

Math 482

Survival Guide

By: Jared Fowler

Originally created during the summer semester of 2014, this guide contains a collection of my personal study notes, programs, and experiments which pertain to California State University Northridge's Math 482, Combinational Algorithms with Professor John Dye.

There is no particular order to this document, however, pages will be labeled with the appropriate topic. I should also note that this guide does not contain every single topic taught by Professor Dye, instead, it contains topics that I found particularly challenging, or new ideas that I hadn't considered before. I highly recommend getting a copy of the required text (Cormen). I found it to be extremely helpful while learning the required algorithms. (Not so much the written explanations but the pictures.) I am assuming that the reader already has a sufficient background in computer science.

Graphing Calculators are allowed during the tests. Many of the algorithms can be easily programmed into your calculator. I will not always provide calculator assembly code, but I will provide source code written in c++, java, or c#. (Code compiled on Visual Studio)

Original Author: Jared Fowler. CEO J-Soft

Computer Science Major.

Associates in Mathematics and Liberal Arts from Moorpark College.

jaredwfowler@hotmail.com

Publish Date: July 6, 2014

This document is open for additions, corrections, etc. Please log any changes in the table provided below. Also be sure to update the edition on the front cover.

Additions/Edits:

NAME	EDIT/ADDITION	DATE

Table of Contents

I.	Binary Heaps Parts 1-2.....	pgs. 3-10
II.	Binary Josephus.....	pgs. 11
III.	BST Permutation/Pattern/Delete.....	pgs. 12-13
IV.	Probability.....	pgs. 14
V.	Instant Insanity.....	pgs. 15

Binary Heaps Part 1

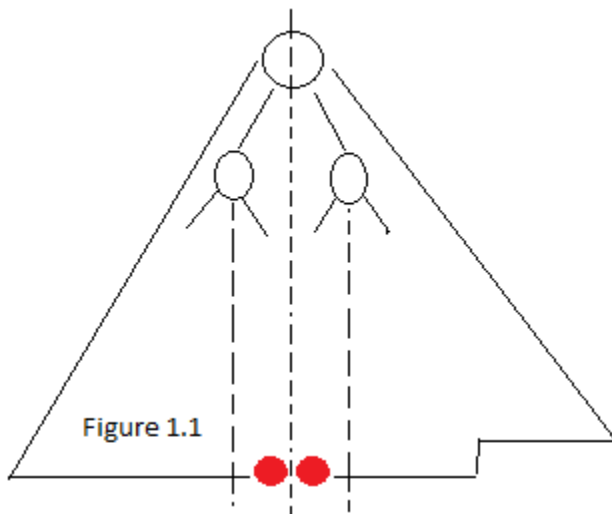
Question:

Given n Nodes with k being the k^{th} smallest key, what is k 's max/min height and depth?

We begin by realizing one rule and one observation:

Rule 1: Max Height cannot exceed $\lceil \log_2 k \rceil$

Observation 1: Many times we will want to find the max height by dipping into the last row, which may or may not be complete. We need to be careful while doing this. While all leaves are related ultimately by the root node, the individual leaves, though they may be placed right next to each other, may have a very distant relation. For example, consider figure 1.1 below:



The two red nodes, though next to each other in the heap, are only related to each other by the root! We can further see that each child will further be divided into another separate tree.

To continue, let's just use some numerical values for n and k , just to make things clear. $n = 7111$, and $k = 144$

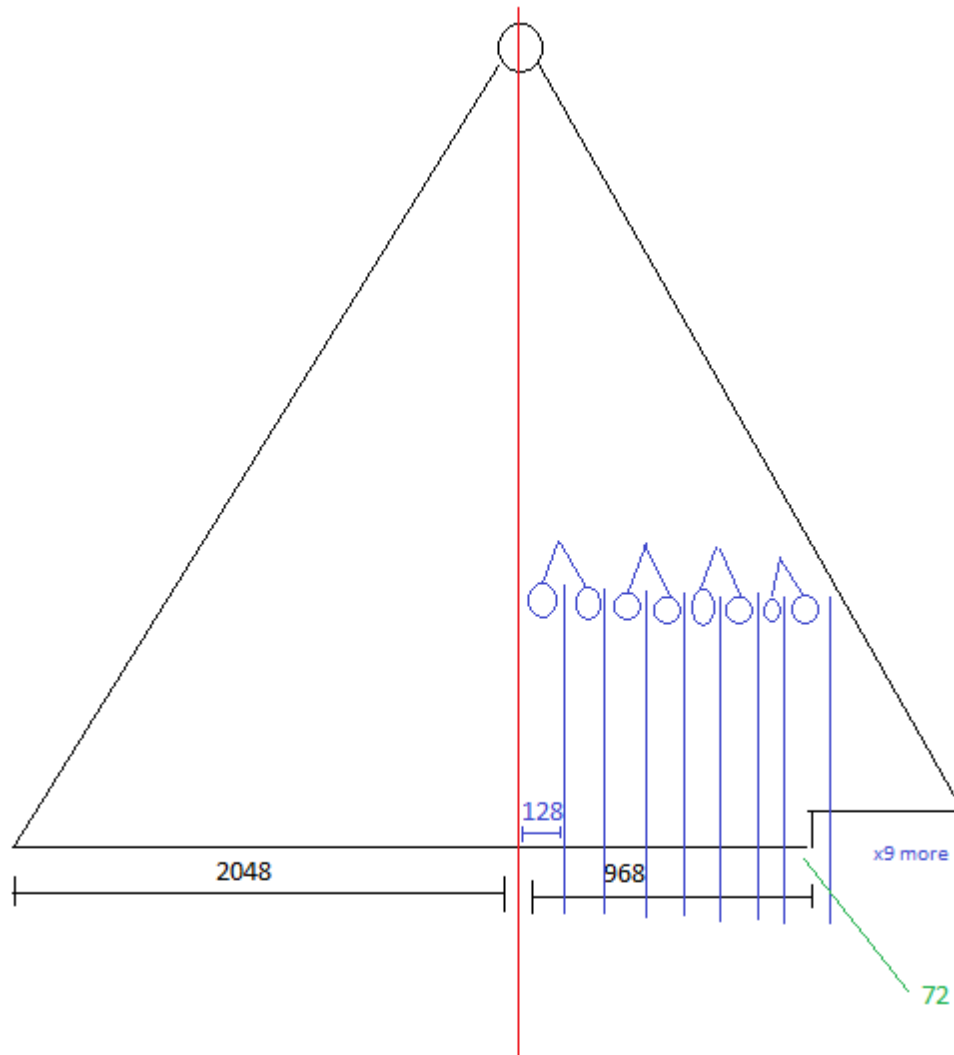
Max Height:

We start by finding the height of both the tree and k .

$$\log_2 7111 = 12.79 \sim 12$$

$$\log_2 144 = 7.17 \sim 7$$

We can visualize our heap to look something like this:



We are able to calculate the exact number of nodes used on the last level to be $2024 + 968 = 3016$.

We now note that in order for k to have maximum height 7, according to rule 1, there will have to exist nodes on its 7th level. (Obviously). The 7th level cannot be full, however, because that would make k greater than the 144th smallest node. We therefore have to rely upon the “leftovers” as defined in green writing above. We find the leftovers by simply partitioning the used nodes in the last row of the tree, by how many nodes a tree of height $\lfloor \log_2 k \rfloor$, in this case 7, would use in its last row. We can either do $3016 \% 128$, or $968 \% 128$. (where $\%$ is mod)

This problem now becomes trivial. We know that the total node count of tree k before the last row will be $2^7 - 1$. We subtract this from k . $144 - 127 = 17$. This means that we can only accommodate 17 nodes or less on the last row before surpassing k . Because there are 72 nodes leftover, which is greater than 17, we cannot use them!

Luckily the alternative solution is simple. $\lfloor \log_2 k \rfloor - 1 = 6$. As seen, if we cannot use the full potential height, we just use one less.

Min Height:

I’ll let the reader convince him or herself about the following reasoning. The minimum height is really determined only by a single factor: Is there enough nodes greater than the k_{th} to get back to the tree’s root? Remember, this is a binary **HEAP**. So, we take $n - k = a$, and in this case we find ‘ a ’ to be much larger than $\log_2 144$, therefore making the min height 0.

For clarity, let's pretend that k was the 7110th smallest, a.k.a. the second largest. We would take $7,111 - 7,110 = 1$. Clearly 1 is not greater than 12, ($\log_2 7110$), so the min height results as $12 - 1 = 11$.

Max Depth:

Subtract the tree height from the min height. Done!

Min Depth:

This is really easy now that we've found the max height. We need to determine if we used any of the "leftovers" to get to our max height. If this is the case, we can just take the tree height minus max height. If we were unable to use these values, all hope isn't lost yet! Remember how we partitioned the last row by $2^{(\lg k)}$? If there are enough nodes on the next level up that don't have children that could serve as a full base to tree of root k , then our min depth will also be $\text{treeheight} - \lg k$. Else, if neither of these cases hold, we are forced to have a full row on the very bottom and have $(\text{treeheight} + 1) - \lg k$.

So, for this case we did not use the leftovers. We now check if there is enough on the next level up to sustain a full base. There is indeed plenty, around 9 more to be exact, which means we can take $12 - 7 = 5$.

NOTE: There does exist some cases where what is written above may not work, however, these other cases are handled in the following C++ code. I encourage the reader to study it. Calculator Programmable.

```
/*Created By: Jared Fowler  
June 2014  
Math 482, Prof. John Dye
```

```
A common question includes finding a specific kth smallest node's  
max/min height and depth. An algorithm to solve this is presented below.*/
```

```
#include <iostream>  
#include <cmath>  
using namespace std;  
  
int main(){  
    //Variables to be changed  
    int k = 6;  
    int n = 8;  
  
    //Do not modify below this line!  
    //Other Variables  
    int max_height = -1;  
    int min_height = -1;  
    int max_depth = -1;  
    int min_depth = -1;  
  
    int lg_k = log(k) / log(2);    //lg_2(k)  
    int lg_n = log(n) / log(2);    //lg_2(n)  
  
    int last_row_leaves = n - (pow(2, lg_n) - 1);    //Number of used leaves on the last row  
    int completeBoost = 0; //Used in the situation that k has a complete tree under it.
```

```

//Min Height: Same Simple Case For All!
/*This is really determined by how many nodes are
greater than the kth value. Imagine following the left most branch.
The only factor that would limit a nodes min height is if there wasn't
enought nodes greater than it to get back to the root. In this case,
we would need to check if there is enough unused nodes on the bottom row
to accomodate a tree with parent k.*/
int numberLarger = n - k;
min_height = lg_n - numberLarger;

if (min_height <= 0){
    min_height = 0;
}
else{
    int last_row_not_used = pow(2, lg_n) - last_row_leaves;
    if (last_row_not_used >= pow(2, min_height)){
        min_height--;
    }
}

//Max Depth is just as easy to find. We subtract the tree height from min height
max_depth = lg_n - min_height;

#if(0) //Uses case 3
//*****
/*Case 1: K is the parent of a complete tree (sub-tree)
This is probably the easiest case... */

//Step 1A: Determine if there is a complete tree under k
int lg_k_ciel = (log(k + 1) / log(2)) + .999999;
if ((lg_k+1) == lg_k_ciel){

}

#endif

//We do need to take into consideration a k node with a complete tree under it.
int matcher = (pow(2, lg_k + 1) - 1);
if (k == matcher){
    completeBoost = 1;
}

//*****
/*Case 2: K is the root of the entire tree.*/

if (k == n){

    min_height = lg_n;
    max_height = lg_n;
    min_depth = 0;
    max_depth = 0;
}

//*****
/*Case 3: K is neither of the two cases above. This is the most
difficult case, as we need to consider k's height dipping into the
last row of leaves that may be incomplete!*/

else{

    /*Our first objective, and probably the most difficult, is to find the max height.
There are only two possible answers: a) floor(lg k) or b) floor(lg k) - 1 .
To obtain the first answer, we need to dip down into the last, incomplete row. We need
to be careful, however, because we just can't pick and choose which nodes we want to

```

include... For example, say we wanted the last two nodes from the bottom row of a heap that consists of 5 nodes. If you draw this out, you can easily see that you cannot include the second to last node without including the entire tree!!!*/

/*We need to find out how many nodes, if any, on the bottom row could be used towards k's height. Because we fill a binary heap from left to right, and because we want to include as few as possible nodes on the bottom row in order to produce the max height, we need to, therefore, read the binary heap from right to left. We know that k's bottom level could have a max of $\text{floor}(\lg k)$ nodes... ,so, let's mod the bottom by this number and find out how many we are left with. "Leftovers"

Note: Realize that each level of the binary tree can be split in half again and again into smaller leaf sets of subtrees which roots are further up the tree.*/

//Step 3A: Get the leftovers as described above.

int leftovers = (last_row_leaves % (static_cast<int>(pow(2, lg_k))));

//There is the possibility that this will return 0.. we don't want this, so we add back on full bottom of

k
if (leftovers == 0){
 leftovers += (static_cast<int>(pow(2, lg_k)));
}

/*Surprisingly, we can now find the max height with one last test. If the number of leftovers exceeds what could be added to k's complete tree, at the second to last level, without surpassing k, then max height is option a, else, it's option b*/

//Step 3B: Determine if we can use the leftovers

if ((k - (static_cast<int>(pow(2, lg_k)) - 1)) >= leftovers){
 max_height = lg_k;

}
else{
 max_height = lg_k - 1;
}

/*Min Depth: This is really easy now that we've found the max height. We need to determine if we used any of the leftovers

to get to our max height. If this is the case, we can just take the tree height minus max height. If we were unable to use

these values, all hope isn't lost yet! Remember how we partitioned the last row by $2^{(\lg k)}$? If there are enough nodes on the

next level up that don't have children that could serve as a full base to tree of root k, then our min depth will also be

treeheight - lgk . Else, if neither of these cases hold, we are forced to have a full row on the very bottom and have (treeheight + 1) - lgk*/

//Step 3C: Find the min depth

//Do we have enough childless parents on second to last row?

if ((pow(2, lg_n) - last_row_leaves) / (pow(2, lg_k)) > 1){
 min_depth = lg_n - lg_k - completeBoost;

}
else{
 min_depth = lg_n - max_height;
}

}

cout << "Nodes: " << n << " , Value: " << k << "th smallest" << endl << endl;

cout << "Max Height: " << max_height << endl;

cout << "Min Height: " << min_height << endl;

cout << "Max Depth: " << max_depth << endl;

cout << "Min Depth: " << min_depth << endl;

cout << endl << endl;

return 0;}

Binary Heaps Part 2

Question:

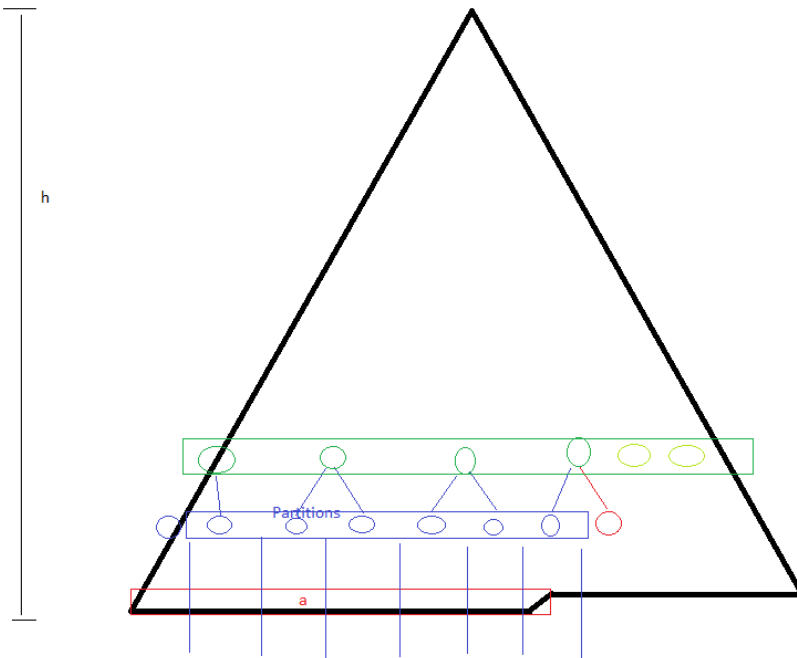
Given a binary max heap of n nodes, find how many nodes have a height of k .

We begin by partitioning the last row of the heap by how many leaves a node with height k would have. To do this, we will need to find the following:

$$\lfloor \log_2 n \rfloor = h ; \text{height of the tree}$$

$$2^k = p ; \text{leaves of a tree of height } k$$

$$n - (2^h - 1) = a ; \text{leaves on bottom row of tree}$$



The partitioning can be seen by the blue lines in the figure to the left. The number of partitions is of course found by dividing 'a' by 'p'. You may notice that this will not always come out to a whole number. In this case, we want to take the ceiling of the number because height is determined by the longest arm, not by the lowest complete row.

$$\left\lceil \frac{a}{p} \right\rceil = y$$

In this case, 'y' is not only represented by the number of blue partitions, but also by the number of blue nodes.

Now, the only other nodes that could share the same height must exist one level above the ones we just found. (This is assuming that we are not working with a complete tree, and that $2^h - a \geq p$.) We take into consideration the number of nodes that exist on this level, and how many of them are not being used as parents of the blue. (Light Greens)

We divide y in half, taking the ceiling of the answer. $\left\lceil \frac{y}{2} \right\rceil = B$. (Dark Green Nodes) To find the number of nodes on the row we take: $2^{h-k-1} = C$. Subtracting C from B , we are left with the number of light green nodes. $C - B = Z$. Finally, we add up the total: Nodes with height k : $Y+Z$

Note: The code presented below is calculator programmable.

```

/*Created By: Jared Fowler
June 2014
Math 482 Combinational Algorithms
Professor John Dye

```

```

A common test question is to find the number of
nodes with height 'n'. This program takes care
of this task.*/

```

```

#include <iostream>
#include <cmath>
using namespace std;

int main(){

    int nodes = 27;
    int height = 2;

    //Step 1: We find the height of the root
    int treeHeight = log(nodes) / log(2);
    //Test if the height is valid
    if (treeHeight < height){
        cout << "A binary heap of " << nodes << " nodes has 0 nodes of height " << height << endl << endl;
        return 0;
    }
    //Step 2: We find out how many nodes are being used on the bottom row of the entire tree
    int usedNodes = nodes - (pow(2, treeHeight) - 1);
    //Step 3: We find out how many nodes are not being used on the bottom row of entire tree
    int freeNodes = pow(2, treeHeight) - usedNodes;

    /*The next few steps are more tricky. First, imagine an incomplete binary tree. We want to
    find the nodes with a height of k, which have height k due to having nodes on the very bottom
    row of the tree. At height k, there will be 2^k nodes under a node with that height. We can
    therefore take the number of nodes at that row "usedNodes" and divide it by 2^k. The cieling
    of this is the number of nodes with a height of k*/

    //Step 4: Get nodes with hieght k that reach into very bottom row
    int total1 = ((usedNodes / (pow(2, height))) + .99999); //Hopefully works 99% of time for cieling

    /*The remaining nodes with height k must be 1 level higher than the ones we just found. We don't
    want to double count, however, so we need to find how many nodes make part of the ones we just counted*/

    //Step 5: Get nodes that have already counted nodes as subtrees
    int parentOfCounted = ((total1 / 2.0) + .99999);

    /*If this number is equal to the number of nodes in that row of the tree, then we are done. Also confirm
    that we are not going into a depth < 0*/

    //Step 6: Are we done?
    if (parentOfCounted == (pow(2, treeHeight - height - 1))){
        cout << "A binary heap of " << nodes << " nodes has " << total1 << " nodes of height " << height << endl
    << endl;
    }
    else if ((treeHeight - height - 1) < 0){
        cout << "A binary heap of " << nodes << " nodes has " << total1 << " nodes of height " << height << endl
    << endl;
    }

    /*How many more nodes are on that row? Find it, and subtract it from the 'parentOfCounted'*/

    //Step 7: Find the remaining nodes
    else{
        int total2 = pow(2, treeHeight - height - 1) - parentOfCounted;
        cout << "A binary heap of " << nodes << " nodes has " << total1 + total2 << " nodes of height " << height
    << endl << endl;
    }

    return 0;
}

```

Binary Josephus

Question:

Find the ultimate, penultimate, and antepenultimate survivors of Josephus ($n, 2$).

Notice that the question is strictly asking for a binary Josephus. ((n, k) where $k = 2$) You might come across something other than a binary Josephus, but most likely nothing too difficult to solve by hand. According to professor Dye, there currently doesn't exist any formula to solve for what we want where k is not equal to 2.

```
//Get the log_2
logV = log(n) / log(2);

//Find the last element, the winner.
last = 2 * (n - pow(2, logV)) + 1; //Correct For ALL!!!

//SECOND TO LAST
second = last + pow(2, logV - 1);
//If this value is over n, we do a different technique
if (second > n)
    second = last - pow(2, logV);

//THIRD TO LAST
third = second + pow(2, logV - 1);
//If this value is over n, we do a different technique
if (third > n)
    third = second - pow(2, logV);
//If the above value goes below or equal to zero, we do a different technique
if (third <= 0)
    third = second + pow(2, logV - 2);
```

The algorithm above, written in C++, solves a Binary Josephus. This algorithm has been tested and found to be working with $4 \leq n \leq 15,000$. (I ran it alongside a brute force method and compared the results.) Credit for this algorithm should also go to Natalia Alonso and Donald Eckels. (It is also very possible that this algorithm exists somewhere online.)

BST Permutation/Pattern Stuff

Question(s):

What is the average height of a bst tree with the input list {a, b, c, d, ...}?

What is the probability of having a bst of height k given any permutation of {a, b, c, d, ...}?

How many different looking trees of height k are there given an input list of {a, b, c, d, ...}?

What is the new average height of a bst if a random node is deleted?

...etc.

Note: We are working under the assumption that the input list has no duplicates. In reality, it doesn't even matter what the input list contains, as long as none of the numbers are equal to each other.

Ok, I don't know any fancy or smart way of answering the first question, nor the second. Normally what you want to do is take different cases, for example, when 'a' is the root, or when 'b' is the root, etc. NOTE: You are pretty much guaranteed to see some symmetry here. If your input list has 12 elements, you only need to do the first 6 and then multiply by 2. Symmetrical cases will work from the outer two nodes working inward. 1-12 ; 2-11; 3-10 ; etc.

Concerning bst patterns, we have a useful formula called the Catalan Formula. $\frac{(2n)!}{(n+1)!(n!)}$, where $C_n = \Theta\left(\frac{4^n}{n^2}\right)$

Unfortunately, the question above requires a bit more work than just plug 'n chugging with this formula.

I wrote a few programs which answered the above questions by brute force. (Stepped through every single permutation, made trees, counted, destroyed, repeat.) Below are the results:

Permutation Heights:

	k=1	2	3	4	5	6	7	8	9	10
n=2	2	-	-	-	-	-	-	-	-	-
3	2	4	-	-	-	-	-	-	-	-
4	0	16	8	-	-	-	-	-	-	-
5	0	40	64	16	-	-	-	-	-	-
6	0	80	400	208	32	-	-	-	-	-
7	0	80	2240	2048	608	64	-	-	-	-
8	0	0	11360	18816	8352	1664	128	-	-	-
9	0	0	55040	168768	10448	30016	4352	256	-	-
10	0	0	253440	1508032	1277568	479040	99200	11008	512	-

Pattern Heights:

	k=1	2	3	4	5	6	7	8	9	10
n=2	2	-	-	-	-	-	-	-	-	-
3	1	1	-	-	-	-	-	-	-	-
4	0	6	8	-	-	-	-	-	-	-
5	0	6	20	16	-	-	-	-	-	-
6	0	4	40	56	32	-	-	-	-	-
7	0	1	68	152	144	64	-	-	-	-
8	0	0	94	376	480	352	128	-	-	-
9	0	0	114	844	1440	1376	832	256	-	-
10	0	0	116	1744	4056	4736	3712	1920	512	-
11	0	0	94	3340						1024

Average Height after Random Delete:

	New Average After Random Delete	Specific Node Rank Delete k=1	2	3	4	5	6	7	8
n=2	0								
3	1	1	1	1					
4	1.64583	1.67	1.5	1.75	1.67				
5	2.3133	2.333	2.233	2.25	2.417	2.333			
6	2.775	2.8	2.73	2.65	2.75	2.92	2.8		
7	3.24218	3.27	3.23	3.15	3.15	3.25	3.38	3.27	
8	3.64648	3.67	3.65	3.58	3.54	3.59	3.675	3.7875	3.67
9	3.99439								

When graphed, the above table does seem to make out a logarithmic curvature as would be expected, but does not constitute any simple formula or algorithm.

The next few pages contain source code for 3 of the programs I created and used to find the above table results.

Probability

I don't consider myself to be an expert on probability, and for that reason I will not attempt to go into any form of lengthy explanation concerning these problems, instead, I will simply list out some of the more common questions, which can be found in the back exams, along with the method of solving them.

Q: Given n initially empty bins, how many empty bins do we expect after k ball tosses?

$$N - \left(1 - \left(1 - \left(\frac{1}{N}\right)\right)^M\right), \text{ where } N = \# \text{ bins}, M = \# \text{ ball tosses}$$

Q: How many ball tosses are needed in order to fill n initially empty bins?

$$\sum_{i=1}^N \frac{N}{N-i+1}, \text{ where } N = \# \text{ bins}$$

Q: B-Day Problem. How many people n are needed for us to expect there to be at least two people born on the same day?

$$\frac{\binom{n}{k}}{D} = 1, \text{ } n = \# \text{ people}, k = \# \text{ shared BirthDays}, D = \# \text{ days in given year}$$

Q: Given n initially empty bins, how many tosses before 2 balls in 1 bin?

$$\sum_{I=2}^{N+1} \frac{N!}{(N-I+1)!} * (I-1) * \frac{1}{N^I}$$

Q: Given 6 coin tosses, what is the expected number of tosses until we get 3 heads and 3 tails?

*Notice that for many of these questions we want the **expected** value.*

For this question, we first use the binomial theorem. $\binom{n}{k} p^k (1-p)^{n-k}$

This will give us p , which we can now plug into the geometric expected value,

$$\text{which is what this question is really asking for: } E[X] = \frac{1}{p}$$






Instant Insanity

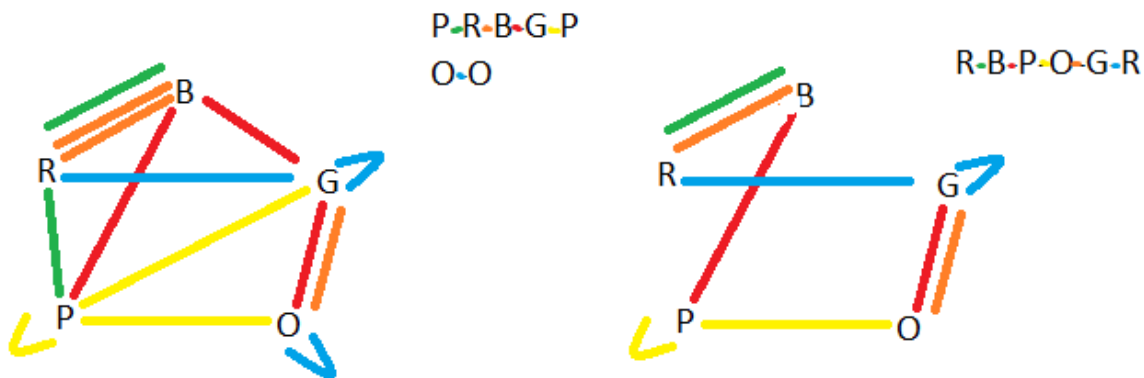
These can be very difficult, tricky, and time consuming. They seem to come in two categories: 5 colors and 12 colors. If you get one with 12 colors, good luck. Fortunately, however, there are a few helpful tricks I can give the reader on how to solve a 4-6 color instant insanity problem.

<http://www.jaapsch.net/puzzles/insanity.htm>

The link above is a good reference to what I'm about to explain. In fact, there is only one small change that I make in order to ease the problem. Instead of lines labeled by numbers, I use different color lines, each color representing a single cube. I **HIGHLY** recommend keeping 5-6 distinct colored pencils or pens handy for the tests!

Q: Does there exist a solution?

Cube	Opposite Pair One	Opposite Pair Two	Opposite Pair Three
I 	b-g	g-o	b-p
II 	g-o	b-r	b-r
III 	o-p	p-p	g-p
IV 	p-r	b-r	g-r
V 	g-g	g-p	o-o



NOTE: As a last resort (you don't have sufficient time on the test) just write NO SOLUTION. You at least then have a 50-50 chance of getting it right!