)$ or $(i, j-1)$.

**Solution:** A brute-force approach is to enumerate all possible paths. This can be done using recursion. However, there is a combinatorial explosion in the number of paths, which leads to huge time complexity.

The problem statement asks for the number of paths, so we focus our attention on that. A key observation is that because paths must advance down or right, the number of ways to get to the bottom-right entry is the number of ways to get to the entry immediately above it, plus the number of ways to get to the entry immediately to its left. Let's treat the origin $(0, 0)$ as the top-left entry. Generalizing, the number of ways to get to $(i, j)$ is the number of ways to get to $(i-1, j)$ plus the number of ways to get to $(i, j-1)$. (If $i = 0$ or $j = 0$, there is only one way to get to $(i, j)$ from the origin.) This is the basis for a recursive algorithm to count the number of paths. Implemented naively, the algorithm has exponential time complexity—it repeatedly recurses on the same locations. The solution is to cache results. For example, the number of ways to get to $(i, j)$ for the configuration in Figure 16.5 is cached in a matrix as shown in Figure 16.6 on the facing page.

```
def number_of_ways(n, m):
    def compute_number_of_ways_to_xy(x, y):
        if x == y == 0:
            return 1

        if number_of_ways[x][y] == 0:
            ways_top = 0 if x == 0 else compute_number_of_ways_to_xy(x - 1, y)
            ways_left = 0 if y == 0 else compute_number_of_ways_to_xy(x, y - 1)
            number_of_ways[x][y] = ways_top + ways_left
```

```
    return number_of_ways[x][y]

number_of_ways = [[0] * m for _ in range(n)]
return compute_number_of_ways_to_xy(n - 1, m - 1)
```

The time complexity is $O(nm)$, and the space complexity is $O(nm)$, where $n$ is the number of rows and $m$ is the number of columns.

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 3 | 6 | 10 | 15 |
| 1 | 4 | 10 | 20 | 35 |
| 1 | 5 | 15 | 35 | 70 |

**Figure 16.6:** The number of ways to get from $(0,0)$ to $(i,j)$ for $0 \leq i,j \leq 4$.

A more analytical way of solving this problem is to use the fact that each path from $(0,0)$ to $(n-1,m-1)$ is a sequence of $m-1$ horizontal steps and $n-1$ vertical steps. There are $\binom{n+m-2}{n-1} = \binom{n+m-2}{m-1} = \frac{(n+m-2)!}{(n-1)!(m-1)!}$ such paths.

**Variant:** Solve the same problem using $O(\min(n,m))$ space.

**Variant:** Solve the same problem in the presence of obstacles, specified by a Boolean 2D array, where the presence of a true value represents an obstacle.

**Variant:** A fisherman is in a rectangular sea. The value of the fish at point $(i,j)$ in the sea is specified by an $n \times m$ 2D array $A$. Write a program that computes the maximum value of fish a fisherman can catch on a path from the upper leftmost point to the lower rightmost point. The fisherman can only move down or right, as illustrated in Figure 16.7.

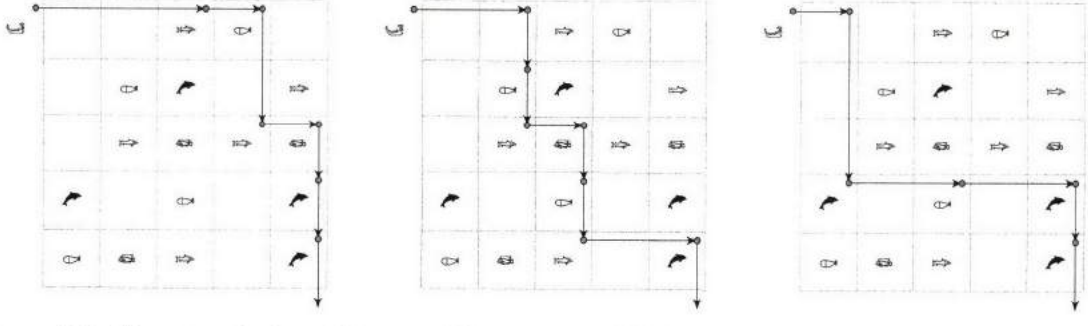


**Figure 16.7:** Alternate paths for a fisherman. Different types of fish have different values, which are known to the fisherman.

**Variant:** Solve the same problem when the fisherman can begin and end at any point. He must still move down or right. (Note that the value at $(i,j)$ may be negative.)

**Variant:** A decimal number is a sequence of digits, i.e., a sequence over $\{0,1,2,\ldots,9\}$. The sequence has to be of length 1 or more, and the first element in the sequence cannot be 0. Call a decimal

number $D$ *monotone* if $D[i] \leq D[i+1], 0 \leq i < |D|$. Write a program which takes as input a positive integer $k$ and computes the number of decimal numbers of length $k$ that are monotone.

**Variant:** Call a decimal number $D$, as defined above, *strictly monotone* if $D[i] < D[i+1], 0 \leq i < |D|$. Write a program which takes as input a positive integer $k$ and computes the number of decimal numbers of length $k$ that are strictly monotone.

### 16.4 Compute the binomial coefficients

The symbol $\binom{n}{k}$ is the short form for the expression $\frac{n(n-1)\cdots(n-k+1)}{k(k-1)\cdots(3)(2)(1)}$. It is the number of ways to choose a $k$-element subset from an $n$-element set. It is not obvious that the expression defining $\binom{n}{k}$ always yields an integer. Furthermore, direct computation of $\binom{n}{k}$ from this expression quickly results in the numerator or denominator overflowing if integer types are used, even if the final result fits in a 32-bit integer. If floats are used, the expression may not yield a 32-bit integer.

Design an efficient algorithm for computing $\binom{n}{k}$ which has the property that it never overflows if the final result fits in the integer word size.

*Hint:* Write an equation.

**Solution:** A brute-force approach would be to compute $n(n-1)\cdots(n-k+1)$, then $k(k-1)\cdots(3)(2)(1)$, and finally divide the former by the latter. As pointed out in the problem introduction, this can lead to overflows, even when the final result fits in the integer word size.

It is tempting to proceed by pairing terms in the numerator and denominator that have common factors and cancel them out. This approach is unsatisfactory because of the need to factor numbers, which itself is challenging.

A better approach is to avoid multiplications and divisions entirely. Fundamentally, the binomial coefficient counts the number of subsets of size $k$ in a set of size $n$. We could enumerate $k$-sized subsets of $\{0, 1, 2, \ldots, n-1\}$ sets using recursion, as in Solution 15.5 on Page 226. The idea is as follows. Consider the $n$th element in the initial set. A subset of size $k$ will either contain this element, or not contain it. This is the basis for a recursive enumeration—find all subsets of size $k-1$ amongst the first $n-1$ elements and add the $n$th element into these sets, and then find all subsets of size $k$ amongst the first $n-1$ elements. The union of these two subsets is all subsets of size $k$.

However, since all we care about is the *number* of such subsets, we can do much better complexity-wise. The recursive enumeration also implies that the binomial coefficient must satisfy the following formula:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

This identity yields a straightforward recursion for $\binom{n}{k}$. The base cases are $\binom{r}{r}$ and $\binom{r}{0}$, both of which are 1. The individual results from the subcalls are 32-bit integers and if $\binom{n}{k}$ can be represented by a 32-bit integer, they can too, so it is not possible for intermediate results to overflow.

For example, $\binom{5}{2} = \binom{4}{2} + \binom{4}{1}$. Expanding $\binom{4}{2}$, we get $\binom{4}{2} = \binom{3}{2} + \binom{3}{1}$. Expanding $\binom{3}{2}$, we get $\binom{3}{2} = \binom{2}{2} + \binom{2}{1}$. Note that $\binom{2}{2}$ is a base case, returning 1. Continuing this way, we get $\binom{4}{2}$, which is 6 and $\binom{4}{1}$, which is 4, so $\binom{5}{2} = 6 + 4 = 10$.

Naively implemented, the above recursion will have repeated subcalls with identical arguments and exponential time complexity. This can be avoided by caching intermediate results.

```
def compute_binomial_coefficient(n, k):
```

```
def compute_x_choose_y(x, y):
    if y in (0, x):
        return 1

    if x_choose_y[x][y] == 0:
        without_y = compute_x_choose_y(x - 1, y)
        with_y = compute_x_choose_y(x - 1, y - 1)
        x_choose_y[x][y] = without_y + with_y
    return x_choose_y[x][y]

x_choose_y = [[0] * (k + 1) for _ in range(n + 1)]
return compute_x_choose_y(n, k)
```

The number of subproblems is $O(nk)$ and once $\binom{n-1}{k}$ and $\binom{n-1}{k-1}$ are known, $\binom{n}{k}$ can be computed in $O(1)$ time, yielding an $O(nk)$ time complexity. The space complexity is also $O(nk)$; it can easily be reduced to $O(k)$.

### 16.5 Search for a sequence in a 2D array

Suppose you are given a 2D array of integers (the "grid"), and a 1D array of integers (the "pattern"). We say the pattern occurs in the grid if it is possible to start from some entry in the grid and traverse adjacent entries in the order specified by the pattern till all entries in the pattern have been visited. The entries adjacent to an entry are the ones directly above, below, to the left, and to the right, assuming they exist. For example, the entries adjacent to $(3,4)$ are $(3,3)$, $(3,5)$, $(2,4)$ and $(4,4)$. It is acceptable to visit an entry in the grid more than once.

As an example, if the grid is

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 5 \\ 5 & 6 & 7 \end{bmatrix}$$

and the pattern is $\langle 1,3,4,6 \rangle$, then the pattern occurs in the grid—consider the entries $\langle (0,0),(1,0),(1,1),(2,1) \rangle$. However, $\langle 1,2,3,4 \rangle$ does not occur in the grid.

Write a program that takes as arguments a 2D array and a 1D array, and checks whether the 1D array occurs in the 2D array.

*Hint:* Start with length 1 prefixes of the 1D array, then move on to length $2,3,\ldots$ prefixes.

**Solution:** A brute-force approach might be to enumerate all 1D subarrays of the 2D subarray. This has very high time complexity, since there are many possible subarrays. Its inefficiency stems from not using the target 1D subarray to guide the search.

Let the 2D array be $A$, and the 1D array be $S$. Here is a guided way to search for matches. Let's say we have a suffix of $S$ to match, and a starting point to match from. If the suffix is empty, we are done. Otherwise, for the suffix to occur from the starting point, the first entry of the suffix must equal the entry at the starting point, and the remainder of the suffix must occur starting at a point adjacent to the starting point.

For example, when searching for $\langle 1,3,4,6 \rangle$ starting at $(0,0)$, since we match 1 with $A[0][0]$, we would continue searching for $\langle 3,4,6 \rangle$ from $(0,1)$ (which fails immediately, since $A[0][1] \neq 3$) and from $(1,0)$. Since $A[0][1] = 3$, we would continue searching for $\langle 4,6 \rangle$ from $(0,1)$'s neighbors, i.e., from $(0,0)$ (which fails immediately), then from $(1,1)$ (which eventually leads to success).

In the program below, we cache intermediate results to avoid repeated calls to the recursion with identical arguments.

```
def is_pattern_contained_in_grid(grid, S):
    def is_pattern_suffix_contained_starting_at_xy(x, y, offset):
        if len(S) == offset:
            # Nothing left to complete.
            return True

        # Check if (x, y) lies outside the grid.
        if (0 <= x < len(grid) and 0 <= y < len(grid[x])
                and grid[x][y] == S[offset]
                and (x, y, offset) not in previous_attempts and any(
                    is_pattern_suffix_contained_starting_at_xy(
                        x + a, y + b, offset + 1)
                    for a, b in ((-1, 0), (1, 0), (0, -1), (0, 1)))):
            return True
        previous_attempts.add((x, y, offset))
        return False

    # Each entry in previous_attempts is a point in the grid and suffix of
    # pattern (identified by its offset). Presence in previous_attempts
    # indicates the suffix is not contained in the grid starting from that
    # point.
    previous_attempts = set()
    return any(
        is_pattern_suffix_contained_starting_at_xy(i, j, 0)
        for i in range(len(grid)) for j in range(len(grid[i])))
```

The complexity is $O(nm|S|)$, where $n$ and $m$ are the dimensions of $A$—we do a constant amount of work within each call to the match function, except for the recursive calls, and the number of calls is not more than the number of entries in the $2D$ array.

**Variant:** Solve the same problem when you cannot visit an entry in $A$ more than once.

**Variant:** Enumerate all solutions when you cannot visit an entry in $A$ more than once.

## 16.6 The knapsack problem

A thief breaks into a clock store. Each clock has a weight and a value, which are known to the thief. His knapsack cannot hold more than a specified combined weight. His intention is to take clocks whose total value is maximum subject to the knapsack's weight constraint.

His problem is illustrated in Figure 16.8 on the next page. If the knapsack can hold at most 130 ounces, he cannot take all the clocks. If he greedily chooses clocks, in decreasing order of value-to-weight ratio, he will choose $P, H, O, B, I$, and $L$ in that order for a total value of \$669. However, $\{H, J, O\}$ is the optimum selection, yielding a total value of \$695.

Write a program for the knapsack problem that selects a subset of items that has maximum value and satisfies the weight constraint. All items have integer weights and values. Return the value of the subset.

*Hint:* Greedy approaches are doomed.

**Solution:** Greedy strategies such as picking the most valuable clock, or picking the clock with maximum value-to-weight ratio, do not always give the optimum solution.

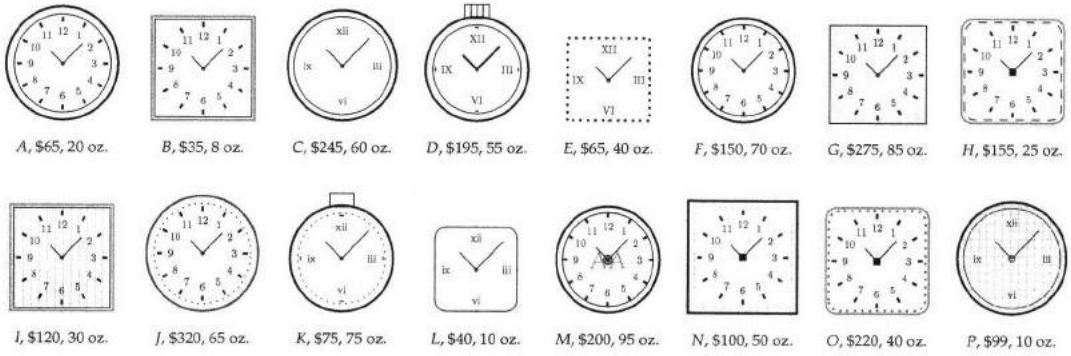



**Figure 16.8:** A clock store.

We can always get the optimum solution by considering all subsets, e.g., using Solution 15.4 on Page 225. This has exponential time complexity in the number of clocks. However, brute-force enumeration is wasteful because it ignores the weight constraint. For example, no subset that includes Clocks $F$ and $G$ can satisfy the weight constraint.

The better approach is to simultaneously consider the weight constraint. For example, what is the optimum solution if a given clock is chosen, and what is the optimum solution if that clock is not chosen? Each of these can be solved recursively with the implied weight constraint. For the given example, if we choose the Clock $A$, we need to find the maximum value of clocks from Clocks $B$–$P$ with a capacity of 130 – 20 and add \$65 to that value. If we do not choose Clock $A$, we need to find the maximum value of clocks from Clocks $B$–$P$ with a capacity of 130. The larger of these two values is the optimum solution.

More formally, let the clocks be numbered from 0 to $n-1$, with the weight and the value of the $i$th clock denoted by $w_i$ and $v_i$. Denote by $V[i][w]$ the optimum solution when we are restricted to Clocks $0, 1, 2, \ldots, i-1$ and can carry $w$ weight. Then $V[i][w]$ satisfies the following recurrence:

$$V[i][w] = \begin{cases} \max\left(V[i-1][w], V[i-1][w-w_i] + v_i\right), & \text{if } w_i \leq w; \\ V[i-1][w], & \text{otherwise.} \end{cases}$$

We take $i = 0$ or $w = 0$ as bases cases—for these, $V[i][w] = 0$.

We demonstrate the above algorithm in Figure 16.9 on the next page. Suppose there are four items,  ,  ,  ,  whose value and weight are given in Figure 16.9(a) on the following page. Then the corresponding $V$ table is given in Figure 16.9(b) on the next page. As an example, the extreme lower right entry is the maximum of the entry above (70) and 30 plus the entry for  , ,  with capacity 5 – 2 (50), i.e., 80.

```
Item = collections.namedtuple('Item', ('weight', 'value'))


def optimum_subject_to_capacity(items, capacity):
    # Returns the optimum value when we choose from items[:k + 1] and have a
    # capacity of available_capacity.
    def optimum_subject_to_item_and_capacity(k, available_capacity):
        if k < 0:
            # No items can be chosen.
            return 0

        if V[k][available_capacity] == -1:
```

| Item | Value | Weight |
|---|---|---|
|  | $60 | 5 oz. |
|  | $50 | 3 oz. |
|  | $70 | 4 oz. |
|  | $30 | 2 oz. |

(a) Value-weight table.

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
|  | 0 | 0 | 0 | 0 | 0 | 60 |
|  | 0 | 0 | 0 | 50 | 50 | 60 |
|  | 0 | 0 | 0 | 50 | 70 | 70 |
|  | 0 | 0 | 30 | 50 | 70 | 80 |

(b) Knapsack table for the items in (a). The columns correspond to capacities from 0 to 5.

**Figure 16.9:** A small example of the knapsack problem. The knapsackcapacity is 5 oz.

```
        without_curr_item = optimum_subject_to_item_and_capacity(
            k - 1, available_capacity)
        with_curr_item = (0 if available_capacity < items[k].weight else (
            items[k].value + optimum_subject_to_item_and_capacity(
                k - 1, available_capacity - items[k].weight)))
        V[k][available_capacity] = max(without_curr_item, with_curr_item)
    return V[k][available_capacity]

# V[i][j] holds the optimum value when we choose from items[:i + 1] and have
# a capacity of j.
V = [[-1] * (capacity + 1) for _ in items]
return optimum_subject_to_item_and_capacity(len(items) - 1, capacity)
```

The algorithm computes $V[n-1][w]$ in $O(nw)$ time, and uses $O(nw)$ space.

**Variant:** Solve the same problem using $O(w)$ space.

**Variant:** Solve the same problem using $O(C)$ space, where $C$ is the number of weights between 0 and $w$ that can be achieved. For example, if the weights are 100, 200, 200, 500, and $w = 853$, then $C = 9$, corresponding to the weights $0, 100, 200, 300, 400, 500, 600, 700, 800$.

**Variant:** Solve the fractional knapsack problem. In this formulation, the thief can take a fractional part of an item, e.g., by breaking it. Assume the value of a fraction of an item is that fraction times the value of the item.

**Variant:** In the "divide-the-spoils-fairly" problem, two thieves who have successfully completed a burglary want to know how to divide the stolen items into two groups such that the difference between the value of these two groups is minimized. For example, they may have stolen the clocks in Figure 16.8 on the preceding page, and would like to divide the clocks between them so that the difference of the dollar value of the two sets is minimized. For this instance, an optimum split is $\{A, G, J, M, O, P\}$ to one thief and the remaining to the other thief. The first set has value \$1179, and the second has value \$1180. An equal split is impossible, since the sum of the values of all the clocks is odd. Write a program to solve the divide-the-spoils-fairly problem.

**Variant:** Solve the divide-the-spoils-fairly problem with the additional constraint that the thieves have the same number of items.

**Variant:** The US President is elected by the members of the Electoral College. The number of electors per state and Washington, D.C., are given in Table 16.2 on the next page. All electors from each state

as well as Washington, D.C., cast their vote for the same candidate. Write a program to determine if a tie is possible in a presidential election with two candidates.

**Table 16.2:** Electoral college votes.

| State | Electors | State | Electors | State | Electors |
|---|---|---|---|---|---|
| Alabama | 9 | Louisiana | 8 | Ohio | 18 |
| Alaska | 3 | Maine | 4 | Oklahoma | 7 |
| Arizona | 11 | Maryland | 10 | Oregon | 7 |
| Arkansas | 6 | Massachusetts | 11 | Pennsylvania | 20 |
| California | 55 | Michigan | 16 | Rhode Island | 4 |
| Colorado | 9 | Minnesota | 10 | South Carolina | 9 |
| Connecticut | 7 | Mississippi | 6 | South Dakota | 3 |
| Delaware | 3 | Missouri | 10 | Tennessee | 11 |
| Florida | 29 | Montana | 3 | Texas | 38 |
| Georgia | 16 | Nebraska | 5 | Utah | 6 |
| Hawaii | 4 | Nevada | 6 | Vermont | 3 |
| Idaho | 4 | New Hampshire | 4 | Virginia | 13 |
| Illinois | 20 | New Jersey | 14 | Washington | 12 |
| Indiana | 11 | New Mexico | 5 | West Virginia | 5 |
| Iowa | 6 | New York | 29 | Wisconsin | 10 |
| Kansas | 6 | North Carolina | 15 | Wyoming | 3 |
| Kentucky | 8 | North Dakota | 3 | Washington, D.C. | 3 |

## 16.7 The bedbathandbeyond.com problem

Suppose you are designing a search engine. In addition to getting keywords from a page's content, you would like to get keywords from Uniform Resource Locators (URLs). For example, bedbathandbeyond.com yields the keywords "bed, bath, beyond, bat, hand": the first two coming from the decomposition "bed bath beyond" and the latter two coming from the decomposition "bed bat hand beyond".

Given a dictionary, i.e., a set of strings, and a name, design an efficient algorithm that checks whether the name is the concatenation of a sequence of dictionary words. If such a concatenation exists, return it. A dictionary word may appear more than once in the sequence. For example, "a", "man", "a", "plan", "a", "canal" is a valid sequence for "amanaplanacanal".

*Hint:* Solve the generalized problem, i.e., determine for each prefix of the name whether it is the concatenation of dictionary words.

**Solution:** The natural approach is to use recursion, i.e., find dictionary words that begin the name, and solve the problem recursively on the remainder of the name. Implemented naively, this approach has very high time complexity on some input cases, e.g., if the name is $N$ repetitions of "AB" followed by "C" and the dictionary words are "A", "B", and "AB" the time complexity is exponential in $N$. For example, for the string "ABABC", then we recurse on the substring "ABC" twice (once from the sequence "A", "B" and once from "AB").

The solution is straightforward—cache intermediate results. The cache keys are prefixes of the string. The corresponding value is a Boolean denoting whether the prefix can be decomposed into a sequence of valid words.

It's easy to determine if a string is a valid word—we simply store the dictionary in a hash table. A prefix of the given string can be decomposed into a sequence of dictionary words exactly if it is a dictionary word, or there exists a shorter prefix which can be decomposed into a sequence of dictionary words and the difference of the shorter prefix and the current prefix is a dictionary word.

For example, for "amanaplanacanal":

(1.) the prefix "a" has a valid decomposition (since "a" is a dictionary word),

(2.) the prefix "am" has a valid decomposition (since "am" is a dictionary word),

(3.) the prefix "ama" has a valid decomposition (since "a" has a valid decomposition, and "am" is a dictionary word), and

(4.) "aman" has a valid decomposition (since "am" has a valid decomposition and "an" is a dictionary word).

Skipping ahead,

5. "amanapl" does not have a valid decomposition (since none of "l", "pl", apl", etc. are dictionary words), and
6. "amanapla" does not have a valid decomposition (since the only dictionary word ending the string is "a" and "amanapl" does not have a valid decomposition).

The algorithm tells us if we can break a given string into dictionary words, but does not yield the words themselves. We can obtain the words with a little more book-keeping. Specifically, if a prefix has a valid decomposition, we record the length of the last dictionary word in the decomposition.

```
def decompose_into_dictionary_words(domain, dictionary):
    # When the algorithm finishes, last_length[i] != -1 indicates domain[:i +
    # 1] has a valid decomposition, and the length of the last string in the
    # decomposition is last_length[i].
    last_length = [-1] * len(domain)
    for i in range(len(domain)):
        # If domain[:i + 1] is a dictionary word, set last_length[i] to the
        # length of that word.
        if domain[:i + 1] in dictionary:
            last_length[i] = i + 1

        # If last_length[i] = -1 look for j < i such that domain[: j + 1] has a
        # valid decomposition and domain[j + 1:i + 1] is a dictionary word. If
        # so, record the length of that word in last_length[i].
        if last_length[i] == -1:
            for j in range(i):
                if last_length[j] != -1 and domain[j + 1:i + 1] in dictionary:
                    last_length[i] = i - j
                    break

    decompositions = []
    if last_length[-1] != -1:
        # domain can be assembled by dictionary words.
        idx = len(domain) - 1
        while idx >= 0:
            decompositions.append(domain[idx + 1 - last_length[idx]:idx + 1])
            idx -= last_length[idx]
        decompositions = decompositions[::-1]
    return decompositions
```

Let $n$ be the length of the input string $s$. For each $k < n$ we check for each $j < k$ whether the substring $s[j+1,k]$ is a dictionary word, and each such check requires $O(k-j)$ time. This implies the time complexity is $O(n^3)$. We can improve the time complexity as follows. Let $W$ be the length of the longest dictionary word. We can restrict $j$ to range from $k-W$ to $k-1$ without losing any decompositions, so the time complexity improves to $O(n^2W)$.

If we want all possible decompositions, we can store all possible values of $j$ that gives us a correct break with each position. Note that the number of possible decompositions can be exponential here. This is illustrated by the string "itsitsitsits...".

**Variant:** Palindromic decompositions were described in Problem 15.7 on Page 228. Observe every string $s$ has at least one palindromic decomposition, which is the trivial one consisting of the individual characters. For example, if $s$ is "0204451881" then "0", "2", "0", "4", "4", "5", "1", "8", "8", "1" is such a trivial decomposition. The minimum decomposition of $s$ is "020", "44", "5", "1881". How would you compute a palindromic decomposition of a string $s$ that uses a minimum number of substrings?

### 16.8 Find the minimum weight path in a triangle

A sequence of integer arrays in which the $n$th array consists of $n$ entries naturally corresponds to a triangle of numbers. See Figure 16.10 for an example.

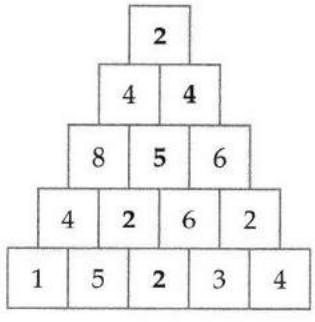



**Figure 16.10:** A number triangle.

Define a path in the triangle to be a sequence of entries in the triangle in which adjacent entries in the sequence correspond to entries that are adjacent in the triangle. The path must start at the top, descend the triangle continuously, and end with an entry on the bottom row. The weight of a path is the sum of the entries.

Write a program that takes as input a triangle of numbers and returns the weight of a minimum weight path. For example, the minimum weight path for the number triangle in Figure 16.10 is shown in bold face, and its weight is 15.

*Hint:* What property does the prefix of a minimum weight path have?

**Solution:** A brute-force approach is to enumerate all possible paths. Although these paths are quite easy to enumerate, there are $2^{n-1}$ such paths in a number triangle with $n$ rows.

A far better way is to consider entries in the $i$th row. For any such entry, if you look at the minimum weight path ending at it, the part of the path that ends at the previous row must also be a minimum weight path. This gives us a DP solution. We iteratively compute the minimum weight of a path ending at each entry in Row $i$. Since after we complete processing Row $i$, we do not need the results for Row $i-1$ to process Row $i+1$, we can reuse storage.

```
def minimum_path_weight(triangle):
    min_weight_to_curr_row = [0]
    for row in triangle:
```

```
        min_weight_to_curr_row = [
            row[j] +
            min(min_weight_to_curr_row[max(j - 1, 0)],
                min_weight_to_curr_row[min(j, len(min_weight_to_curr_row) - 1)])
            for j in range(len(row))
        ]
    return min(min_weight_to_curr_row)
```

The time spent per element is $O(1)$ and there are $1 + 2 + \cdots + n = n(n+1)/2$ elements, implying an $O(n^2)$ time complexity. The space complexity is $O(n)$.

## 16.9 Pick up coins for maximum gain

In the pick-up-coins game, an even number of coins are placed in a line, as in Figure 16.11. Two players take turns at choosing one coin each—they can only choose from the two coins at the ends of the line. The game ends when all the coins have been picked up. The player whose coins have the higher total value wins. A player cannot pass his turn.


**Figure 16.11:** A row of coins.

Design an efficient algorithm for computing the maximum total value for the starting player in the pick-up-coins game.

*Hint:* Relate the best play for the first player to the best play for the second player.

**Solution:** First of all, note that greedily selecting the maximum of the two end coins does not yield the best solution. If the coins are $5, 25, 10, 1$, if the first player is greedy and chooses 5, the second player can pick up the 25. Now the first player will choose the 10, and the second player gets the 1, so the first player has a total of 15 and the second player has a total of 26. A better move for the first player is picking up 1. Now the second player is forced to expose the 25, so the first player will achieve 26.

The drawback of greedy selection is that it does not consider the opportunities created for the second player. Intuitively, the first player wants to balance selecting high coins with minimizing the coins available to the second player.

The second player is assumed to play the best move he possibly can. Therefore, the second player will choose the coin that maximizes *his* revenue. Call the sum of the coins selected by a player his revenue. Let $R(a, b)$ be the maximum revenue a player can get when it is his turn to play, and the coins remaining on the table are at indices $a$ to $b$, inclusive. Let $C$ be an array representing the line of coins, i.e., $C[i]$ is the value of the $i$th coin. If the first player selects the coin at $a$, since the second player plays optimally, the first player will end up with a total revenue of $C[a] + S(a+1, b) - R(a+1, b)$, where $S(a, b)$ is the sum of the coins from positions $a$ to $b$, inclusive. If he selects the coin at $b$, he will end up with a total revenue of $C[b] + S(a, b-1) - R(a, b-1)$. Since the first player wants to maximize revenue, he chooses the greater of the two, i.e., $R(a, b) = \max(C[a]+S(a+1, b)-R(a+1, b), C[b]+S(a, b-1)-R(a, b-1))$. This recursion for $R$ can be solved easily using DP.

Now we present a slightly different recurrence for $R$. Since the second player seeks to maximize his revenue, and the total revenue is a constant, it is equivalent for the the second player to move

so as to minimize the first player's revenue. Therefore, $R(a, b)$ satisfies the following equations:

$$R(a,b) = \begin{cases} \max\left( \begin{array}{l} C[a] + \min\left( \begin{array}{l} R(a+2,b), \\ R(a+1,b-1) \end{array} \right), \\ C[b] + \min\left( \begin{array}{l} R(a+1,b-1), \\ R(a,b-2) \end{array} \right) \end{array} \right), & \text{if } a \leq b; \\ 0, & \text{otherwise.} \end{cases}$$

In essence, the strategy is to minimize the maximum revenue the opponent can gain. The benefit of this "min-max" recurrence for $R(a, b)$, compared to our first formulation, is that it does not require computing $S(a + 1, b)$ and $S(a, b - 1)$.

For the coins $\langle 10, 25, 5, 1, 10, 5 \rangle$, the optimum revenue for the first player is 31. Some of subproblems encountered include computing the optimum revenue for $\langle 10, 25 \rangle$ (which is 25), $\langle 5, 1 \rangle$ (which is 5), and $\langle 5, 1, 10, 5 \rangle$ (which is 15).

For the coins in Figure 16.11 on the facing page, the maximum revenue for the first player is 140¢, i.e., whatever strategy the second player uses, the first player can guarantee a revenue of at least 140¢. In contrast, if both players always pick the more valuable of the two coins available to them, the first player will get only 120¢.

In the program below, we solve for $R$ using DP.

```
def maximum_revenue(coins):
    def compute_maximum_revenue_for_range(a, b):
        if a > b:
            # No coins left.
            return 0

        if maximum_revenue_for_range[a][b] == 0:
            max_revenue_a = coins[a] + min(
                compute_maximum_revenue_for_range(a + 2, b),
                compute_maximum_revenue_for_range(a + 1, b - 1))
            max_revenue_b = coins[b] + min(
                compute_maximum_revenue_for_range(a + 1, b - 1),
                compute_maximum_revenue_for_range(a, b - 2))
            maximum_revenue_for_range[a][b] = max(max_revenue_a, max_revenue_b)
        return maximum_revenue_for_range[a][b]

    maximum_revenue_for_range = [[0] * len(coins) for _ in coins]
    return compute_maximum_revenue_for_range(0, len(coins) - 1)
```

There are $O(n^2)$ possible arguments for $R(a, b)$, where $n$ is the number of coins, and the time spent to compute $R$ from previously computed values is $O(1)$. Hence, $R$ can be computed in $O(n^2)$ time.

## 16.10 Count the number of moves to climb stairs

You are climbing stairs. You can advance 1 to $k$ steps at a time. Your destination is exactly $n$ steps up.

Write a program which takes as inputs $n$ and $k$ and returns the number of ways in which you can get to your destination. For example, if $n = 4$ and $k = 2$, there are five ways in which to get to the destination:

- four single stair advances,

- two single stair advances followed by a double stair advance,
- a single stair advance followed by a double stair advance followed by a single stair advance,
- a double stair advance followed by two single stairs advances, and
- two double stair advances.

*Hint:* How many ways are there in which you can take the last step?

**Solution:** A brute-force enumerative solution does not make sense, since there are an exponential number of possibilities to consider.

Since the first advance can be one step, two steps,..., $k$ steps, and all of these lead to different ways to get to the top, we can write the following equation for the number of steps $F(n,k)$:

$$F(n,k) = \sum_{i=1}^{k} F(n-i,k)$$

For the working example, $F(4,2) = F(4-2,2) + F(4-1,2)$. Recursing, $F(4-2,2) = F(4-2-2,2) + F(4-2-1,2)$. Both $F(0,2)$ and $F(1,2)$ are base-cases, with a value of 1, so $F(4-2,2) = 2$. Continuing with $F(4-1,2)$, $F(4-1,2) = F(4-1-2,2) + F(4-1-1,2)$. The first term is a base-case, with a value of 1. The second term has already been computed—its value is 2. Therefore, $F(4-1,2) = 3$, and $F(4,2) = 3+2$.

In the program below, we cache values of $F(i,k)$, $0 \leq i \leq n$ to improve time complexity.

```
def number_of_ways_to_top(top, maximum_step):
    def compute_number_of_ways_to_h(h):
        if h <= 1:
            return 1

        if number_of_ways_to_h[h] == 0:
            number_of_ways_to_h[h] = sum(
                compute_number_of_ways_to_h(h - i)
                for i in range(1, min(maximum_step, h) + 1))
        return number_of_ways_to_h[h]

    number_of_ways_to_h = [0] * (top + 1)
    return compute_number_of_ways_to_h(top)
```

We take $O(k)$ time to fill in each entry, so the total time complexity is $O(kn)$. The space complexity is $O(n)$.

## 16.11 The pretty printing problem

Consider the problem of laying out text using a fixed width font. Each line can hold no more than a fixed number of characters. Words on a line are to be separated by exactly one blank. Therefore, we may be left with whitespace at the end of a line (since the next word will not fit in the remaining space). This whitespace is visually unappealing.

Define the *messiness* of the end-of-line whitespace as follows. The messiness of a single line ending with $b$ blank characters is $b^2$. The total messiness of a sequence of lines is the sum of the messinesses of all the lines. A sequence of words can be split across lines in different ways with different messiness, as illustrated in Figure 16.12 on the next page.

```
I have inserted a large number of␣␣␣
new examples from the papers for the
Mathematical Tripos during the last␣
twenty years, which should be useful
to Cambridge students.␣␣␣␣␣␣␣␣␣␣␣␣␣␣
```

(a) Messiness = $3^2 + 0^2 + 1^2 + 0^2 + 14^2 = 206$.

```
I have inserted a large number␣␣␣␣␣␣␣
of new examples from the papers␣␣␣␣␣␣
for the Mathematical Tripos during␣␣␣
the last twenty years, which should␣
be useful to Cambridge students.␣␣␣␣␣
```

(b) Messiness = $6^2 + 5^2 + 2^2 + 1^2 + 4^2 = 82$.

**Figure 16.12:** Two layouts for the same sequence of words; the line length $L$ is 36.

Given text, i.e., a string of words separated by single blanks, decompose the text into lines such that no word is split across lines and the messiness of the decomposition is minimized. Each line can hold no more than a specified number of characters.

*Hint:* Focus on the last word and the last line.

**Solution:** A greedy approach is to fit as many words as possible in each line. However, some experimentation shows this is suboptimal. See Figure 16.13 for an example. In essence, the greedy algorithm does not spread words uniformly across the lines, which is the key requirement of the problem. Adding a new word may necessitate moving earlier words.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Line 1 | a | | b | | c |
| Line 2 | d | | | | |

(a) Greedy placement: Line 1 has a messiness of $0^2$, and Line 2 has a messiness of $4^2$.

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| Line 1 | a | | b | | |
| Line 2 | c | | d | | |

(b) Optimum placement: Line 1 has a messiness of $2^2$, and Line 2 has a messiness of $2^2$.

**Figure 16.13:** Assuming the text is "a b c d" and the line length is 5, placing words greedily results in a total messiness of 16, as seen in (a), whereas the optimum placement has a total messiness of 8, as seen in (b).

Suppose we want to find the optimum placement for the $i$th word. As we have just seen, we cannot trivially extend the optimum placement for the first $i-1$ words to an optimum placement for the $i$th word. Put another way, the placement that results by removing the $i$th word from an optimum placement for the first $i$ words is not always an optimum placement for the first $i-1$ words. However, what we can say is that if in the optimum placement for the $i$th word the last line consists of words $j, j+1, \ldots, i$, then in this placement, the first $j-1$ words must be placed optimally.

In an optimum placement of the first $i$ words, the last line consists of *some* subset of words ending in the $i$th word. Furthermore, since the first $i$ words are assumed to be optimally placed, the placement of words on the lines prior to the last one must be optimum. Therefore, we can write a recursive formula for the minimum messiness, $M(i)$, when placing the first $i$ words. Specifically, $M(i)$, equals $\min_{j \le i} f(j,i) + M(j-1)$, where $f(j,i)$ is the messiness of a single line consisting of words $j$ to $i$ inclusive.

We give an example of this approach in Figure 16.14 on the next page. The optimum placement of "aaa bbb c d ee" is shown in Figure 16.14(a) on the following page. The optimum placement of "aaa bbb c d ee ff" is shown in Figure 16.14(b) on the next page.

To determine the optimum placement for "aaa bbb c d ee ff ggggggg", we consider two cases—the final line is "ff ggggggg", and the final line is "ggggggg". (No more cases are possible, since we

cannot fit "ee ff ggggggg" on a single line.)

If the final line is "ff ggggggg", then the "aaa bb c d ee" must be placed as in Figure 16.14(a). If the final line is "ggggggg", then "aaa bbb c d ee ff" must be placed as in Figure 16.14(b). These two cases are shown in Figure 16.14(c) and Figure 16.14(d), respectively. Comparing the two, we see Figure 16.14(c) has the lower messiness and is therefore optimum.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| a | a | a | | b | b | b | | | | |
| c | | d | | e | e | | | | | |

(a) Optimum placement for "aaa bbb c d ee".

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| a | a | a | | b | b | b | | | | |
| c | | d | | e | e | | f | f | | |

(b) Optimum placement for "aaa bbb c d ee ff".

| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| a | a | a | | b | b | b | | | | |
| c | | d | | e | e | | f | f | | |
| g | g | g | g | g | g | g | | | | |

(c) Optimum placement for "aaa bbb c d ee ff ggggggg" when the final line is "ggggggg".

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| a | a | a | | b | b | b | | | | |
| c | | d | | e | e | | | | | |
| f | f | | g | g | g | g | g | g | g | |

(d) Optimum placement for "aaa bbb c d ee ff ggggggg" when the final line is "ff ggggggg".

**Figure 16.14:** Solving the pretty printing problem for text "aaa bb c d ee ff ggggggg" and line length 11.

The recursive computation has exponential complexity, because it visits identical subproblems repeatedly. The solution is to cache the values for $M$.

```
def minimum_messiness(words, line_length):
    num_remaining_blanks = line_length - len(words[0])
    # min_messiness[i] is the minimum messiness when placing words[0:i + 1].
    min_messiness = ([num_remaining_blanks**2] + [float('inf')] *
                     (len(words) - 1))
    for i in range(1, len(words)):
        num_remaining_blanks = line_length - len(words[i])
        min_messiness[i] = min_messiness[i - 1] + num_remaining_blanks**2
        # Try adding words[i - 1], words[i - 2], ...
        for j in reversed(range(i)):
            num_remaining_blanks -= len(words[j]) + 1
            if num_remaining_blanks < 0:
                # Not enough space to add more words.
                break
            first_j_messiness = 0 if j - 1 < 0 else min_messiness[j - 1]
            current_line_messiness = num_remaining_blanks**2
            min_messiness[i] = min(min_messiness[i],
                                   first_j_messiness + current_line_messiness)
    return min_messiness[-1]
```

Let $L$ be the line length. Then there can certainly be no more than $L$ words on a line, so the amount of time spent processing each word is $O(L)$. Therefore, if there are $n$ words, the time complexity is $O(nL)$. The space complexity is $O(n)$ for the cache.

**Variant:** Solve the same problem when the messiness is the sum of the messinesses of all but the last line.

**Variant:** Suppose the messiness of a line ending with $b$ blank characters is defined to be $b$. Can you solve the messiness minimization problem in $O(n)$ time and $O(1)$ space?

The problem of finding the longest nondecreasing subsequence in a sequence of integers has implications to many disciplines, including string matching and analyzing card games. As a concrete instance, the length of a longest nondecreasing subsequence for the array in Figure 16.15 is 4. There are multiple longest nondecreasing subsequences, e.g., $\langle 0, 4, 10, 14\rangle$ and $\langle 0, 2, 6, 9\rangle$. Note that elements of non-decreasing subsequence are not required to immediately follow each other in the original sequence.

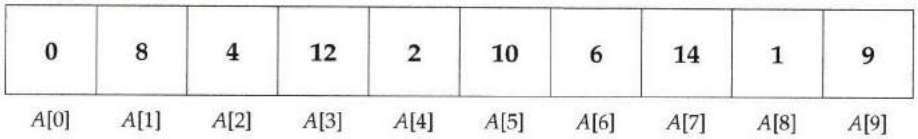



**Figure 16.15:** An array whose longest nondecreasing subsequences are of length 4.

Write a program that takes as input an array of numbers and returns the length of a longest nondecreasing subsequence in the array.

*Hint:* Express the longest nondecreasing subsequence ending at an entry in terms of the longest nondecreasing subsequence appearing in the subarray consisting of preceding elements.

**Solution:** A brute-force approach would be to enumerate all possible subsequences, testing each one for being nondecreasing. Since there are $2^n$ subsequences of an array of length $n$, the time complexity would be huge. Some heuristic pruning can be applied, but the program grows very cumbersome.

If we have processed the initial set of entries of the input array, intuitively this should help us when processing the next entry. For the given example, if we know the lengths of the longest nondecreasing subsequences that end at Indices $0, 1, \ldots, 5$ and we want to know the longest nondecreasing subsequence that ends at Index 6, we simply look for the longest subsequence ending at an entry in $A[0, 5]$ whose value is less than or equal to $A[6]$. For Index 6 in the example in Figure 16.15, there are two such longest subsequences, namely the ones ending at Index 2 and Index 4, both of which yield nondecreasing subsequences of length 3 ending at $A[6]$.

Generalizing, define $L[i]$ to be the length of the longest nondecreasing subsequence of $A$ that ends at and includes Index $i$. As an example, for the array in Figure 16.15, $L[6] = 3$. The longest nondecreasing subsequence that ends at Index $i$ is of length 1 (if $A[i]$ is smaller than all preceding entries) or has some element, say at Index $j$, as its penultimate entry, in which case the subsequence restricted to $A[0, j]$ must be the longest subsequence ending at Index $j$. Based on this, $L[i]$ is either 1 (if $A[i]$ is less than all previous entries), or $1 + \max\{L[j] | j < i \text{ and } A[j] \leq A[i]\}$.

We can use this relationship to compute $L$, recursively or iteratively. If we want the sequence as well, for each $i$, in addition to storing the length of the nondecreasing sequence ending at $i$, we store the index of the last element of the subsequence that we extended to get the value assigned to $L[i]$.

Applying this algorithm to the example in Figure 16.15, we compute $L$ as follows:

(1.) $L[0] = 1$ (since there are no entries before Entry 0)
(2.) $L[1] = 1 + \max(L[0]) = 2$ (since $A[0] \leq A[1]$)
(3.) $L[2] = 1 + \max(L[0]) = 2$ (since $A[0] \leq A[2]$ and $A[1] > A[2]$)
(4.) $L[3] = 1 + \max(L[0], L[1], L[2]) = 3$ (since $A[0], A[1]$ and $A[2] \leq A[3]$)
(5.) $L[4] = 1 + \max(L[0]) = 2$ (since $A[0] \leq A[4], A[1], A[2]$ and $A[3] > A[4]$)

(6.) $L[5] = 1 + \max(L[0], L[1], L[2], L[4]) = 3$ (since $A[0], A[1], A[2], A[4] \leq A[5]$ and $A[3] > A[5]$)
(7.) $L[6] = 1 + \max(L[0], L[2], L[4]) = 3$ (since $A[0], A[2], A[4] \leq A[6]$ and $A[1], A[3], A[5] > A[6]$)
(8.) $L[7] = 1 + \max(L[0], L[1], L[2], L[3], L[4], L[5], L[6]) = 4$ (since $A[0]$,$A[1]$,$A[2]$, $A[3]$, $A[4]$, $A[5]$, $A[6] \leq A[7]$)
(9.) $L[8] = 1 + \max(L[0]) = 2$ (since $A[0] \leq A[8]$ and $A[1]$,$A[2]$, $A[3]$, $A[4]$, $A[5]$, $A[6]$, $A[7] > A[8]$)
(10.) $L[9] = 1 + \max(L[0], L[1], L[2], L[4], L[6], L[8]) = 4$ (since $A[0]$, $A[1]$, $A[2]$, $A[4]$, $A[6]$, $A[8] \leq A[9]$ and $A[3]$, $A[5]$, $A[7] > A[9]$)

Therefore the maximum length of a longest nondecreasing subsequence is 4. There are multiple longest nondecreasing sequences of length 4, e.g., $\langle 0, 8, 12, 14\rangle$, $\langle 0, 2, 6, 9\rangle$, $\langle 0, 4, 6, 9\rangle$.

Here is an iterative implementation of the algorithm.

```
def longest_nondecreasing_subsequence_length(A):
    # max_length[i] holds the length of the longest nondecreasing subsequence
    # of A[:i + 1].
    max_length = [1] * len(A)
    for i in range(1, len(A)):
        max_length[i] = max(1 + max(
            [max_length[j] for j in range(i)
             if A[i] >= A[j]], default=0), max_length[i])
    return max(max_length)
```

The time complexity is $O(n^2)$ (each $L[i]$ takes $O(n)$ time to compute), and the space complexity is $O(n)$ (to store $L$).

**Variant:** Write a program that takes as input an array of numbers and returns a longest nondecreasing subsequence in the array.

**Variant:** Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ to be *alternating* if $a_i < a_{i+1}$ for even $i$ and $a_i > a_{i+1}$ for odd $i$. Given an array of numbers $A$ of length $n$, find a longest subsequence $\langle i_0, \ldots, i_{k-1}\rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}]\rangle$ is alternating.

**Variant:** Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ to be *weakly alternating* if no three consecutive terms in the sequence are increasing or decreasing. Given an array of numbers $A$ of length $n$, find a longest subsequence $\langle i_0, \ldots, i_{k-1}\rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}]\rangle$ is weakly alternating.

**Variant:** Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ to be *convex* if $a_i < \frac{a_{i-1}+a_{i+1}}{2}$, for $1 \leq i \leq n-2$. Given an array of numbers $A$ of length $n$, find a longest subsequence $\langle i_0, \ldots, i_{k-1}\rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}]\rangle$ is convex.

**Variant:** Define a sequence of numbers $\langle a_0, a_1, \ldots, a_{n-1}\rangle$ to be *bitonic* if there exists $k$ such that $a_i < a_{i+1}$, for $0 \leq i < k$ and $a_i > a_{i+1}$, for $k \leq i < n-1$. Given an array of numbers $A$ of length $n$, find a longest subsequence $\langle i_0, \ldots, i_{k-1}\rangle$ such that $\langle A[i_0], A[i_1], \ldots, A[i_{k-1}]\rangle$ is bitonic.

**Variant:** Define a sequence of points in the plane to be *ascending* if each point is above and to the right of the previous point. How would you find a maximum ascending subset of a set of points in the plane?

**Variant:** Compute the longest nondecreasing subsequence in $O(n \log n)$ time.

CHAPTER

# 17 Greedy Algorithms and Invariants

*The intended function of a program, or part of a program, can be specified by making general assertions about the values which the relevant variables will take* after *execution of the program.*

— *"An Axiomatic Basis for Computer Programming,"* C. A. R. HOARE, 1969

### *Greedy algorithms*

A greedy algorithm is an algorithm that computes a solution in steps; at each step the algorithm makes a decision that is locally optimum, and it never changes that decision.

A greedy algorithm does not necessarily produce the optimum solution. As example, consider making change for 48 pence in the old British currency where the coins came in 30, 24, 12, 6, 3, and 1 pence denominations. Suppose our goal is to make change using the smallest number of coins. The natural greedy algorithm iteratively chooses the largest denomination coin that is less than or equal to the amount of change that remains to be made. If we try this for 48 pence, we get three coins—30 + 12 + 6. However, the optimum answer would be two coins—24 + 24.

In its most general form, the coin changing problem is NP-hard, but for some coinages, the greedy algorithm is optimum—e.g., if the denominations are of the form $\{1, r, r^2, r^3\}$. (An *ad hoc* argument can be applied to show that the greedy algorithm is also optimum for US coinage.) The general problem can be solved in pseudo-polynomial time using DP in a manner similar to Problem 16.6 on Page 246.

Sometimes, there are multiple greedy algorithms for a given problem, and only some of them are optimum. For example, consider $2n$ cities on a line, half of which are white, and the other half are black. Suppose we need to pair white with black cities in a one-to-one fashion so that the total length of the road sections required to connect paired cities is minimized. Multiple pairs of cities may share a single section of road, e.g., if we have the pairing $(0, 4)$ and $(1, 2)$ then the section of road between Cities 0 and 4 can be used by Cities 1 and 2.

The most straightforward greedy algorithm for this problem is to scan through the white cities, and, for each white city, pair it with the closest unpaired black city. This algorithm leads to suboptimum results. Consider the case where white cities are at 0 and at 3 and black cities are at 2 and at 5. If the white city at 3 is processed first, it will be paired with the black city at 2, forcing the cities at 0 and 5 to pair up, leading to a road length of 5. In contrast, the pairing of cities at 0 and 2, and 3 and 5 leads to a road length of 4.

A slightly more sophisticated greedy algorithm does lead to optimum results: iterate through all the cities in left-to-right order, pairing each city with the nearest unpaired city of opposite color. Note that the pairing for the first city must be optimum, since if it were to be paired with any other city, we could always change its pairing to be with the nearest black city without adding any road. This observation can be used in an inductive proof of overall optimality.

### *Greedy algorithms boot camp*

For US currency, wherein coins take values 1, 5, 10, 25, 50, 100 cents, the greedy algorithm for making change results in the minimum number of coins. Here is an implementation of this algorithm. Note that once it selects the number of coins of a particular value, it never changes that selection; this is the hallmark of a greedy algorithm.

```
def change_making(cents):
    COINS = [100, 50, 25, 10, 5, 1]
    num_coins = 0
    for coin in COINS:
        num_coins += cents // coin
        cents %= coin
    return num_coins
```

We perform 6 iterations, and each iteration does a constant amount of computation, so the time complexity is $O(1)$.

| |
|---|
| A greedy algorithm is often the right choice for an **optimization problem** where there's a natural set of **choices to select from**. |
| It's often easier to conceptualize a greedy algorithm recursively, and then **implement** it using iteration for higher performance. |
| Even if the greedy approach does not yield an optimum solution, it can give insights into the optimum algorithm, or serve as a heuristic. |
| Sometimes the correct greedy algorithm is **not obvious**. |

**Table 17.1:** Top Tips for Greedy Algorithms

## 17.1 Compute an optimum assignment of tasks

We consider the problem of assigning tasks to workers. Each worker must be assigned exactly two tasks. Each task takes a fixed amount of time. Tasks are independent, i.e., there are no constraints of the form "Task 4 cannot start before Task 3 is completed." Any task can be assigned to any worker.

We want to assign tasks to workers so as to minimize how long it takes before all tasks are completed. For example, if there are 6 tasks whose durations are $5, 2, 1, 6, 4, 4$ hours, then an optimum assignment is to give the first two tasks (i.e., the tasks with duration 5 and 2) to one worker, the next two (1 and 6) to another worker, and the last two tasks (4 and 4) to the last worker. For this assignment, all tasks will finish after $\max(5 + 2, 1 + 6, 4 + 4) = 8$ hours.

Design an algorithm that takes as input a set of tasks and returns an optimum assignment.

*Hint:* What additional task should be assigned to the worker who is assigned the longest task?

**Solution:** Simply enumerating all possible sets of pairs of tasks is not feasible—there are too many of them. (The precise number assignments is $\binom{n}{2}\binom{n-2}{2}\binom{n-4}{2}\cdots\binom{4}{2}\binom{2}{2} = n!/2^{n/2}$, where $n$ is the number of tasks.)

Instead we should look more carefully at the structure of the problem. Extremal values are important—the task that takes the longest needs the most help. In particular, it makes sense to pair the task with longest duration with the task of shortest duration. This intuitive observation can be understood by looking at any assignment in which a longest task is not paired with a shortest task.

By swapping the task that the longest task is currently paired with with the shortest task, we get an assignment which is at least as good.

Note that we are not claiming that the time taken for the optimum assignment is the sum of the maximum and minimum task durations. Indeed this may not even be the case, e.g., if the two longest duration tasks are close to each other in duration and the shortest duration task takes much less time than the second shortest task. As a concrete example, if the task durations are $1, 8, 9, 10$, the optimum delay is $8 + 9 = 17$, not $1 + 10$.

In summary, we sort the set of task durations, and pair the shortest, second shortest, third shortest, etc. tasks with the longest, second longest, third longest, etc. tasks. For example, if the durations are $5, 2, 1, 6, 4, 4$, then on sorting we get $1, 2, 4, 4, 5, 6$, and the pairings are $(1, 6)$, $(2, 5)$, and $(4, 4)$.

```
PairedTasks = collections.namedtuple('PairedTasks', ('task_1', 'task_2'))


def optimum_task_assignment(task_durations):
    task_durations.sort()
    return [
        PairedTasks(task_durations[i], task_durations[-i])
        for i in range(len(task_durations) // 2)
    ]
```

The time complexity is dominated by the time to sort, i.e., $O(n \log n)$.

## 17.2 Schedule to minimize waiting time

A database has to respond to a set of client SQL queries. The service time required for each query is known in advance. For this application, the queries must be processed by the database one at a time, but can be done in any order. The time a query waits before its turn comes is called its waiting time.

Given service times for a set of queries, compute a schedule for processing the queries that minimizes the total waiting time. Return the minimum waiting time. For example, if the service times are $\langle 2, 5, 1, 3 \rangle$, if we schedule in the given order, the total waiting time is $0 + (2) + (2 + 5) + (2 + 5 + 1) = 17$. If however, we schedule queries in order of decreasing service times, the total waiting time is $0 + (5) + (5 + 3) + (5 + 3 + 2) = 23$. As we will see, for this example, the minimum waiting time is 10.

*Hint:* Focus on extreme values.

**Solution:** We can solve this problem by enumerating all schedules and picking the best one. The complexity is very high—$O(n!)$ to be precise, where $n$ is the number of queries.

Intuitively, it makes sense to serve the short queries first. The justification is as follows. Since the service time of each query adds to the waiting time of all queries remaining to be processed, if we put a slow query before a fast one, by swapping the two we improve the waiting time for all queries between the slow and fast query, and do not change the waiting time for the other queries. We do increase the waiting time of the slow query, but this cancels out with the decrease in the waiting time of the fast query. Hence, we should sort the queries by their service time and then process them in the order of nondecreasing service time.

For the given example, the best schedule processes queries in increasing order of service times. It has a total waiting time of $0 + (1) + (1 + 2) + (1 + 2 + 3) = 10$. Note that scheduling queries with longer service times, which we gave as an earlier example, is the worst approach.

```
def minimum_total_waiting_time(service_times):
    # Sort the service times in increasing order.
    service_times.sort()
    total_waiting_time = 0
    for i, service_time in enumerate(service_times):
        num_remaining_queries = len(service_times) - (i + 1)
        total_waiting_time += service_time * num_remaining_queries
    return total_waiting_time
```

The time complexity is dominated by the time to sort, i.e., $O(n \log n)$.

### 17.3 The interval covering problem

Consider a foreman responsible for a number of tasks on the factory floor. Each task starts at a fixed time and ends at a fixed time. The foreman wants to visit the floor to check on the tasks. Your job is to help him minimize the number of visits he makes. In each visit, he can check on all the tasks taking place at the time of the visit. A visit takes place at a fixed time, and he can only check on tasks taking place at exactly that time. For example, if there are tasks at times $[0,3], [2,6], [3,4], [6,9]$, then visit times $0, 2, 3, 6$ cover all tasks. A smaller set of visit times that also cover all tasks is $3, 6$. In the abstract, you are to solve the following problem.

You are given a set of closed intervals. Design an efficient algorithm for finding a minimum sized set of numbers that covers all the intervals.

*Hint:* Think about extremal points.

**Solution:** Note that we can restrict our attention to numbers which are endpoints without losing optimality. A brute-force approach might be to enumerate every possible subset of endpoints, checking for each one if it covers all intervals, and if so, determining if it has a smaller size than any previous such subset. Since a set of size $k$ has $2^k$ subsets, the time complexity is very high.

We could simply return the left end point of each interval, which is fast to compute but, as the example in the problem statement showed, may not be the minimum number of visit times. Similarly, always greedily picking an endpoint which covers the most intervals may also lead to suboptimum results, e.g., consider the six intervals $[1,2], [2,3], [3,4], [2,3], [3,4], [4,5]$. The point 3 appears in four intervals, more than any other point. However, if we choose 3, we do not cover $[1,2]$ and $[4,5]$, so we need two additional points to cover all six intervals. If we pick 2 and 4, each individually covers just three intervals, but combined they cover all six.

It is a good idea to focus on extreme cases. In particular, consider the interval that ends first, i.e., the interval whose right endpoint is minimum. To cover it, we must pick a number that appears in it. Furthermore, we should pick its right endpoint, since any other intervals covered by a number in the interval will continue to be covered if we pick the right endpoint. (Otherwise the interval we began with cannot be the interval that ends first.) Once we choose that endpoint, we can remove all other covered intervals, and continue with the remaining set.

The above observation leads to the following algorithm. Sort all the intervals, comparing on right endpoints. Select the first interval's right endpoint. Iterate through the intervals, looking for the first one not covered by this right endpoint. As soon as such an interval is found, select its right endpoint and continue the iteration.

For the given example, $[1,2], [2,3], [3,4], [2,3], [3,4], [4,5]$, after sorting on right endpoints we get $[1,2], [2,3], [2,3], [3,4], [3,4], [4,5]$. The leftmost right endpoint is 2, which covers the first three intervals. Therefore, we select 2, and as we iterate, we see it covers $[1,2], [2,3], [2,3]$. When we get

to [3,4], we select its right endpoint, i.e., 4. This covers [3,4],[3,4],[4,5]. There are no remaining intervals, so {2,4} is a minimum set of points covering all intervals.

```
Interval = collections.namedtuple('Interval', ('left', 'right'))


def find_minimum_visits(intervals):
    # Sort intervals based on the right endpoints.
    intervals.sort(key=operator.attrgetter('right'))
    last_visit_time, num_visits = float('-inf'), 0
    for interval in intervals:
        if interval.left > last_visit_time:
            # The current right endpoint, last_visit_time, will not cover any
            # more intervals.
            last_visit_time = interval.right
            num_visits += 1
    return num_visits
```

Since we spend $O(1)$ time per index, the time complexity after the initial sort is $O(n)$, where $n$ is the number of intervals. Therefore, the time taken is dominated by the initial sort, i.e., $O(n \log n)$.

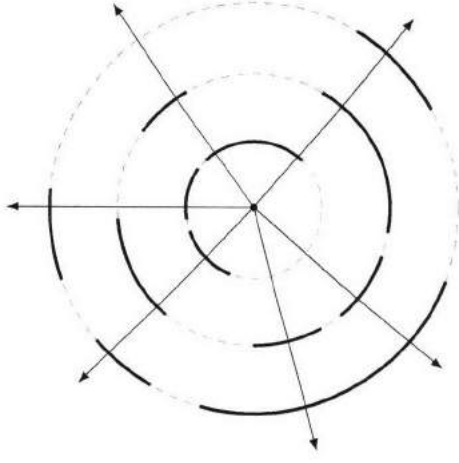


**Figure 17.1:** An instance of the minimum ray covering problem, with 12 partially overlapping arcs. Arcs have been drawn at different distances for illustration. For this instance, six cameras are sufficient, corresponding to the six rays.

**Variant:** You are responsible for the security of a castle. The castle has a circular perimeter. A total of $n$ robots patrol the perimeter—each robot is responsible for a closed connected subset of the perimeter, i.e., an arc. (The arcs for different robots may overlap.) You want to monitor the robots by installing cameras at the center of the castle that look out to the perimeter. Each camera can look along a ray. To save cost, you would like to minimize the number of cameras. See Figure 17.1 for an example.

**Variant:** There are a number of points in the plane that you want to observe. You are located at the point (0, 0). You can rotate about this point, and your field-of-view is a fixed angle. Which direction should you face to maximize the number of visible points?

### *Invariants*

A common approach to designing an efficient algorithm is to use invariants. Briefly, an invariant is a condition that is true during execution of a program. This condition may be on the values of the

variables of the program, or on the control logic. A well-chosen invariant can be used to rule out potential solutions that are suboptimal or dominated by other solutions.

For example, binary search, maintains the invariant that the space of candidate solutions contains all possible solutions as the algorithm executes.

Sorting algorithms nicely illustrate algorithm design using invariants. Intuitively, selection sort is based on finding the smallest element, the next smallest element, etc. and moving them to their right place. More precisely, we work with successively larger subarrays beginning at index 0, and preserve the invariant that these subarrays are sorted, their elements are less than or equal to the remaining elements, and the entire array remains a permutation of the original array.

As a more sophisticated example, consider Solution 14.7 on Page 208, specifically the $O(k)$ algorithm for generating the first $k$ numbers of the form $a + b\sqrt{2}$. The key idea there is to process these numbers in sorted order. The queues in that code maintain multiple invariants: queues are sorted, duplicates are never present, and the separation between elements is bounded.

### *Invariants boot camp*

Suppose you were asked to write a program that takes as input a sorted array and a given target value and determines if there are two entries in the array that add up to that value. For example, if the array is $\langle -2, 1, 2, 4, 7, 11 \rangle$, then there are entries adding to 6, to 0, and to 13, but not to 10.

The brute-force algorithm for this problem consists of a pair of nested for loops. Its complexity is $O(n^2)$, where $n$ is the length of the input array. A faster approach is to add each element of the array to a hash table, and test for each element $e$ if $K - e$, where $K$ is the target value, is present in the hash table. While reducing time complexity to $O(n)$, this approach requires $O(n)$ additional storage for the hash.

The most efficient approach uses invariants: maintain a subarray that is guaranteed to hold a solution, if it exists. This subarray is initialized to the entire array, and iteratively shrunk from one side or the other. The shrinking makes use of the sortedness of the array. Specifically, if the sum of the leftmost and the rightmost elements is less than the target, then the leftmost element can never be combined with some element to obtain the target. A similar observation holds for the rightmost element.

```
def has_two_sum(A, t):
    i, j = 0, len(A) - 1

    while i <= j:
        if A[i] + A[j] == t:
            return True
        elif A[i] + A[j] < t:
            i += 1
        else:  # A[i] + A[j] > t.
            j -= 1
    return False
```

The time complexity is $O(n)$, where $n$ is the length of the array. The space complexity is $O(1)$, since the subarray can be represented by two variables.

## 17.4 The 3-sum problem

Design an algorithm that takes as input an array and a number, and determines if there are three entries in the array (not necessarily distinct) which add up to the specified number. For example, if the array is $\langle 11, 2, 5, 7, 3 \rangle$ then there are three entries in the array which add up to 21 (3, 7, 11 and

| Identifying the right invariant is an art. The key strategy to determine whether to use an invariant when designing an algorithm is to work on **small examples** to hypothesize the invariant.<br><br>Often, the invariant is a subset of the set of input space, e.g,. a subarray. |
|---|

**Table 17.2:** Top Tips for Invariants

5, 5, 11). (Note that we can use 5 twice, since the problem statement said we can use the same entry more than once.) However, no three entries add up to 22.

*Hint:* How would you check if a given array entry can be added to two more entries to get the specified number?

**Solution:** The brute-force algorithm is to consider all possible triples, e.g., by three nested for-loops iterating over all entries. The time complexity is $O(n^3)$, where $n$ is the length of the array, and the space complexity is $O(1)$.

Let $A$ be the input array and $t$ the specified number. We can improve the time complexity to $O(n^2)$ by storing the array entries in a hash table first. Then we iterate over pairs of entries, and for each $A[i] + A[j]$ we look for $t - (A[i] + A[j])$ in the hash table. The space complexity now is $O(n)$.

We can avoid the additional space complexity by first sorting the input. Specifically, sort $A$ and for each $A[i]$, search for indices $j$ and $k$ such that $A[j] + A[k] = t - A[i]$. We can do each such search in $O(n \log n)$ time by iterating over $A[j]$ values and doing binary search for $A[k]$.

We can improve the time complexity to $O(n)$ by starting with $A[0] + A[n-1]$. If this equals $t - A[i]$, we're done. Otherwise, if $A[0] + A[n-1] < t - A[i]$, we move to $A[1] + A[n-1]$—there is no chance of $A[0]$ pairing with any other entry to get $t - A[i]$ (since $A[n-1]$ is the largest value in $A$). Similarly, if $A[0] + A[n-1] > t - A[i]$, we move to $A[0] + A[n-2]$. This approach eliminates an entry in each iteration, and spends $O(1)$ time in each iteration, yielding an $O(n)$ time bound to find $A[j]$ and $A[k]$ such that $A[j] + A[k] = t - A[i]$, if such entries exist. The invariant is that if two elements which sum to the desired value exist, they must lie within the subarray currently under consideration.

For the given example, after sorting the array is $\langle 2, 3, 5, 7, 11 \rangle$. For entry $A[0] = 2$, to see if there are $A[j]$ and $A[k]$ such that $A[0]+A[j]+A[k] = 21$, we search for two entries that add up to $21-2 = 19$.

The code for this approach is shown below.

```
def has_three_sum(A, t):
    A.sort()
    # Finds if the sum of two numbers in A equals to t - a.
    return any(has_two_sum(A, t - a) for a in A)
```

The additional space needed is $O(1)$, and the time complexity is the sum of the time taken to sort, $O(n \log n)$, and then to run the $O(n)$ algorithm to find a pair in a sorted array that sums to a specified value, which is $O(n^2)$ overall.

**Variant:** Solve the same problem when the three elements must be distinct. For example, if $A = \langle 5, 2, 3, 4, 3 \rangle$ and $t = 9$, then $A[2] + A[2] + A[2]$ is not acceptable, $A[2] + A[2] + A[4]$ is not acceptable, but $A[1] + A[2] + A[3]$ and $A[1] + A[3] + A[4]$ are acceptable.

**Variant:** Solve the same problem when $k$, the number of elements to sum, is an additional input.

**Variant:** Write a program that takes as input an array of integers $A$ and an integer $T$, and returns a 3-tuple $(A[p], A[q], A[r])$ where $p, q, r$ are all distinct, minimizing $|T - (A[p] + A[q] + A[r])|$, and

$A[p] \leq A[r] \leq A[s]$.

**Variant:** Write a program that takes as input an array of integers $A$ and an integer $T$, and returns the number of 3-tuples $(p, q, r)$ such that $A[p] + A[q] + A[r] \leq T$ and $A[p] \leq A[q] \leq A[r]$.

### 17.5 Find the majority element

Several applications require identification of elements in a sequence which occur more than a specified fraction of the total number of elements in the sequence. For example, we may want to identify the users using excessive network bandwidth or IP addresses originating the most Hypertext Transfer Protocol (HTTP) requests. Here we consider a simplified version of this problem.

You are reading a sequence of strings. You know *a priori* that more than half the strings are repetitions of a single string (the "majority element") but the positions where the majority element occurs are unknown. Write a program that makes a single pass over the sequence and identifies the majority element. For example, if the input is $\langle b, a, c, a, a, b, a, a, c, a \rangle$, then $a$ is the majority element (it appears in 6 out of the 10 places).

*Hint:* Take advantage of the existence of a majority element to perform elimination.

**Solution:** The brute-force approach is to use a hash table to record the repetition count for each distinct element. The time complexity is $O(n)$, where $n$ is the number of elements in the input, but the space complexity is also $O(n)$.

Randomized sampling can be used to identify a majority element with high probability using less storage, but is not exact.

The intuition for a better algorithm is as follows. We can group entries into two subgroups—those containing the majority element, and those that do not hold the majority element. Since the first subgroup is given to be larger in size than the second, if we see two entries that are different, at most one can be the majority element. By discarding both, the difference in size of the first subgroup and second subgroup remains the same, so the majority of the remaining entries remains unchanged.

The algorithm then is as follows. We have a candidate for the majority element, and track its count. It is initialized to the first entry. We iterate through remaining entries. Each time we see an entry equal to the candidate, we increment the count. If the entry is different, we decrement the count. If the count becomes zero, we set the next entry to be the candidate.

Here is a mathematical justification of the approach. Let's say the majority element occurred $m$ times out of $n$ entries. By the definition of majority element, $\frac{m}{n} > \frac{1}{2}$. At most one of the two distinct entries that are discarded can be the majority element. Hence, after discarding them, the ratio of the number of remaining majority elements to the total number of remaining elements is either $\frac{m}{(n-2)}$ (neither discarded element was the majority element) or $\frac{(m-1)}{(n-2)}$ (one discarded element was the majority element). It is simple to verify that if $\frac{m}{n} > \frac{1}{2}$, then both $\frac{m}{(n-2)} > \frac{1}{2}$ and $\frac{(m-1)}{(n-2)} > \frac{1}{2}$.

For the given example, $\langle b, a, c, a, a, b, a, a, c, a \rangle$, we initialize the candidate to $b$. The next element, $a$ is different from the candidate, so the candidate's count goes to 0. Therefore, we pick the next element $c$ to be the candidate, and its count is 1. The next element, $a$, is different so the count goes back to 0. The next element is $a$, which is the new candidate. The subsequent $b$ decrements the count to 0. Therefore the next element, $a$, is the new candidate, and it has a nonzero count till the end.

```
def majority_search(input_stream):
    candidate, candidate_count = None, 0
```

```
for it in input_stream:
    if candidate_count == 0:
        candidate, candidate_count = it, candidate_count + 1
    elif candidate == it:
        candidate_count += 1
    else:
        candidate_count -= 1
return candidate
```

Since we spend $O(1)$ time per entry, the time complexity is $O(n)$. The additional space complexity is $O(1)$.

The code above assumes a majority word exists in the sequence. If no word has a strict majority, it still returns a word from the stream, albeit without any meaningful guarantees on how common that word is. We could check with a second pass whether the returned word was a majority. Similar ideas can be used to identify words that appear more than $n/k$ times in the sequence, as discussed in Problem 24.33 on Page 399.

## 17.6 The gasup problem

In the gasup problem, a number of cities are arranged on a circular road. You need to visit all the cities and come back to the starting city. A certain amount of gas is available at each city. The amount of gas summed up over all cities is equal to the amount of gas required to go around the road once. Your gas tank has unlimited capacity. Call a city *ample* if you can begin at that city with an empty tank, refill at it, then travel through all the remaining cities, refilling at each, and return to the ample city, without running out of gas at any point. See Figure 17.2 for an example.

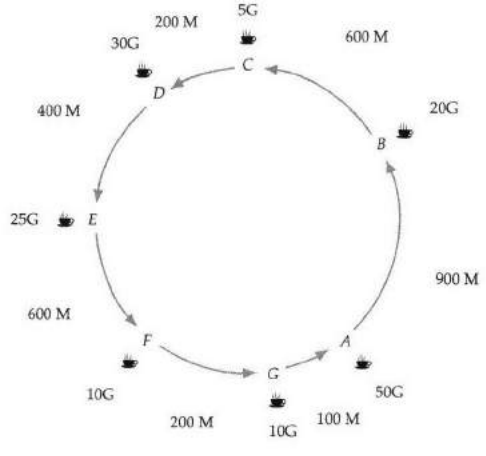



**Figure 17.2:** The length of the circular road is 3000 miles, and your vehicle gets 20 miles per gallon. The distance noted at each city is how far it is from the next city. For this configuration, we can begin with no gas at City $D$, and complete the circuit successfully, so $D$ is an ample city.

Given an instance of the gasup problem, how would you efficiently compute an ample city? You can assume that there exists an ample city.

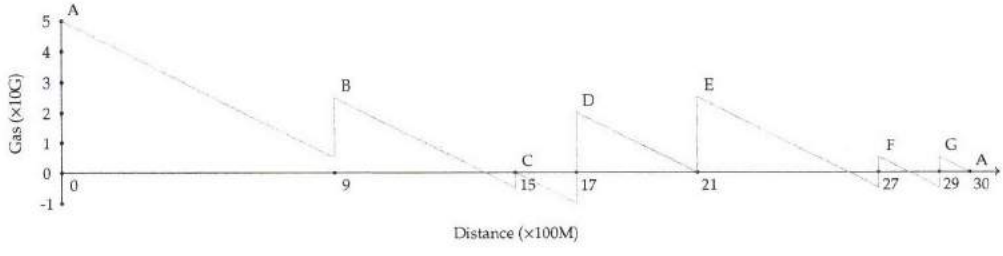
*Hint:* Think about starting with more than enough gas to complete the circuit without gassing up. Track the amount of gas as you perform the circuit, gassing up at each city.

**Solution:** The brute-force approach is to simulate the traversal from each city. This approach has time $O(n^2)$ time complexity, where $n$ is the number of cities.

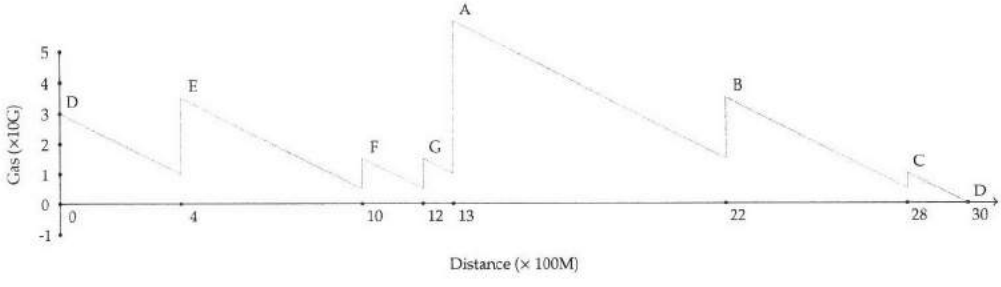
Greedy approaches, e.g., finding the city with the most gas, the city closest to the next city, or the city with best distance-to-gas ratio, do not work. For the given example, $A$ has the most gas, but you cannot get to $C$ starting from $A$. The city $G$ is closest to the next city ($A$), and has the lowest distance-to-gas ratio (100/10) but it cannot get to $D$.

We can gain insight by looking at the graph of the amount of gas as we perform the traversal. See Figure 17.3 for an example. The amount of gas in the tank could become negative, but we ignore the physical impossibility of that for now. These graphs are the same up to a translation about the Y-axis and a cyclic shift about the X-axis.

In particular, consider a city where the amount of gas in the tank is minimum when we enter that city. Observe that it does not depend where we begin from—because graphs are the same up to translation and shifting, a city that is minimum for one graph will be a minimum city for all graphs. Let $z$ be a city where the amount of gas in the tank before we refuel at that city is minimum. Now suppose we pick $z$ as the starting point, with the gas present at $z$. Since we never have less gas than we started with at $z$, and when we return to $z$ we have 0 gas (since it's given that the total amount of gas is just enough to complete the traversal) it means we can complete the journey without running out of gas. Note that the reasoning given above demonstrates that there always exists an ample city.




(a) Gas vs. distance, starting at $A$.




(b) Gas vs. distance, starting at $D$.

**Figure 17.3:** Gas as a function of distance for different starting cities for the configuration in Figure 17.2 on the preceding page.

The computation to determine $z$ can be easily performed with a single pass over all the cities simulating the changes to amount of gas as we advance.

```
MPG = 20


# gallons[i] is the amount of gas in city i, and distances[i] is the
# distance city i to the next city.
def find_ample_city(gallons, distances):
    remaining_gallons = 0
```

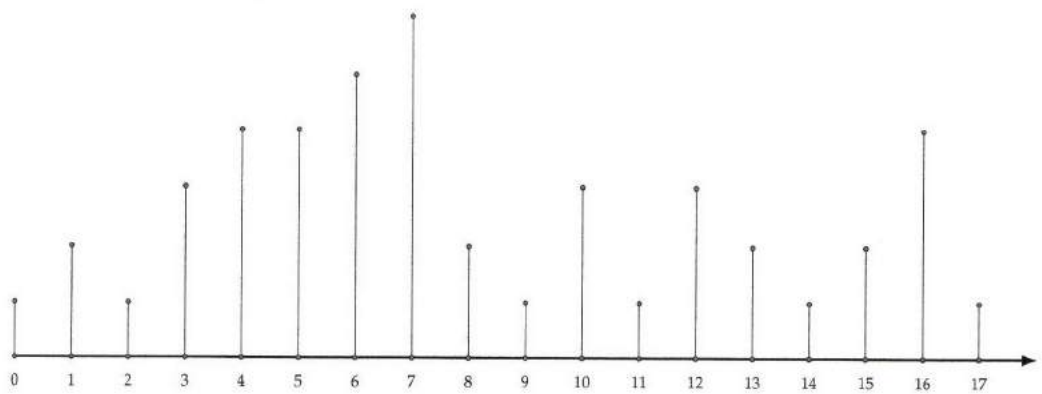
```
CityAndRemainingGas = collections.namedtuple('CityAndRemainingGas',
                                             ('city', 'remaining_gallons'))
city_remaining_gallons_pair = CityAndRemainingGas(0, 0)
num_cities = len(gallons)
for i in range(1, num_cities):
    remaining_gallons += gallons[i - 1] - distances[i - 1] // MPG
    if remaining_gallons < city_remaining_gallons_pair.remaining_gallons:
        city_remaining_gallons_pair = CityAndRemainingGas(
            i, remaining_gallons)
return city_remaining_gallons_pair.city
```

The time complexity is $O(n)$, and the space complexity is $O(1)$.

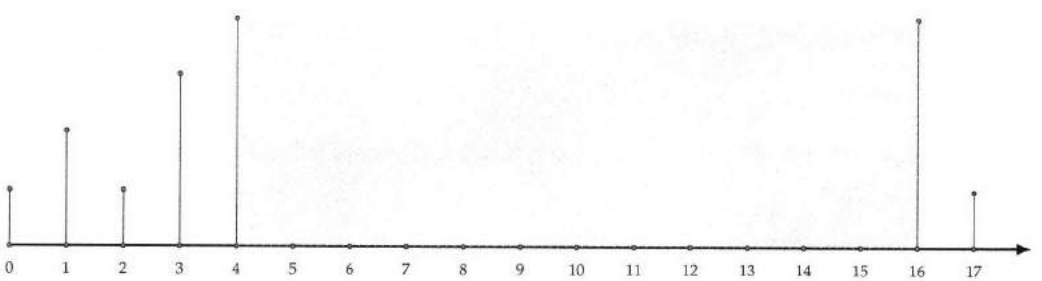
**Variant:** Solve the same problem when you cannot assume that there exists an ample city.

## 17.7 Compute the maximum water trapped by a pair of vertical lines

An array of integers naturally defines a set of lines parallel to the $Y$-axis, starting from $x = 0$ as illustrated in Figure 17.4(a). The goal of this problem is to find the pair of lines that together with the $X$-axis "trap" the most water. See Figure 17.4(b) for an example.




(a) A graphical depiction of the array $\langle 1, 2, 1, 3, 4, 4, 5, 6, 2, 1, 3, 1, 3, 2, 1, 2, 4, 1 \rangle$.




(b) The shaded area between 4 and 16 is the maximum water that can be trapped by the array in (a).

**Figure 17.4:** Example of maximum water trapped by a pair of vertical lines.

Write a program which takes as input an integer array and returns the pair of entries that trap the maximum amount of water.

*Hint:* Start with 0 and $n - 1$ and work your way in.

**Solution:** Let $A$ be the array, and $n$ its length. There is a straightforward $O(n^2)$ brute-force solution—for each pair of indices $(i, j), i < j$, the water trapped by the corresponding lines is $(j - i) \times \min(A[i], A[j])$, which can easily be computed in $O(1)$ time. The maximum of all these is the desired quantity.

In an attempt to achieve a better time complexity, we can try divide-and-conquer. We find the maximum water that can be trapped by the left half of $A$, the right half of $A$, and across the center of $A$. Finding the maximum water trapped by an element on the left half and the right half entails considering combinations of the $n/2$ entries from left half and $n/2$ entries on the right half. Therefore, the time complexity $T(n)$ of this divide-and-conquer approach satisfies $T(n) = 2T(n/2) + O(n^2/4)$ which solves to $T(n) = O(n^2)$. This is no better than the brute-force solution, and considerably trickier to code.

A good starting point is to consider the widest pair, i.e., 0 and $n-1$. We record the corresponding amount of trapped water, i.e., $((n-1) - 0) \times \min(A[0], A[n-1])$. Suppose $A[0] > A[n-1]$. Then for any $k > 0$, the water trapped between $k$ and $n-1$ is less than the water trapped between 0 and $n-1$, so going forward, we only need to focus on the maximum water that can be trapped between 0 and $n-2$. The converse is true if $A[0] \leq A[n-1]$—we need never consider 0 again.

We use this approach iteratively to continuously reduce the subarray that must be explored, while recording the most water trapped so far. In essence, we are exploring the best way in which to trade-off width for height.

For the given example, we begin with $(0, 17)$, which has a capacity of $1 \times 17 = 17$. Since the height of the left line is less than or equal to the height of the right line, we advance to $(1, 17)$. The capacity is $1 \times 16 = 16$. Since $2 > 1$, we advance to $(1, 16)$. The capacity is $2 \times 15 = 30$. Since $2 < 4$, we advance to $(2, 16)$. The capacity is $1 \times 14 = 14$. Since $1 < 4$, we advance to $(3, 16)$. The capacity is $3 \times 13 = 39$. Since $3 < 4$, we advance to $(4, 16)$. The capacity is $4 \times 12 = 48$. Future iterations, which we do not show, do not surpass 48, which is the result.

```
def get_max_trapped_water(heights):
    i, j, max_water = 0, len(heights) - 1, 0
    while i < j:
        width = j - i
        max_water = max(max_water, width * min(heights[i], heights[j]))
        if heights[i] > heights[j]:
            j -= 1
        else:  # heights[i] <= heights[j].
            i += 1
    return max_water
```

We iteratively eliminate one line or two lines at a time, and we spend $O(1)$ time per iteration, so the time complexity is $O(n)$.

### 17.8 Compute the largest rectangle under the skyline

You are given a sequence of adjacent buildings. Each has unit width and an integer height. These buildings form the skyline of a city. An architect wants to know the area of a largest rectangle contained in this skyline. See Figure 17.5 on the next page for an example.

Let $A$ be an array representing the heights of adjacent buildings of unit width. Design an algorithm to compute the area of the largest rectangle contained in this skyline.

*Hint:* How would you efficiently find the largest rectangle which includes the $i$th building, and has height $A[i]$?

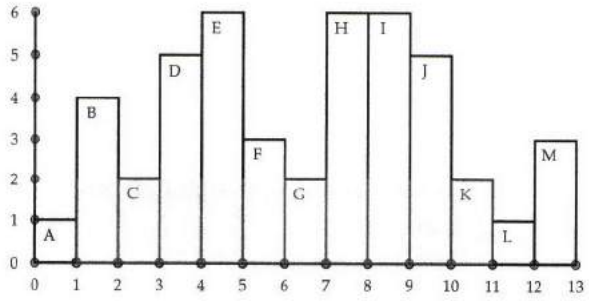



**Figure 17.5:** A collection of unit-width buildings, and the largest contained rectangle. The text label identifying the building is just below and to the right of its upper left-hand corner. The shaded area is the largest rectangle under the skyline. Its area is $2 \times (11 - 1)$. Note that the tallest rectangle is from 7 to 9, and the widest rectangle is from 0 to 1, but neither of these are the largest rectangle under the skyline.

**Solution:** A brute-force approach is to take each $(i, j)$ pair, find the minimum of subarray $A[i, j]$, and multiply that by $j - i + 1$. This has time complexity $O(n^3)$, where $n$ is the length of $A$. This can be improved to $O(n^2)$ by iterating over $i$ and then $j \geq i$ and tracking the minimum height of buildings from $i$ to $j$, inclusive. However, there is no reasonable way to further refine this algorithm to get the time complexity below $O(n^2)$.

Another brute-force solution is to consider for each $i$ the furthest left and right we can go without dropping below $A[i]$ in height. In essence, we want to know the largest rectangle that is "supported" by Building $i$, which can be viewed as acting like a "pillar" of that rectangle. For the given example, the largest rectangle supported by $G$ extends from 1 to 11, and the largest rectangle supported by $F$ extends from 3 to 6.

We can easily determine the largest rectangle supported by Building $i$ with a single forward and a single backward iteration from $i$. Since $i$ ranges from 0 to $n - 1$, the time complexity of this approach is $O(n^2)$.

This brute-force solution can be refined to get a much better time complexity. Specifically, suppose we iterate over buildings from left to right. When we process Building $i$, we do not know how far to the right the largest rectangle it supports goes. However, we do know that the largest rectangles supported by earlier buildings whose height is greater than $A[i]$ cannot extend past $i$, since Building $i$ "blocks" these rectangles. For example, when we get to $F$ in Figure 17.5, we know that the largest rectangles supported by Buildings $B$, $D$, and $E$ (whose heights are greater than $F$'s height) cannot extend past 5.

Looking more carefully at the example, we see that there's no need to consider $B$ when examining $F$, since C has already blocked the largest rectangle supported by $B$. In addition, there's no reason to consider C: since $G$'s height is the same as $C$'s height, and $C$ has not been blocked yet, $G$ and $C$ will have the same largest supported rectangle. Generalizing, as we advance through the buildings, all we really need is to keep track of is buildings that have not been blocked yet. Additionally, we can replace existing buildings whose height equals that of the current building with the current building. Call these buildings the set of active pillars.

Initially there are no buildings in the active pillar set. As we iterate from 0 to 12, the active pillar sets are $\{A\}$, $\{A, B\}$, $\{A, C\}$, $\{A, C, D\}$, $\{A, C, D, E\}$, $\{A, C, F\}$, $\{A, G\}$, $\{A, G, H\}$, $\{A, G, I\}$, $\{A, G, J\}$, $\{A, K\}$, $\{L\}$, and $\{L, M\}$.

Whenever a building is removed from the active pillar set, we know exactly how far to the right the largest rectangle that it supports goes to. For example, when we reach C we know $B$'s supported rectangle ends at 2, and when we reach $F$, we know that $D$ and $E$'s largest supported rectangles end

at 5.

When we remove a blocked building from the active pillar set, to find how far to the left its largest supported rectangle extends we simply look for the closest active pillar that has a lower height. For example, when we reach $F$, the active pillars are $\{A, C, D, E\}$. We remove $E$ and $D$ from the set, since both are taller than $F$. The largest rectangle supported by $E$ has height 6 and begins after $D$, i.e., at 4; its area is $6 \times (5 - 4) = 6$. The largest rectangle supported by $D$ has height 5 and begins after $C$, i.e., at 3; its area is $5 \times (5 - 3) = 10$.

There are a number of data structures that can be used to store the active pillars and the right data structure is key to making the above algorithm efficient. When we process a new building we need to find the buildings in the active pillar set that are blocked. Because insertion and deletion into the active pillar set take place in last-in first-out order, a stack is a reasonable choice for maintaining the set. Specifically, the rightmost building in the active pillar set appears at the top. Using a stack also makes it easy to find how far to the left the largest rectangle that's supported by a building in the active pillar set extends—we simply look at the building below it in the stack. For example, when we process $F$, the stack is $A, C, D, E$, with $E$ at the top. Comparing $F$'s height with $E$, we see $E$ is blocked so the largest rectangle under $E$ ends at where $F$ begins, i.e., at 5. Since the next building in the stack is $D$, we know that the largest rectangle under $E$ begins where $D$ ends, i.e., at 4.

The algorithm described above is almost complete. The missing piece is what to do when we get to the end of the iteration. The stack will not be empty when we reach the end—at the very least it will contain the last building. We deal with these elements just as we did before, the only difference being that the largest rectangle supported by each building in the stack ends at $n$, where $n$ is the number of elements in the array. For the given example, the stack contains $L$ and $M$ when we get to the end, and the largest supported rectangles for both of these end at 13.

```
def calculate_largest_rectangle(heights):
    pillar_indices, max_rectangle_area = [], 0
    # By appending [0] to heights, we can uniformly handle the computation for
    # rectangle area here.
    for i, h in enumerate(heights + [0]):
        while pillar_indices and heights[pillar_indices[-1]] >= h:
            height = heights[pillar_indices.pop()]
            width = i if not pillar_indices else i - pillar_indices[-1] - 1
            max_rectangle_area = max(max_rectangle_area, height * width)
        pillar_indices.append(i)
    return max_rectangle_area
```

The time complexity is $O(n)$. When advancing through buildings, the time spent for building is proportional to the number of pushes and pops performed when processing it. Although for some buildings, we may perform multiple pops, in total we perform at most $n$ pushes and at most $n$ pops. This is because in the advancing phase, an entry $i$ is added at most once to the stack and cannot be popped more than once. The time complexity of processing remaining stack elements after the advancing is complete is also $O(n)$ since there are at most $n$ elements in the stack, and the time to process each one is $O(1)$. Thus, the overall time complexity is $O(n)$. The space complexity is $O(n)$, which is the largest the stack can grow to, e.g., if buildings appear in ascending order.

**Variant:** Find the largest square under the skyline.

CHAPTER

# 18 Graphs

*Concerning these bridges, it was asked whether anyone could arrange a route in such a way that he would cross each bridge once and only once.*

— *"The solution of a problem relating to the geometry of position,"* L. EULER, 1741

Informally, a graph is a set of vertices and connected by edges. Formally, a directed graph is a set $V$ of *vertices* and a set $E \subset V \times V$ of edges. Given an edge $e = (u, v)$, the vertex $u$ is its *source*, and $v$ is its *sink*. Graphs are often decorated, e.g., by adding lengths to edges, weights to vertices, a start vertex, etc. A directed graph can be depicted pictorially as in Figure 18.1.

A *path* in a directed graph from $u$ to vertex $v$ is a sequence of vertices $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ where $v_0 = u$, $v_{n-1} = v$, and each $(v_i, v_{i+1})$ is an edge. The sequence may consist of a single vertex. The *length* of the path $\langle v_0, v_1, \ldots, v_{n-1} \rangle$ is $n - 1$. Intuitively, the length of a path is the number of edges it traverses. If there exists a path from $u$ to $v$, $v$ is said to be *reachable* from $u$. For example, the sequence $\langle a, c, e, d, h \rangle$ is a path in the graph represented in Figure 18.1.

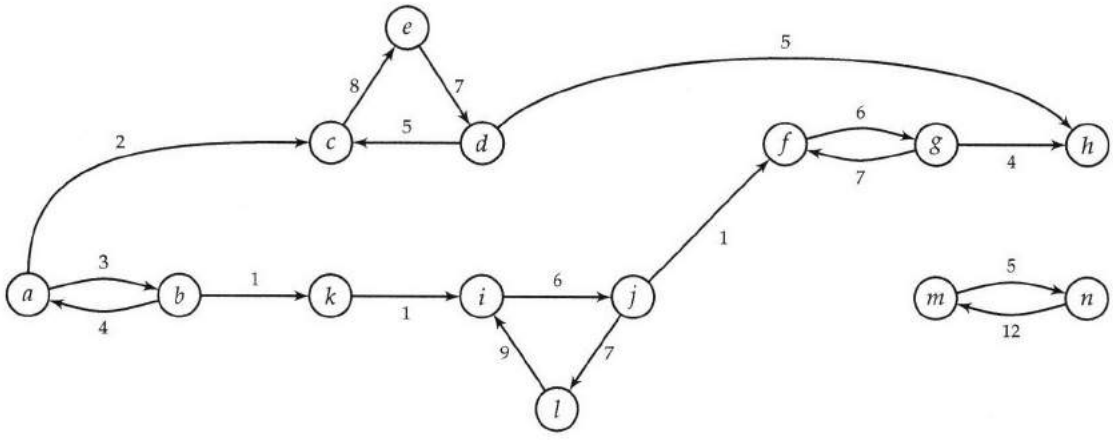



**Figure 18.1:** A directed graph with weights on edges.

A *directed acyclic graph* (*DAG*) is a directed graph in which there are no *cycles*, i.e., paths which contain one or more edges and which begin and end at the same vertex. See Figure 18.2 on the next page for an example of a directed acyclic graph. Vertices in a DAG which have no incoming edges are referred to as *sources*; vertices which have no outgoing edges are referred to as *sinks*. A *topological ordering* of the vertices in a DAG is an ordering of the vertices in which each edge is from a vertex earlier in the ordering to a vertex later in the ordering. Solution 18.8 on Page 286 uses the notion of topological ordering.

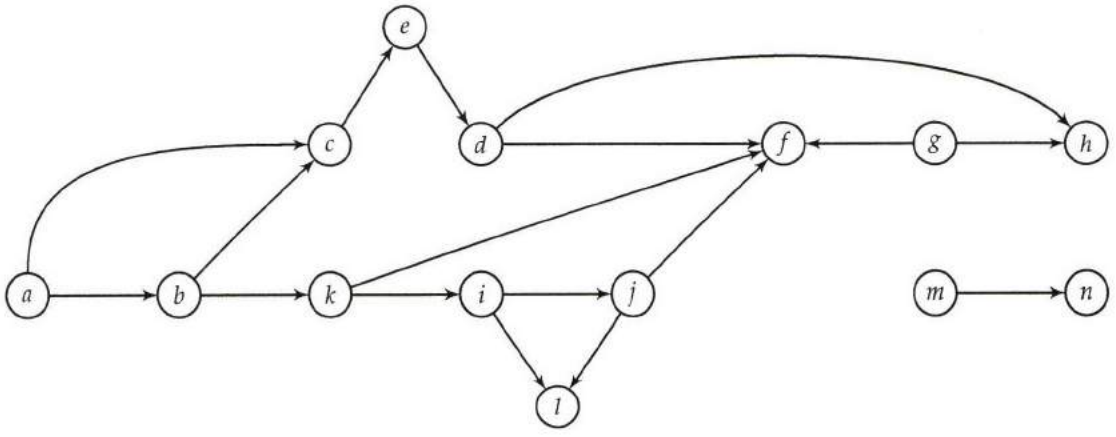



**Figure 18.2:** A directed acyclic graph. Vertices $a, g, m$ are sources and vertices $l, f, h, n$ are sinks. The ordering $\langle a, b, c, e, d, g, h, k, i, j, f, l, m, n \rangle$ is a topological ordering of the vertices.

An undirected graph is also a tuple $(V, E)$; however, $E$ is a set of unordered pairs of vertices. Graphically, this is captured by drawing arrowless connections between vertices, as in Figure 18.3.

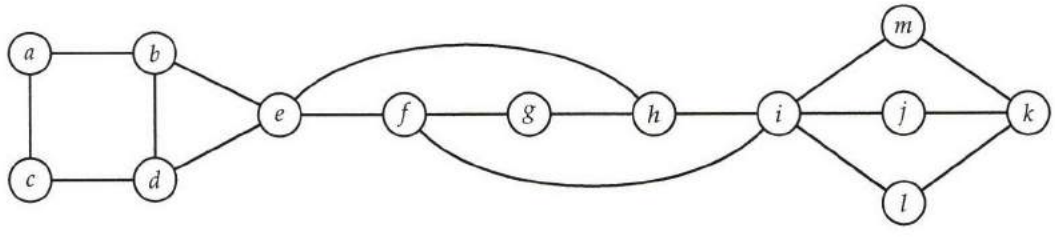



**Figure 18.3:** An undirected graph.

If $G$ is an undirected graph, vertices $u$ and $v$ are said to be *connected* if $G$ contains a path from $u$ to $v$; otherwise, $u$ and $v$ are said to be *disconnected*. A graph is said to be connected if every pair of vertices in the graph is connected. A *connected component* is a maximal set of vertices $C$ such that each pair of vertices in $C$ is connected in $G$. Every vertex belongs to exactly one connected component.

For example, the graph in Figure 18.3 is connected, and it has a single connected component. If edge $(h, i)$ is removed, it remains connected. If additionally $(f, i)$ is removed, it becomes disconnected and there are two connected components.

A directed graph is called *weakly connected* if replacing all of its directed edges with undirected edges produces an undirected graph that is connected. It is *connected* if it contains a directed path from $u$ to $v$ or a directed path from $v$ to $u$ for every pair of vertices $u$ and $v$. It is *strongly connected* if it contains a directed path from $u$ to $v$ and a directed path from $v$ to $u$ for every pair of vertices $u$ and $v$.

Graphs naturally arise when modeling geometric problems, such as determining connected cities. However, they are more general, and can be used to model many kinds of relationships.

A graph can be implemented in two ways—using *adjacency lists* or an *adjacency matrix*. In the adjacency list representation, each vertex $v$, has a list of vertices to which it has an edge. The adjacency matrix representation uses a $|V| \times |V|$ Boolean-valued matrix indexed by vertices, with

a 1 indicating the presence of an edge. The time and space complexities of a graph algorithm are usually expressed as a function of the number of vertices and edges.

A *tree* (sometimes called a free tree) is a special sort of graph—it is an undirected graph that is connected but has no cycles. (Many equivalent definitions exist, e.g., a graph is a free tree if and only if there exists a unique path between every pair of vertices.) There are a number of variants on the basic idea of a tree. A rooted tree is one where a designated vertex is called the root, which leads to a parent-child relationship on the nodes. An ordered tree is a rooted tree in which each vertex has an ordering on its children. Binary trees, which are the subject of Chapter 9, differ from ordered trees since a node may have only one child in a binary tree, but that node may be a left or a right child, whereas in an ordered tree no analogous notion exists for a node with a single child. Specifically, in a binary tree, there is position as well as order associated with the children of nodes.

As an example, the graph in Figure 18.4 is a tree. Note that its edge set is a subset of the edge set of the undirected graph in Figure 18.3 on the facing page. Given a graph $G = (V, E)$, if the graph $G' = (V, E')$ where $E' \subset E$, is a tree, then $G'$ is referred to as a spanning tree of $G$.

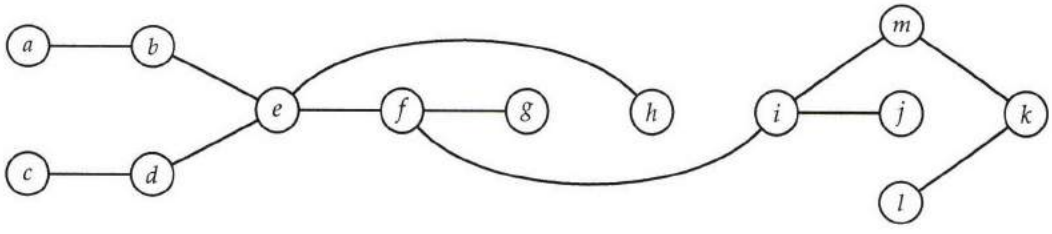



**Figure 18.4:** A tree.

### *Graphs boot camp*

Graphs are ideal for modeling and analyzing relationships between pairs of objects. For example, suppose you were given a list of the outcomes of matches between pairs of teams, with each outcome being a win or loss. A natural question is as follows: given teams $A$ and $B$, is there a sequence of teams starting with $A$ and ending with $B$ such that each team in the sequence has beaten the next team in the sequence?

A slick way of solving this problem is to model the problem using a graph. Teams are vertices, and an edge from one team to another indicates that the team corresponding to the source vertex has beaten the team corresponding to the destination vertex. Now we can apply graph reachability to perform the check. Both DFS and BFS are reasonable approaches—the program below uses DFS.

```
MatchResult = collections.namedtuple('MatchResult', ('winning_team',
                                                     'losing_team'))


def can_team_a_beat_team_b(matches, team_a, team_b):
    def build_graph():
        graph = collections.defaultdict(set)
        for match in matches:
            graph[match.winning_team].add(match.losing_team)
        return graph

    def is_reachable_dfs(graph, curr, dest, visited=set()):
        if curr == dest:
```

```
            return True
        elif curr in visited or curr not in graph:
            return False
        visited.add(curr)
        return any(is_reachable_dfs(graph, team, dest) for team in graph[curr])

    return is_reachable_dfs(build_graph(), team_a, team_b)
```

The time complexity and space complexity are both $O(E)$, where $E$ is the number of outcomes.

| |
|---|
| It's natural to use a graph when the problem involves **spatially connected** objects, e.g., road segments between cities.<br><br>**More generally**, consider using a graph when you need to analyze **any binary relationship**, between objects, such as interlinked webpages, followers in a social graph, etc. In such cases, quite often the problem can be reduced to a well-known graph problem.<br><br>Some graph problems entail **analyzing structure**, e.g., looking for cycles or connected components. **DFS** works particularly well for these applications.<br><br>Some graph problems are related to **optimization**, e.g., find the shortest path from one vertex to another. **BFS, Dijkstra's shortest path algorithm, and minimum spanning tree** are examples of graph algorithms appropriate for optimization problems. |

**Table 18.1:** Top Tips for Graphs

### *Graph search*

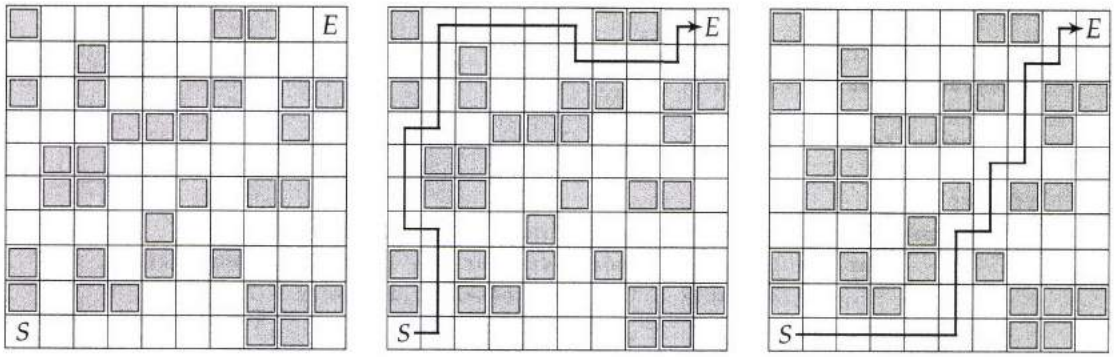
Computing vertices which are reachable from other vertices is a fundamental operation which can be performed in one of two idiomatic ways, namely depth-first search (DFS) and breadth-first search (BFS). Both have linear time complexity—$O(|V| + |E|)$ to be precise. In the worst-case there is a path from the initial vertex covering all vertices without any repeats, and the DFS edges selected correspond to this path, so the space complexity of DFS is $O(|V|)$ (this space is implicitly allocated on the function call stack). The space complexity of BFS is also $O(|V|)$, since in a worst-case there is an edge from the initial vertex to all remaining vertices, implying that they will all be in the BFS queue simultaneously at some point.

DFS and BFS differ from each other in terms of the additional information they provide, e.g., BFS can be used to compute distances from the start vertex and DFS can be used to check for the presence of cycles. Key notions in DFS include the concept of *discovery time* and *finishing time* for vertices.

## 18.1 Search a maze

It is natural to apply graph models and algorithms to spatial problems. Consider a black and white digitized image of a maze—white pixels represent open areas and black spaces are walls. There are two special white pixels: one is designated the entrance and the other is the exit. The goal in this problem is to find a way of getting from the entrance to the exit, as illustrated in Figure 18.5 on the next page.

Given a 2D array of black and white entries representing a maze with designated entrance and exit points, find a path from the entrance to the exit, if one exists.




(a) A maze. (b) A path from entrance to exit. (c) A shortest path from entrance to exit.

**Figure 18.5:** An instance of the maze search problem, with two solutions, where $S$ and $E$ denote the entrance and exit, respectively.

*Hint:* Model the maze as a graph.

**Solution:** A brute-force approach would be to enumerate every possible path from entry to exit. However, we know from Solution 16.3 on Page 242 that the number of paths is astronomical. Of course, pruning helps, since we can stop as soon as a path hits a black pixel, but the worse-case behavior of enumerative approaches is still very bad.

Another approach is to perform a random walk moving from a white pixel to a random adjacent white pixel. Given enough time this will find a path, if one exists. However, it repeats visits, which retards the progress. The random walk does suggest the right way—we should keep track of pixels that we have already visited. This is exactly what DFS and BFS do to ensure progress.

This suggests modeling the maze as a graph. Each vertex corresponds to a white pixel. We will index the vertices based on the coordinates of the corresponding pixel, i.e., vertex $v_{i,j}$ corresponds to the white entry at $(i, j)$ in the 2D array. Edges model adjacent white pixels.

Now, run a DFS starting from the vertex corresponding to the entrance. If at some point, we discover the exit vertex in the DFS, then there exists a path from the entrance to the exit. If we implement recursive DFS then the path would consist of all the vertices in the call stack corresponding to previous recursive calls to the DFS routine.

This problem can also be solved using BFS from the entrance vertex on the same graph model. The BFS tree has the property that the computed path will be a shortest path from the entrance. However BFS is more difficult to implement than DFS since in DFS, the compiler implicitly handles the DFS stack, whereas in BFS, the queue has to be explicitly coded. Since the problem did not call for a shortest path, it is better to use DFS.

```
WHITE, BLACK = range(2)

Coordinate = collections.namedtuple('Coordinate', ('x', 'y'))


def search_maze(maze, s, e):
    # Perform DFS to find a feasible path.
    def search_maze_helper(cur):
        # Checks cur is within maze and is a white pixel.
        if not (0 <= cur.x < len(maze) and 0 <= cur.y < len(maze[cur.x])
                and maze[cur.x][cur.y] == WHITE):
```

```
            return False
        path.append(cur)
        maze[cur.x][cur.y] = BLACK
        if cur == e:
            return True

        if any(
                map(search_maze_helper, (Coordinate(
                    cur.x - 1, cur.y), Coordinate(cur.x + 1, cur.y), Coordinate(
                        cur.x, cur.y - 1), Coordinate(cur.x, cur.y + 1)))):
            return True
        # Cannot find a path, remove the entry added in path.append(cur).
        del path[-1]
        return False

    path = []
    if not search_maze_helper(s):
        return []  # No path between s and e.
    return path
```

The time complexity is the same as that for DFS, namely $O(|V| + |E|)$.

## 18.2 Paint a Boolean matrix

Let $A$ be a Boolean 2D array encoding a black-and-white image. The entry $A(a, b)$ can be viewed as encoding the color at entry $(a, b)$. Call two entries adjacent if one is to the left, right, above or below the other. Note that the definition implies that an entry can be adjacent to at most four other entries, and that adjacency is symmetric, i.e., if $e0$ is adjacent to entry $e1$, then $e1$ is adjacent to $e0$.

Define a path from entry $e0$ to entry $e1$ to be a sequence of adjacent entries, starting at $e0$, ending at $e1$, with successive entries being adjacent. Define the region associated with a point $(i, j)$ to be all points $(i', j')$ such that there exists a path from $(i, j)$ to $(i', j')$ in which all entries are the same color. In particular this implies $(i, j)$ and $(i', j')$ must be the same color.

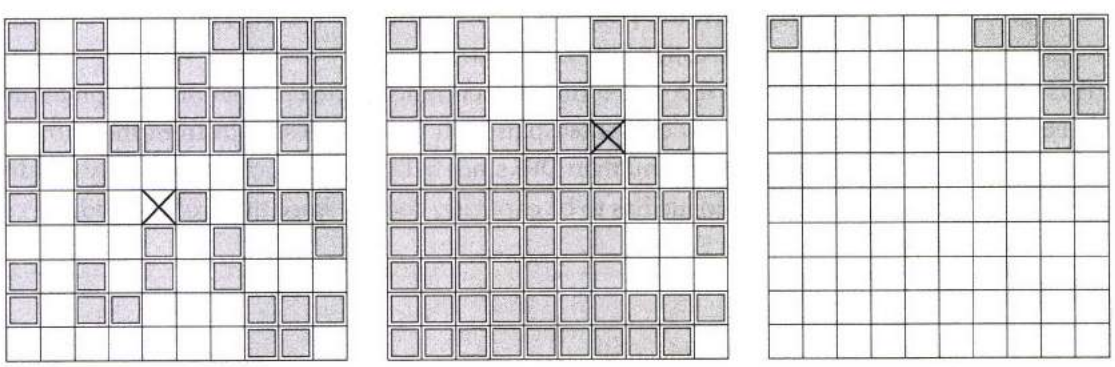



**Figure 18.6:** The color of all squares associated with the first square marked with a × in (a) have been recolored to yield the coloring in (b). The same process yields the coloring in (c).

Implement a routine that takes an $n \times m$ Boolean array $A$ together with an entry $(x, y)$ and flips the color of the region associated with $(x, y)$. See Figure 18.6 for an example of flipping.

*Hint:* Solve this conceptually, then think about implementation optimizations.

**Solution:** As with Solution 18.1 on Page 276, graph search can overcome the complexity of enumerative and random search solutions. Specifically, entries can be viewed as vertices, with vertices corresponding to adjacent entries begin connected by edges.

For the current problem, we are searching for all vertices whose color is the same as that of $(x, y)$ that are reachable from $(x, y)$. Breadth-first search is natural when starting with a set of vertices. Specifically, we can use a queue to store such vertices. The queue is initialized to $(x, y)$. The queue is popped iteratively. Call the popped point $p$. First, we record $p$'s initial color, and then flip its color. Next we examine $p$ neighbors. Any neighbor which is the same color as $p$'s initial color is added to the queue. The computation ends when the queue is empty. Correctness follows from the fact that any point that is added to the queue is reachable from $(x, y)$ via a path consisting of points of the same color, and all points reachable from $(x, y)$ via points of the same color will eventually be added to the queue.

```
def flip_color(x, y, A):
    Coordinate = collections.namedtuple('Coordinate', ('x', 'y'))
    color = A[x][y]
    q = collections.deque([Coordinate(x, y)])
    A[x][y] = 1 - A[x][y]  # Flips.
    while q:
        x, y = q.popleft()
        for d in (0, 1), (0, -1), (1, 0), (-1, 0):
            next_x, next_y = x + d[0], y + d[1]
            if (0 <= next_x < len(A) and 0 <= next_y < len(A[next_x])
                    and A[next_x][next_y] == color):
                # Flips the color.
                A[next_x][next_y] = 1 - A[next_x][next_y]
                q.append(Coordinate(next_x, next_y))
```

The time complexity is the same as that of BFS, i.e., $O(mn)$. The space complexity is a little better than the worst-case for BFS, since there are at most $O(m + n)$ vertices that are at the same distance from a given entry.

We also provide a recursive solution which is in the spirit of DFS. It does not need a queue but implicitly uses a stack, namely the function call stack.

```
def flip_color(x, y, A):
    color = A[x][y]
    A[x][y] = 1 - A[x][y]  # Flips.
    for d in (0, 1), (0, -1), (1, 0), (-1, 0):
        next_x, next_y = x + d[0], y + d[1]
        if (0 <= next_x < len(A) and 0 <= next_y < len(A[next_x])
                and A[next_x][next_y] == color):
            flip_color(next_x, next_y, A)
```

The time complexity is the same as that of DFS.

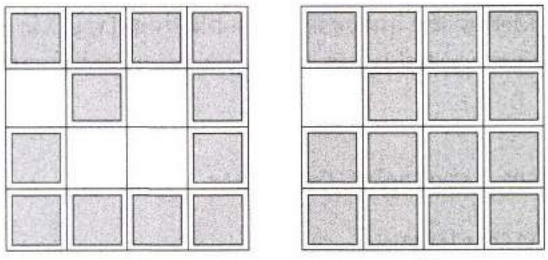
Both the algorithms given above differ slightly from traditional BFS and DFS algorithms. The reason is that we have a color field already available, and hence do not need the auxiliary color field traditionally associated with vertices BFS and DFS. Furthermore, since we are simply determining reachability, we only need two colors, whereas BFS and DFS traditionally use three colors to track state. (The use of an additional color makes it possible, for example, to answer questions about cycles in directed graphs, but that is not relevant here.)

**Variant:** Design an algorithm for computing the black region that contains the most points.

**Variant:** Design an algorithm that takes a point $(a, b)$, sets $A(a, b)$ to black, and returns the size of the black region that contains the most points. Assume this algorithm will be called multiple times, and you want to keep the aggregate run time as low as possible.

## 18.3 Compute enclosed regions

This problem is concerned with computing regions within a 2D grid that are enclosed. See Figure 18.7 for an illustration of the problem.



(a) (b)

**Figure 18.7:** Three of the four white squares in (a) are *enclosed*, i.e., there is no path from any of them to the boundary that only passes through white squares. (b) shows the white squares that are not enclosed.

The computational problem can be formalized using 2D arrays of $B$s (blacks) and $W$s (whites). Figure 18.7(a) is encoded by

$$A = \begin{bmatrix} B & B & B & B \\ W & B & W & B \\ B & W & W & B \\ B & B & B & B \end{bmatrix}.$$

Figure 18.7(b) is encoded by

$$\begin{bmatrix} B & B & B & B \\ W & B & B & B \\ B & B & B & B \\ B & B & B & B \end{bmatrix}.$$

Let $A$ be a 2D array whose entries are either $W$ or $B$. Write a program that takes $A$, and replaces all $W$s that cannot reach the boundary with a $B$.

*Hint:* It is easier to compute the complement of the desired result.

**Solution:** It is easier to focus on the inverse problem, namely identifying $W$s that can reach the boundary. The reason that the inverse is simpler is that if a $W$ is adjacent to a $W$ that is can reach the boundary, then the first $W$ can reach it too. The $W$s on the boundary are the initial set. Subsequently, we find $W$s neighboring the boundary $W$s, and iteratively grow the set. Whenever we find a new $W$ that can reach the boundary, we need to record it, and at some stage search for new $W$s from it. A queue is a reasonable data structure to track $W$s to be processed. The approach amounts to breadth-first search starting with a set of vertices rather than a single vertex.

```
def fill_surrounded_regions(board):
    n, m = len(board), len(board[0])
    q = collections.deque([
        (i, j) for k in range(n) for i, j in ((k, 0), (k, m - 1))
    ] + [(i, j) for k in range(m) for i, j in ((0, k), (n - 1, k))])
    while q:
        x, y = q.popleft()
        if 0 <= x < n and 0 <= y < m and board[x][y] == 'W':
            board[x][y] = 'T'
            q.extend([(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)])
    board[:] = [['B' if c != 'T' else 'W' for c in row] for row in board]
```

The time and space complexity are the same as those for BFS, namely $O(mn)$, where $m$ and $n$ are the number of rows and columns in $A$.

## 18.4 Deadlock detection

High performance database systems use multiple processes and resource locking. These systems may not provide mechanisms to avoid or prevent deadlock: a situation in which two or more competing actions are each waiting for the other to finish, which precludes all these actions from progressing. Such systems must support a mechanism to detect deadlocks, as well as an algorithm for recovering from them.

One deadlock detection algorithm makes use of a "wait-for" graph to track which other processes a process is currently blocking on. In a wait-for graph, processes are represented as nodes, and an edge from process $P$ to $Q$ implies $Q$ is holding a resource that $P$ needs and thus $P$ is waiting for $Q$ to release its lock on that resource. A cycle in this graph implies the possibility of a deadlock. This motivates the following problem.

Write a program that takes as input a directed graph and checks if the graph contains a cycle.

*Hint:* Focus on "back" edges.

**Solution:** We can check for the existence of a cycle in $G$ by running DFS on $G$. Recall DFS maintains a color for each vertex. Initially, all vertices are white. When a vertex is first discovered, it is colored gray. When DFS finishes processing a vertex, that vertex is colored black.

As soon as we discover an edge from a gray vertex back to a gray vertex, a cycle exists in $G$ and we can stop. Conversely, if there exists a cycle, once we first reach vertex in the cycle (call it $v$), we will visit its predecessor in the cycle (call it $u$) before finishing processing $v$, i.e., we will find an edge from a gray to a gray vertex. In summary, a cycle exists if and only if DFS discovers an edge from a gray vertex to a gray vertex. Since the graph may not be strongly connected, we must examine each vertex, and run DFS from it if it has not already been explored.

```
class GraphVertex:

    white, gray, black = range(3)

    def __init__(self):
        self.color = GraphVertex.white
        self.edges = []
```

```
def is_deadlocked(G):
    def has_cycle(cur):
        # Visiting a gray vertex means a cycle.
        if cur.color == GraphVertex.gray:
            return True

        cur.color = GraphVertex.gray  # Marks current vertex as a gray one.
        # Traverse the neighbor vertices.
        if any(next.color != GraphVertex.black and has_cycle(next)
               for next in cur.edges):
            return True
        cur.color = GraphVertex.black  # Marks current vertex as black.
        return False

    return any(vertex.color == GraphVertex.white and has_cycle(vertex)
               for vertex in G)
```

The time complexity of DFS is $O(|V| + |E|)$: we iterate over all vertices, and spend a constant amount of time per edge. The space complexity is $O(|V|)$, which is the maximum stack depth—if we go deeper than $|V|$ calls, some vertex must repeat, implying a cycle in the graph, which leads to early termination.

**Variant:** Solve the same problem for an undirected graph.

**Variant:** Write a program that takes as input an undirected graph, which you can assume to be connected, and checks if the graph remains connected if any one edge is removed.

### 18.5 Clone a graph

Consider a vertex type for a directed graph in which there are two fields: an integer label and a list of references to other vertices. Design an algorithm that takes a reference to a vertex $u$, and creates a copy of the graph on the vertices reachable from $u$. Return the copy of $u$.

*Hint:* Maintain a map from vertices in the original graph to their counterparts in the clone.

**Solution:** We traverse the graph starting from $u$. Each time we encounter a vertex or an edge that is not yet in the clone, we add it to the clone. We recognize new vertices by maintaining a hash table mapping vertices in the original graph to their counterparts in the new graph. Any standard graph traversal algorithm works—the code below uses breadth first search.

```
class GraphVertex:
    def __init__(self, label):
        self.label = label
        self.edges = []


def clone_graph(G):
    if not G:
        return None

    q = collections.deque([G])
    vertex_map = {G: GraphVertex(G.label)}
    while q:
        v = q.popleft()
```

```
        for e in v.edges:
            # Try to copy vertex e.
            if e not in vertex_map:
                vertex_map[e] = GraphVertex(e.label)
                q.append(e)
            # Copy edge v->e.
            vertex_map[v].edges.append(vertex_map[e])
    return vertex_map[G]
```

The space complexity is $O(|V| + |E|)$, which is the space taken by the result. Excluding the space for the result, the space complexity is $O(|V|)$—this comes from the hash table, as well as the BFS queue.

### 18.6 Making wired connections

Consider a collection of electrical pins on a printed circuit board (PCB). For each pair of pins, there may or may not be a wire joining them. This is shown in Figure 18.8, where vertices correspond to pins, and edges indicate the presence of a wire between pins. (The significance of the colors is explained later.)

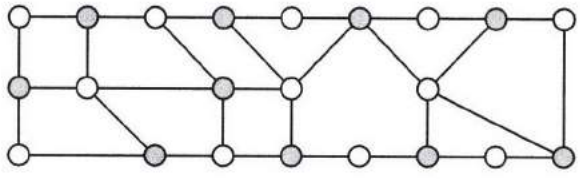


**Figure 18.8:** A set of pins and wires between them.

Design an algorithm that takes a set of pins and a set of wires connecting pairs of pins, and determines if it is possible to place some pins on the left half of a PCB, and the remainder on the right half, such that each wire is between left and right halves. Return such a division, if one exists. For example, the light vertices and dark vertices in Figure 18.8 are such division.

*Hint:* Model as a graph and think about the implication of an odd length cycle.

**Solution:** A brute-force approach might be to try all partitions of the pins into two sets. However, the number of such partitions is very high.

A better approach is to use connectivity information to guide the partitioning. Assume the pins are numbered from 0 to $p - 1$. Create an undirected graph $G$ whose vertices are the pins. Add an edge between pairs of vertices if the corresponding pins are connected by a wire. For simplicity, assume $G$ is connected; if not, the connected components can be analyzed independently.

Run BFS on $G$ beginning with any vertex $v_0$. Assign $v_0$ arbitrarily to lie on the left half. All vertices at an odd distance from $v_0$ are assigned to the right half.

When performing BFS on an undirected graph, all newly discovered edges will either be from vertices which are at a distance $d$ from $v_0$ to undiscovered vertices (which will then be at a distance $d + 1$ from $v_0$) or from vertices which are at a distance $d$ to vertices which are also at a distance $d$. First, assume we never encounter an edge from a distance $k$ vertex to a distance $k$ vertex. In this case, each wire is from a distance $k$ vertex to a distance $k + 1$ vertex, so all wires are between the left and right halves.

If any edge is from a distance $k$ vertex to a distance $k$ vertex, we stop—the pins cannot be partitioned into left and right halves as desired. The reason is as follows. Let $u$ and $v$ be such vertices. Consider the first common ancestor $a$ in the BFS tree of $u$ and $v$ (such an ancestor must exist since the search started at $v_0$). The paths $p_u$ and $p_v$ in the BFS tree from $a$ to $u$ and $v$ are of equal length; therefore, the cycle formed by going from $a$ to $u$ via $p_u$, then through the edge $(u, v)$, and then back to $a$ from $v$ via $p_v$ has an odd length. A cycle in which the vertices can be partitioned into two sets must have an even number of edges—it has to go back and forth between the sets and terminate at the starting vertex, and each back and forth adds two edges. Therefore, the vertices in an odd length cycle cannot be partitioned into two sets such that all edges are between the sets.

```
class GraphVertex:
    def __init__(self):
        self.d = -1
        self.edges = []


def is_any_placement_feasible(G):
    def bfs(s):
        s.d = 0
        q = collections.deque([s])

        while q:
            for t in q[0].edges:
                if t.d == -1:  # Unvisited vertex.
                    t.d = q[0].d + 1
                    q.append(t)
                elif t.d == q[0].d:
                    return False
            del q[0]
        return True

    return all(bfs(v) for v in G if v.d == -1)
```

The complexity is the same as for BFS, i.e., $O(p + w)$ time complexity, where $w$ is the number of wires, and $O(p)$ space complexity.

Graphs that can be partitioned as described above are known as bipartite graphs. Another term for such graphs is 2-colorable (since the vertices can be assigned one of two colors without neighboring vertices having the same color).

## 18.7 Transform one string to another

Let $s$ and $t$ be strings and $D$ a dictionary, i.e., a set of strings. Define $s$ to *produce* $t$ if there exists a sequence of strings from the dictionary $P = \langle s_0, s_1, \ldots, s_{n-1} \rangle$ such that the first string is $s$, the last string is $t$, and adjacent strings have the same length and differ in exactly one character. The sequence $P$ is called a *production sequence*. For example, if the dictionary is {bat, cot, dog, dag, dot, cat}, then ⟨cat, cot, dot, dog⟩ is production sequence.

Given a dictionary $D$ and two strings $s$ and $t$, write a program to determine if $s$ produces $t$. Assume that all characters are lowercase alphabets. If $s$ does produce $t$, output the length of a shortest production sequence; otherwise, output $-1$.

*Hint:* Treat strings as vertices in an undirected graph, with an edge between $u$ and $v$ if and only if the corresponding strings differ in one character.

**Solution:** A brute-force approach may be to explore all strings that differ in one character from the starting string, then two characters from the starting string, etc. The problem with this approach is that it may explore lots of strings that are outside the dictionary.

A better approach is to be more focused on dictionary words. In particular, it's natural to model this problem using graphs. The vertices correspond to strings from the dictionary and the edge $(u, v)$ indicates that the strings corresponding to $u$ and $v$ differ in exactly one character. Note that the relation "differs in one character" is symmetric, so the graph is undirected.

For the given example, the vertices would be {bat, cot, dog, dag, dot, cat}, and the edges would be {(bat, cat), (cot, dot), (cot, cat), (dog, dag), (dog, dot)}.

A production sequence is simply a path in $G$, so what we need is a shortest path from $s$ to $t$ in $G$. Shortest paths in an undirected graph are naturally computed using BFS.

```
# Uses BFS to find the least steps of transformation.
def transform_string(D, s, t):
    StringWithDistance = collections.namedtuple(
        'StringWithDistance', ('candidate_string', 'distance'))
    q = collections.deque([StringWithDistance(s, 0)])
    D.remove(s)  # Marks s as visited by erasing it in D.

    while q:
        f = q.popleft()
        # Returns if we find a match.
        if f.candidate_string == t:
            return f.distance  # Number of steps reaches t.

        # Tries all possible transformations of f.candidate_string.
        for i in range(len(f.candidate_string)):
            for c in string.ascii_lowercase:  # Iterates through 'a' ~ 'z'.
                cand = f.candidate_string[:i] + c + f.candidate_string[i + 1:]
                if cand in D:
                    D.remove(cand)
                    q.append(StringWithDistance(cand, f.distance + 1))
    return -1  # Cannot find a possible transformations.
```

The number of vertices is $d$, the number of words in the dictionary. The number of edges is, in the worst-case, $O(d^2)$. The time complexity is that of BFS, namely $O(d+d^2) = O(d^2)$. If the string length $n$ is less than $d$ then the maximum number of edges out of a vertex is $O(n)$, implying an $O(nd)$ bound.

**Variant:** An addition chain exponentiation program for computing $x^n$ is a finite sequence $\langle x, x^{i_1}, x^{i_2}, \ldots, x^n \rangle$ where each element after the first is either the square of some previous element or the product of any two previous elements. For example, the term $x^{15}$ can be computed by the following two addition chain exponentiation programs.

$$P1 = \langle x, x^2 = (x)^2, x^4 = (x^2)^2, x^8 = (x^4)^2, x^{12} = x^8x^4, x^{14} = x^{12}x^2, x^{15} = x^{14}x \rangle$$
$$P2 = \langle x, x^2 = (x)^2, x^3 = x^2x, x^5 = x^3x^2, x^{10} = (x^5)^2, x^{15} = x^{10}x^5 \rangle$$

It is not obvious, but the second program, $P2$, is the shortest addition chain exponentiation program for computing $x^{15}$.

Given a positive integer $n$, how would you compute a shortest addition chain exponentiation program to evaluate $x^n$?

### *Advanced graph algorithms*

Up to this point we looked at basic search and combinatorial properties of graphs. The algorithms we considered were all linear time complexity and relatively straightforward—the major challenge was in modeling the problem appropriately.

There are four classes of complex graph problems that can be solved efficiently, i.e., in polynomial time. Most other problems on graphs are either variants of these or, very likely, not solvable by polynomial time algorithms. These four classes are:

- *Shortest path*—given a graph, directed or undirected, with costs on the edges, find the minimum cost path from a given vertex to all vertices. Variants include computing the shortest paths for all pairs of vertices, and the case where costs are all nonnegative.
- *Minimum spanning tree*—given an undirected graph $G = (V, E)$, assumed to be connected, with weights on each edge, find a subset $E'$ of the edges with minimum total weight such that the subgraph $G' = (V, E')$ is connected.
- *Matching*—given an undirected graph, find a maximum collection of edges subject to the constraint that every vertex is incident to at most one edge. The matching problem for bipartite graphs is especially common and the algorithm for this problem is much simpler than for the general case. A common variant is the maximum weighted matching problem in which edges have weights and a maximum weight edge set is sought, subject to the matching constraint.
- *Maximum flow*—given a directed graph with a capacity for each edge, find the maximum flow from a given source to a given sink, where a flow is a function mapping edges to numbers satisfying conservation (flow into a vertex equals the flow out of it) and the edge capacities. The minimum cost circulation problem generalizes the maximum flow problem by adding lower bounds on edge capacities, and for each edge, a cost per unit flow.

These four problem classes have polynomial time algorithms and can be solved efficiently in practice for very large graphs. Algorithms for these problems tend to be specialized, and the natural approach does not always work best. For example, it is natural to apply divide-and-conquer to compute the MST as follows. Partition the vertex set into two subsets, compute MSTs for the subsets independently, and then join these two MSTs with an edge of minimum weight between them. Figure 18.9 on the facing page shows how this algorithm can lead to suboptimal results.

In this chapter we restrict our attention to shortest-path problems.

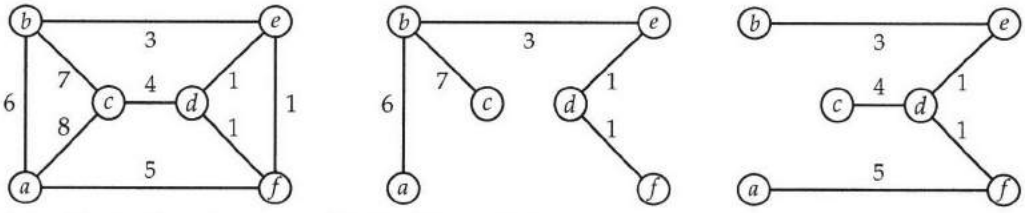
## 18.8 Team photo day—2

How would you generalize your solution to Problem 13.9 on Page 193, to determine the largest number of teams that can be photographed simultaneously subject to the same constraints?

*Hint:* Form a DAG in which paths correspond to valid placements.

**Solution:** Let $G$ be the DAG with vertices corresponding to the teams as follows and edges from vertex $X$ to $Y$ iff Team $X$ can be placed behind Team $Y$.

Every sequence of teams where a team can be placed behind its predecessor corresponds to a path in $G$. To find the longest such sequence, we simply need to find the longest path in the




(a) A weighted undirected graph.

(b) An MST built by divide-and-conquer from the MSTs on $\{a, b, c\}$ and $\{d, e, f\}$. The edge $(b, e)$ is the lightest edge connecting the two MSTs.

(c) An optimum MST.

**Figure 18.9:** Divide-and-conquer applied to the MST problem is suboptimum—the MST in (b) has weight 18, but the MST in (c) has weight 14.

DAG $G$. We can do this, for example, by topologically ordering the vertices in $G$; the longest path terminating at vertex $v$ is the maximum of the longest paths terminating at $v$'s fan-ins concatenated with $v$ itself.

```
class GraphVertex:
    def __init__(self):
        self.edges = []
        self.max_distance = 0


def find_largest_number_teams(G):
    def build_topological_ordering():
        def dfs(cur):
            cur.max_distance = 1
            for next in cur.edges:
                if not next.max_distance:
                    dfs(next)
            vertex_order.append(cur)

        vertex_order = []
        for g in G:
            if not g.max_distance:
                dfs(g)
        return vertex_order

    def find_longest_path(vertex_order):
        max_distance = 0
        while vertex_order:
            u = vertex_order.pop()
            max_distance = max(max_distance, u.max_distance)
            for v in u.edges:
                v.max_distance = max(v.max_distance, u.max_distance + 1)
        return max_distance

    return find_longest_path(build_topological_ordering())
```

The topological ordering computation is $O(|V| + |E|)$ and dominates the computation time. Clearly $|V|$ is the number of teams. The number of edges $E$ depends on the heights, but can be as high as $O(|V|^2)$, e.g., when there is a path of length $|V| - 1$.

**Variant:** Let $\mathcal{T} = \{T_0, T_1, \ldots, T_{n-1}\}$ be a set of tasks. Each task runs on a single generic server. Task $T_i$ has a duration $\tau_i$, and a set $P_i$ (possibly empty) of tasks that must be completed before $T_i$ can be started. The set is *feasible* if there does not exist a sequence of tasks $\langle T_0, T_1, \ldots, T_{n-1}, T_0 \rangle$ starting and ending at the same task such that for each consecutive pair of tasks, the first task must be completed before the second task can begin.

Given an instance of the task scheduling problem, compute the least amount of time in which all the tasks can be performed, assuming an unlimited number of servers. Explicitly check that the system is feasible.

CHAPTER

# 19 Parallel Computing

*The activity of a computer must include the proper reacting to a possibly great variety of messages that can be sent to it at unpredictable moments, a situation which occurs in all information systems in which a number of computers are coupled to each other.*

— "Cooperating sequential processes," E. W. DIJKSTRA, 1965

Parallel computation has become increasingly common. For example, laptops and desktops come with multiple processors which communicate through shared memory. High-end computation is often done using clusters consisting of individual computers communicating through a network.

Parallelism provides a number of benefits:

- High performance—more processors working on a task (usually) means it is completed faster.
- Better use of resources—a program can execute while another waits on the disk or network.
- Fairness—letting different users or programs share a machine rather than have one program run at a time to completion.
- Convenience—it is often conceptually more straightforward to do a task using a set of concurrent programs for the subtasks rather than have a single program manage all the subtasks.
- Fault tolerance—if a machine fails in a cluster that is serving web pages, the others can take over.

Concrete applications of parallel computing include graphical user interfaces (GUI) (a dedicated thread handles UI actions while other threads are, for example, busy doing network communication and passing results to the UI thread, resulting in increased responsiveness), Java virtual machines (a separate thread handles garbage collection which would otherwise lead to blocking, while another thread is busy running the user code), web servers (a single logical thread handles a single client request), scientific computing (a large matrix multiplication can be split across a cluster), and web search (multiple machines crawl, index, and retrieve web pages).

The two primary models for parallel computation are the shared memory model, in which each processor can access any location in memory, and the distributed memory model, in which a processor must explicitly send a message to another processor to access its memory. The former is more appropriate in the multicore setting and the latter is more accurate for a cluster. The questions in this chapter are focused on the shared memory model.

Writing correct parallel programs is challenging because of the subtle interactions between parallel components. One of the key challenges is races—two concurrent instruction sequences access the same address in memory and at least one of them writes to that address. Other challenges to correctness are

- starvation (a processor needs a resource but never gets it; e.g., Problem 19.6 on Page 296),
- deadlock (Thread $A$ acquires Lock $L1$ and Thread $B$ acquires Lock $L2$, following which $A$ tries to acquire $L2$ and $B$ tries to acquire $L1$), and

- livelock (a processor keeps retrying an operation that always fails).

Bugs caused by these issues are difficult to find using testing. Debugging them is also difficult because they may not be reproducible since they are usually load dependent. It is also often true that it is not possible to realize the performance implied by parallelism—sometimes a critical task cannot be parallelized, making it impossible to improve performance, regardless of the number of processors added. Similarly, the overhead of communicating intermediate results between processors can exceed the performance benefits.

The problems in this chapter focus on thread-level parallelism. Problems concerned with parallelism on distributed memory architectures, e.g., cluster computing, are usually not meant to be coded; they are popular design and architecture problems. Problems 20.9 on Page 310, 20.10 on Page 310, 20.11 on Page 311, and 20.17 on Page 316 cover some aspects of cluster-level parallelism.

### *Parallel computing boot camp*

A semaphore is a very powerful synchronization construct. Conceptually, a semaphore maintains a set of permits. A thread calling *acquire()* on a semaphore waits, if necessary, until a permit is available, and then takes it. A thread calling *release()* on a semaphore adds a permit and notifies threads waiting on that semaphore, potentially releasing a blocking acquirer.

```
import threading


class Semaphore():
    def __init__(self, max_available):
        self.cv = threading.Condition()
        self.MAX_AVAILABLE = max_available
        self.taken = 0

    def acquire(self):
        self.cv.acquire()
        while (self.taken == self.MAX_AVAILABLE):
            self.cv.wait()
        self.taken += 1
        self.cv.release()

    def release(self):
        self.cv.acquire()
        self.taken -= 1
        self.cv.notify()
        self.cv.release()
```

| |
|---|
| Start with an algorithm that **locks aggressively** and is easily seen to be correct. Then **add back concurrency**, while ensuring the **critical** parts are locked.<br><br>When analyzing parallel code, assume a worst-case thread scheduler. In particular, it may choose to schedule the same thread repeatedly, it may alternate between two threads, it may starve a thread, etc.<br><br>Try to work at a **higher level of abstraction**. In particular, know the **concurrency libraries**—don't implement your own **semaphores, thread pools, deferred execution**, etc. (You should know how these features are implemented, and implement them if asked to.) |

**Table 19.1:** Top Tips for Concurrency

The program below is part of an online spell correction service. Clients send as input a string, and the service returns an array of strings in its dictionary that are closest to the input string (this array could be computed, for example, using Solution 16.2 on Page 239). The service caches results to improve performance. Critique the implementation and provide a solution that overcomes its limitations.

```
class SpellCheckService:
    w_last = closest_to_last_word = None

    @staticmethod
    def service(req, resp):
        w = req.extract_word_to_check_from_request()
        if w != SpellCheckService.w_last:
            SpellCheckService.w_last = w
            SpellCheckService.closest_to_last_word = closest_in_dictionary(w)
        resp.encode_into_response(SpellCheckService.closest_to_last_word)
```

*Hint:* Look for races, and lock as little as possible to avoid reducing throughput.

**Solution:** The solution has a race condition. Suppose clients $A$ and $B$ make concurrent requests, and the service launches a thread per request. Suppose the thread for request $A$ finds that the input string is present in the cache, and then, immediately after that check, the thread for request $B$ is scheduled. Suppose this thread's lookup fails, so it computes the result, and adds it to the cache. If the cache is full, an entry will be evicted, and this may be the result for the string passed in request $A$. Now when request $A$ is scheduled back, it does a lookup for the value corresponding to its input string, expecting it to be present (since it checked that that string is a key in the cache). However, the cache will return null.

A thread-safe solution would be to synchronize every call to the service. In this case, only one thread could be executing the method and there are no races between cache reads and writes. However, it also leads to poor performance—only one thread can execute the service call at a time.

The solution is to lock just the part of the code that operates on the cached values—specifically, the check on the cached value and the updates to the cached values:

In the program below, multiple threads can be concurrently computing closest strings. This is good because the calls take a long time (this is why they are cached). Locking ensures that the read assignment on a hit and write assignment on completion are atomic.

```
class SpellCheckService:
    w_last = closest_to_last_word = None
    lock = threading.Lock()

    @staticmethod
    def service(req, resp):
        w = req.extract_word_to_check_from_request()
        result = None
        with SpellCheckService.lock:
            if w == SpellCheckService.w_last:
                result = SpellCheckService.closest_to_last_word.copy()
        if result is None:
            result = closest_in_dictionary(w)
            with SpellCheckService.lock:
                SpellCheckService.w_last = w
```

```
                SpellCheckService.closest_to_last_word = result
        resp.encode_into_response(result)
```

**Variant:** Threads 1 to $n$ execute a method called *critical()*. Before this, they execute a method called *rendezvous()*. The synchronization constraint is that only one thread can execute *critical()* at a time, and all threads must have completed executing *rendezvous()* before *critical()* can be called. You can assume $n$ is stored in a variable $n$ that is accessible from all threads. Design a synchronization mechanism for the threads. All threads must execute the same code. Threads may call *critical()* multiple times, and you should ensure that a thread cannot call *critical()* a $(k+1)$th time until all other threads have completed their $k$th calls to *critical()*.

## 19.2 Analyze two unsynchronized interleaved threads

Threads $t1$ and $t2$ each increment an integer variable $N$ times, as shown in the code below. This program yields nondeterministic results. Usually, it prints $2N$ but sometimes it prints a smaller value. The problem is more pronounced for large $N$. As a concrete example, on one run the program output 1320209 when $N = 1000000$ was specified at the command line.

```
N = 1000000
counter = 0


def increment_thread():
    global counter
    for _ in range(N):
        counter = counter + 1


t1 = threading.Thread(target=increment_thread)
t2 = threading.Thread(target=increment_thread)

t1.start()
t2.start()
t1.join()
t2.join()

print(counter)
```

What are the maximum and minimum values that could be printed by the program as a function of $N$?

*Hint:* Be as perverse as you can when scheduling the threads.

**Solution:** First, note that the increment code is unguarded, which opens up the possibility of its value being determined by the order in which threads that write to it are scheduled by the thread scheduler.

The maximum value is $2N$. This occurs when the thread scheduler runs one thread to completion, followed by the other thread.

When $N = 1$, the minimum value for the count variable is 1: $t1$ reads, $t2$ reads, $t1$ increments and writes, then $t2$ increments and writes. When $N > 1$, the final value of the count variable must be at least 2. The reasoning is as follows. There are two possibilities. A thread, call it $T$, performs a read-increment-write-read-increment-write without the other thread writing between reads, in

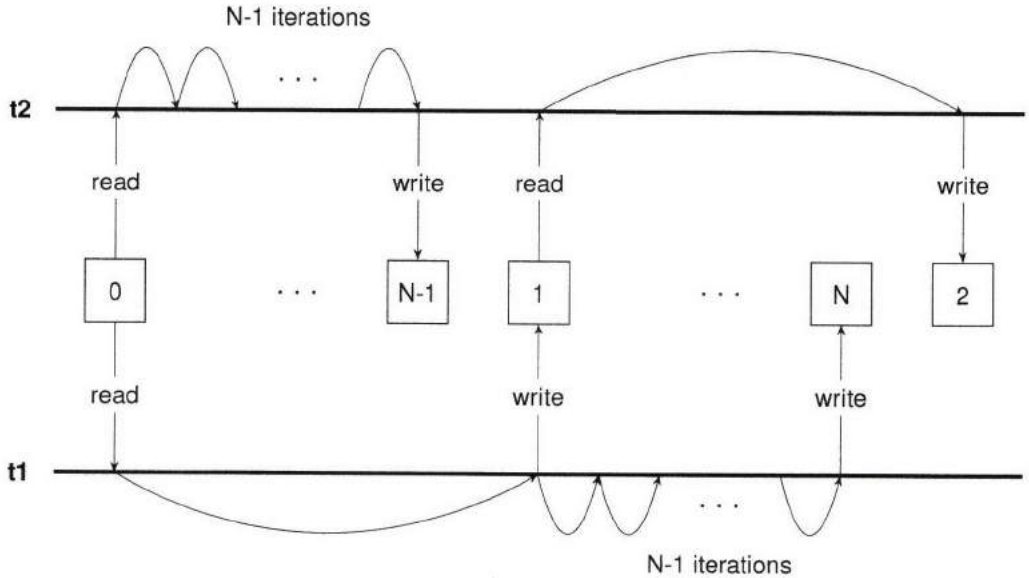



**Figure 19.1:** Worst-case schedule for two unsynchronized threads.

which case the written value is at least 2. If the other thread now writes a 1, it has not yet completed, so it will increment at least once more. Otherwise, $T$'s second read returns a value of 1 or more (since the other thread has performed at least one write).

The lower bound of 2 is achieved according to the following thread schedule:

- $t1$ loads the value of the counter, which is 0.
- $t2$ executes the loop $N-1$ times.
- $t1$ doesn't know that the value of the counter changed and writes 1 to it.
- $t2$ loads the value of the counter, which is 1.
- $t1$ executes the loop for the remaining $N-1$ iterations.
- $t2$ doesn't know that the value of the counter has changed, and writes 2 to the counter.

This schedule is depicted in Figure 19.1.

## 19.3 Implement synchronization for two interleaving threads

Thread $t1$ prints odd numbers from 1 to 100; Thread $t2$ prints even numbers from 1 to 100.

Write code in which the two threads, running concurrently, print the numbers from 1 to 100 in order.

*Hint:* The two threads need to notify each other when they are done.

**Solution:** A brute-force solution is to use a lock which is repeatedly captured by the threads. A single variable, protected by the lock, indicates who went last. The drawback of this approach is that it employs the busy waiting antipattern: processor time that could be used to execute a different task is instead wasted on useless activity.

Below we present a solution based on the same idea, but one that avoids busy locking by using

```
class OddEvenMonitor(threading.Condition):

    ODD_TURN = True
    EVEN_TURN = False
```

```
    def __init__(self):
        super().__init__()
        self.turn = self.ODD_TURN

    def wait_turn(self, old_turn):
        with self:
            while self.turn != old_turn:
                self.wait()

    def toggle_turn(self):
        with self:
            self.turn ^= True
            self.notify()


class OddThread(threading.Thread):
    def __init__(self, monitor):
        super().__init__()
        self.monitor = monitor

    def run(self):
        for i in range(1, 101, 2):
            self.monitor.wait_turn(OddEvenMonitor.ODD_TURN)
            print(i)
            self.monitor.toggle_turn()


class EvenThread(threading.Thread):
    def __init__(self, monitor):
        super().__init__()
        self.monitor = monitor

    def run(self):
        for i in range(2, 101, 2):
            self.monitor.wait_turn(OddEvenMonitor.EVEN_TURN)
            print(i)
            self.monitor.toggle_turn()
```

## 19.4 Implement a thread pool

The following program, implements part of a simple HTTP server:

```
SERVERPORT = 8080


def main():
    serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    serversock.bind(('', SERVERPORT))
    serversock.listen(5)
    while True:
        sock, addr = serversock.accept()
        process_req(sock)
```

Suppose you find that the program has poor performance because it frequently blocks on I/O. What steps could you take to improve the program's performance? Feel free to use any utilities from the standard library, including concurrency classes.

*Hint:* Use multithreading, but control the number of threads.

**Solution:** The first attempt to solve this problem might be to launch a new thread per request rather than process the request itself:

```
SERVERPORT = 8080

serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversock.bind(('', SERVERPORT))
serversock.listen(5)
while True:
    sock, addr = serversock.accept()
    threading.Thread(target=process_req, args=(sock, )).start()
```

The problem with this approach is that we do not control the number of threads launched. A thread consumes a nontrivial amount of resources, such as the time taken to start and end the thread and the memory used by the thread. For a lightly-loaded server, this may not be an issue but under load, it can result in exceptions that are challenging, if not impossible, to handle.

The right trade-off is to use a *thread pool*. As the name implies, this is a collection of threads, the size of which is bounded. A thread pool can be implemented relatively easily using a blocking queue, i.e., a queue which blocks the writing thread on a put until the queue is empty. However, since the problem statement explicitly allows us to use library routines, we can use

```
SERVERPORT = 8080
NTHREADS = 2

executor = concurrent.futures.ThreadPoolExecutor(max_workers=NTHREADS)
serversock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversock.bind(('', SERVERPORT))
serversock.listen(5)
while True:
    sock, addr = serversock.accept()
    executor.submit(process_req, sock)
```

## 19.5 Deadlock

When threads need to acquire multiple locks to enter a critical section, deadlock can result. As an example, suppose both $T1$ and $T2$ need to acquire locks $L$ and $M$. If $T1$ first acquires $L$, and then $T2$ then acquires $M$, they end up waiting on each other forever.

Identify a concurrency bug in the program below, and modify the code to resolve the issue.

```
class Account:

    _global_id = 0

    def __init__(self, balance):
        self._balance = balance
        self._id = Account._global_id
        Account._global_id += 1
        self._lock = threading.RLock()
```

```
    def get_balance(self):
        return self._balance

    @staticmethod
    def transfer(acc_from, acc_to, amount):
        th = threading.Thread(target=acc_from._move, args=(acc_to, amount))
        th.start()

    def _move(self, acc_to, amount):
        with self._lock:
            if amount > self._balance:
                return False
            acc_to._balance += amount
            self._balance -= amount
            print('returning True')
            return True
```

**Solution:** Suppose $U1$ initiates a transfer to $U2$, and immediately afterwards, $U2$ initiates a transfer to $U1$. Since each transfer takes place in a separate thread, it's possible for the first thread to lock $U1$ and then the second thread to be scheduled in and take the lock $U2$. The program is now deadlocked—each of the two threads is waiting for the lock held by the other thread.

One solution is to have a global lock which is acquired by the transfer method. The drawback is that it blocks transfers that are unrelated, e.g., $U3$ cannot transfer to $U4$ if there is a pending transfer from $U5$ to $U6$.

The canonical way to avoid deadlock is to have a global ordering on locks and acquire them in that order. Since accounts have a unique integer id, the update below is all that is needed to solve the deadlock.

```
lock1 = self._lock if self._id < acc_to._id else acc_to._lock
lock2 = acc_to._lock if self._id < acc_to._id else self._lock
# Does not matter if lock1 equals lock2: since recursive_mutex locks
# are reentrant, we will re-acquire lock2.
with lock1, lock2:
```

### 19.6 The readers-writers problem

Consider an object $s$ which is read from and written to by many threads. (For example, $s$ could be the cache from Problem 19.1 on Page 291.) You need to ensure that no thread may access $s$ for reading or writing while another thread is writing to $s$. (Two or more readers may access $s$ at the same time.)

One way to achieve this is by protecting $s$ with a mutex that ensures that two threads cannot access $s$ at the same time. However, this solution is suboptimal because it is possible that a reader $R1$ has locked $s$ and another reader $R2$ wants to access $s$. Reader $R2$ does not have to wait until $R1$ is done reading; instead, $R2$ should start reading right away.

This motivates the first readers-writers problem: protect $s$ with the added constraint that no reader is to be kept waiting if $s$ is currently opened for reading.

Implement a synchronization mechanism for the first readers-writers problem.

*Hint:* Track the number of readers.

**Solution:** We want to keep track of whether the string is being read from, as well as whether the string is being written to. Additionally, if the string is being read from, we want to know the number of concurrent readers. We achieve this with a pair of locks—a read lock and a write lock—and a read counter locked by the read lock.

A reader proceeds as follows. It locks the read lock, increments the counter, and releases the read lock. After it performs its reads, it locks the read lock, decrements the counter, and releases the read lock. A writer locks the write lock, then performs the following in an infinite loop. It locks the read lock, checks to see if the read counter is 0; if so, it performs its write, releases the read lock, and breaks out of the loop. Finally, it releases the write lock. As in Solution 19.3 on Page 293, we use wait-notify primitives to avoid busy waiting.

```
# LR and LW are class attributes in the RW class.
# They serve as read and write locks. The integer
# variable read_count in RW tracks the number of readers.
class Reader(threading.Thread):

    def run(self):
        while True:
            with RW.LR:
                RW.read_count += 1


            print(RW.data)
            with RW.LR:
                RW.read_count -= 1
                RW.LR.notify()
            do_something_else()


class Writer(threading.Thread):

    def run(self):
        while True:
            with RW.LW:
                done = False
                while not done:
                    with RW.LR:
                        if RW.read_count == 0:
                            RW.data += 1
                            done = True
                        else:
                            # use wait/notify to avoid busy waiting
                            while RW.read_count != 0:
                                RW.LR.wait()
            do_something_else()
```

## 19.7 The readers-writers problem with write preference

Suppose we have an object s as in Problem 19.6 on the preceding page. In the solution to Problem 19.6 on the facing page, a reader $R1$ may have the lock; if a writer $W$ is waiting for the lock and then a reader $R2$ requests access, $R2$ will be given priority over $W$. If this happens often enough, $W$ will starve. Instead, suppose we want $W$ to start as soon as possible.

This motivates the second readers-writers problem: protect s with "writer-preference", i.e., no writer, once added to the queue, is to be kept waiting longer than absolutely necessary.

Implement a synchronization mechanism for the second readers-writers problem.

*Hint:* Force readers to acquire a write lock.

**Solution:** We want to give writers the preference. We achieve this by modifying Solution 19.6 on the previous page to have a reader start by locking the write lock and then immediately releasing it. In this way, a writer who acquires the write lock is guaranteed to be ahead of the subsequent readers.

**Variant:** The specifications to Problems 19.6 on Page 296 and 19.7 on the previous page allow starvation—the first may starve writers, the second may starve readers. The third readers-writers problem adds the constraint that neither readers nor writers should starve. Implement a synchronization mechanism for the third readers-writers problem.

### 19.8 Implement a Timer class

Consider a web-based calendar in which the server hosting the calendar has to perform a task when the next calendar event takes place. (The task could be sending an email or a Short Message Service (SMS).) Your job is to design a facility that manages the execution of such tasks.

Develop a timer class that manages the execution of deferred tasks. The timer constructor takes as its argument an object which includes a run method and a string-valued name field. The class must support—(1.) starting a thread, identified by name, at a given time in the future; and (2.) canceling a thread, identified by name (the cancel request is to be ignored if the thread has already started).

*Hint:* There are two aspects—data structure design and concurrency.

**Solution:** The two aspects to the design are the data structures and the locking mechanism.

We use two data structures. The first is a min-heap in which we insert key-value pairs: the keys are run times and the values are the thread to run at that time. A dispatch thread runs these threads; it sleeps from call to call and may be woken up if a thread is added to or deleted from the pool. If woken up, it advances or retards its remaining sleep time based on the top of the min-heap. On waking up, it looks for the thread at the top of the min-heap—if its launch time is the current time, the dispatch thread deletes it from the min-heap and executes it. It then sleeps till the launch time for the next thread in the min-heap. (Because of deletions, it may happen that the dispatch thread wakes up and finds nothing to do.)

The second data structure is a hash table with thread ids as keys and entries in the min-heap as values. If we need to cancel a thread, we go to the min-heap and delete it. Each time a thread is added, we add it to the min-heap; if the insertion is to the top of the min-heap, we interrupt the dispatch thread so that it can adjust its wake up time.

Since the min-heap is shared by the update methods and the dispatch thread, we need to lock it. The simplest solution is to have a single lock that is used for all read and writes into the min-heap and the hash table.

### 19.9 Test the Collatz conjecture in parallel

In Problem 12.11 on Page 176 and its solution we introduced the Collatz conjecture and heuristics for checking it. In this problem, you are to build a parallel checker for the Collatz conjecture.

Specifically, assume your program will run on a multicore machine, and threads in your program will be distributed across the cores. Your program should check the Collatz conjecture for every integer in $[1, U]$ where $U$ is an input to your program.

Design a multi-threaded program for checking the Collatz conjecture. Make full use of the cores available to you. To keep your program from overloading the system, you should not have more than $n$ threads running at a time.

*Hint:* Use multithreading for performance—take care to minimize threading overhead.

**Solution:** Heuristics for pruning checks on individual integers are discussed in Solution 12.11 on Page 176. The aim of this problem is implementing a multi-threaded checker. We could have a master thread launch $n$ threads, one per number, starting with $1, 2, \ldots, x$. The master thread would keep track of what number needs to be processed next, and when a thread returned, it could re-assign it the next unchecked number.

The problem with this approach is that the time spent executing the check in an individual thread is very small compared to the overhead of communicating with the thread. The natural solution is to have each thread process a subrange of $[1, U]$. We could do this by dividing $[1, U]$ into $n$ equal sized subranges, and having Thread $i$ handle the $i$th subrange.

The heuristics for checking the Collatz conjecture take longer on some integers than others, and in the strategy above there is the potential of a situation arising where one thread takes much longer to complete than the others, which leads to most of the cores being idle.

A good compromise is to have threads handle smaller intervals, which are still large enough to offset the thread overhead. We can maintain a work-queue consisting of unprocessed intervals, and assigning these to returning threads.

```
# Performs basic unit of work
def worker(lower, upper):
    for i in range(lower, upper + 1):
        assert collatz_check(i, set())
    print('(%d,%d)' % (lower, upper))


# Checks an individual number
def collatz_check(x, visited):
    if x == 1:
        return True
    elif x in visited:
        return False
    visited.add(x)
    if x & 1:  # odd number
        return collatz_check(3 * x + 1, visited)
    else:  # even number
        return collatz_check(x >> 1, visited)  # divide by 2


    # Uses the library thread pool for task assignment and load balancing
    executor = concurrent.futures.ProcessPoolExecutor(max_workers=NTHREADS)
    with executor:
        for i in range(N // RANGESIZE):
            executor.submit(worker, i * RANGESIZE + 1, (i + 1) * RANGESIZE)
```

# Part III

# Domain Specific Problems

CHAPTER

# 20 Design Problems

*Don't be fooled by the many books on complexity or by the many complex and arcane algorithms you find in this book or elsewhere. Although there are no textbooks on simplicity, simple systems work and complex don't.*

*— "Transaction Processing: Concepts and Techniques,"*
J. GRAY, 1992

You may be asked in an interview how to go about creating a set of services or a larger system, possibly on top of an algorithm that you have designed. These problems are fairly open-ended, and many can be the starting point for a large software project.

In an interview setting when someone asks such a question, you should have a conversation in which you demonstrate an ability to think creatively, understand design trade-offs, and attack unfamiliar problems. You should sketch key data structures and algorithms, as well as the technology stack (programming language, libraries, OS, hardware, and services) that you would use to solve the problem.

The answers in this chapter are presented in this context—they are meant to be examples of good responses in an interview and are not comprehensive state-of-the-art solutions.

We review patterns that are useful for designing systems in Table 20.1. Some other things to keep in mind when designing a system are implementation time, extensibility, scalability, testability, security, internationalization, and IP issues.

**Table 20.1:** System design patterns.

| Design principle | Key points |
|---|---|
| Algorithms and Data Structures | Identify the basic algorithms and data structures |
| Decomposition | Split the functionality, architecture, and code into manageable, reusable components. |
| Scalability | Break the problem into subproblems that can be solved relatively independently on different machines. Shard data across machines to make it fit. Decouple the servers that handle writes from those that handle reads. Use replication across the read servers to gain more performance. Consider caching computation and later look it up to save work. |

## *Decomposition*

Good decompositions are critical to successfully solving system-level design problems. Functionality, architecture, and code all benefit from decomposition.

For example, in our solution to designing a system for online advertising (Problem 20.15 on Page 315), we decompose the goals into categories based on the stake holders. We decompose the architecture itself into a front-end and a back-end. The front-end is divided into user management, web page design, reporting functionality, etc. The back-end is made up of middleware, storage, database, cron services, and algorithms for ranking ads. The design of TEX (Problem 20.6 on Page 307) and Connexus (Problem 20.14 on Page 314) also illustrate such decompositions.

Decomposing code is a hallmark of object-oriented programming. The subject of design patterns is concerned with finding good ways to achieve code-reuse. Broadly speaking, design patterns are grouped into creational, structural, and behavioral patterns. Many specific patterns are very natural—strategy objects, adapters, builders, etc., appear in a number of places in our codebase. Freeman *et al.*'s *"Head First Design Patterns"* is, in our opinion, the right place to study design patterns.

### *Scalability*

In the context of interview questions parallelism is useful when dealing with scale, i.e., when the problem is too large to fit on a single machine or would take an unacceptably long time on a single machine. The key insight you need to display is that you know how to decompose the problem so that

- each subproblem can be solved relatively independently, and
- the solution to the original problem can be efficiently constructed from solutions to the subproblems.

Efficiency is typically measured in terms of central processing unit (CPU) time, random access memory (RAM), network bandwidth, number of memory and database accesses, etc.

Consider the problem of sorting a petascale integer array. If we know the distribution of the numbers, the best approach would be to define equal-sized ranges of integers and send one range to one machine for sorting. The sorted numbers would just need to be concatenated in the correct order. If the distribution is not known then we can send equal-sized arbitrary subsets to each machine and then merge the sorted results, e.g., using a min-heap. Details are given in Solution 20.9 on Page 310.

The solutions to Problems 20.8 on Page 308 and 20.17 on Page 316 also illustrate the use of parallelism.

Caching is a great tool whenever computations are repeated. For example, the central idea behind dynamic programming is caching results from intermediate computations. Caching is also extremely useful when implementing a service that is expected to respond to many requests over time, and many requests are repeated. Workloads on web services exhibit this property. Solution 19.1 on Page 291 sketches the design of an online spell correction service; one of the key issues is performing cache updates in the presence of concurrent requests. Solution 19.9 on Page 299 shows how multithreading combines with caching in code which tests the Collatz hypothesis.

## 20.1 Design a spell checker

Designing a good spelling correction system can be challenging. We discussed spelling correction in the context of edit distance (Problem 16.2 on Page 239). However, in that problem, we only computed the Levenshtein distance between a pair of strings. A spell checker must find a set of words that are closest to a given word from the entire dictionary. Furthermore, the Levenshtein

distance may not be the right distance function when performing spelling correction—it does not take into account the commonly misspelled words or the proximity of letters on a keyboard.

How would you build a spelling correction system?

*Hint:* Start with an appropriate notion of distance between words.

**Solution:** The basic idea behind most spelling correction systems is that the misspelled word's Levenshtein distance from the intended word tends to be very small (one or two edits). Hence, if we keep a hash table for all the words in the dictionary and look for all the words that have a Levenshtein distance of 2 from the text, it is likely that the intended word will be found in this set. If the alphabet has $m$ characters and the search text has $n$ characters, we need to perform $O(n^2m^2)$ hash table lookups. More precisely, for a word of length $n$, we can pick any two characters and change them to any other character in the alphabet. The total number of ways of selecting any two characters is $n(n-1)/2$, and each character can be changed to one of $(m-1)$ other chars. Therefore, the number of lookups is $n(n-1)(m-1)^2/2$.

The intersection of the set of all strings at a distance of two or less from a word and the set of dictionary words may be large. It is important to provide a ranked list of suggestions to the users, with the most likely candidates are at the beginning of the list. There are several ways to achieve this:

- Typing errors model—often spelling mistakes are a result of typing errors. Typing errors can be modeled based on keyboard layouts.
- Phonetic modeling—a big class of spelling errors happen when the person spelling it knows how the words sounds but does not know the exact spelling. In such cases, it helps to map the text to phonemes and then find all the words that map to the same phonetic sequence.
- History of refinements—often users themselves provide a great amount of data about the most likely misspellings by first entering a misspelled word and then correcting it. This historic data is often immensely valuable for spelling correction.
- Stemming—often the size of a dictionary can be reduced by keeping only the stemmed version of each word. (This entails stemming the query text.)

### 20.2 Design a solution to the stemming problem

When a user submits the query "computation" to a search engine, it is quite possible he might be interested in documents containing the words "computers", "compute", and "computing" also. If you have several keywords in a query, it becomes difficult to search for all combinations of all variants of the words in the query.

Stemming is the process of reducing all variants of a given word to one common root, both in the query string and in the documents. An example of stemming would be mapping {*computers*, *computer*, *compute*} to *compute*. It is almost impossible to succinctly capture all possible variants of all words in the English language but a few simple rules can get us most cases.

Design a stemming algorithm that is fast and effective.

*Hint:* The examples are suggestive of general rules.

**Solution:** Stemming is a large topic. Here we mention some basic ideas related to stemming—this is in no way a comprehensive discussion on stemming approaches.

Most stemming systems are based on simple rewrite rules, e.g., remove suffixes of the form "es", "s", and "ation". Suffix removal does not always work. For example, wolves should be stemmed to wolf. To cover this case, we may have a rule that replaces the suffix "ves" with "f".

Most rules amount to matching a set of suffixes and applying the corresponding transformation to the string. One way of efficiently performing this is to build a finite state machine based on all the rules.

A more sophisticated system might have exceptions to the broad rules based on the stem matching some patterns. The Porter stemmer, developed by Martin Porter, is considered to be one of the most authoritative stemming algorithms in the English language. It defines several rules based on patterns of vowels and consonants.

Other approaches include the use of stochastic methods to learn rewrite rules and $n$-gram based approaches where we look at the surrounding words to determine the correct stemming for a word.

### 20.3 Plagiarism detector

Design an efficient algorithm that takes as input a set of text files and returns pairs of files which have substantial commonality.

*Hint:* Design a hash function which can incrementally hash $S[i, i+k-1]$ for $i = 0, 1, 2, \ldots$.

**Solution:** We will treat each file as a string. We take a pair of files as having substantial commonality if they have a substring of length $k$ in common, where $k$ is a measure of commonality. (Later, we will have a deeper discussion as to the validity of this model.)

Let $l_i$ be the length of the string corresponding to the $i$th file. For each such string we can compute $l_i - k + 1$ hash codes, one for *each* $k$ length substring.

We insert these hash codes in a hash table $G$, recording which file each code corresponds to, and the offset the corresponding substring appears at. A collision indicates that the two length-$k$ substrings are potentially the same.

Since we are computing hash code for each $k$ length substring, it is important for efficiency to have a hash function which can be updated incrementally when we delete one character and add another. Solution 6.13 on Page 80 describes such a hash function.

In addition, it is important to have many slots in $G$, so that collisions of unequal strings is rare. A total of $\sum_{i=1}^{|S|}(l_i - k + 1)$ strings are added to the hash table. If $k$ is small relative to the string length and $G$ has significantly fewer slots than the total number of characters in the strings, then we are certain to have collisions.

If it is not essential to return an exact answer, we can save storage by only considering a subset of substrings, e.g., those whose hash codes have 0s in the last $b$ bits. This means that on average we consider $\frac{1}{2^b}$ of the total set of substrings (assuming the hash function does a reasonable job of spreading keys).

The solution presented above can lead to many false positives. For example, if each file corresponds to an HTML page, all pages with a common embedded script of length $k$ or longer will show up as having substantial overlap. We can account for this through preprocessing, e.g., parsing the documents and removing headers. (This may require multiple passes with some manual inspection driving the process.) This solution will also not work if, for example, the files are similar in that they are the exact same program, but with ids renamed. (It is however, resilient to code blocks being

moved around.) By normalizing ids, we can cast the problem back into one of finding common substrings.

The approach we have just described is especially effective when the number of files is very large and the files are spread over many servers. In particular, the map-reduce framework can be used to achieve the effect of a single $G$ spread over many servers.

## 20.4 Pair users by attributes

You are building a social network where each user specifies a set of attributes. You would like to pair each user with another unpaired user that specified the same set of attributes.

You are given a sequence of users where each user has a unique 32-bit integer key and a set of attributes specified as strings. When you read a user, you should pair that user with another previously read user with identical attributes who is currently unpaired, if such a user exists. If the user cannot be paired, you should add him to the unpaired set.

*Hint:* Map sets of attributes to strings.

**Solution:** First, we examine the algorithmic core of this problem. Later we will consider implementation details, particularly those related to scale.

A brute-force algorithm is to compare each new user's attributes with the attributes of users in the unpaired set. This leads to $O(n^2)$ time complexity, where $n$ is the number of users.

To improve the time complexity, we need a way to find users who have the same attributes as the new user. A hash table whose keys are subsets of attributes and values are users is the perfect choice. This leaves us with the problem of designing a hash function which is suitable for subsets of attributes.

If the total number of possible attributes is small, we can represent a subset of attributes with a bit array, where each index represents a specific attribute. Specifically, we store a 1 at an index if that attribute is present. We can then use any hash function for bit arrays. The time complexity to process the $n$ users is a hash lookup per user, i.e., $O(nm)$, where $m$ is the number of distinct attributes. The space complexity is also $O(nm)$—$n$ entries, each of size $m$.

If the set of possible attributes is large, and most users have a small subset of those attributes, the bit vector representation for subsets is inefficient from both a time and space perspective. A better approach to represent sparse subsets is directly record the elements. To ensure equal subsets have equal hash codes, we need to represent the subsets in a unique way. One approach is to sort the elements. We can view the sorted sequence of attributes as a single string, namely the concatenation the individual elements. We then use a hash function for strings. For example, if the possible attributes are {USA, Senior, Income, Prime Customer}, and a user has attributes {USA, Income}, we represent his set of attributes as the string "Income,USA".

The time complexity of this approach is $O(M)$, where $M$ is the sum of the sizes of the attribute sets for all users.

Now we consider implementation issues. Suppose the network is small enough that a single machine can store it. For such a system, you can use database with Users table, Attributes table, and a join table between them. Then for every attribute you will find matches (or no match) and based on criteria you can decide if user could join a group or start a new one. If you do not like to use a database and want to reinvent the wheel, you still can use similar approach and get a list

of matching user IDs for every attribute. To do it use reverse indexes, with a string hash for quick lookup in storage. Assuming arrays are returned sorted there is an easy way to merge arrays for multiple attributes and make a decision about matching to a group / starting new group.

Now suppose the network is too large to fit on a single machine. Because of number of users it makes sense to process problem on multiple machines—each will store subset of attributes and may return results as user IDs. We will need to make two merge operations:

- Identical searches—when a single attribute lookup was performed on many machines. We will need to merge returned sorted IDs into big sorted list of IDs.
- Searches by different attributes—we will need to merge those attributes if they all are present in all lists (or if the match criteria is above X%, e.g. 3/4 are matching).

Since in all likelihood we do not need to perform real-time matching we could use the Consumer-Producer pattern: pick up users from a queue and perform a search, thereby limit the number of simultaneous requests.

### 20.5 Design a system for detecting copyright infringement

YouTV.com is a successful online video sharing site. Hollywood studios complain that much of the material uploaded to the site violates copyright.

Design a feature that allows a studio to enter a set $V$ of videos that belong to it, and to determine which videos in the YouTV.com database match videos in $V$.

*Hint:* Normalize the video format and create signatures.

**Solution:** If we replaced videos everywhere with documents, we could use the techniques in Solution 20.3 on Page 304, where we looked for near duplicate documents by computing hash codes for each length-$k$ substring.

Videos differ from documents in that the same content may be encoded in many different formats, with different resolutions, and levels of compression.

One way to reduce the duplicate video problem to the duplicate document problem is to re-encode all videos to a common format, resolution, and compression level. This in itself does not mean that two videos of the same content get reduced to identical files—the initial settings affect the resulting videos. However, we can now "signature" the normalized video.

A trivial signature would be to assign a 0 or a 1 to each frame based on whether it has more or less brightness than average. A more sophisticated signature would be a 3 bit measure of the red, green, and blue intensities for each frame. Even more sophisticated signatures can be developed, e.g., by taking into account the regions on individual frames. The motivation for better signatures is to reduce the number of false matches returned by the system, and thereby reduce the amount of time needed to review the matches.

The solution proposed above is algorithmic. However, there are alternative approaches that could be effective: letting users flag videos that infringe copyright (and possibly rewarding them for their effort), checking for videos that are identical to videos that have previously been identified as infringing, looking at meta-information in the video header, etc.

**Variant:** Design an online music identification service.

## 20.6 Design TEX

The TEX system for typesetting beautiful documents was designed by Don Knuth. Unlike GUI based document editing programs, TEX relies on a markup language, which is compiled into a device independent intermediate representation. TEX formats text, lists, tables, and embedded figures; supports a very rich set of fonts and mathematical symbols; automates section numbering, cross-referencing, index generation; exports an API; and much more.

How would you implement TEX?

*Hint:* There are two aspects—building blocks (e.g., fonts, symbols) and hierarchical layout.

**Solution:** Note that the problem does not ask for the design of TEX, which itself is a complex problem involving feature selection, and language design. There are a number of issues common to implementing any such program: programming language selection, lexing and parsing input, error handling, macros, and scripting.

Two key implementation issues specific to TEX are a specifying fonts and symbols (e.g., A, b, $f$, $\sum$, $\oint$, $\uplus$), and assembling a document out of components.

Focusing on the second aspect, a reasonable abstraction is to use a rectangular bounding box to describe components. The description is hierarchical: each individual symbol is a rectangle, lines and paragraphs are made out of these rectangles and are themselves rectangles, as are section titles, tables and table entries, and included images. A key algorithmic problem is to assemble these rectangles, while preserving hard constraints on layout, and soft constraints on aesthetics. See also Problem 16.11 on Page 254 for an example of the latter.

Turning our attention to symbol specification, the obvious approach is to use a 2D array of bits to represent each symbol. This is referred to as a bit-mapped representation. The problem with bit-mapped fonts is that the resolution required to achieve acceptable quality is very high, which leads to huge documents and font-libraries. Different sizes of the same symbol need to be individually mapped, as do italicized and bold-face versions.

A better approach is to define symbols using mathematical functions. A reasonable approach is to use a language that supports quadratic and cubic functions, and elementary graphics transformations (rotation, interpolation, and scaling). This approach overcomes the limitations of bit-mapped fonts—parameters such as aspect ratio, font slant, stroke width, serif size, etc. can be programmed.

Other implementation issues include enabling cross-referencing, automatically creating indices, supporting colors, and outputting standard page description formats (e.g., PDF)).

Donald Knuth's book *"Digital Typography"* describes in great detail the design and implementation of TEX.

## 20.7 Design a search engine

Keyword-based search engines maintain a collection of several billion documents. One of the key computations performed by a search engine is to retrieve all the documents that contain the keywords contained in a query. This is a nontrivial task in part because it must be performed in a few tens of milliseconds.

Here we consider a smaller version of the problem where the collection of documents can fit within the RAM of a single computer.

Given a million documents with an average size of 10 kilobytes, design a program that can efficiently return the subset of documents containing a given set of words.

*Hint:* Build on the idea of a book's index.

**Solution:** The predominant way of doing this is to build inverted indices. In an inverted index, for each word, we store a sequence of locations where the word occurs. The sequence itself could be represented as an array or a linked list. Location is defined to be the document ID and the offset in the document. The sequence is stored in sorted order of locations (first ordered by document ID, then by offset). When we are looking for documents that contain a set of words, what we need to do is find the intersection of sequences for each word. Since the sequences are already sorted, the intersection can be done in time proportional to the aggregate length of the sequences. We list a few optimizations below.

- *Compression*—compressing the inverted index helps both with the ability to index more documents as well as memory locality (fewer cache misses). Since we are storing sorted sequences, one way of compressing is to use delta compression where we only store the difference between the successive entries. The deltas can be represented in fewer bits.
- *Caching*—the distribution of queries is usually fairly skewed and it helps a great deal to cache the results of some of the most frequent queries.
- *Frequency-based optimization*—since search results often do not need to return every document that matches (only top ten or so), only a fraction of highest quality documents can be used to answer most of the queries. This means that we can make two inverted indices, one with the high quality documents that stays in RAM and one with the remaining documents that stays on disk. This way if we can keep the number of queries that require the secondary index to a small enough number, then we can still maintain a reasonable throughput and latency.
- *Intersection order*—since the total intersection time depends on the total size of sequences, it would make sense to intersect the words with smaller sets first. For example, if we are looking for "USA GDP 2009", it would make sense to intersect the lists for GDP and 2009 before trying to intersect the sequence for USA.

We could also build a multilevel index to improve accuracy on documents. For a high priority web page, we can decompose the page into paragraphs and sentences, which are indexed individually. That way the intersections for the words might be within the same context. We can pick results with closer index values from these sequences. See the sorted array intersection problem 13.1 on Page 182 and digest problem 12.6 on Page 168 for related issues.

### 20.8 Implement PageRank

The PageRank algorithm assigns a rank to a web page based on the number of "important" pages that link to it. The algorithm essentially amounts to the following:

(1.) Build a matrix $A$ based on the hyperlink structure of the web. Specifically, $A_{ij} = \frac{1}{d_j}$ if page $j$ links to page $i$; here $d_j$ is the number of distinct pages linked from page $j$.

(2.) Find $X$ satisfying $X = \epsilon[\mathbf{1}] + (1-\epsilon)AX$. Here $\epsilon$ is a constant, e.g., $\frac{1}{7}$, and $[\mathbf{1}]$ represents a column vector of 1s. The value $X[i]$ is the rank of the $i$th page.

The most commonly used approach to solving the above equation is to start with a value of $X$, where each component is $\frac{1}{n}$ (where $n$ is the number of pages) and then perform the following

iteration: $X_k = \epsilon[\mathbf{1}] + (1 - \epsilon)AX_{k-1}$. The iteration stops when the $X_k$ converges, i.e., the difference from $X_k$ to $X_{k+1}$ is less than a specified threshold.

Design a system that can compute the ranks of ten billion web pages in a reasonable amount of time.

*Hint:* This must be performed on an ensemble of machines. The right data structures will simplify the computation.

**Solution:** Since the web graph can have billions of vertices and it is mostly a sparse graph, it is best to represent the graph as an adjacency list. Building the adjacency list representation of the graph may require a significant amount of computation, depending upon how the information is collected. Usually, the graph is constructed by downloading the pages on the web and extracting the hyperlink information from the pages. Since the URL of a page can vary in length, it is often a good idea to represent the URL by a hash code.

The most expensive part of the PageRank algorithm is the repeated matrix multiplication. Usually, it is not possible to keep the entire graph information in a single machine's RAM. Two approaches to solving this problem are described below.

- Disk-based sorting—we keep the column vector $X$ in memory and load rows one at a time. Processing Row $i$ simply requires adding $A_{i,j}X_j$ to $X_j$ for each $j$ such that $A_{i,j}$ is not zero. The advantage of this approach is that if the column vector fits in RAM, the entire computation can be performed on a single machine. This approach is slow because it uses a single machine and relies on the disk.
- Partitioned graph—we use $n$ servers and partition the vertices (web pages) into $n$ sets. This partition can be computed by partitioning the set of hash codes in such a way that it is easy to determine which vertex maps to which machine. Given this partitioning, each machine loads its vertices and their outgoing edges into RAM. Each machine also loads the portion of the PageRank vector corresponding to the vertices it is responsible for. Then each machine does a local matrix multiplication. Some of the edges on each machine may correspond to vertices that are owned by other machines. Hence the result vector contains nonzero entries for vertices that are not owned by the local machine. At the end of the local multiplication it needs to send updates to other hosts so that these values can be correctly added up. The advantage of this approach is that it can process arbitrarily large graphs.

PageRank runs in minutes on a single machine on the graph consisting of the multi-million pages that constitute Wikipedia. It takes roughly 70 iterations to converge on this graph. Anecdotally, PageRank takes roughly 200 iterations to converge on the graph consisting of the multi-billion pages that constitute the World-Wide Web.

When operating on a graph the scale of the webgraph, PageRank must be run on many thousands of machines. In such situations, it is very likely that a machine will fail, e.g., because of a defective power supply. The widely used Map-Reduce framework handles efficient parallelization as well as fault-tolerance. Roughly speaking, fault-tolerance is achieved by replicating data across a distributed file system, and having the master reassign jobs on machines that are not responsive.

## 20.9 Design TeraSort and PetaSort

Modern datasets are huge. For example, it is estimated that a popular social network contains over two trillion distinct items.

How would you sort a billion 1000 byte strings? How about a trillion 1000 byte strings?

*Hint:* Can a trillion 1000 byte strings fit on one machine?

**Solution:** A billion 1000 byte strings cannot fit in the RAM of a single machine, but can fit on the hard drive of a single machine. Therefore, one approach is to partition the data into smaller blocks that fit in RAM, sort each block individually, write the sorted block to disk, and then combine the sorted blocks. The sorted blocks can be merged using for example Solution 10.1 on Page 134. The UNIX `sort` program uses these principles when sorting very large files, and is faster than direct implementations of the merge-based algorithm just described.

If the data consists of a trillion 1000 byte strings, it cannot fit on a single machine—it must be distributed across a cluster of machines. The most natural interpretation of sorting in this scenario is to organize the data so that lookups can be performed via binary search. Sorting the individual datasets is not sufficient, since it does not achieve a global ordering—lookups entail a binary search on each machine. The straightforward solution is to have one machine merge the sorted datasets, but then that machine will become the bottleneck.

A solution which does away with the bottleneck is to first reorder the data so that the $i$th machine stores strings in a range, e.g., Machine 3 is responsible for strings that lie between *daily* and *ending*. The range-to-machine mapping $R$ can be computed by sampling the individual files and sorting the sampled values. If the sampled subset is small enough, it can be sorted by a single machine. The techniques in Solution 11.8 on Page 153 can be used to determine the ranges assigned to each machine. Specifically, let $A$ be the sorted array of sampled strings. Suppose there are $M$ machines. Define $r_i = iA[n/M]$, where $n$ is the length of $A$. Then Machine $i$ is responsible for strings in the range $[r_i, r_{i+1})$. If the distribution of the data is known *a priori*, e.g., it is uniform, the sampling step can be skipped.

The reordering can be performed in a completely distributed fashion by having each machine route the strings it begins with to the responsible machines.

After reordering, each machine sorts the strings it stores. Consequently queries such as lookups can be performed by using $R$ to determine which individual machine to forward the lookup to.

## 20.10 Implement distributed throttling

You have $n$ machines ("crawlers") for downloading the entire web. The responsibility for a given URL is assigned to the crawler whose ID is Hash(URL) mod $n$. Downloading a web page takes away bandwidth from the web server hosting it.

Implement crawling under the constraint that in any given minute your crawlers do not request more than $b$ bytes from any website.

*Hint:* Use a server to coordinate the crawl.

**Solution:** This problem, as posed, is ambiguous.

- Since we usually download one file in one request, if a file is greater than $b$ bytes, there is no way we can meet the constraint of serving fewer than $b$ bytes every minute, unless we can work with the lower layers of the network stack such as the transport layer or the network layer. Often the system designer could look at the distribution of file sizes and conclude that this problem happens so infrequently that we do not care. Alternatively, we may choose to download no more than the first $b$ bytes of any file.
- Given that the host's bandwidth is a resource for which there could be contention, one important design choice to be made is how to resolve a contention. Do we let requests get served in first-come first-served order or is there a notion of priority? Often crawlers have a built-in notion of priority based on how important the document is to the users or how fresh the current copy is.

One way of doing this could be to maintain a permission server with which each crawler checks to see if it is okay to hit a particular host. The server can keep an account of how many bytes have been downloaded from the server in the last minute and not permit any crawler to hit the server if we are already close to the quota. If we do not care about priority, then we can keep the interface synchronous where a server requests permission to download a file and it immediately gets approved or denied. If we care about priorities, then the server may enqueue the request and inform the crawler when the file is available for download. The queues at the permission server may be based on priorities.

If the permission server becomes a bottleneck, we can use multiple permission servers such that the responsibility of a given host is decided by applying a hash function to the host name and assigning it to a particular server based on the hash code.

### 20.11 Design a scalable priority system

Maintaining a set of prioritized jobs in a distributed system can be tricky. Applications include a search engine crawling web pages in some prioritized order, as well as event-driven simulation in molecular dynamics. In both cases the number of jobs is in the billions and each has its own priority.

Design a system for maintaining a set of prioritized jobs that implements the following API: (1.) insert a new job with a given priority; (2.) delete a job; (3.) find the highest priority job. Each job has a unique ID. Assume the set cannot fit into a single machine's memory.

*Hint:* How would you partition jobs across machines? Is it always essential to operate on the highest priority job?

**Solution:** If we have enough RAM on a single machine, the most simple solution would be to maintain a min-heap where entries are ordered by their priority. An additional hash table can be used to map jobs to their corresponding entry in the min-heap to make deletions fast.

A more scalable solution entails partitioning the problem across multiple machines. One approach is to apply a hash function to the job ids and partition the resulting hash codes into ranges, one per machine. Insert as well as delete require communication with just one server. To do extract-min, we send a lookup minimum message to all the machines, infer the min from their responses, and then delete it.

At a given time many clients may be interested in the highest priority event, and it is challenging to distribute this problem well. If many clients are trying to do this operation at the same time, we

may run into a situation where most clients will find that the min event they are trying to extract has already been deleted. If the throughput of this service can be handled by a single machine, we can make one server solely responsible for responding to all the requests. This server can prefetch the top hundred or so events from each of the machines and keep them in a heap.

In many applications, we do not need strong consistency guarantees. We want to spend most of our resources taking care of the highest priority jobs. In this setting, a client could pick one of the machines at random, and request the highest priority job. This would work well for the distributed crawler application. It is not suited to event-driven simulation because of dependencies.

Another other issue to consider is resilience: if a node fails, all list of work on that node fails as well. It is better to have nodes to contain overlapped lists and the dispatching node in this case will handle duplicates. The lost of a node shouldn't result in full re-hashing—the replacement node should handle only new jobs. Consistent hashing can be used to achieve this.

A front-end caching server can become a bottleneck. This can be avoided by using replication, i.e., multiple servers which duplicate each other. There could be possible several ways to coordinate them: use non-overlapping lists, keep a blocked job list, return a random job from the jobs with highest priority.

### 20.12 Create photomosaics

A photomosaic is built from a collection of images called "tiles" and a target image. The photomosaic is another image which approximates the target image and is built by juxtaposing the tiles. Quality is defined by human perception.

Design a program that produces high quality mosaics with minimal compute time.

*Hint:* How would you define the distance between two images?

**Solution:** A good way to begin is to partition the image into $s \times s$-sized squares, compute the average color of each such image square, and then find the tile that is closest to it in the color space. Distance in the color space can be the $L2$-distance over the Red-Green-Blue (RGB) intensities for the color. As you look more carefully at the problem, you might conclude that it would be better to match each tile with an image square that has a similar structure. One way could be to perform a coarse pixelization ($2 \times 2$ or $3 \times 3$) of each image square and finding the tile that is "closest" to the image square under a distance function defined over all pixel colors. In essence, the problem reduces to finding the closest point from a set of points in a $k$-dimensional space.

Given $m$ tiles and an image partitioned into $n$ squares, then a brute-force approach would have $O(mn)$ time complexity. You could improve on this by first indexing the tiles using an appropriate search tree. You can also run the matching in parallel by partitioning the original image into subimages and searching for matches on the subimages independently.

### 20.13 Implement Mileage Run

Airlines often give customers who fly frequently with them a "status". This status allows them early boarding, more baggage, upgrades to executive class, etc. Typically, status is a function of miles flown in the past twelve months. People who travel frequently by air sometimes want to take

a round trip flight simply to maintain their status. The destination is immaterial—the goal is to minimize the cost-per-mile (cpm), i.e., the ratio of dollars spent to miles flown.

Design a system that will help its users find mileage runs.

*Hint:* Partition the implied features into independent tasks.

**Solution:** There are two distinct aspects to the design. The first is the user-facing portion of the system. The second is the server backend that gets flight-price-distance information and combines it with user input to generate the alerts.

We begin with the user-facing portion. For simplicity, we illustrate it with a web-app, with the realization that the web-app could also be written as a desktop or mobile app. The web-app has the following components: a login page, a manage alerts page, a create an alert page, and a results page. For such a system we would like defer to a single-sign-on login service such as that provided by Google or Facebook. The management page would present login information, a list of alerts, and the ability to create an alert.

One reasonable formulation of an alert is that it is an origin city, a target cpm, and optionally, a date or range of travel dates. The results page would show flights satisfying the constraints. Note that other formulations are also possible, such as how frequently to check for flights, a set of destinations, a set of origins, etc.

The classical approach to implement the web-app front end is through dynamically generated HTML on the server, e.g., through Java Server Pages. It can be made more visually appealing and intuitive by making appropriate use of cascaded style sheets, which are used for fonts, colors, and placements. The UI can be made more efficient through the use of Javascript to autocomplete common fields, and make attractive date pickers.

Modern practice is to eschew server-side HTML generation, and instead have a single-page application, in which Javascript reads and writes JavaScript Object Notation (JSON) objects to the server, and incrementally updates the single-page based. The AngularJS framework supports this approach.

The web-app backend server has four components: gathering flight data, matching user-generated alerts to this data, persisting data and alerts, and generating the responses to browser initiated requests.

Flight data can be gathered via "scraping" or by subscribing to a flight data service. Scraping refers to extraction of data from a website. It can be quite involved—some of the issues are parsing the results from the website, filling in form data, and running the Javascript that often populates the actual results on a page. Selenium is a Java library that can programmatically interface to the Firefox browser, and is appropriate for scraping sites that are rich in Javascript. Most flight data services are paid. ITA software provides a very widely used paid aggregated flight data feed service. The popular Kayak site provides an Extensible Markup Language (XML) feed of recently discovered fares, which can be a good free alternative. Flight data does not include the distance between airports, but there are websites which return the distance between airport codes which can be used to generate the cpm for a flight.

There are a number of common web application frameworks—essentially libraries that handle many common tasks—that can be used to generate the server. Java and Python are very commonly used for writing the backend for web applications.

Persistence of data can be implemented through a database. Most web application frameworks provide support for automating the process of reading and writing objects from and to a database. Finally, web application frameworks can route incoming HTTP requests to appropriate code—this is through a configuration file matching URLs to methods. The framework provides convenience methods for accessing HTTP fields and writing results. Frameworks also provide HTTP templating mechanisms, wherein developers intersperse HTML with snippets of code that dynamically add content to the HTML.

Web application frameworks typically implement cron functionality, wherein specified functions are executed at a regular interval. This can be used to periodically scrape data and check if the condition of an alert is matched by the data.

Finally, the web app can be deployed via a platform-as-a-service such as Amazon Web Services and Google App Engine.

### 20.14 Implement Connexus

How would you design Connexus, a system by which users can share pictures? Address issues around access control (public, private), picture upload, organizing pictures, summarizing sets of pictures, allowing feedback, and displaying geographical and temporal information for pictures.

*Hint:* Think about the UI, in particular UI widgets that make for an engaging product.

**Solution:** There are three aspects to Connexus. The first is the server backend, used to store images, and meta-data, such as author, comments, hashtags, GPS coordinates, etc., as well as run `cron` jobs to identify trending streams. The technology for this is similar to Mileage Run (Solution 20.13 on the previous page), with the caveat that a database is not suitable for storing large binary objects such as images, which should be stored on a local or remote file system. (The database will hold references to the file.)

The web UI is also similar to Mileage Run, with a login page, a management page, and pages for displaying images. Images can be grouped based on a concept of a stream, with comment boxes annotating streams. Facebook integration would make it easier to share links to streams, and post new images as status updates. Search capability and discussion boards also enhance the user experience.

Some UI features that are especially appealing are displaying images by location on a zoomable map, slider UI controls to show subsets of images in a stream based on the selected time intervals, a file upload dialog with progress measures, support for multiple simultaneous uploads, and drag-and-drop upload. All these UI widgets are provided by, for example, the jQuery-UI Javascript library. (This library also makes the process of creating autocompletion on text entry fields trivial.)

Connexus is an application that begs for a mobile client. A smartphone provides a camera, location information, and push notifications. These can be used to make it easier to create and immediately upload geo-tagged images, find nearby images, and be immediately notified on updates and comments. The two most popular mobile platforms, iOS and Android, have APIs rich in UI widgets and media access. Both can use serialization formats such as JSON or protocol buffers to communicate with the server via remote procedure calls layered over HTTP.

Jingle, a search engine startup, has been very successful at providing a high-quality Internet search service. A large number of customers have approached Jingle and asked it to display paid advertisements for their products and services alongside search results.

Design an advertising system for Jingle.

*Hint:* Consider the stakeholders separately.

**Solution:** Reasonable goals for such a system include

- providing users with the most relevant ads,
- providing advertisers the best possible return on their investment, and
- minimizing the cost and maximizing the revenue to Jingle.

Two key components for such a system are:

- The front-facing component, which advertisers use to create advertisements, organize campaigns, limit when and where ads are shown, set budgets, and create performance reports.
- The ad-serving system, which selects which ads to show on the searches.

The front-facing system can be a fairly conventional web application, i.e., a set of web pages, middleware that responds to user requests, and a database. Key features include:

- User authentication—a way for users to create accounts and authenticate themselves. Alternatively, use an existing single sign-on login service, e.g., Facebook or Google.
- User input—a set of form elements to let advertisers specify ads, advertising budget, and search keywords to bid on.
- Performance reports—a way to generate reports on how the advertiser's money is being spent.
- Customer service—even the best of automated systems require occasional human interaction, e.g., ways to override limits on keywords. This requires an interface for advertisers to contact customer service representatives, and an interface for those representatives to interact with the system.

The whole front-end system can be built using, for example, HyperText Markup Language (HTML) and JavaScript. A commonly used approach is to use a LAMP stack on the server-side: Linux as the OS, Apache as the HTTP server, MySQL as the database software, and PHP for the application logic.

The ad-serving system is less conventional. The ad-serving system would build a specialized data structure, such as a decision tree, from the ads database. It chooses ads from the database of ads based on their "relevance" to the search. In addition to keywords, the ad-serving systems can use knowledge of the user's search history, how much the advertiser is willing to pay, the time of day, user locale, and type of browser. Many strategies can be envisioned here for estimating relevance, such as, using information retrieval or machine learning techniques that learn from past user interactions.

The ads could be added to the search results by embedding JavaScript in the results page. This JavaScript pulls in the ads from the ad-serving system directly. This helps isolate the latency of serving search results from the latency of serving ad results.

There are many more issues in such a system: making sure there are no inappropriate images using an image recognition API; using a link verification to check if keywords really corresponds to a real site; serving up images from a content-delivery network; and having a fallback advertisement to show if an advertisement cannot be found.

### 20.16 Design a recommendation system

Jingle wants to generate more page views on its news site. A product manager has the idea to add to each article a sidebar of clickable snippets from articles that are likely to be of interest to someone reading the current article.

Design a system that automatically generates a sidebar of related articles.

*Hint:* This problem can be solved with various degrees of algorithmic sophistication: none at all, simple frequency analysis, or machine learning.

**Solution:** The key technical challenge in this problem is to come up with the list of articles—the code for adding these to a sidebar is trivial.

One suggestion might be to add articles that have proved to be popular recently. Another is to have links to recent news articles. A human reader at Jingle could tag articles which he believes to be significant. He could also add tags such as finance, sports, and politics, to the articles. These tags could also come from the HTML meta-tags or the page title.

We could also provide randomly selected articles to a random subset of readers and see how popular these articles prove to be. The popular articles could then be shown more frequently.

On a more sophisticated level, Jingle could use automatic textual analysis, where a similarity is defined between pairs of articles—this similarity is a real number and measures how many words are common to the two. Several issues come up, such as the fact that frequently occurring words such as "for" and "the" should be ignored and that having rare words such as "arbitrage" and "diesel" in common is more significant than having say, "sale" and "international".

Textual analysis has problems, such as the fact that two words may have the same spelling but completely different meanings (anti-virus means different things in the context of articles on acquired immune deficiency syndrome (AIDS) and computer security). One way to augment textual analysis is to use collaborative filtering—using information gleaned from many users. For example, by examining cookies and timestamps in the web server's log files, we can tell what articles individual users have read. If we see many users have read both $A$ and $B$ in a single session, we might want to recommend $B$ to anyone reading $A$. For collaborative filtering to work, we need to have many users.

### 20.17 Design an optimized way of distributing large files

Jingle is developing a search feature for breaking news. New articles are collected from a variety of online news sources such as newspapers, bulletin boards, and blogs, by a single lab machine at Jingle. Every minute, roughly one thousand articles are posted and each article is 100 kilobytes.

Jingle would like to serve these articles from a data center consisting of a 1000 servers. For performance reasons, each server should have its own copy of articles that were recently added. The data center is far away from the lab machine.

Design an efficient way of copying one thousand files each 100 kilobytes in size from a single lab server to each of 1000 servers in a distant data center.

*Hint:* Exploit the data center.