```
62    Random rand = new Random();
63    return rand.nextInt(max + 1 - min) + min;
64  }
65
66  /* Swap values at index i and j. */
67  void swap(int[] array, int i, int j) {
68    int t = array[i];
69    array[i] = array[j];
70    array[j] = t;
71  }
72
73  /* Get largest element in array between left and right indices. */
74  int max(int[] array, int left, int right) {
75    int max = Integer.MIN_VALUE;
76    for (int i = left; i <= right; i++) {
77      max = Math.max(array[i], max);
78    }
79    return max;
80  }
```

If the elements are not unique, we can tweak this algorithm slightly to accommodate this.

Approach 4: Selection Rank Algorithm (if elements are not unique)

The major change that needs to be made is to the `partition` function. When we partition the array around a pivot element, we now partition it into three chunks: less than `pivot`, equal to `pivot`, and greater than `pivot`.

This requires minor tweaks to `rank` as well. We now compare the size of left and middle partitions to `rank`.

```
1   class PartitionResult {
2     int leftSize, middleSize;
3     public PartitionResult(int left, int middle) {
4       this.leftSize = left;
5       this.middleSize = middle;
6     }
7   }
8
9   int[] smallestK(int[] array, int k) {
10    if (k <= 0 || k > array.length) {
11      throw new IllegalArgumentException();
12    }
13
14    /* Get item with rank k - 1. */
15    int threshold = rank(array, k - 1);
16
17    /* Copy elements smaller than the threshold element. */
18    int[] smallest = new int[k];
19    int count = 0;
20    for (int a : array) {
21      if (a < threshold) {
22        smallest[count] = a;
23        count++;
24      }
25    }
26
27    /* If there's still room left, this must be for elements equal to the threshold
```

```
28      * element. Copy those in. */
29     while (count < k) {
30        smallest[count] = threshold;
31        count++;
32     }
33
34     return smallest;
35  }
36
37  /* Find value with rank k in array. */
38  int rank(int[] array, int k) {
39     if (k >= array.length) {
40        throw new IllegalArgumentException();
41     }
42     return rank(array, k, 0, array.length - 1);
43  }
44
45  /* Find value with rank k in sub array between start and end. */
46  int rank(int[] array, int k, int start, int end) {
47     /* Partition array around an arbitrary pivot. */
48     int pivot = array[randomIntInRange(start, end)];
49     PartitionResult partition = partition(array, start, end, pivot);
50     int leftSize = partition.leftSize;
51     int middleSize = partition.middleSize;
52
53     /* Search portion of array. */
54     if (k < leftSize) { // Rank k is on left half
55        return rank(array, k, start, start + leftSize - 1);
56     } else if (k < leftSize + middleSize) { // Rank k is in middle
57        return pivot; // middle is all pivot values
58     } else { // Rank k is on right
59        return rank(array, k - leftSize - middleSize, start + leftSize + middleSize,
60                    end);
61     }
62  }
63
64  /* Partition result into < pivot, equal to pivot -> bigger than pivot. */
65  PartitionResult partition(int[] array, int start, int end, int pivot) {
66     int left = start; /* Stays at (right) edge of left side. */
67     int right = end;  /* Stays at (left) edge of right side. */
68     int middle = start; /* Stays at (right) edge of middle. */
69     while (middle <= right) {
70        if (array[middle] < pivot) {
71           /* Middle is smaller than the pivot. Left is either smaller or equal to
72            * the pivot. Either way, swap them. Then middle and left should move by
73            * one. */
74           swap(array, middle, left);
75           middle++;
76           left++;
77        } else if (array[middle] > pivot) {
78           /* Middle is bigger than the pivot. Right could have any value. Swap them,
79            * then we know that the new right is bigger than the pivot. Move right by
80            * one. */
81           swap(array, middle, right);
82           right--;
83        } else if (array[middle] == pivot) {
```

```
84          /* Middle is equal to the pivot. Move by one. */
85          middle++;
86        }
87      }
88
89      /* Return sizes of left and middle. */
90      return new PartitionResult(left - start, right - left + 1);
91  }
```

Notice the change made to `smallestK` too. We can't simply copy all elements less than or equal to `threshold` into the array. Since we have duplicates, there could be many more than k elements that are less than or equal to `threshold`. (We also can't just say "okay, only copy k elements over." We could inadvertently fill up the array early on with "equal" elements, and not leave enough space for the smaller ones.)

The solution for this is fairly simple: only copy over the smaller elements first, then fill up the array with equal elements at the end.

17.15 Longest Word: Given a list of words, write a program to find the longest word made of other words in the list.

pg 188

SOLUTION

This problem seems complex, so let's simplify it. What if we just wanted to know the longest word made of *two* other words in the list?

We could solve this by iterating through the list, from the longest word to the shortest word. For each word, we would split it into all possible pairs and check if both the left and right side are contained in the list.

The pseudocode for this would look like the following:

```
1   String getLongestWord(String[] list) {
2     String[] array = list.SortByLength();
3     /* Create map for easy lookup */
4     HashMap<String, Boolean> map = new HashMap<String, Boolean>;
5
6     for (String str : array) {
7       map.put(str, true);
8     }
9
10    for (String s : array) {
11      // Divide into every possible pair
12      for (int i = 1; i < s.length(); i++) {
13        String left = s.substring(0, i);
14        String right = s.substring(i);
15        // Check if both sides are in the array
16        if (map[left] == true && map[right] == true) {
17          return s;
18        }
19      }
20    }
21    return str;
22  }
```

This works great for when we just want to know composites of two words. But what if a word could be formed by any number of other words?

In this case, we could apply a very similar approach, with one modification: rather than simply looking up if the right side is in the array, we would recursively see if we can build the right side from the other elements in the array.

The code below implements this algorithm:

```
String printLongestWord(String arr[]) {
  HashMap<String, Boolean> map = new HashMap<String, Boolean>();
  for (String str : arr) {
    map.put(str, true);
  }
  Arrays.sort(arr, new LengthComparator()); // Sort by length
  for (String s : arr) {
    if (canBuildWord(s, true, map)) {
      System.out.println(s);
      return s;
    }
  }
  return "";
}

boolean canBuildWord(String str, boolean isOriginalWord,
                     HashMap<String, Boolean> map) {
  if (map.containsKey(str) && !isOriginalWord) {
    return map.get(str);
  }
  for (int i = 1; i < str.length(); i++) {
    String left = str.substring(0, i);
    String right = str.substring(i);
    if (map.containsKey(left) && map.get(left) == true &&
        canBuildWord(right, false, map)) {
      return true;
    }
  }
  map.put(str, false);
  return false;
}
```

Note that in this solution we have performed a small optimization. We use a dynamic programming/ memoization approach to cache the results between calls. This way, if we repeatedly need to check if there's any way to build "`testingtester`," we'll only have to compute it once.

A boolean flag `isOriginalWord` is used to complete the above optimization. The method `canBuildWord` is called for the original word and for each substring, and its first step is to check the cache for a previously calculated result. However, for the original words, we have a problem: `map` is initialized to `true` for them, but we don't want to return `true` (since a word cannot be composed solely of itself). Therefore, for the original word, we simply bypass this check using the `isOriginalWord` flag.

17.16 The Masseuse: A popular masseuse receives a sequence of back-to-back appointment requests and is debating which ones to accept. She needs a 15-minute break between appointments and therefore she cannot accept any adjacent requests. Given a sequence of back-to-back appointment requests (all multiples of 15 minutes, none overlap, and none can be moved), find the optimal (highest total booked minutes) set the masseuse can honor. Return the number of minutes.

EXAMPLE

Input: `{30, 15, 60, 75, 45, 15, 15, 45}`

Output: `180 minutes ({30, 60, 45, 45}).`

pg 188

SOLUTION

Let's start with an example. We'll draw it visually to get a better feel for the problem. Each number indicates the number of minutes in the appointment.

Alternatively, we could have also divided all the values (including the break) by 15 minutes, to give us the array `{5, 7, 8, 5, 6, 9}`. This would be equivalent, but now we would want a 1-minute break.

The best set of appointments for this problem has 330 minutes total, formed with $\{r_0 = 75, r_2 = 120, r_5 = 135\}$. Note that we've intentionally chosen an example in which the best sequence of appointments was not formed through a strictly alternating sequence.

We should also recognize that choosing the longest appointment first (the "greedy" strategy) would not necessarily be optimal. For example, a sequence like `{45, 60, 45, 15}` would not have 60 in the optimal set.

Solution #1: Recursion

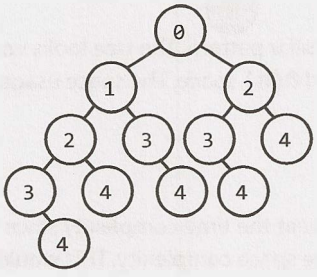
The first thing that may come to mind is a recursive solution. We have essentially a sequence of choices as we walk down the list of appointments: Do we use this appointment or do we not? If we use appointment `i`, we must skip appointment `i + 1` as we can't take back-to-back appointments. Appointment `i + 2` is a possibility (but not necessarily the best choice).

```
1  int maxMinutes(int[] massages) {
2    return maxMinutes(massages, 0);
3  }
4
5  int maxMinutes(int[] massages, int index) {
6    if (index >= massages.length) { // Out of bounds
7      return 0;
8    }
9
10   /* Best with this reservation. */
11   int bestWith = massages[index] + maxMinutes(massages, index + 2);
12
13   /* Best without this reservation. */
14   int bestWithout = maxMinutes(massages, index + 1);
15
16   /* Return best of this subarray, starting from index. */
17   return Math.max(bestWith, bestWithout);
18 }
```

The runtime of this solution is $O(2^n)$ because at each element we're making two choices and we do this n times (where n is the number of massages).

The space complexity is $O(n)$ due to the recursive call stack.

We can also depict this through a recursive call tree on an array of length 5. The number in each node represents the `index` value in a call to `maxMinutes`. Observe that, for example, `maxMinutes(massages, 0)` calls `maxMinutes(massages, 1)` and `maxMinutes(massages, 2)`.



As with many recursive problems, we should evaluate if there's a possibility to memoize repeated subproblems. Indeed, there is.

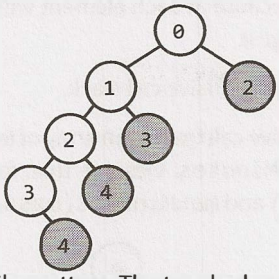
Solution #2: Recursion + Memoization

We will repeatedly call `maxMinutes` on the same inputs. For example, we'll call it on index 2 when we're deciding whether to take appointment 0. We'll also call it on index 2 when we're deciding whether to take appointment 1. We should memoize this.

Our memo table is just a mapping from `index` to the max minutes. Therefore, a simple array will suffice.

```
1  int maxMinutes(int[] massages) {
2    int[] memo = new int[massages.length];
3    return maxMinutes(massages, 0, memo);
4  }
5
6  int maxMinutes(int[] massages, int index, int[] memo) {
7    if (index >= massages.length) {
8      return 0;
9    }
10
11   if (memo[index] == 0) {
12     int bestWith = massages[index] + maxMinutes(massages, index + 2, memo);
13     int bestWithout = maxMinutes(massages, index + 1, memo);
14     memo[index] = Math.max(bestWith, bestWithout);
15   }
16
17   return memo[index];
18 }
```

To determine the runtime, we'll draw the same recursive call tree as before but gray-out the calls that will return immediately. The calls that will never happen will be deleted entirely.



If we drew a bigger tree, we'd see a similar pattern. The tree looks very linear, with one branch down to the left. This gives us an O(n) runtime and O(n) space. The space usage comes from the recursive call stack as well as from the memo table.

Solution #3: Iterative

Can we do better? We certainly can't beat the time complexity since we have to look at each appointment. However, we might be able to beat the space complexity. This would mean not solving the problem recursively.

Let's look at our first example again.

$r_0 = 30$	$r_1 = 15$	$r_2 = 60$	$r_3 = 75$	$r_4 = 45$	$r_5 = 15$	$r_6 = 15$	$r_7 = 45$

As we noted in the problem statement, we cannot take adjacent appointments.

There's another observation, though, that we can make: We should never *skip* three consecutive appointments. That is, we might skip r_1 and r_2 if we wanted to take r_0 and r_3. But we would never skip r_1, r_2, and r_3. This would be suboptimal since we could always improve our set by grabbing that middle element.

This means that if we take r_0, we know we'll definitely skip r_1 and definitely take either r_2 or r_3. This substantially limits the options we need to evaluate and opens the door to an iterative solution.

Let's think about our recursive + memoization solution and try to reverse the logic; that is, let's try to approach it iteratively.

A useful way to do this is to approach it from the back and move toward the start of the array. At each point, we find the solution for the subarray.

- `best(7)`: What's the best option for {$r_7 = 45$}? We can get 45 min. if we take r_7, so `best(7) = 45`
- `best(6)`: What's the best option for {$r_6 = 15$, ...}? Still 45 min., so `best(6) = 45`.
- `best(5)`: What's the best option for {$r_5 = 15$, ...}? We can either:
  - » take $r_5 = 15$ and merge it with `best(7) = 45`, or:
  - » take `best(6) = 45`.

  The first gives us 60 minutes, `best(5) = 60`.
- `best(4)`: What's the best option for {$r_4 = 45$, ...}? We can either:
  - » take $r_4 = 45$ and merge it with `best(6) = 45`, or:
  - » take `best(5) = 60`.

  The first gives us 90 minutes, `best(4) = 90`.
- `best(3)`: What's the best option for {$r_3 = 75$, ...}? We can either:
  - » take $r_3 = 75$ and merge it with `best(5) = 60`, or:

- » take `best(4) = 90`.

The first gives us 135 minutes, `best(3) = 135`.

- `best(2)`: What's the best option for $\{r_2 = 60, \ldots\}$? We can either:
  - » take $r_2 = 60$ and merge it with `best(4) = 90`, or:
  - » take `best(3) = 135`.

  The first gives us 150 minutes, `best(2) = 150`.
- `best(1)`: What's the best option for $\{r_1 = 15, \ldots\}$? We can either:
  - » take $r_1 = 15$ and merge it with `best(3) = 135`, or:
  - » take `best(2) = 150`.

  Either way, `best(1) = 150`.
- `best(0)`: What's the best option for $\{r_0 = 30, \ldots\}$? We can either:
  - » take $r_0 = 30$ and merge it with `best(2) = 150`, or:
  - » take `best(1) = 150`.

  The first gives us 180 minutes, `best(0) = 180`.

Therefore, we return 180 minutes.

The code below implements this algorithm.

```
1  int maxMinutes(int[] massages) {
2    /* Allocating two extra slots in the array so we don't have to do bounds
3     * checking on lines 7 and 8. */
4    int[] memo = new int[massages.length + 2];
5    memo[massages.length] = 0;
6    memo[massages.length + 1] = 0;
7    for (int i = massages.length - 1; i >= 0; i--) {
8      int bestWith = massages[i] + memo[i + 2];
9      int bestWithout = memo[i + 1];
10     memo[i] = Math.max(bestWith, bestWithout);
11   }
12   return memo[0];
13 }
```

The runtime of this solution is `O(n)` and the space complexity is also `O(n)`.

It's nice in some ways that it's iterative, but we haven't actually "won" anything here. The recursive solution had the same time and space complexity.

Solution #4: Iterative with Optimal Time and Space

In reviewing the last solution, we can recognize that we only use the values in the memo table for a short amount of time. Once we are several elements past an index, we never use that element's index again.

In fact, at any given index `i`, we only need to know the best value from `i + 1` and `i + 2`. Therefore, we can get rid of the memo table and just use two integers.

```
1  int maxMinutes(int[] massages) {
2    int oneAway = 0;
3    int twoAway = 0;
4    for (int i = massages.length - 1; i >= 0; i--) {
5      int bestWith = massages[i] + twoAway;
6      int bestWithout = oneAway;
```

```
        int current = Math.max(bestWith, bestWithout);
        twoAway = oneAway;
        oneAway = current;
    }
    return oneAway;
}
```

This gives us the most optimal time and space possible: O(n) time and O(1) space.

Why did we look backward? It's a common technique in many problems to walk backward through an array.

However, we can walk forward if we want. This is easier for some people to think about, and harder for others. In this case, rather than asking "What's the best set that starts with a[i]?", we would ask "What's the best set that ends with a[i]?"

17.17 Multi Search: Given a string b and an array of smaller strings T, design a method to search b for each small string in T.

pg 189

SOLUTION

Let's start with an example:

```
T = {"is", "ppi", "hi", "sis", "i", "ssippi"}
b = "mississippi"
```

Note that in our example, we made sure to have some strings (like "is") that appear multiple times in b.

Solution #1

The naive solution is reasonably straightforward. Just search through the bigger string for each instance of the smaller string.

```
HashMapList<String, Integer> searchAll(String big, String[] smalls) {
    HashMapList<String, Integer> lookup =
        new HashMapList<String, Integer>();
    for (String small : smalls) {
        ArrayList<Integer> locations = search(big, small);
        lookup.put(small, locations);
    }
    return lookup;
}

/* Find all locations of the smaller string within the bigger string. */
ArrayList<Integer> search(String big, String small) {
    ArrayList<Integer> locations = new ArrayList<Integer>();
    for (int i = 0; i < big.length() - small.length() + 1; i++) {
        if (isSubstringAtLocation(big, small, i)) {
            locations.add(i);
        }
    }
    return locations;
}

/* Check if small appears at index offset within big. */
boolean isSubstringAtLocation(String big, String small, int offset) {
    for (int i = 0; i < small.length(); i++) {
        if (big.charAt(offset + i) != small.charAt(i)) {
```

```
26          return false;
27        }
28      }
29      return true;
30  }
31
32  /* HashMapList<String, Integer> is a HashMap that maps from Strings to
33   * ArrayList<Integer>. See appendix for implementation. */
```

We could have also used a `substring` and `equals` function, instead of writing `isAtLocation`. This is slightly faster (though not in terms of big O) because it doesn't require creating a bunch of substrings.

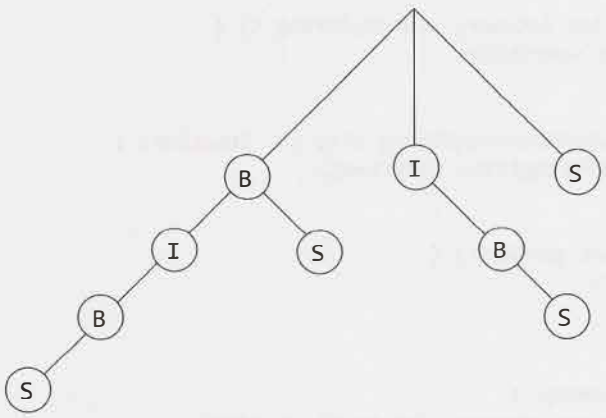
This will take $O(kbt)$ time, where k is the length of the longest string in T, b is the length of the bigger string, and t is the number of smaller strings within T.

Solution #2

To optimize this, we should think about how we can tackle all the elements in T at once, or somehow re-use work.

One way is to create a trie-like data structure using each suffix in the bigger string. For the string `bibs`, the suffix list would be: `bibs`, `ibs`, `bs`, `s`.

The tree for this is below.



Then, all you need to do is search in the suffix tree for each string in T. Note that if "B" were a word, you would come up with two locations.

```
1   HashMapList<String, Integer> searchAll(String big, String[] smalls) {
2     HashMapList<String, Integer> lookup = new HashMapList<String, Integer>();
3     Trie tree = createTrieFromString(big);
4     for (String s : smalls) {
5       /* Get terminating location of each occurrence.*/
6       ArrayList<Integer> locations = tree.search(s);
7
8       /* Adjust to starting location. */
9       subtractValue(locations, s.length());
10
11      /* Insert. */
12      lookup.put(s, locations);
13    }
14    return lookup;
```