Recursions

August 8, 2020

1 Recursions

https://opendsa-server.cs.vt.edu/ODSA/Books/CS2/html/RecIntro.html

1.0.1 Table of Contents

1.1 Recursion Introduction

- an algorithm (or a function in a computer program) is recursive if it invokes itself to do part of its work
- the process of solving a large problem by reducing it to one or more sub-problems which are identical in structure to the original problem and somewhat simpler to solve
  - eventually sub-problems become so simple that they can be solved without further division
- for a recursive approach to be successful, the recursive "call to itself" must be on a smaller problem than the one originally attempted
- in general, recursive algorithm must have two parts:
  1. **base case** - handles a simple input that has direct answer (stops recursion)
     - must have at least one or more base cases
  2. **general case** - recursive part that contains one or more recursive calls to the algorithm
     - must have at least one or more general cases
     - each general case leads to one of the base cases

1.2 pros

- makes it possible to solve complex problems with concise solutions that may directly map the recursive problem statement (Tower of Hanoii problem)
- focus on base case and how to combine the sub-problems (process of delegating tasks)

1.3 cons

- not intuitive to understand and may not have a counterpart in everyday problem solving
  - must adopt the attitude that sub-problems take care of themselves (taking a leap of faith)
- typically less efficient compared to iterative counterparts
  - lots of function calls have computational and memory overhead

- must be able to understand how to read and write recursive functions

1.4 Real-world example

1.4.1 example 1

- a project manager divides a large project and delegates sub-tasks to many workers
  - s/he simply uses the reports from each worker to submit a final report of completion
  - assuming each sub-task is similar

1.4.2 example 2

- Let's say you're in the last row of a movie theater and want to know the row number. How can you find out?

1.5 Writing a recursive function

- think of the smallest version of the problem and solve it and expand it to the larger
- steps:
  1. write a function header in a way that you can call it to solve the smallest problem
  - call the function with the smallest problem case
    - mark this as your base case in function implementation
  - expand the function header to solve a generic larger problem if necessary
    - add this as the general case
    - use the solution from the base case if necessary
- example:

```
returnType functionName(inputData) {
    if (base case) {
        // answer directly found and return if necessary
    }
    else {
        // call functionName(smallerInputData)
        // use the result if functionName returns a value
        // return your final answer
    }
}
```

1.5.1 Exercise 1

- Write a recursive solution to a countdown problem (countdown to "Happy New Year!", "Blast Off!", etc.)

[1]:
```
#include <iostream>
using namespace std;
```

[2]:
```
void countDown(int n) {
    // base case
    if (n == 0)
        cout << "Blast Off!" << endl;
```

```
    // general case
    else {
        cout << n << endl;
        countDown(n-1);
    }
}
```

[3]:
```
countDown(10);
```

```
10
9
8
7
6
5
4
3
2
1
Blast Off!
```

1.5.2 visualize it at pythontutor.com: https://goo.gl/LB3ET6

1.5.3 Compile and run demo-programs/recursions/countDown.cpp

1.5.4 Exercise 2

- Find sum of n elements stored in an array using recursive solution.
  - using loop would be easier, right?

[1]:
```
// demonstrate it in the class from scratch
#include <iostream>
using namespace std;
```

[2]:
```
// sum of 1 number is that number
// sum of n number is firstNumber + sum(remainingNumbers)
// which one is base and which one is general?
int sum(int nums[], int startIndex, int endIndex) {
    if (startIndex == endIndex)
        return nums[startIndex];
    else {
        int s = nums[startIndex] + sum(nums, startIndex + 1, endIndex);
        return s;
    }
}
```

[3]:
```
int s = 0;
int nums[] = {100};
```

[4]:
```
int nums1[] = {10, 20, 30};
```

[5]:
```
s = sum(nums1, 0, 2);
cout << "sum = " << s << endl;
```

```
sum = 60
```

1.5.5 Exercise 3

- Find maximum/minimum value in an array of n elements using recursive solution
- Practice it!!

[6]:
```
int max(int nums[], int i, int j) {
    // FIMXE: base case
    // FIXME: general case
    return 0;
}
```

1.5.6 Exercise 4

- Find a factorial of n (n!) using recursive solution
- Definition:
  - 1! = 1
  - n! = n * (n-1)!

[1]:
```
// solve exercise 4
long fact(int n) {
    if (n == 1)
        return 1;
    else
        return n*fact(n-1);
}
```

visualize it at pythontutor.com: https://goo.gl/qL17S8

1.5.7 Exercise 5

- Compute the Fibonacci sequence for a given number. The Fibonacci Sequence is the series of numbers: 1, 1, 2, 3, 5, 8, 13, 21, …
- Definition:
  - fib(1) = 1
  - fib(2) = 1
  - fib(n) = fin(n-1) + fib(n-2)

[11]:
```
// solve exercise 5
int fib(int n) {
    if (n == 1) //base case
        return 1;
```

```
    else if (n == 2) // base case
        return 1;
    else //general case
        return fib(n-1) + fib(n-2);
}
```

[12]:
```
cout << "fib(10) = " << fib(10) << endl;
```

```
fib(10) = 55
```

1.5.8 how many time is the fib() called for fib(10)?

- visualize it with pythontutor.com for answer: https://goo.gl/a6RrDW

1.5.9 better way... using memoization technique

https://en.wikipedia.org/wiki/Memoization - optimization technique where you save/remember/cache the previously calculated answer to avoid recalculating for the same input

[2]:
```
#include <iostream>
#include <unordered_map>

using namespace std;
```

[14]:
```
unordered_map<int, int> FibDict;
int call = 0;
```

[15]:
```
// solve exercise 5 using memoization technique
// first run as it is and see how the value of call
// then use memoization technique and run and see the value of call
int fibMemoized(int n) {
    call += 1;
    if (n == 1) //base case
        return 1;
    else if (n == 2) // base case
        return 1;
    else {//general case
        // check if fib of n is already calculated
        //auto search = FibDict.find(n);
        //if (search != FibDict.end())
        //    return search->second;
        int ans = fibMemoized(n-1) + fibMemoized(n-2);
        //FibDict[n] = ans;
        return ans;
    }
}
```

[16]:
```
cout << "fib(10) = " << fibMemoized(10) << endl;
cout << "total fibMemoized call = " << call << endl;
```

```
fib(10) = 55
total fibMemoized call = 109
```

1.5.10 Exercise 6

- Use memoization technique to improve recursive factorial function.
- Practice it!

1.5.11 Exercise 7

- Find greatest common divisor (gcd) of a given two integers using recursive solution.
- Definition:
  - gcd(n, 0) = n
  - gcd(n, m) = gcd(m, n%m)
- Solve it!

1.5.12 Exercise 8

- Print the steps needed to solve Tower of Hanoi puzzle: https://www.khanacademy.org/computing/computer-science/algorithms/towers-of-hanoi/a/towers-of-hanoi
- recursive algorithm:
- let's say disks are numbered from 1 to n (n is the largest)
  1. move n-1 disks from peg A to peg B with the help of peg C
  2. move disk n from peg A to peg C
  3. move n-1 disks from peg B to peg C with the help of peg 1

[3]:
```
void moveDisks(int n, char from, char helper, char dest)
{
    if (n >= 1)
    {
        moveDisks(n-1, from, dest, helper);
        cout << "Move disk #" << n << " from peg " << from << " to peg " <<␣
↪dest << endl;
        moveDisks(n-1, helper, from, dest);
    }
}
```

[4]:
```
moveDisks(2, 'A', 'B', 'C');
```

```
Move disk #1 from peg A to peg B
Move disk #2 from peg A to peg C
Move disk #1 from peg B to peg C
```

1.5.13 How long will it take to move 64 golden disks?

- total moves $\approx 2^{64} \approx 1.8446744e+19$
- if monks work 24/7 moving 1 disk/second non-stop and not making a single mistake
  - would take: $5,848,682,331$ ($> 5B$ centuries)
- computer doing the same @ $1B$ moves/second would take about 500 years!

1.5.14 Exercise 9 - Cumulative Sum

- Write a recursive solution to find and return the sum of the values from 1 to n.
- test cases:
  - assert(sum(5) == 15);
  - assert(sum(3) == 6);

1.5.15 Exercise 10 - Add Odd Values

- Write a recursive solution to add and return the sum of the positive odd values from 1 to n given some positive integer n.
- test cases:
  - assert(addOdd(1) == 1);
  - assert(addOdd(2) == 1);
  - assert(addOdd(3) == 4);
  - assert(addOdd(7) == 16);

1.5.16 Exercise 11 - Sum of the Digits

- Write a recursive solution to add and return the sum of the digits in a given non-negative integer.
- test cases:
  - assert(sumOfDigits(1234) == 10);
  - assert(sumOfDigits(999) == 27);

1.5.17 Exercise 12 - Count Characters

- Write a recursive solution to count the number of times a letter 'A' appears in given string str.
- test cases:
  - assert(countChr("ctcowcAt") == 1);
  - assert(countChr("AAbAt") == 3);

[ ]:

```
```