Chris McCord
CS 411
HW 4

A.)
```
count = 0
for ( i = 0; i <= str.size() - 2; i++ ) {
  if ( str[i] == 'A' ) {
    for ( j = 0; j <= str.size() - 1; j++ ) {
      if ( str[j] == 'B' )
        count = count + 1
    }
  }
}
return count;
```

- Brute force efficiency class is $\Theta(n^2)$, because we loop through every character in the string ~~[illegible]~~ n times.

```
~~count~~ count, countAs = 0;
for ( i = 0, i <= str.size() - 1 ) {
  if ( str[i] == 'A' )
    countAs = countAs + 1;
  if ( str[i] == 'B' )
    count = count + countAs;
}
return count;
```

- Optimized is linear because we go through the length of the string at most 1 time.

C.) a.) $T(n) = 4T(n/2) + n$, $T(1) = 1$

$a = 4$, $b = 2$, $c = 1$, $d = 1$.

$4 > 2^1$, therefore the order of growth is $\Theta(n^2)$

b.) $T(n) = 4T(n/2) + n^2$, $T(1) = 1$

$a = 4$, $b = 2$, $c = 1$, $d = 2$

$4 = 2^2$, therefore the order of growth is $\Theta(n^2 \log n)$

c.) $T(n) = 4T(n/2) + n^3$, $T(1) = 1$

$a = 4$, $b = 2$, $c = 1$, $d = 3$

$4 < 2^3$, therefore the order of growth is $\Theta(n3)$

D.) Use divide-and-conquer.
Select a nut and try all of the bolts to find the matching one. Seperate the bolts that are smaller and lager than the nut into seperate piles. Do the same thing with the other nuts until each bolt has a nut. Efficiency cost is $O(n \log n)$.

E.) a.) Preorder - a b d e c f

b.) inorder - d b e a c f

c.) Postorder - d e b c f a

common = 0

F.) Max common = 0

```
for(i=0; i<=2n-1; i++){
    if ( sorted list[i] is left endpoint )
        common = common+1;
    if (sorted list[i] is right endpoint)
        common = common - 1;
    if (max common < common)
        max common = common;
}
return max common;
```

- a_i's and b_i's can be sorted in O(n log n) time and the loop executes 2n times. Therefore the running time is O(n log n).

G.
```
max prod(int n) {
int x = 0;
    if(n/2 == 0)
        x = n/2
    else  x = L n/2 ];
return x;
```

Constant time.

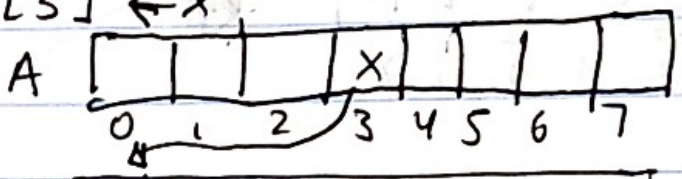
H. Take a binary Tree and two vertices, u & v of T
return True if u is an ancestor of v and false if
u is not an ancestor.

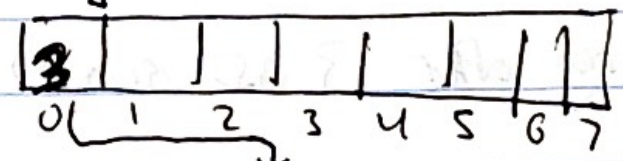
```
pre = 0;
create stack S;
Push root of T into S;
while(!S.empty()){
    pop x from S;
    Preorder(x) = pre;
    pre = pre + 1;
    if (left Tree is not empty)
        push it onto S.
    if(right tree is not empty) push it onto S.
}
post = 0;
empty S;
Push root of T onto S;
while(!S.empty()){
    pop x from S
    if (T_left is not empty) push it onto S;
    if (T_right is not empty) push it onto S;
    Postorder(x) = post;
    post = post + 1;
}
if((Preorder[u] ≤ Preorder(v)) && (postorder(u) ≥ postorder(v))
        return true;
else
    return false;
```

J.) The English language only has 26 letters. Creating a hashing function that only uses the first letter means that we can only have 26 addresses, which is very low. Therefore, it's not a good hashing algorithm.

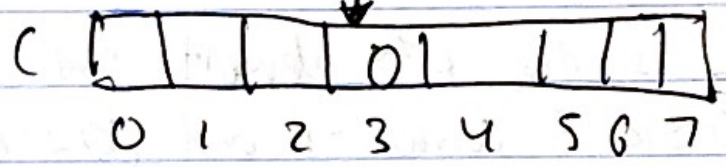
I.) counter = 0

A[3] ← X

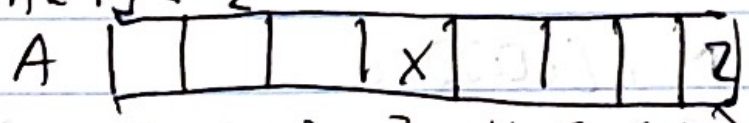


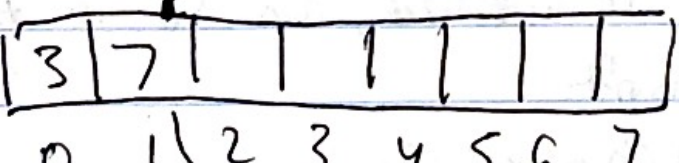


counter = 1

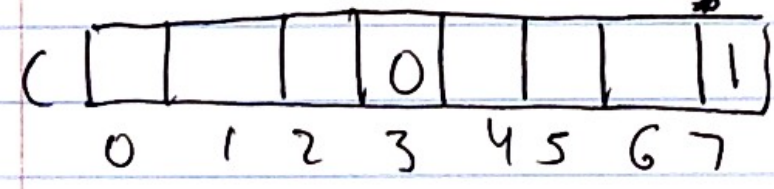


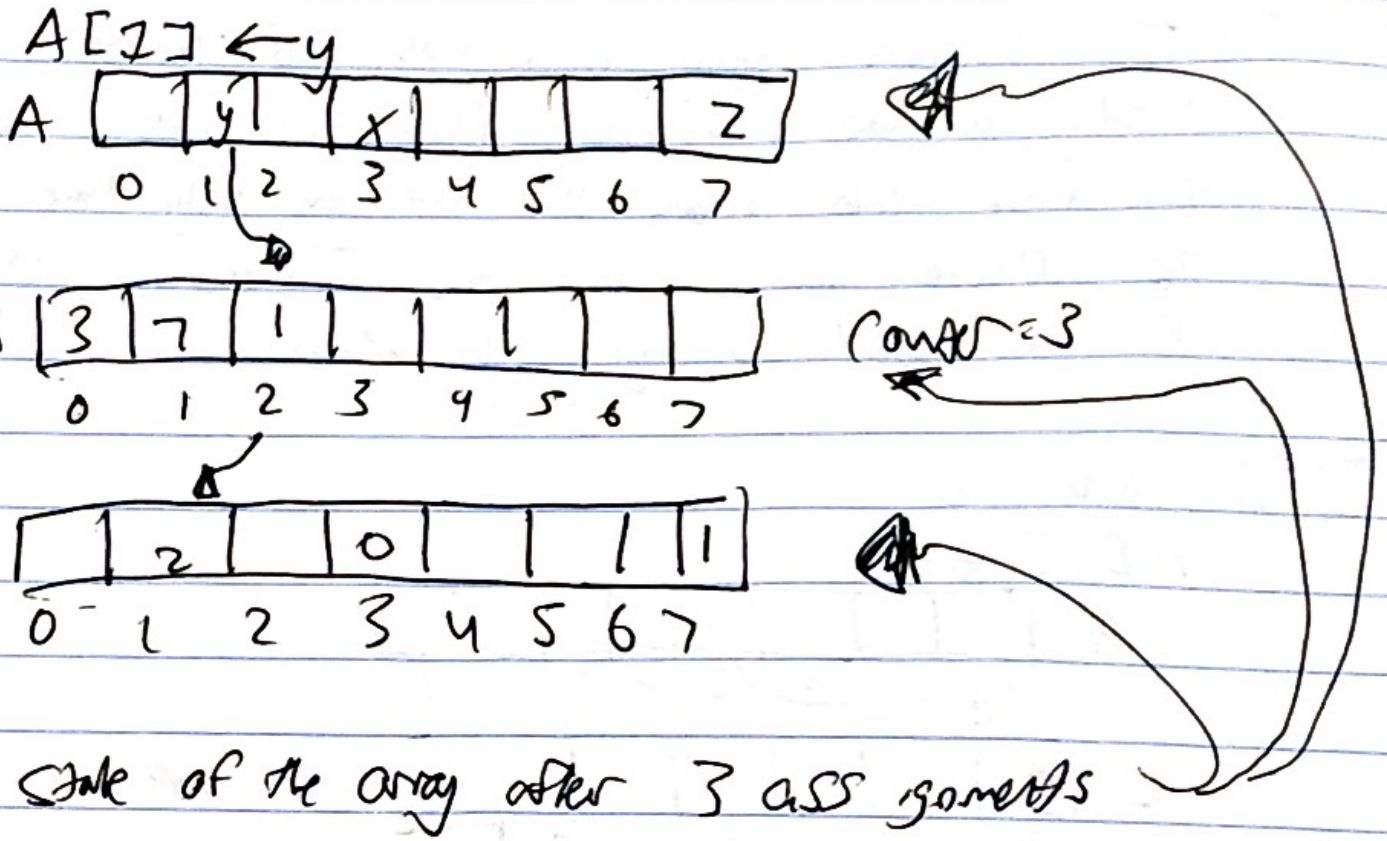
A[7] ← 2





counter = 2





State of the array after 3 assignments

b.) ~~B[K]~~ A[i] is the K^{th} element that is initialized, then $0 \leq K \leq count - 1$ and C[i] = K.

B[K] = i or B[C[i]] = i

to See if A[i] is initialized, check B[C[i]]
If B[C[i]] is initialized, then so is A[i].
Otherwise it isn't.

Comparison Counting Sort.