APCS Notes

Yicheng Wang

2014-2015

Contents

1 2015-02-05

1.1 Do Now

Figure out what the following code does.

```
public void printme(int n) {
    if (N > 0) {
        printme(n - 1);
        System.out.println(n);
    }
}
```

It should print out an increasing sequence of numbers from 1-N.

1.2 Stack and ROP

The stack on top is the current function, and each layer beneath that is the function that called the current function.

1.3 Recursion

Simple recursive problem: FACTORIAL
Hallmarks of a recursive solutioin:

- Base Case: thing that stops the program, simple case you know the answer of. In the case of factorials, factorial(0) = 1
- Reduction Case: You need to alternate the variable in some sort of way, for example, we should do n * factorial(n - 1)
- Recursion: function A need to eventually call A

Final code:

```
public int factorial(n) {
    if (n == 0) {
        return 1; // Base Case
    }
    else {
        return n * factorial(n - 1); // Reduction Step
    }
}
```

2 2015-02-06

2.1 Traditional Recursion Examples

2.1.1 Fibonacci Numbers

1, 1, 2, 3, 5, 8, 13 ...

Base Case: if n ¡ 2, return 1
Reduction Step: fib(n) = fib(n - 1) + fib(n - 2)
Example Code:

```
public int fib(int n) {
    if (n < 2) {
        return 1;
    }

    else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

2.1.2 List/String Manipulation

Example, finding the length of a substring.
Base case: "" has length of 0
Reduction Step: 1 + "cdr" of the string, i.e. s.substring(1);
Example Code:

```
public int lenStr(String s) {
    if (s.equals("")) {
        return 0;
    }
    else {
        return 1 + lenStr(s.substring(1));
    }
}
```

3 2015-02-09

3.1 Getting out of a Maze

1. Maze vs. Laybrith: Laybrith may not have choices, mazes have choices.
2. Strategies in solving the maze:
    - Doesn't work due to loops
3. Greek Way of Solving Mazes:
    - Invented by Odysseus, bring a thread, use process of elimination to loop through all possible intersection.
    - Works really well.

Maze solving in java

1. This is clearly a recursive solution.

2. Using a recursive solution we can easily trace back the stack to find the previous intersection

3. We represent our map as a char array of paths ('#')

4. Base case: location is a wall OR location of exit

5. Reduction step: call solve at a different (x,y) location, if it's a dead end, it'll be peeled off the stack due to the "return"

6. We will try to solve the maze systematically, x+/-1, y+/-1.

4 2015-02-11

4.1 Blind Search

Trying all possibilities until we find our solutions. Also called exhaustive search, linear search, recursive search, search with backtracking.

The maze algorithm we wrote is sometimes known as a depth first search. It's because you say in one path for as long as possible. Advantage is if the solution is deep, this works pretty well. However, if the solution is closer, breath first search goes n steps in all possible directions.

4.2 State Space Search

Search algorithm to solve a problem, searching for a "state space."

Idea is if you have a problem, you can describe said problem as a "state."

State is a configuration of the world, such as turtle in netLogo.

The world is made up of many states, and one can transition from one state to the next.

State Space Search: The series of states one has to go through to get to the desirable final state. (like exit in the maze thing)

4.3 Graph Theory

Graphs are collections of edges and nodes. Nodes represent the states, edges marks the transition between one node to the next.

5 2015-02-12

5.1 Space State Search

Examples of space state search:

- Maze path-finding
- 15 puzzle

- Cube
- Chess algorithm – More complex
  - two players!
  - high branching factor
  - harder base case
  - uses mini-max searching: best for me and worst for you
- Description of an entity

5.2 Implicit Data Structure

When we did the maze solver, we used an explicit data structure → the 2D array. However, the graph of the transition is also a data structure, it's implicit though, just running in the background. Whenever we call solve, it creates a node on the stack. However, if a call returns, it destroy that branch of the graph.

As the program runs, we can imagine it as a graph.

5.3 PROJECT

Do the diagonistic exam in the Barron's Review book, do as follows:

- First do it under test condition
- Go back and tried to fix your problem

Option 1. Knight Tour:
Given a $N \times N$ board, find a path such that the knight visits each square once without repetition. (Start with 5×5);

Option 2. N-Queen:
A way to place N mutually non-attacking queens on a $N \times N$ chessboard.

6 2015-02-13

Other problem, third variation, one can also do the 15 puzzle.

6.1 System.out.printf

When one is doing a knight's tour, the print output may be distorted if one moves from single digit to double digit. We can leave a placeholder in the format string, such as %s, %d, etc.

Example code:

```
...
System.out.printf("%s, helloWorld\n %s\n", "title", "more stuff!");
...

OUTPUT:

title, helloWorld
more stuff!
```

However, printf can take multiple types and it is possible to save spaces for formatters. Like the following:

```
...
System.out.printf("%3d\n%3d", 1, 123);
...

OUTPUT:

  1
123
```

7 2015-02-25

Iteninary:

- Tomorrow - Knight's Tour
- Tuesday - USACO

Today, we're going back to sorting.

8 Merge Sort

So far, we've covered 3 algorithms: selection sort, insertion sort, and bubble sort. They are all linear algorithms. Selection sort selects the ith index and puts the ith smallest/largest elements there, whereas insertion sort inserts the ith smallest/largest element so far into the ith index of the list.

HOWEVER, we're lazy and need to do this the 6 years old way. If we're sorting a deck of unsorted cards, we'll split the deck in half and give it to more people, so on, and so on. This continues until everyone has 1 card at hand. The process then goes backwards and the people merges the sorted lists passed on to him/her. This is known as a merge sort, it is a **divide and conquer algorithm.**

Sample code:

```
...
    public static int[] mergeSort(int[] data) {
        if (data.length == 1) {
            return data;
        }

```

```
7         else {
8             int[] A = Arrays.copyOfRange(data, 0, data.length / 2);
9             int[] B = Arrays.copyOfRange(data, data.length / 2, data.
    length);
10
11            int[] AS = mergeSort(A);
12            int[] BS = mergeSort(B);
13            return merge(AS, BS);
14        }
15    }
16
17    public static int[] merge(int[] A, int[] B) {
18        int[] result = new int[A.length + B.length];
19        int position = 0;
20        int APos = 0;
21        int BPos = 0;
22        while (APos < A.length && BPos < B.length) {
23            if (A[APos] < B[BPos]) {
24                result[position] = A[APos];
25                APos++;
26            }
27            else {
28                result[position] = B[BPos];
29                BPos++;
30            }
31            position++;
32        }
33
34        for (int i = APos ; i < A.length ; i++) {
35            result[position] = A[i];
36            position++;
37        }
38        for (int i = BPos ; i < B.length ; i++) {
39            result[position] = B[i];
40            position++;
41        }
42        return result;
43    }
44
45 ...
```

9 2015-03-04

9.1 On the Algorithm of the Merge Sort

In insertion and selection sorts, the average operation has a complexity of n^2. This is because the inner loop runs through the list (n operations) and the outer loop instructs us to run through the inner loop n times, hence the total operations is n^2.

For merge sort, each step we split the list in half and sort the smaller part first. If we imagine this as a tree, we divide the list in half each time. So the vertical step takes $\log_2 n$ steps. Then at each step, we copy the array once so there is also a n component to the total

complexity. Then, the sum of each level's merge is also going to be n because merging takes the same amount of work as splitting. Therefore, the total complexity is:

$$O(n) = n \log n$$

Let's look at the different rate of growth of the two curves (n^2 and $n \log n$). Take, 1,000,000 as n, $n^2 = 1.0 \times 10^{12}$, but $n \log n = 6 \times 10^6$

9.2 Big O Notation

A function $f(n)$ is said to be $O(g(n))$ if there exists some constant k such that $kg(n) > f(n)$ over the long term.

Note that the Big O Notation is always and **Upper bound**, and is extracted from the worst-case scenario of the function. However, it is a very tight upper bound.

10 2015-03-05

We shall write an algorithm to find the k^{th} smallest element.

A few ways to do this:

One, we can recursively delete the smallest data point, but this isn't very efficient. In fact, this is a $O(n^2)$ algorithm.

The other, cheap (my) way is to first merge sort it and then just do a lookup for the element. This has the complexity of only $O(n \log n)$.

Note that for hard problems, it is very hard for complexity to go below $O(n \log n)$. Therefore, merge sorting something is practically free, complexity-wise. Case in point: is it worthwhile to first sort an array and then use binary search with the merge sort or just use the linear search.

However, there is a more efficient way of doing this.

We can use a binary-search-like algorithm.

11 2015-03-09

11.1 Efficiency of Quick Select

Given that we are lucky, and if the pivot is good such that it bisects the array, on the first operation we only go through $\frac{n}{2}$ operations after going through the entire array. Then it just divides by 2 each time, so the final operation number is:

$$n + \frac{1}{2}n + \frac{1}{4}n + ... = 2n$$

So the quickselect runs on linear time!

12 2015-03-11

12.1 Linked List

A linked list is comprised of elements/nodes and a way to get to the "start node." A node contains some data and information on how to get to the next node. There are different forms of linked list. We are going to make the scheme list.

12.2 Scheme-like list in Java

```
1  public class Node {
2      private String data;
3      private Node next; // Self-referencing data structure
4
5      public Node(string s) {
6          data = s;
7      }
8
9      public void setData(String s) {
10         data = s;
11     }
12
13     public String getData() {
14         return data;
15     }
16
17     public void (Node n) {
18         next = n;
19     }
20
21     public Node getNext() {
22         return next;
23     }
24
25     public String toString() {
26         return data;
27     }
28 }
29
30 public class Driver {
31     public static void main(String[] args) {
32         Node n1 = new Node("Hello");
33         Node n2 = new Node("World");
34
35         System.out.println(n1.getNext().getData()); // prints out "World"
36
37         n.getNext().setData("pickle") // changes n2 to pickle
38
39         System.out.println(n1.getNext().getData()); // prints out "pickle"
40
41         n2.setNext("abc") // Creates a new node and point the next of n2
       to it.
```

```
42
43          n1.getNext().getNext().setNext("blah");
44
45          n0 = new Node("Start");
46          n0.setNext(n1); // Prefixes n0 to the list
47          // If we want n1 to refer to the start of the list, we do:
48          n1 = n0;
49
50          // Right now the list looks like this:
51          // "Start" --> "Hello" --> "pickle" --> "blah"
52          System.out.println(n1);
53      }
54  }
```

13 2015-03-11

13.1 Print out every element in a linked list

To print everything from an array, we do:

```
1  int i = 0;
2  while (i < A.length) {
3      System.out.println(A[i]);
4      i++;
5  }
```

To do the same for a linked list:

```
1  Node tmp = n; // starting node
2  while (tmp != null) {
3      System.out.println(tmp);
4      tmp = tmp.getNext();
5  }
6
7  // With a for loop:
8  for (Node tmp = n ; tmp != null ; tmp = tmp.getNext()) {
9      System.out.println(tmp);
10 }
11
12 // And recursively:
13 public static print(Node n) {
14     if (n.getNext() == null) {
15         System.out.println(n);
16         return;
17     }
18
19     else {
20         System.out.println(n);
21         print(n.getNext())
22     }
23 }
```

14 2015-03-23

14.1 Data Structures

14.1.1 STACKS

Stack based objects add on the top and take from the top. This is known as a last-in-first-out (**LIFO**) data structure. Stacks are concepts, it's NOT a basic data structure.

Traditionally, a stack has certain operations:

```
1 myStack<Integer> s = new myStack<Integer>();
2 s.push(10); // Push takes whatever your parameter is and pushes it onto
3             // the top of the stack
4
5 s.push(5); // Now 10 is no longer the top, now 5 is on top
6
7 i = s.pop(); // Take the item off of the top of the stack and return it.
8 System.out.println(i); // In this case i = 5 and 10 is back on top
```

push and **pop** are the two BASIC operations that all stack classes have to have, but they usually gives out other methods such as **empty** (check if empty), **peek** or **top** (returns but doesn't remove), **size** (give how many elements are in the stack), etc.

14.2 Assignment

Make a class called myStack that mimics a stack, make methods push, pop, empty and peek. Use this as an underlying linked list.

15 2015-03-25

15.1 Queue

A queue, unlike a stack, is what is known as a first-in-first-out (**FIFO**) data structure. The basic tradition terms for a queue are called **enqueue** and **dequeue**. Enqueue is the equivalent of push and dequeue is the equivalent to pop. There may also be **empty** and **front**.

Front in a queue is more important than that of a stack, because one cannot simply pop and then push for the queue. But front is still not considered a basic command.

15.2 Assignment

Implement the queue with a single linked-list, and enqueue and dequeue should be able to run at constant time.

16 2015-03-31

16.1 Maze

Review on the maze algorithm we did a few weeks ago:

1. 2D array of chars
2. Recursive Deepth First Search

However, if we had infinite resources, it may be better to do a breadth first search. This is a good strat when you suspect that the answer is closer by. This is known as breadth first search.

Example maze:

```
    ---
    |B|
    ---
    |C|
  -----------
  |D|A|E|F|G|
  -----------
    |H|
    ---
    |I|
    ---------
    |J|K|M|I|
    ---------
```

We create a concept called the frontier, which is the "next square to be explored."

If we start at square A in the example maze, the frontier would be consisted of E, C, D, and H. However, for argument's sake let's say E was the first square to be explored, after it is taken off of the frontier. the next square to be explored would be one of C, D and H instead of F. So the frontier should be stored as a FIFO system, i.e. a queue.

Basic code snippet:

```
public void solve(Node position) {
    while (!frontier.empty()) {
        current = frontier.dequeue();
        if (current.equals(exit)) {
            System.exit(0);
        }
        ... ... ...
    }
}
```

17 2015-4-22

17.1 Boring Definition Day

17.1.1 Graph/Trees

Trees are gateways to more powers

Tree is a specification of graph, a graph is a collection of nodes (or vertices) and edges.

A node holds stuff and an edge connects nodes.

A tree is an acyclic graph such that one node is defined as "the root."

A tree is either empty or has one or more nodes, each node can have 0 or more children and one node is designated as the root.

Leaf: A node with 0 children.

Siblings: Nodes that share a parent (1 level up only)

Path: A sequence of edges/nodes that takes you from 1 place to another

Height: The length of the longest root-to-leaf path

Full: A tree is full means that every single node is either a leaf or has the fullest amount of children

Balance: THe same number of children and node on both sides.

17.1.2 Binary Tree

Binary Tree: A tree in which each node is either a leaf or has 1 or 2 chidren.

18 2015-4-23

18.1 Design of the Node

The tree is based on a double-linked list, each node should have two private Nodes left, and right and generic data.

18.2 Binary Search Tree

It is a binary tree such that given any Node w/ value v, all the node in the left subtree has values ¡ V and all nodes in the right subtree have values greater than V.

18.3 Sample Algorithms

```
1  public Node Search (Node T, Integer i) {
2      while (T != null) {
3          int c = T.getData().compareTo(i);
4          if (c == 0) {
5              return T;
6          }
7          T = (c > 0) ? T.getRight() : T.getLeft();
8      }
9      return null;
10 }
11
12 public insert (Node T) {
13     Node tmp1 = root;
14     Node tmp2 = null;
15     while (tmp1 != null) {
16         int c = T.getData().compareTo(tmp1.getData());
```

```
17          tmp2 = tmp1;
18          tmp1 = (c > 0) ? tmp1.getRight() : tmp1.getLeft();
19      }
20      int c = T.getData().compareTo(tmp2.getData());
21      if (c > 0) {
22          tmp2.setRight(T);
23      }
24      else {
25          tmp2.setLeft(T);
26      }
27  }
```

19 2015-4-29

19.1 Remove in a Tree

```
1  10
2  -5
3  |-2
4  ||-4
5  |-8
6  -20
7   -17
8   |-15
9   |-18
10  -25
11   -100
12    -101
```

There are three situations when we try to delete a node, each one harder than the next. However, we need to use the same piggy-back algorithm as used in insertion.

19.1.1 Leaf

It is very easy to remove a leaf, we simply set the parent's left/right to null.

19.1.2 1 Child

This is a little bit more work, but still not that much. We basically take out the node as if it's a linked list. i.e. parent.setRight(parent.getRight().getRight()).

19.1.3 2 Children

For example, if we want to remove the node 20 from our tree, we need to find the largest node on the left side of the tree, which is 18, and remove it and add it in place of the twenty. This preserves our tree structure.

20 2015-05-11

20.1 Do Now

Find a way to find the number of nodes in a binary search tree.
It is easy to do this recursively, we simple do the following:

```
1  public int nodeCount() {
2      return nodeCountH(root);
3  }
4
5  private int nodeCountH(Node current) {
6      if (current == null) {
7          return 0;
8      }
9      else {
10         return 1 + nodeCountH(current.getLeft()) + nodeCountH(current.
       getRight());
11     }
12 }
```

Do the following functions:

1. Max value from the tree
2. Height of the tree
3. Longest leaf-to-leaf path
4. Split Dupes
    - You have a tree, but somewhere in the tree a parent and a child have the same value
    - When you do that you have to insert a node with value of 1 less than that value.

21 2015-05-12

21.1 Heap

A min heap is a binary tree such that all children are greater than their parents and is as full as possible left and right. A max heap is the same except all childrens are less than their parents.

21.1.1 findMin

Note the finding the min is a min-heap is a constant-time operation. We simply take the root value.

21.1.2 removeMin

However, if we remove the min, we have several problems:

- We have a tree without a root
- We have a heap that doesn't follow the heap property.

To solve this we do the following:

1. Remove the root
2. Replace the root with the lowest rightest value
3. Pushdown
    - swap the parent with the smaller of its children until it reaches the leaf.

21.1.3 heapSort

heapSort can be performed by repeated removing the minimum and using the heap's self-balancing property to sort the data.

21.1.4 insert

Instead of push down, we sift up. We attach the new data to the lowest leftest node and for each parent we replace it with the smaller of its two children. We then move up until we hit the root.

21.2 Algorithm Efficiency

Find min: $O(1)$
Remove min / pushdown: $O(\log n)$
Heapsort: $O(n \log n)$
Insert: $O(\log n)$

22 2015-05-14

22.1 Array-Heap

Note that by the nature of binary trees, the two child of the parent node at index n is $2n$ and $2n + 1$. To change any array and convert it into a heap, we start at the last element of the array, push down, from there. This creates a mini-heap with root being the last element. Then we move to the top one element at a time, expanding the "heap" portion of the array. Eventually creating the heap.

To do the heapsort, instead of pulling out the root, we swap the root with the last element of the array. Then we basically shrink the "heap" part of the array.

23 2015-05-26

In a weighted graph, breath first search doesn't quite work. Because it assumed that every step is 1. A-Star doesn't take consideration of the edge length either.

23.1 Dijkstra's Path Finding Algorithm

Single Source Shortest Path Algorithm – Gives the shortest path from one node.
We start building a solution set, begin the algorithm from the beginning node:

1. initialize all unreached points to infinity
2. Add new node with the lowest edge length to the solution set.
3. For each node not in the solution set, recalculate the minimum.