# Triple Step (8.1)

Constraints:

* Hopping 1 step means taking every step? Yes
* The child can freely alternate between the 1-step, 2-steps, and 3-steps options in one run

Example:

* n = 3
* Options:
  + 3 x 1 step
  + 1 x 3 steps
  + 1 x 2 steps + 1 x 1 step
  + 1 x 1 step + 1 x 2 steps
* Answer: 4 possible ways

~~Bottom-up approach:~~

* n = 0
  + 0 ways
* n = 1
  + 1 way
* n = 2
  + 2 ways
    - 1 1
    - 2
* n = 3
  + 4 ways
    - 1 1 1
    - 3
    - 2 1
    - 1 2
* n = 4
  + 7 ways
    - 1 1 1 1
    - 3 1
    - 1 3
    - 2 2
    - 1 1 2
    - 1 2 1
    - 2 1 1

~~Top-down approach:~~

* The number to run up n steps is:
  + If n == 0, then 0
  + Else, if n == 1, then 1
  + Else, if n == 2, then 2
  + Else, if n == 3, then 4
  + Else, it's the number of ways to run up 3 steps times the number of ways to run up n-3 steps

**Top-down approach:**

* The number to run up n steps is:
  + The number to run up n-3 steps + the number to run up n-2 steps + the number to run up n-1 steps
* Why? Because the last hop is either a 3-hop, 2-hop, or 1-hop
  + The option of a 3-hop last hop can be achieved in f(n-3) ways
  + The option of a 2-hop last sep can be achieved in a f(n-2) ways
  + The option of a 1-hop last step can be achieved in f(n-1) ways
* For example, if we know the number to run up n-3 steps, we don't need to bother that now we could also take a 1-hop and a 2-hop, or three 1-hops, etc., because these possibilities would be already included in the number of ways to run up n-1 and n-1 steps.

Implementation:

class TripleStep {

public static void main(String[] args) {

int n = Integer.parseInt(args[0]);

System.out.println(calc(n));

}

private static int calc(int n) {

switch (n) {

case 1: return 1;

case 2: return 2;

case 3: return 4;

}

return calc(n-3) + calc(n-2) + calc(n-1);

}

}

Complexity:

* Draw the recursive call tree
  + Each node has 3 children
  + If the method is called with n, there are n-2 levels in the tree
  + The max. number of nodes in the tree is 3^(n-2)-1
* Thus, the complexity is O(3^n).

**Optimisation (top-down dynamic programming):**

* Whenever we calculated f(n), cache it for further calls of the f method
  + That is, we have to pass the cache with the method calls

class TripleStep2 {

public static void main(String[] args) {

int n = Integer.parseInt(args[0]);

System.out.println(calc(n));

}

private static int calc(int n) {

return calcRec(n, new int[n+1]);

}

private static int calcRec(int n, int[] memo) {

switch (n) {

case 1: return 1;

case 2: return 2;

case 3: return 4;

}

if (memo[n] == 0) {

memo[n] = calcRec(n-1, memo) + calcRec(n-2, memo) + calcRec(n-3, memo);

}

return memo[n];

}

}

Complexity:

* Draw the recursive call tree and indicate which calls return their value from the cache
* Now there are stil n-2 levels in the tree, but none of these levels is full
  + Actually, there's only a single path going from the root to the base cases at the deepest level of the tree
* The number of nodes in the tree is roughly 3(n-3)
  + n-3 nodes on the path from the root until before the deepest leaf, and each of these nodes has three children
* Thus, the time complexity is O(n)
* The space complexity is also O(n), since the recursion tree has n-2 levels, thus there are up to n-2 functions on the call stack
  + This is a common problem with recursive implementations

**Optimisation (bottom-up dynamic programming):**

* The goal is to keep the time complexity at O(n), but reduce the space complexity to O(1)

class TripleStep3 {

public static void main(String[] args) {

int n = Integer.parseInt(args[0]);

System.out.println(calc(n));

}

private static int calc(int n) {

switch (n) {

case 1: return 1;

case 2: return 2;

case 3: return 4;

}

int a = 1; // Result of calc(1)

int b = 2; // Result of calc(2)

int c = 4; // Result of calc(3)

for (int i = 4; i <= n; i++) {

int tmp = a + b + c;

a = b;

b = c;

c = tmp;

}

return c;

}

}

* The time complexity clearly is O(n), as there is only one iteration of n-3 steps
* The space complexity is O(1), there are only three variables and one function call (no recursion)

# Robot in a Grid (8.2)

Constraints:

* Is the robot on the grid or inside the cells? Inside the cells, only the cells are important
* The field consists of r x c cells
* Must there be a path? No, it's possible that no path exists
* In each cell (except the destination cell) there are 4 possibilities
  + It's possible to move right and down
  + It's possible to move right, but not down
  + It's possible to move down, but not right
  + It's not possible to move neither right nor down (this means that no path exists

Idea:

* First, we have to check if there exists a path or not
* There exists a path from the start cell to any other cell if and only if:
  + This cell is "allowed" (not an "off-limit" cell)
  + There exists a path from the destination cell to the cell to the left of it OR there exists a path from the destination cell to the cell on top of it

Implementation:

class Robot {

public static void main(String[] args) {

int r = Integer.parseInt(args[0]);

int c = Integer.parseInt(args[1]);

hasPath(r,c);

}

private static boolean hasPath(int row, int col) {

if (row < 1 || col < 1) return false;

if (row == 1 && col == 1) return true;

return isAllowed(row,col) &&

(hasPath(row, col-1) || hasPath(row-1, col));

}

}

Complexity:

* Draw the recursive call tree
* The call tree has r+c+1 levels
  + Because at each level, either r or c decreases by one, that is, the sum r+c decreases by one (so, r+c+1 calls until 0)
* Each node has at most 2 children
* Thus, the complexity is O(2^(r+c))

Optimised (top-down dynamic programming):

class Robot2 {

public static void main(String[] args) {

int r = Integer.parseInt(args[0]);

int c = Integer.parseInt(args[1]);

hasPath(r,c);

}

private static boolean hasPath(int row, int col) {

return hasPathRec(row, col, new int[row+1][col+1]);

}

private static boolean hasPathRec(int row, int col, int[][] memo) {

if (row < 1 || col < 1) return false;

if (row == 1 && col == 1) return true;

if (memo[row][col] == 0) {

if (isAllowed(row,col) && (hasPathRec(row, col-1, memo) || hasPath(row-1, col, memo)))

memo[row][col] = 1;

else

memo[row][col] = -1;

}

return memo[row][col] == 1;

}

}

Complexity:

* Draw the recursive call tree and indicate the function calls that can take their result from the cache
* How many nodes (which are recursive calls and not cache hits) are there in the tree? There is one node (recursive call) for each cell of the grid. There are r\*c cells in the grid
* Thus, the complexity is O(r\*c)

# Power Set (8.4)

Constraints:

* A set of n elements has 2^n subsets (including the empty set and the subset containing all elements of n)
* The method takes as input a set and returns a list of sets

Idea:

* All the subsets of a set of n elements are:
  + The union of the first element with all the subsets of the remaining n-1 elements PLUS
  + The subsets of the remaining n-1 elements
* Base case: if the set is empty, the list of all the subsets of this set contains only the empty set
* Recursion:
  + Remove one element from the set
  + Calculate the list of subsets of the set consisting of the remaining elements
  + Put in the list of all subsets:
    - All the subsets of the remaining elements
    - The union of the removed element with all the subsets of the remaining elements

Complexity:

* We have no overlapping subproblems here, so probably we cannot improve the complexity of the algorithm by dynamic programming
* If the set has n elements, we have n+1 recursive calls until the empty set
* ~~Therefore, the time complexity is O(n)~~
* ~~The space complexity is also O(n), because of the recursive calls~~
* There are 2^n subsets, and each subsets contains n/2 elements on average
* That is, there are n \* 1/2 \* 2^n = n \* 2^(n-1) elements across all the subsets
  + That's the minimum number of steps and space that we need
* Thus, the time and space complexity is O(n\*2^n)

Implementation:

class PowerSet {

public static void main(String[] args) {

Set<Integer> set = new HashSet<>();

Scanner sc = new Scanner(System.in);

while (sc.hasNextInt()) {

set.add(sc.nextInt());

}

List<Set<Integer>> subsets = getSubsets(set);

for (Set<Integer> s : subsets) {

System.out.println(s);

}

}

private static List<Set<Integer>> getSubsets(Set<Integer> set) {

if (set.isEmpty()) {

List<Set<Integer>> subsets = new ArrayList<>();

subsets.add(new HashSet<>());

return subsets;

}

Iterator<Integer> it = set.iterator();

int element = it.next();

set.remove(element);

List<Set<Integer>> list = getSubsets(set);

List<Set<Integer>> newList = new ArrayList<>();

for (Set<Integer> s : list) {

Set<Integer> newSet = new HashSet(s);

newSet.add(element);

newList.add(newSet);

}

list.addAll(newList);

return list;

}

}

# Towers of Hanoi (8.6)

Constraints:

* At the beginning the disk are ordered in pyramid form on the first tower.
* A disk can be moved from one tower to any other tower that is either empty, or has a disk on the top that is LARGER than the moved disk
  + A disk can be moved to a tower that does NOT contain a smaller disk at its top
* Before moving another disk, must the previously moved disk be put on a tower? Yes

Idea:

* We can model the problem with three stacks 0, 1, and 2
  + At the beginning, stack 0 holds the integers N to 1, with 1 being at the head of the stack
  + The other two stacks are empty
* Base case: stack 0 holds only one disk
  + Move this disk from stack 0 to stack 2
* Case stack 0 has n disks:
  + Solve problem for n-1 with destination stack 1 (not 2)
    - All the disks on top of the largest disks are now on stack 1
  + Move disk n to stack 2
  + Solve problem for n-1 with start stack 1 and destination stack 2

Implementation:

import java.util.Stack;

class TowersOfHanoi {

private static Stack<Integer> stack0 = new Stack<>();

private static Stack<Integer> stack1 = new Stack<>();

private static Stack<Integer> stack2 = new Stack<>();

public static void main(String[] args) {

int n = Integer.parseInt(args[0]);

for (int i = n; i <= 1; i--) {

stack0.push(i);

}

hanoi(n, stack0, stack1, stack2);

System.out.println("Stack 0: " + stack0);

System.out.println("Stack 1: " + stack1);

System.out.println("Stack 2: " + stack2);

}

private static void hanoi(int n, Stack<Integer> from, Stack<Integer> tmp, Stack<Integer> to) {

if (n == 1) {

move(from, to);

return;

}

hanoi(n-1, from, to, tmp);

move(from, to);

hanoi(n-1; tmp, from, to);

}

private static void move(Stack<Integer> from, Stack<Integer> to) {

to.push(from.pop());

}

}

Complexity:

* For each call hanoi(n), we have two calls hanoi(n-1)
* There are n levels in the recursive call tree until reaching the base case n=1
* Therefore, there are (2^n)-1 recursive calls, and the time complexity is O(2^n)
* The space complexity is also O(2^n), because each recursive calls uses some space on the call stack

# Parens (8.9)

Example

* Base case: n=1
  + Output:()
* n=2
  + Output: ()(), (())
* n=3
  + Output: ()()(), (())(), ()(()), (()()), ((()))

Idea:

* The combinations for n can be obtained by first calculating the combinations for n-1, and then inserting the additional pair "()" into every possible spot of all the combinations of the n-1 solution
  + If there are n-1 pairs, there are 2\*(n-1)+1 spots to insert an additional pair
* This produces many duplicate combinations
* How to remove the duplicates?
  + We can insert every new combination in a hash table and then for every new combination check whether we already have it in the hash table
* How to represent the combinations of parentheses pairs?
  + Easy: as strings of 2n characters
* If we have the solution of n-1 (a set of strings of 2(n-1) characters):
  + For each string:
    - Create a new string of 2n characters (solution of n)
    - Iterate i from 0 to 2(n-1)
      * Insert the substring from 0 to i (exclusive) of the n-1 solution into the new string
      * Then add the new parentheses pair "()" to the new string
      * Then, add the substring from i to "end" of the n-1 solution to the new string (the part of the n-1 solution that has not been previously added)
    - Check if this solution is contained in the set of new combinations
    - If yes, continue to the next set
    - Else, if no, add the string to the set
  + Return the set

Implementation:

import java.util.Set;

import java.util.HashSet;

import java.util.Arrays;

class Parens {

public static void main(String[] args) {

int n = Integer.parseInt(args[0]);

Set<String> parens = parens(n);

int i = 1;

for (String s : parens) {

System.out.println(i + ": " + s);

i++;

}

}

private static Set<String> parens(int n) {

// Base case

if (n == 0) return new HashSet<>(Arrays.asList(""));

// Recursion

Set<String> setPrev = parens(n-1);

Set<String> setNew = new HashSet<>();

for (String prev : setPrev) {

for (int i = 0; i <= 2\*(n-1); i++) {

StringBuilder sb = new StringBuilder(2\*n);

sb.append(prev.substring(0, i));

sb.append("()");

sb.append(prev.substring(i));

setNew.add(sb.toString()); // Adds only if not already exists

}

}

return setNew;

}

}

Complexity:

* There are n recursive calls
* The max. number of combinations for n is 2\*(n-1)+1
* n goes down from n to 1
* The complexity is O(n!)
* In any case, the complexity is more than exponential (e.g O(2^n)). This can be seen by calculating some results for e.g. n=10 (n=13 has 742,900 combinations)