Recursion Problems: Group A

The following problems all involve recursion, memoization, or dynamic programming. Try to see if you can come up with the most efficient solutions possible!

1. Generate all strings of n pairs of balanced parentheses. For example, if $n = 3$, you'd generate the strings ((())), (()()), (())(), ()(()), ()()().

2. You are given a pyramid of numbers like the one shown here:

   137

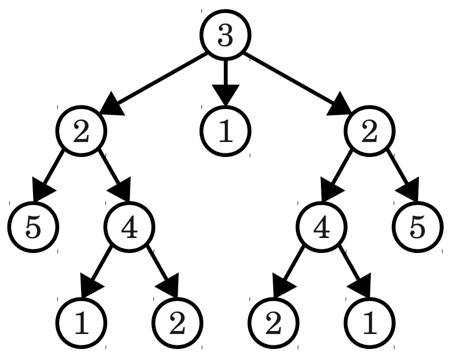
   42 -15

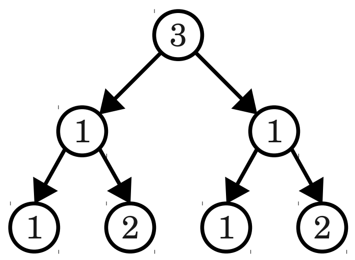
   -4 13 45

   21 14 -92 33

   Values in the pyramid can be both positive or negative. A path from the top of the pyramid to the bottom consists of starting at the top of the pyramid and taking steps diagonally left or diagonally right down to the bottom of the pyramid. The cost of a path is the sum of all the values in the pyramid. Find the path from the top of the pyramid to the bottom with the highest total cost.

3. A *palindromic tree* is a tree that is the same when it's mirrored around the root. For example, the left tree below is a palindromic tree and the right tree below is not:

   Given a tree, determine whether it is a palindromic tree.

4. The Fibonacci strings are a series of recursively-defined strings. F_0 is the string **a**, F_1 is the string **bc**, and F_{n+2} is the concatenation of F_n and F_{n+1}. For example, F_2 is **abc**, F_3 is **bcabc**, F_4 is **abcbcabc**, etc. Given a number n and an index k, return the kth character of the string F_n.

Recursion Problems: Group B

The following problems all involve recursion, memoization, or dynamic programming. Try to see if you can come up with the most efficient solutions possible!

1. Given a number n, generate all distinct ways to write n as the sum of positive integers. For example, with $n = 4$, the options are 4, 3 + 1, 2 + 2, 2 + 1 + 1, and 1 + 1 + 1 + 1.
2. In a binary tree, a *common value subtree* is a complete subtree where every node has the same value. (A complete subtree is a subtree consisting of a node and all its children). Determine the largest common value subtree in a nonempty binary tree.
3. Suppose you have a multiway tree where each node has an associated integer value. Find a set of nodes with the maximum possible sum, subject to the constraint that you cannot choose a node and any of its children at the same time.
4. Suppose that you have a group of people that you need to assign into different houses. For each house, you know the number of people that the house can hold. Additionally, you know that some people *insist* that they not be put into the same house as some other people. Given the list of pairs of people that can't be put into houses and the house capacities, determine how to distribute the people into the houses, or report that it's impossible.

Recursion Problem Solutions: Group A

1. Generate all strings of n pairs of balanced parentheses. For example, if $n = 3$, you'd generate the strings ((())), (()()), (())(), ()(()), ()()().

There are many possible solutions to this problem. I'll outline two of them here.

Option One: Enumerate all strings of n copies of (and n copies of) and, for each, check whether or not those strings are balanced strings of parentheses. For each one that is a string of balanced parentheses, output it. Here is some pseudocode for this:

```
function allBalancedStrings(n) {
    allBalancedStringsRec(n, n, "")
}
function allBalancedStringsRec(numOpensLeft, numClosesLeft, soFar) {
    if numOpensLeft is 0 and numClosesLeft is 0:
        print soFar if soFar is balanced
    else
        if numOpensLeft > 0:
            allBalancedStringsRec(numOpensLeft - 1, numClosesLeft, soFar + '(')
        if numClosesLeft > 0
            allBalancedStringsRec(numOpensLeft, numClosesLeft - 1, soFar + ')')
}
```

This approach works, but isn't ideal because it generates a large number of imbalanced strings. It turns out to generate exactly $n + 1$ times more strings than it should,* which given that there are exponentially many possible strings is a lot of overhead!

Option Two: Use the following recursive insight. Any string of n pairs of balanced parentheses will have an open parenthesis in the first position. This will then get matched against some other close parenthesis, splitting the string into two pieces, as shown here:

(*paren-group-one*) *paren-group-two*

Therefore, one way to generate all strings of n balanced parentheses is the following. For all numbers i ranging from 0 up to $n - 1$, generate all possible strings of i balanced parentheses and all possible strings of $n - 1 - i$ parentheses. Then, for each combination of a string of i parentheses and a string of $n - 1 - i$ parentheses, parenthesize the first string and append the second. This is shown here:

```
function allBalancedStrings(n) {
    if n is 0, return a list containing the empty string.
    for i from 0 to n - 1, inclusive:
        for each string x in allBalancedStrings(i):
            for each string y in allBalancedStrings(n - 1 - i):
                append '(' + x + ')' + y to the result list.
    return the resulting list.
}
```

Assuming that you memoize the results to avoid generating the same strings multiple times, this will generate every string exactly once.

* We're not expecting anyone to be able to come up with this figure off the top of their heads. You pretty much have to look this up or already know something about how many strings of balanced parentheses are possible.

2. You are given a pyramid of numbers like the one shown here:

```
      137
    42   -15
  -4   13   45
21   14  -92   33
```

   Values in the pyramid can be both positive or negative. A path from the top of the pyramid to the bottom consists of starting at the top of the pyramid and taking steps diagonally left or diagonally right down to the bottom of the pyramid. The cost of a path is the sum of all the values in the pyramid. Find the path from the top of the pyramid to the bottom with the highest total cost.

There are 2^n possible paths from the top to the bottom in a pyramid of height n (do you see why?), so brute-forcing the answer won't be at all efficient. Fortunately, you don't have to!

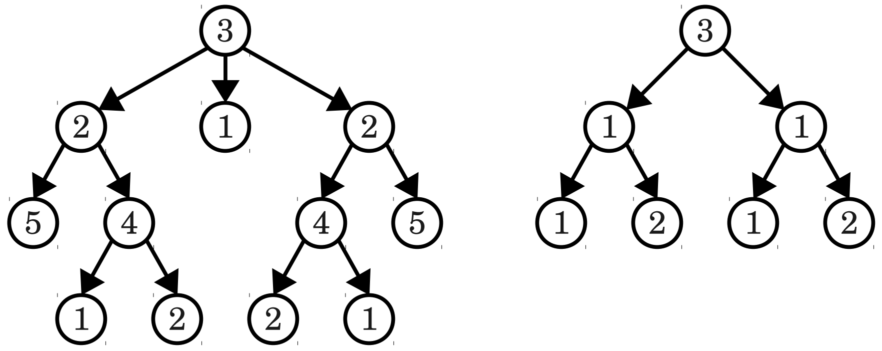
The main observation necessary to solve this problem efficiently is that the first step is either to the left or to the right, and the rest of the path should be an optimal path from your new location to the destination. Therefore, you can recursively compute the cost of the best possible path from the left and right children of the starting location, then combine that with information about the values of those children to get the cost of the optimal paths starting with a step left or a step right. From there, you can determine the optimal solution by simply taking the better of the two.

Here is some pseudocode to determine the *cost* of an optimal solution; I'll leave the task of actually finding the solution as an exercise.

```
function optimalPath(t) { // Note: Very inefficient; see below for details.
    if t is a triangle of height 1, return the only value in the triangle.
    return the maximum of t.left.value  + optimalPath(t.left) and
                          t.right.value + optimalPath(t.right)
}
```

This approach will recompute the the optimal paths for many of the internal nodes (do you see why?), so it is not at all efficient. In fact, it runs in time O(2^n). However, if we modify it by either memoizing the results or using dynamic programming to compute the values from the bottom of the tree upward, that overhead is eliminated. In that case, we spend only O(1) time per value in the triangle, so the runtime is linear in the number of elements in the triangle.

3. A *palindromic tree* is a tree that is the same when it's mirrored around the root. For example, the left tree below is a palindromic tree and the right tree below is not:



   Given a tree, determine whether it is a palindromic tree.

One simple approach is to compute the mirror of the original tree, then determine whether the mirror of the original tree is equal to the original tree. Here's some pseudocode for this:

```
function mirrorTree(t) {
    if t is null, return null.
    let result be a new node with value t.value
    for each child c of t, in reverse order:
        append mirrorTree(c) to result's child list
    return result
}
function treesEqual(t₁, t₂) {
    if both t₁ and t₂ are null, return true.
    if either t₁ or t₂ are null, return false.
    if t₁.value is not t₂.value, return false.
    if t₁ and t₂ have different numbers of children, return false.
    for each pair of children (c₁, c₂) in (t₁.children, t₂.children):
        if treesEqual(c₁, c₂) is false, return false.
    return true
}
function isPalindromicTree(t) {
    return treesEqual(t, mirrorTree(t))
}
```

The `mirrorTree` and `treesEqual` functions each do O(1 + num children) work per node in the tree. Summing up across all nodes in the tree, this works out to work linear in the number of nodes in the trees, and therefore this approach runs in linear time. However, this approach requires $O(n)$ space because it makes a copy of the tree. Another approach would be to check whether the tree is a mirror of itself in-place, which is conveniently left as an exercise to the reader. ☺

4. The Fibonacci strings are a series of recursively-defined strings. F_0 is the string **a**, F_1 is the string **bc**, and F_{n+2} is the concatenation of F_n and F_{n+1}. For example, F_2 is **abc**, F_3 is **bcabc**, F_4 is **abcbcabc**, etc. Given a number n and an index k, return the kth character of the string F_n.

This is a problem where the naïve solution won't work for large n and k. For example, if $n = 100$, then F_{100} has about 3×10^{21} characters, which certainly won't fit into memory. However, since all you need to do is produce characters at specific positions in the string, you don't need to store all the Fibonacci strings in memory.

Notice that the nth Fibonacci string has length f_{n+3}, where f_{n+3} is the $(n+3)$rd Fibonacci number. You can check this by looking at the first few Fibonacci strings and, if you'd like, writing a quick proof by induction to confirm it. By the definition of the Fibonacci numbers, we know that f_{n+3} is equal to $f_{n+1} + f_{n+2}$. This means that (if $n \geq 2$) that the first f_{n+1} characters of F_n come from F_{n-2} and the next f_{n+2} characters come from F_{n-1}. This gives a recursive algorithm that works by recursively descending into the first or second portion of the Fibonacci strings:

```
function kthChar(n, k) {
    if n is zero and k is zero, return 'a'
    if n is one and k is zero, return 'b'
    if n is one and k is one, return 'c'
    if k < $f_{n+1}$, return kthChar(n - 2, k)
    return kthChar(n - 1, k - $f_{n+1}$)
}
```

This approach runs in time O(n) plus the additional work to compute the appropriate Fibonacci numbers. If you precompute all Fibonacci numbers up to and including n, which can be done in time O(n), then you can look up individual Fibonacci numbers in time O(1) and the total runtime will be O(n).

As an interesting exercise – since the Fibonacci numbers grow exponentially quickly, if n and k are given as 32-bit or 64-bit integers, you can optimize this code by relying on the fact that k will eventually always be in the first portion of the string. Try thinking about how you might take this into account and see if you get the solution to run in time O(1)!

Recursion Problem Solutions: Group B

1. Given a number n, generate all distinct ways to write n as the sum of positive integers. For example, with $n = 4$, the options are 4, 3 + 1, 2 + 2, 2 + 1 + 1, and 1 + 1 + 1 + 1.

The main challenge in solving this problem is figuring out how to avoid getting duplicate solutions like 3 + 1 and 1 + 3. The approach I've outlined in this solution works by always generating the numbers in nonincreasing order, so we will never generate solutions like 1 + 3.

Here's a solution that works by strengthening the recursion. We will actually answer the harder question "list all distinct ways to write n as the sum of positive integers, where all the integers in the sum are less than or equal to k." Given this, we can make the following insight:

- There is just one way to write 0 as the sum of integers up to and including k: namely, it's the empty sum of no numbers.
- Otherwise, the sum must have at least one integer in it. For each possible first integer i (which must be in the range of 1 to the minimum of n and k), try writing the number as the sum of i, plus all possible ways of summing to $n - i$ using numbers no greater than i.

This gives the following recursive solution:

```
function allPartitionsOf(n) {
      return restrictedAllPartitions(n, n);
}
function restrictedAllPartitions(n, k) {
      if n is zero, the only solution is the empty sum.
      for i from 1 to the minimum of n and k, inclusive:
            record all solutions of the form i + restrictedAllPartitions(n - i, i)
      return those solutions
}
```

This solution will recompute many subproblems, so you could consider using memoization or DP to try to avoid the recomputations. This unfortunately increases the memory usage, so it's up to you to decide whether it's a good idea or not.

2. In a binary tree, a *common value subtree* is a complete subtree where every node has the same value. (A complete subtree is a subtree consisting of a node and all its children). Determine the largest common value subtree in a nonempty binary tree.

For notational simplicity, let's call a common value subtree a CVS. The key observation necessary here is the following:

- A tree with just one node is a CVS whose value is the value in the root.
- A tree with root *r* and subtrees $t_1, t_2, \ldots, t_n$ is a CVS if each of $t_1, t_2, \ldots, t_n$ is a CVS and has the same value as the root's value.

This means that we can do a bottom-up pass over the tree and find all of the CVS's as we go. We can then determine which CVS found this way is the largest. The pseudocode below splits this into several passes over the tree just for simplicity, and can be significantly space-optimized by combining everything together into one pass.

```
function findLargestCVS(t) {
      annotateTreeSizes(t)
      annotateTreeCVS(t)
      return largestCVSIn(t)
}
function annotateTreeSizes(t) {
      if t is a leaf, set t.size = 1
      else:
        call annotateTreeSizes(c) for each child c of t.
        set t.size = 1 + sum(c.size) for each child c of t
}
function annotateCVS(t) {
      if t is a leaf, set t.isCVS to true.
      else:
        call annotateCVS(c) for each child c of t.
        set t.isCVS to whether c.value = t.value and c.isCVS for each child c of t.
}
function largestCVSIn(t) {
      if t.isCVS return t.size
      else
        let dₖ = largestCVSIn(cₖ) for each child cₖ of t.
        return argmax{ dₖ.size } for all dₖ.
}
```

This code runs in time O(n), where n is the number of nodes in the trees. To see this, note that every function call takes time O(1 + num children), so summing up across all nodes in the tree we can charge O(1) total work to each node in the tree. Summing up across all n nodes gives a runtime of O(n).

3. Suppose you have a multiway tree where each node has an associated integer value. Find a set of nodes with the maximum possible sum, subject to the constraint that you cannot choose a node and any of its children at the same time.

The insight necessary to solve this problem is to split the problem into two cases – first, solving this problem when the root node *is* included in the solution, and second when the root node is *not* included in the solution. If we include the root node, then all of its children must not be included, and we'd like optimal solutions for each of its subtrees subject to the restriction that their root nodes aren't included. If we exclude the root node, then for each child, we should take the best solution possible, whether or not we choose to include the root node.

If we code this up using a naïve recursion, we get this solution:

```
function maxSum(t) {
      return maxSumUnrestricted(t)
}

// Gives the largest possible sum that can be made with the tree rooted at t when
// there are no restrictions on whether t must be included.
function maxSumUnrestricted(t) {
      return the max of maxSumRestricted(t, true) and maxSumRestricted(t, false);
}

// Gives the largest possible sum that can be made with the tree rooted at t
// subject to the restriction that the root of t either must be or must not be
// included
function maxSumRestricted(t, include) {
   if t is null, return 0.

   if mustInclude is true:
      return t.value + sum of maxSumRestricted(c, false) for all children c of t.

   if mustInclude is false:
      return the sum of maxSumUnrestricted(c) for all children c of t
}
```

This recursion is correct but highly inefficient – we'll end up recomputing the same subproblems on many subtrees, leading to a very slow runtime. However, there are only $2n$ possible unique calls in a tree of n nodes – for each node, we can either include it or exclude it – and so we can memoize the results or use dynamic programming to compute the results bottom-up. If we do this, the net runtime is only O(n) because we do O(1) work per node O(n) total times, though it now requires O(n) storage space.

4. Suppose that you have a group of people that you need to assign into different houses. For each house, you know the number of people that the house can hold. Additionally, you know that some people *insist* that they not be put into the same house as some other people. Given the list of pairs of people that can't be put into houses and the house capacities, determine how to distribute the people into the houses, or report that it's impossible.

This problem is actually equivalent to the graph coloring problem – you can imagine that you have a node for each person, an edge between two people if they insist on not being in the same house, and one color for each house. Since this problem is, in general, NP-hard, it's probably best to approach this through some kind of backtracking approach. One natural one is the following:

- If all people are assigned, you're done. Report success.
- Otherwise, choose a person.
- Determine what possible houses they can be in by checking where their neighbors have been assigned.
- If there are no options, backtrack.
- Otherwise, for each choice, try that choice. If any of them work, report success; otherwise report failure.