Chapter 3: Recursion

Call Traces and Call Trees

- Recursion definitions are circular
- When reach the end of the list, MUST roll back to find the answer
    - Go forward and backward, therefore it is efficient memory wise
- Rules for recursion
    - Contains recursive case
    - Contains base case
    - Make progress towards base case

Divide-and-Conquer Decompositions

- Divide-and-Conquer Algorithm
    - Solve an N size problem by dividing it into small/similar type problems
        - Continue to divide until the problem is small enough to conquer
        - Solve N size problem by combining the conquered smaller problems
- Down-by-1 D&C
        - One smaller problem is size 1
        - Other one is size n-1
    - Going-Up Recursion
        - D&C by reducing problem size by i (i >= 1) each time such that argument to recursive function goes up each time
    - Going-Down Recursion
        - D&C by reducing problem size by i (i >= 1) each time such that argument to recursive function goes down each time
- Division-in-halves D&C
        - D&C works better if smaller problems if each size is ~n/2
    - Edges & Center Recursion

Tail Recursion

- A function call is said to be tail recursive if there is nothing to do after the function returns except return its value
- Can be optimized by some compilers
- When last executed operation (recursive case) in a function is a recursive call to the function
    - Note: “last executed” may not be last line in code
- A tail recursion is a recursive function where the function calls itself at the end ("tail") of the function in which **no computation is done** after the return of recursive call. Many compilers optimize to change a recursive call to a tail recursive or an iterative call.
- Mutual vs Self Tail Recursion
    - Self-Tail Recursive – It calls itself
    - Mutual-Tail Recursive happens when A calls B, B calls A

      (Any number of functions may be involved)

Infinite Regress

- Recursion never ends
- Could be programmer or user's fault
- There are two reasons that a recursive program can call itself endlessly:

1. There is no base case to stop the recursion/ Forget base case/Incomplete base cases
2. A base case never gets called
3. Never hits base case
    a. Programmer's error
    b. User doesn't pay attention to API (factorial example)
    c. Not enough resources(memory, computational power)

Simple Analysis

- Compilers and Recursion
    - When a function called, a STACK FRAME is pushed on run-time stack
    - Run-time stack keeps info such as:
        - Address to return to
        - Memory for permissions and function variables
        - Memory for function return value
    - The deeper the recursion, the bigger the Run-time stack
        - Stack size -> Space
        - Stack overhead -> Long time to run
- Optimize recursion if possible
    - GCC, some JVMs, Scala optimize self-tail recursion but can't do mutual-tail recursion
    - All functional languages (Lisp, Scheme, Haskell) optimize both S.T.R and M.T.R
    - How?
        - Compilers cam clobber (mostly) previous stack frame with new one – so stack doesn't grow
        - Some do not optimize at all (R, Python)
- How can we remove recursion?
    - Iterations:
        - At each step, update a partial solution (typically by iteration over a variable (i). The partial solution gets close and closer to the final solution at each step.
        - ALL recursion can be redefine as iteration
        - Tail recursion is "easy"
- Common uses for Recursion
    - Defining things – e.g. Context Free Grammar
    - Generating things – e.g. Recursive algorithm to generate Context Free Language
    - Recognizing things – e.g. Recursive descent parser to tell whether a string belongs to the language
    - Proving things – e.g. Recursion Induction

Dynamic Programming

- Dynamic Programming to the rescue
    - Solve problem by breaking into smaller sub-problems
    - Solve each sub-problem only once
    - Store sub-problem solution in some DS(memorization)
        - There are a lot of redundant calculation
    - Next time sub-problem occurs, use stored results
- Top Down Approach
- Bottom Up Approach

-

Chapter 6: Complexity

Complexity Classes

- Analysis of Algorithms
  - Two Algorithms, same task: which is better?
    - Testing the algorithms by running them do not give you the completed picture: Depends on the hardware, compiler which are might be changeable
    - Example: 2 way to Waterloo: fast by SUV, or slower with your own car
      - Trading speed vs cost(ie. scenic)
  - Compiler Algorithms trade-off:
    - Time, space(memory used at once)
    - Disk space, maintainability
    - Can't just time them both
  - Consider algorithms A(n)
    - Implement A(n) in 3 different environments (compilers, Osa,langs)
    - Time each of them on the same hardware

Big-O Notation

- Formal definition
  - f(n) is O(g(n)) if there exist 2 positive integers, K and n_o, such that $|f(n)| <= K|g(n)|$ for all $n>= n_o$
- Generally, measure officially, independent of programming languages, hardware, compiler, etc.
- Express algorithm as a function of problem size ("n")
  - Size of the problem is the size of the input of the problem
- Varies algorithm will have varies time complicity

  $4n^2+3n+7$
  - Big-O ignores everything except the highest order term

    $4n^2+3y+x => O(n^2)$
  - Complexity classes (Page.214)
    - O(1) - constant
    - O(n) - linear
    - $O(n^2)$ - quadratic
    - $O(n^3)$ - cubic
    - O(log n) - logarithmic
    - O(n log n) - logarithmic
    - $O(2^n)$ - exponential
    - $O(c^n)$ - exponential
  - The scale of increasing complexity class

$O(1)<O(\log n)<O(n)<O(n \log n)<O(n^2)<O(n^3)<O(2^n)<O(10^n)$

Analyze algorithms

- Iterative algorithms
  - Summing up the contribution of the individual stages of the computation
  - Usually count number of:
    - Operations

        - Comparisons
        - Loop overhead
        - Point/Array references
        - Function calls
- Recursive algorithms
    - Setting up and solving recurrence relations

Time Complexity

- When we analyze the running time of an algorithm, we will try to come up with the general shape of the curve that characterizes its running time as a function of the problem size.
    - Running times for different algorithms fall into different complexity classes. Each complexity class is characterized by a different family of curves.
- Recursive Version
    - Base case: the terminating scenario in recursion that does not use recursion to produce an answer
        - T(1) = 1
    - Recurrence relation: T(n) = 1+T(n-1)
    - Lower Bound
    - Upper Bound
- For finding the item in the n-item array
    - Best case: A[i]-there is only one comparsion
    - Worst case: A[n]-number of comparsion is the size of the problem
    - Average case: Average of finding item at A[i], A[2] .. A[n]

Space Complexity

- Measure the amount of memory used AT ONCE by algorithm
    - While the algorithm running, at the one given time, used the maximum amount of memory
    - What takes up memory
        - Instruction space(memory to hold compiled version of program, constant for any n)
        - Data space (variables, data structures(hashtrees,hashcode)/allocated memory)
        - Environment space (constant for each function call)
- Three methods for searching:
    - Sequential search: O(n)
    - Binary Search: O(log n)
    - Interpolation search: O(log log n)

Complexity Trade-offs