A2

Jianan Luo
20523403

1. a) 1. Find the node with the assumed name
2. Swap the node we found and the last node
3. Delete the node we found
4. Bubble down the swaped node
The worst case run time is: $O(n)+O(\log n)$
Step 1: $O(n)$
Step 2: $O(1)$
Step 3: $O(1)$
Step 4: $O(\log n)$
∴ The time complexity of the worst case is $O(n)$

b) 1. Find the node with the assumed priority
2. Swap the node we found with the last node
3. Delete the node we found
4. Bubble down the swaped node.
The worst case run tme is: $O(\log n)+O(\log n)$
Step 1: $O(\log n)$
Step 2: $O(1)$
Step 3: $O(1)$
Step 4: $O(\log n)$
∴ The time complexity of the worst case is $O(\log n)$

c) 1. Swap the node at the index and the last node
2. Delete the node at the index
3. Bubble down the swaped node
Step 1 = $O(1)$
Step 2 = $O(1)$
Step 3: $O(\log n)$
The worst case run time is = $O(\log n)$
∴ The time complexity of the worst case is $O(\log n)$
which is still belongs to $O(n)$

2. The algorithm:

```
i = 0
while i != n-1
    if i == keyValue(A[i])
        i++
    else
        swap(A[i], A[keyValue(A[i])])
    endif
endwhile
```

Note: The function KeyValue return the key value of an element in array A.
So this algorithm basically loop from 1 to n-1 and in the loop, check if the current element's index in the array equal to it's value. If it is, the counter of the loop plus 1, Otherwise, swap the current element with the element at th index is the key value of the current element.
For example when i=0 and A[5] = {3, 1, 2, 0, then we do swap → A[5] = {0, 1, 2, 3, 4}.
Then, A[5] = {0, 1, 2, 3, 4}, since i == keyValue(A[i++; We keep doing this until the loop is don

3. The algorithm:

1. convert the array to binary (32 bits) enough to
2. create a loop from 0 to 31, counter is "bitInde
3. we put every elements which "bit index" is 0 to left of th the array. And for those are 1, put them to right of
4. Keep the k^{th} smallest number, delete rest of the arra
5. Convert them back.

This is kind like a binary radix sort

Step 1: $O(n)$
Step 2+3 = 32$O(n)$
Step 4: $O(n)$
Step 5: $O(n)$
∴ The run time will be $O(n)$.

4. a) n-1

We compare the first coin with the second coin. If weight of first is not equal to the weight of second, then we know the light one is counterfeit, and the heavy one is genuine. If their weight is the same we just place the 2nd on a side and keep doing this experiment with other coin, until we found which are genuine and which are counterfeit. Then we pick a genuine coin and a counterfeit coin, make them a pair, we call this base pair. Then we grab 2 unknown coin and compare them with the base pair. If the unknown pair is lighter, than they all counterfeit, if is heavier, than they all genuine, otherwise we need to weigh the pair and check which is genuine, which is counterfeit.

So for the worst case:

1. We found our base case at last
2. For each unknow pair, 1 genuine 1 counterfeit.

They both = n-1.

b) if CompareWeight({C1; C2}) != "both weight same"

if (compareWeight({C1,C2; C3,C4}) == "first weight more"

if compareWeight({C1; C3}) == "first wight more"

add c1 to A[]

else

add C2 to A[]

else if compareWeight({C1,C2; C3,C4}) == "second weight more"

add C3 & C4 to A[]

if compareWeight({C1; C3}) == "second weight more"

add C2 to A[]

else

if compareWeight({C1; C3}) = "first wight more"

add C1 & C4 to A[]

else

add C2 & C3 to A[]

else // C1 = C2

if compareWeight({C3; C4}) == "first weight more"

add C3 to A[]

if compareWeight({C1; C3}) == "both weight same"

add C1 & C2 to A[]

else if compareWeight({C3; C4}) == "second weight more"

add C4 to A[]

if compareWeight({C1; C4}) == "both weight same"

add C1 & C2 to A[]

else

if compareWeight({C1; C3}) == "first weight more"

add C1 & C2 to A[]

else

add C3 & C4 to A[]

return A[]

So the worst case for this algorithm is 3

which is 4-1=3

n-1 = 3 which is exactly match part a

4.c)
```
O(1) { L = "first weight more"
       R = "second weight more"
       M = "both weight the same"
       i = 2
O(n) { while (i != n+1)
         switch (CompareWeight({C1 ; Ci}))
           case L
             add C1 to A[]
             break;
           case R
             add Ci to A[]; add B[] to A[]
             break;
           case M // default
             add Ci to B[]
         endswitch
         i++
       endwhile
O(1) { basePair = C1 + Ci
O(n) { while (i+2 <= n)
         switch (CompareWeight({basePair ; Ci+1, Ci+2}))
           case R
             add Ci+1 & Ci+2 to A[]
           case M
             if (CompareWeight({A[0] ; Ci+1}) != L)
               add Ci+2 to A[]
             else
               add Ci+1 to A[]
             endif ; endswitch
         i+=2
       endwhile
O(1) { i++
       if (i == n)
         if (CompareWeight({A[0] ; Ci}) == M)
           add Ci to A[]
         endif
       endif
       return A[]
```

So the runtime when worst case is: O(n) which match part a)

5.a) Algorithm: return A[0]

Basically this algorithm will just return the first element in the array. Since the array has the size of n and there are at least $[n/2]+1$ x in that array, also it's like a uniform distribution, but with the probability choose dominant is $\frac{\frac{n}{2}+1}{n}$ which can be rewrite as $\frac{1}{2}+\frac{1}{n}$ which is larger than $\frac{1}{2}$. $(\frac{1}{2}+\frac{1}{n} > \frac{1}{2})$. Therefore the probability is at least $\frac{1}{2}$, and running time is O(1), since return A[0] $\in$ O(1).

b) Algorithm:

1. Convert the array to binary (32 bits) enough
2. Create a loop from 0 to 31, counter is "bitIndex"
3. We put every elements with "bit index" is 0 to left the array. And 1 to the right of the array.
4. Convert binary back to dec
5. Return the A[size/2].

Step 1: O(n)

Step 2+3: 32O(n)

Step 4: O(n)

Step 5: O(n)

∴ The run time will be O(n).

This is like a binary radix sort

And since there are at least $[\frac{n}{2}]+1$ terms of dominant, after the array get sorted, the median must be the dominant X, since the worst case is either dominants are the first half of the last half, but either these case, the median is still the dominant.

Therefore the algorithm always returns the correct answer and has expected-case running time O(

```
L = "first weight more"
R = "second weight more"
M = "both weight the same"
i = 2
while (i != N+1)
    switch (comparWeight (A[i], C[i]))
    case L
        add C[i] to A[]
        break;
    case R
        add C[i] to B[]; add B[i] to A[]
        break;
    case M // default
        add C[i] to B[]
    endswitch
    i++
endwhile

if basePair = C[1] + C[2]
while (i+2 <= N)
    switch (comparWeight(basePair, C[i], C[i+1]))
    case R
        add C[i+1] & C[i+2] to A[]
    case M
        if(comparWeight(C[A[0]], C[i+1])==L)
            add C[i+2] to A[]
        else
            add C[i+1] to A[]
        endif
    endswitch
    i++
endwhile

if(i == N)&&
    if((comparWeight(A[0], C[i])) == M)
        add C[i] to A[]
    endif
endif
return A[]
```

a) Pseudocode Algorithm: return A[0]

Basically this algorithm will sort the first element in the array. Since the size of n and there are at least $\lceil n/2 \rceil + 1$ X in that array, also it's like a uniform distribution, but with the probability decrease dominant as $\frac{n/2+1}{n}$ where can be rewrite as $\frac{1}{2} + \frac{1}{n}$ where is bigger than $\frac{1}{2}$. ($\frac{1}{2} + \frac{1}{n} > \frac{1}{2}$). Therefore the probability is at least $\frac{1}{2}$, and running time as $O(1)$. Since return A[0] is $O(1)$.

b) Algorithm:

1. Convert the array to binary (32-bits) array (T)
2. Create a loop from 0 to 31, (counter is "bitIndex")
3. We put every elements with bit index "is" 0 to left of the array. And 1 to the right of the array.
4. Convert binary back to dec
5. Return the A[Size/2] ↑

Time as like a binary radix sort

Step 1: O(n)
Step 2+3: 32O(n)
Step 4: O(n)
Step 5: O(n)

∴ The runn time will be O(n).

And since there are at least $\lceil \frac{n}{2} \rceil + 1$ terms of dominant, after the array get sorted, the median must be the dominant X, since the worst case is either dominant one the first half of the last half, but either those case, the median is still the dominant.

Therefore the algorithm always returns the correct answer and has expected-case running time $O(n)$