title: "coursework2_nkatz01_dsa"
author: "NK"
date: "12 December 2018"

1. Fast computation

I've found this solution here https://stackoverflow.com/questions/10195252/calculate-x-y-in-olog-n but after thoroughly studying it, I think I now understand how it works and could not think of a better solution myself.

```
 public static int computeAtoPowN(int a, int n) {
if (n == 0) {
  return 1;
}
int results = computeAtoPowN(a, n / 2);
if (n % 2 != 0) {
  return a * results * results;
} else {
  return results * results;
}
```

I believe the reason why this solution works is due to the realisation that a exponentiation operation can be broken down in the following. For an odd exponent n:

a^n=a(a^2)^(n-1)/2

(Let the a(a^2) part of the equation, be called operation A.)

and for an even exponent n:

a^n=(a^2)^n/2

(Let the (a^2) part of the equation be called operation B.)

This is what's happening in this algorithm.

- The base case is n==0, returning 1 to the last caller, always resulting in producing a value == to the base, for the second return.
- if/when the caller's n is odd, operation A is carried out (the IF).
- if/when the caller's n is even, operation B is carried out (the second condition - Else).
- and it happens n/2 [for even numbers] because, n is fed to itself n/2 every time
- it happens (n-1)/2 [for odd numbers] because n is an int and so the fraction is discarded everytime the 'division operations' is being performed when the function calls itself.
- yet, whilst in the case where if one before the last call to itself, n was even, coming back from the base case, the sequence of operations will be odd, even (first condition [if], second condition [else] - eg 1,2 where n begins with 2); for a case where one before the last call to itself, n was odd, coming back from the base case, the sequence of operations will always be odd, odd one after the other (first condition [if] twice, - eg 1,3 where n begins with 3), making up that additional multiplication operation (by the base - non-square)

it needs due to its odd, odd integer.

2. Missing Number problem with a Sorted array

```
        public static int missingNumberlogn() {
        int A[] = {1,2,3,4,5,6,8,9,10};//eg

        int start = 0;
        int end = A.length - 1;
        int middle;
        while (end - start > 1) {
          middle = (start + end) / 2;
          if (A[middle] == (middle + 1))
            start = middle + 1;
          else
            end = middle - 1;
        }
        if (start != end && A[end] - A[start] > 1) {// array is size 2
and difference >1
          return (A[start] + 1);
        }

        else if (start != 0) {//array is of size 1 but not the first
element
          if (A[end] - A[end - 1] > 1)// no worries of subtracting
previous element
            return A[end] - 1;
          else
            return A[end] + 1;//number missing is at the end boundary
of sequence. program will reach this if array starts at 1
        }

        else if (A[start + 1] - A[start] > 1)//array is of size 1 and
is the first element of array and difference to second element >1
          return A[start] + 1;
        else
          return A[start] - 1; //number missing is at the beginning
boundary of sequence. program will reach this if array starts at any
number after but not including 1

      }
```

3. Subarray problem

3.1 Iterative algorithm

```
    int[] A =  {1,2,3,4}; //eg
    int[] B = {1,2,5};    //eg
    int count=0;

    for (int i=0; i<B.length; i++){
      for (int j=0; j<A.length; j++)
      {
        if (B[i]==A[j])
        {count++;
        break; }
      }


    }
    if (count==B.length)

    System.out.println("B is sub array of A");

    else
        System.out.println("B is not a sub array of A");

  }
```

I believe this solution works because if not all values are found also in A, then the variable count will not be the same at B.length.

I believe this is a 0(n^2) runtime because it's a loop within a loop.

3.2 Recursive algorithm

To query of B is subarray of A, call the following method with method call recursiveSol(0, A.length - 1, A, B); (The first param refers to first element in B, the second, to the last element in A.)

```
    public static int recursiveSol(int j, int i, int[] A, int[] B) {
    int found = 0;
    if (B.length > A.length) {
      System.out.println("Array B is not sub of Array A");
      return B.length - 1;
    }
    if (i == -1) {// base case - Array A's finished traversing back to
front
      return 0;
    }

    found += recursiveSol(j, i - 1, A, B); // increase indexes in A -
equivalent of inner loop - pass 1 (if matching
                                           // value found in A) or 0
(if matching value not found in A)through the
```

```
                                           // recursive calls back to
the first caller.

    if (B[j] != A[i])
      found = found + 0;

    else

      found = found + 1;

    j++;

    if (i == A.length - 1 && j < B.length)// increase index in B -
equivalent of outer loop
      found += recursiveSol(j, i, A, B); // sum up the 1s gained (or
not) for each element in B - from each "whole loop
                                         // cycle" through A.

    if (j == 1 && i == A.length - 1) {// it has finished reversing
from all the recursion calls
      if (found == B.length)

        System.out.println("B is a subarray of A");
      else
        System.out.println("B is not subarray of A");
    }

    return found;// added advantage: if one wants evaluation above to
be done by outside caller -
                 // that's possible.

  }
```

3.3 Array Sorted (SortedAnd0ofN() uses helper method compareArrays(int[] A, int j, int[] B, int i))

```
        public static void SortedAnd0ofN(){

    int[] A =  {1,2,3,4};//eg used
    int[] B = {5,1,2,3};//eg used
    if (B.length>A.length)//we can do this as we know values aren't
repeated
   { System.out.println("Array B not subof array A");
    return; }
    int i=0;//index counter for B
    int j=0; //index counter for A
    int flag=0;
    while (i<B.length)
      {
        if (flag==1 || j==A.length)//if value of corresponding index
in A is more than value of corresponding index in B, or if all indexes
in A have been visited - terminate the loop
        {
        break; }
        flag=compareArrays(A, j, B, i); //send array ref and index no
to compareArrays func
        System.out.println(flag);
        if (flag==-1)//if value of corresponding index in A is less
than value of corresponding index in B,
          i--;//pass again the current index no of Array B.
        i++;
        j++;//whilst increasing the index of A, regardless.

      }

    if (flag==0)
    System.out.println("Array B is subbary of A");
    else
    System.out.println("Array B is not subbary of A");
  }

  public static int compareArrays( int[] A, int j, int[]  B, int i){
    if (A[j]>B[i])
    return 1;//because Arrays are sorted, we know the chance for the
value to be found in A is gone.
    if (A[j]==B[i])
    return 0;
    else return -1; //we may still find the value further in A.
  }
```