Problem 1

Give an algorithm that sorts n 0/1-valued records (that is, the key of each record is either 0 or 1). Your algorithm should be stable and run in $\mathcal{O}(n)$ time.

```
# this is like a bucket sort
# let A be list to be sorted
list_zero = []
list_one  = []
for i=0 to A.length - 1:
    if A[i] == 0:
        list_zero.append( A[i] )
    else:
        list_one.append( A[i] )

return list_zero + list_one
```

Problem 2

Give an algorithm that sorts n 0/1-valued records. Your algorithm should be an in-place algorithm (that is, the algorithm uses a constant amount of storage space in addition to the original input array) and runs in $\mathcal{O}(n)$ time.

```
# this is like a counting sort
# let A be the array to be sorted
counters = [0, 0]
for i=0 to A.length-1:
    counters[A[i]] = counters[A[i]] + 1

i = 0
for j=0 to counter.length-1:
    for j=0 to counters.length-1:
        A[i] =  j
        i = i + 1

return A
```

Problem 3

Give an algorithm that sorts n 0/1-valued records. Your algorithm should be in-place and stable.

```
# this is bubble sort
# let A be the arry to be sorted
for i=1 to A.length:
    for j=0 to A.length-1:
        if A[j] > A[j+1]:
            swap( A[j], A[j+1] )
```

Problem 4

Can any of your algorithms from the first 3 questions be used to sort n records with b-bit keys using radix sort in $\mathcal{O}(bn)$ time? Explain how if yes or why if not.

The algorithm in Problem 2 could be used. The algorithm provided in Problem 1 is not stable. Finally, the algorithm provided in Problem 3 takes quadratic time.

Problem 5

Augment the binary search tree (BST) data structure so that it supports queries of the form "rank(k)=?", where rank(k) is defined to be the number of nodes in the tree whose keys are less than or equal to k. The running time of each query should be $\mathcal{O}(h)$, where h is the height of the BST. (that is, show how to add some additional fields to the BST data structure so that there is an efficient algorithm that can answer such queries.)

The binary search tree can be augemented with an additional field for each node called *size*. This holds the number of nodes that are in the subtree of a node.

```
rank( node ):
r = size( left( node ) ) + 1
temp = node
while temp != root:
    if temp = right( parent( temp ) ):
        r = r + size( left( parent( temp ) ) ) + 1
    temp = parent( temp )

return r
```

Problem 6

Suppose there are k processes $P_1, \ldots, P_k$, all competing for the exclusive access to a resource (e.g. a communication channel). Suppose we divide time into discrete *rounds*. During each round, if there are at least two processes requestion the resource, then all the processes fail to get the resource. If there is a single process requests the resource at a round, then the request will be fulfilled. The main challenge is there is no communication among these k processes. The purpose of this problem is to design and analyze a *randomized* protocol for these processes so that after a reasonable number of rounds, all processes will get access to the resource (with high probility).

The protocol is very simple: during each round, every process requests the resource with a uniform probability p independently.

(a) For each round and every fixed process, what is the probability that this process successfully accesses the resource? What value of p maximizes this probability?

(b) Let the probability you just derived be r. Prove that $1/ek \leq r \leq 1/2k$.

(c) For a fixed process, what is the probability that is fails to access the resource for all the first t rounds? How do you set the value of t to make this failure probability less than $1/k^3$?

(d) Recall the very useful probability inequality of *union bound* and give a value of t such that the probability that there is a process that has not accessed the resource after t rounds is at most $1/k^2$ (such a probability is extremely small when k is larger).

(a) Let $A[i,r]$ represent some process P_i attempting to access the resource in round r.
Then $Pr[A[i,r]] = p$ and $Pr[\overline{A[i,r]}] = 1-p$.
Now let $S[i,r]$ represent some process P_i successfully accessing the resource in round r.
Then $S[i,r] = A[i,r] \cap (\cap_{i \neq j} \overline{A[j,r]})$ and now,
$Pr[S[i,r]] = Pr[A[i,r]] * \prod_{i \neq j} Pr[\overline{A[j,r]}] = p(1-p)^{k-1}$

The value of p that maximizes this probability is $1/k$.

(b) $\lim_{k \to 2}(1-(1/k))^k = 1/4$ and $\lim_{k \to 2}(1-(1/k))^{(k-1)} = 1/2$.
Thus, $1/ek \leq r \leq 1/2k$ is true.

(c) Let $F[i,r]$ represent some process P_i failing to access the resource for all rounds r.
Then $Pr[F[i,r]] = \cap_{i=1}^{r} \overline{S[i,r]}$ and then,
$Pr[F[i,r]] = (1 - Pr[S[i,r]])^r \leq (1-1/ek)^r$.
Now if we set $r = ek$ then, $Pr[F[i,r]] \leq (1-1/ek)^{\lceil ek \rceil} \leq (1-1/ek)^{ek} \leq 1/e$.
Also if we increase r to $\lceil ek \rceil * c\ ln\ k$,
$Pr[F[i,r]] \leq ((1-1/ek)^{\lceil ek \rceil})^{c\ ln\ k} \leq (1/e)^{c\ ln\ k} = e^{-c\ ln\ k} = k^{-c}$.

(d) Let $F[i,r]$ represent some process P_i failing to access the resource for all rounds r.
Then $Pr[F[i,r]] = \cap_{i=1}^{r} \overline{S[i,r]}$ and then,
$Pr[F[i,r]] = (1 - Pr[S[i,r]])^r \leq (1-1/ek)^r$.
Now if we set $r = ek$ then, $Pr[F[i,r]] \leq (1-1/ek)^{\lceil ek \rceil} \leq (1-1/ek)^{ek} \leq 1/e$.
Also if we increase r to $\lceil ek \rceil * c\ ln\ k$,
$Pr[F[i,r]] \leq ((1-1/ek)^{\lceil ek \rceil})^{c\ ln\ k} \leq (1/e)^{c\ ln\ k} = e^{-c\ ln\ k} = k^{-c}$.