Recitation 4 (HW3)

Online: Xinyi Zhao xz2833@nyu.edu

GCASL 475: Yifan Jin yj2063@nyu.edu

New York University

Basic Algorithms (CSCI-UA.0310-005)

Problem 1

Problem 1 (5+12+5 points)

Let A be a sorted array with n distinct elements. Consider the following algorithm that finds an integer $i \in \{j \dots k\}$ such that $A[i] = x$, or returns **FALSE** if no such integer i exists.

```
1 FIND(A, j, k, x)
2     If j > k Return FALSE
3     Define i := ...
4     If A[i] = ... Return ...
5     If A[i] < ... Return FIND(...)
6     Return FIND(...)
```

(a) Fill in the blanks to complete the algorithm.

(b) Write the recurrence for the running time of the algorithm. Draw the corresponding recursion tree and find the explicit answer to the recursion.
Your answer must use $\Theta(.)$ notation, i.e., you should obtain both an upper bound and a lower bound (proof by strong induction is NOT needed).

(c) We assumed that the elements of A are distinct. Does the algorithm still work if the elements of A are not necessarily distinct? Justify your answer.

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = ... Return ...
5    If A[i] < ... Return FIND(...)
6    Return FIND(...)
```

If the program executes line 6, what does it indicate?

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = ... Return ...
5    If A[i] < ... Return FIND(...)
6    Return FIND(...)
```

If the program executes line 6, what does it indicate?

A[i] > …

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = ... Return ...
5    If A[i] < ... Return FIND(...)
6    Return FIND(...)
```

Thus, this recursive function is divided into three parts:

A[i] is larger than / smaller than / equal to ...

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = ... Return ...
5    If A[i] < ... Return FIND(...)
6    Return FIND(...)
```

What is the value of … (which is compared with A[i])?

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = ... Return ...
5    If A[i] < ... Return FIND(...)
6    Return FIND(...)
```

What is the value of … (which is compared with A[i])?

Recall the function is to find whether value **x** exists in the range of [j,k].

Also, when A[i] = …, there is no further recursion.

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = ... Return ...
5    If A[i] < ... Return FIND(...)
6    Return FIND(...)
```

Thus, we can infer that the comparison is between **A[i]** and **x**.

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = x. . Return ...
5    If A[i] < x. . Return FIND(...)
6    Return FIND(...)
```

Thus, we can infer that the comparison is between **A[i]** and **x**.

Also, when A[i] = x, Return i.

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = x. . Return .i. .
5    If A[i] < x. . Return FIND(...)
6    Return FIND(...)
```

Thus, we can infer that the comparison is between **A[i]** and **x**.

Also, when A[i] = x, Return i.

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = x. . Return .i. .
5    If A[i] < x. . Return FIND(...)
6    Return FIND(...)
```

So how do we design the recursive function?

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = x. . Return .i. .
5    If A[i] < x. . Return FIND(...)
6    Return FIND(...)
```

So how do we design the recursive function?

Recall array A is sorted.

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = x. . Return .i. .
5    If A[i] < x. . Return FIND(...)
6    Return FIND(...)
```

So how do we design the recursive function?

Recall array A is sorted.

If A[i] < x, then **A[j...i] < x**. Thus, x could only possibly exist in **A[i+1,k]**.

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = x. . Return .i. .
5    If A[i] < x. . Return FIND(...)
6    Return FIND(...)
```

So how do we design the recursive function?

Recall array A is sorted.

If A[i] < x, then **A[j...i] < x**. Thus, x could only possibly exist in **A[i+1,k]**.

Similarly, if A[i] > x, then **A[i...k] > x**. Thus, x could only possibly exist in **A[j,i-1]**

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3   Define i := ...
4   If A[i] = x. . Return .i. .
5   If A[i] < x. . Return FIND(A,i+1,k,x
6   Return FIND(A,j,i-1,x
```

So how do we design the recursive function?

Recall array A is sorted.

If A[i] < x, then **A[j...i] < x**. Thus, x could only possibly exist in **A[i+1,k]**.

Similarly, if A[i] > x, then **A[i...k] > x**. Thus, x could only possibly exist in **A[j,i-1]**

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3   Define i := ...
4   If A[i] = x. . Return .i. .
5   If A[i] < x. . Return FIND(A,i+1,k,x
6   Return FIND(A,j,i-1,x
```

The last question is: what is the value of i?

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = x. . Return .i. .
5    If A[i] < x. . Return FIND(A,i+1,k,x
6    Return FIND(A,j,i-1,x
```

The last question is: what is the value of i?

Recall we want to optimize the worst-case running time.

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := ...
4    If A[i] = x. . Return .i. .
5    If A[i] < x. . Return FIND(A,i+1,k,x
6    Return FIND(A,j,i-1,x
```

The last question is: what is the value of i?

Recall **partition** function, it is most optimal if array is divided into two equal parts.

Problem 1

(a) Fill in the blanks to complete the algorithm.

```
3    Define i := .(j+k)/2
4    If A[i] = x. . Return .i. .
5    If A[i] < x. . Return FIND(A,i+1,k,x
6    Return FIND(A,j,i-1,x
```

The last question is: what is the value of i?

Recall **partition** function, it is most optimal if array is divided into two equal parts.

Thus, we set **i = (j+k) / 2**

Problem 1

(b) Write the recurrence for the running time of the algorithm. Draw the corresponding recursion tree and find the explicit answer to the recursion.
Your answer must use $\Theta(.)$ notation, i.e., you should obtain both an upper bound and a lower bound (proof by strong induction is NOT needed).

```
3   Define i := .(j+k)/2
4   If A[i] = x . Return .i. .
5   If A[i] < x . Return FIND(A,i+1,k,x
6   Return FIND(A,j,j-1,x
```

Problem 1

(b) Write the recurrence for the running time of the algorithm. Draw the corresponding recursion tree and find the explicit answer to the recursion.
Your answer must use $\Theta(.)$ notation, i.e., you should obtain both an upper bound and a lower bound (proof by strong induction is NOT needed).

3 Define $i :=$.(j+k)/2
4 **If** $A[i] =$ x . **Return** .i. .
5 **If** $A[i] <$ x . **Return** FIND(A,i+1,k,x
6 **Return** FIND(A,j,j-1,x

Obviously the best-case T(n) = O(1). (e.g A[(n+1)/2] = x)

Usually we pay more attention to the worst-case running time.

Problem 1

(b) Write the recurrence for the running time of the algorithm. Draw the corresponding recursion tree and find the explicit answer to the recursion.
Your answer must use $\Theta(.)$ notation, i.e., you should obtain both an upper bound and a lower bound (proof by strong induction is NOT needed).

```
3   Define i := .(j+k)/2
4   If A[i] = x . Return .i. .
5   If A[i] < x . Return FIND(A,i+1,k,x
6   Return FIND(A,j,j-1,x
```

In the worst case, one recursion calls one sub-recursion.

The size of sub-recursion is half of that of recursion.

All other operations are \theta(1).

Problem 1

(b) Write the recurrence for the running time of the algorithm. Draw the corresponding recursion tree and find the explicit answer to the recursion.
Your answer must use $\Theta(.)$ notation, i.e., you should obtain both an upper bound and a lower bound (proof by strong induction is NOT needed).

```
3   Define i := .(j+k)/2
4   If A[i] = x . Return .i. .
5   If A[i] < x . Return FIND(A,i+1,k,x
6   Return FIND(A,j,j-1,x
```

Thus, in the worst case, the running time is:

$$T(n) = T(n/2) + \theta(1)$$

Problem 1

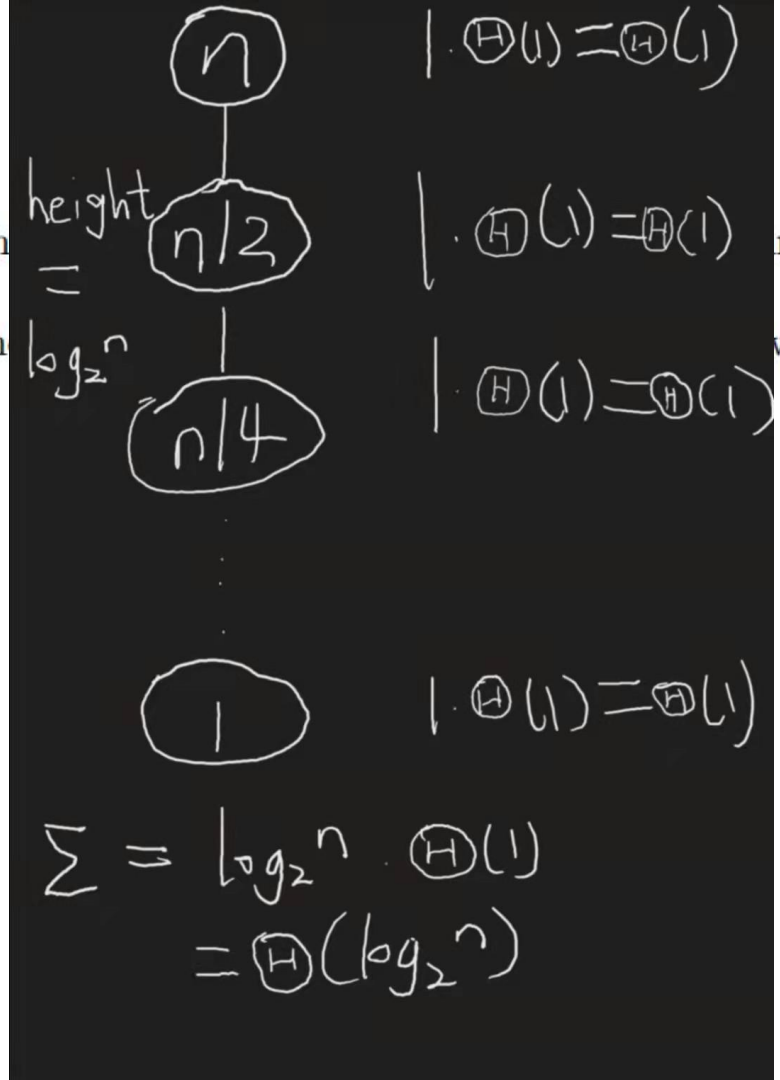
(b) Write the recurrence for the running time of th rsion tree and find the explicit answer to the recursion.
Your answer must use $\Theta(.)$ notation, i.e., you sh ver bound (proof by strong induction is NOT needed).

Worst case:

T(n) = T(n/2) + \theta(1)

Thus,

T(n) = \theta(log n)



Problem 1

(c) We assumed that the elements of A are distinct. Does the algorithm still work if the elements of A are not necessarily distinct? Justify your answer.

Problem 1

(c) We assumed that the elements of A are distinct. Does the algorithm still work if the elements of A are not necessarily distinct? Justify your answer.

It still works.

Even if there are duplicate elements, it still holds that

If A[i] < x, then **A[j…i] < x**. Thus, x could only possibly exist in **A[i+1,k]**.

if A[i] > x, then **A[i…k] > x**. Thus, x could only possibly exist in **A[j,i-1]**

Problem 2

Problem 2 (5+15+5 points)

Given an array $A[1, 2, \ldots, n]$, recall that the PARTITION function in QUICKSORT takes the last element $A[n]$ as the pivot and partitions A into two parts: the left part consisting of the elements which are less than or equal to the pivot and the right part consisting of the rest.

(a) Assume that the elements in A are all the same. What is the running time of QUICKSORT on A? Explain your answer.

You should be convinced by part (a) that the running time is not good! To overcome this issue, we modify the PARTITION function so that after running it, any input array A is partitioned into three parts: the left part consisting of all elements strictly less than the pivot, the middle part consisting of all elements equal to the pivot, and the right part consisting of the rest (i.e., all elements strictly larger than the pivot).

Problem 2

Recall QuickSort

```
Quick Sort ( A[1...n])
  k = Partition ( A[1...n] , pivot = n )
  Quick Sort ( A[1... k-1])
  Quick Sort ( A[k+1 ... n])
```

left subarray

right subarray

k := index of the pivot after the partition

Partition (A[1...n], pivot = n)

idea: build the left part iteratively

```
i = 1
```

i: the index of the next potential left element to be inserted at

n-1 iterations

```
for  j = 1  to  n-1
    if  A[j] ≤ A[n]        ↘ pivot
        swap ( A[i], A[j] )
        i = i+1
swap ( A[i], A[n] )        pivot
return i
```

each iteration takes O(1) time

j: the current element of the array to be decided

A[j] belongs to the left part

inserting A[j] at the end of the left part

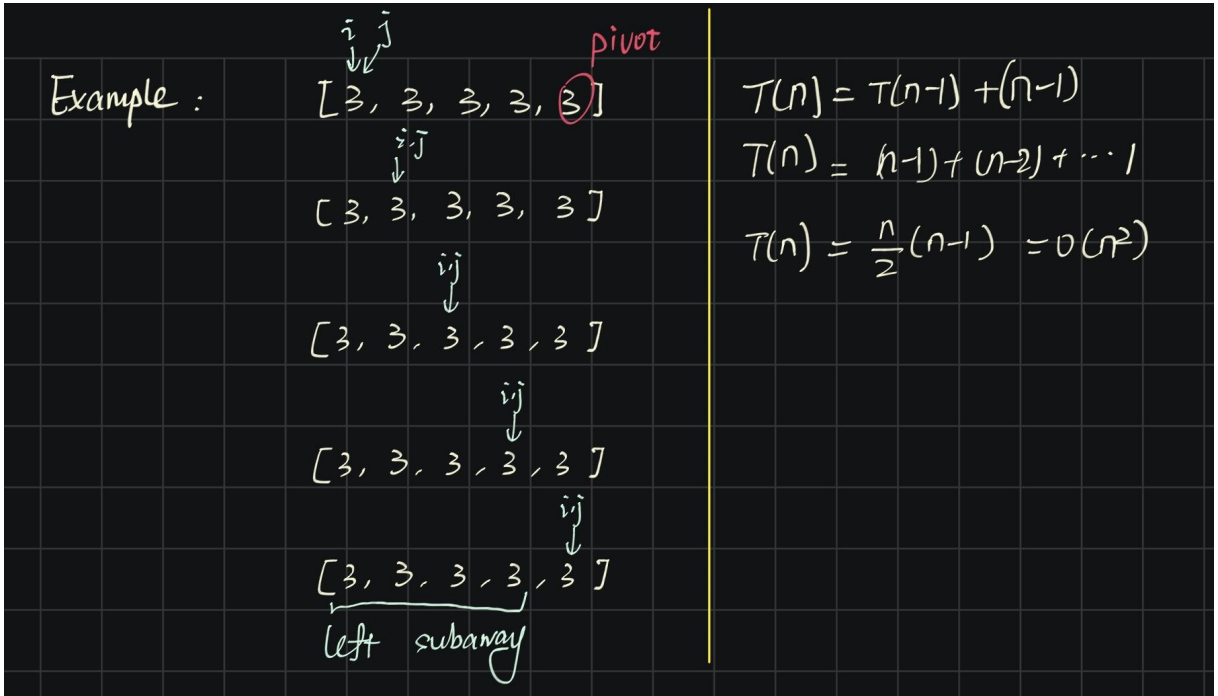
the index i is updated

placing the pivot at the end of the left part

return the index of the pivot

Problem 2

(a) Assume that the elements in A are all the same. What is the running time of QuickSort on A?

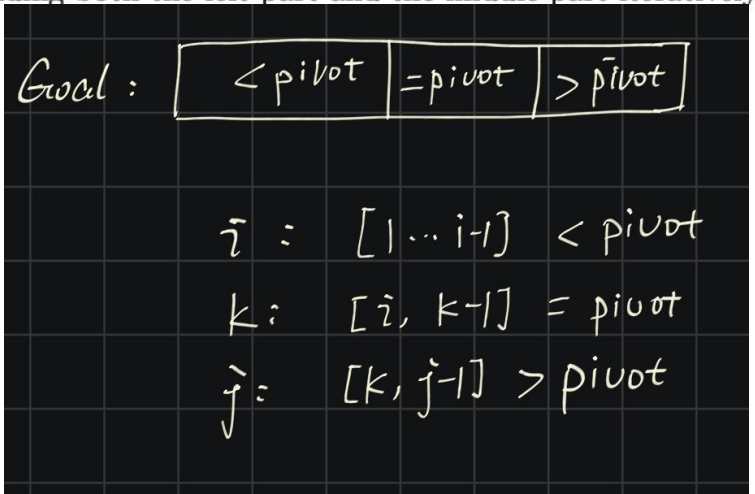


Problem 2

You should be convinced by part (a) that the running time is not good! To overcome this issue, we modify the PARTITION function so that after running it, any input array A is partitioned into three parts: the left part consisting of all elements strictly less than the pivot, the middle part consisting of all elements equal to the pivot, and the right part consisting of the rest (i.e., all elements strictly larger than the pivot).

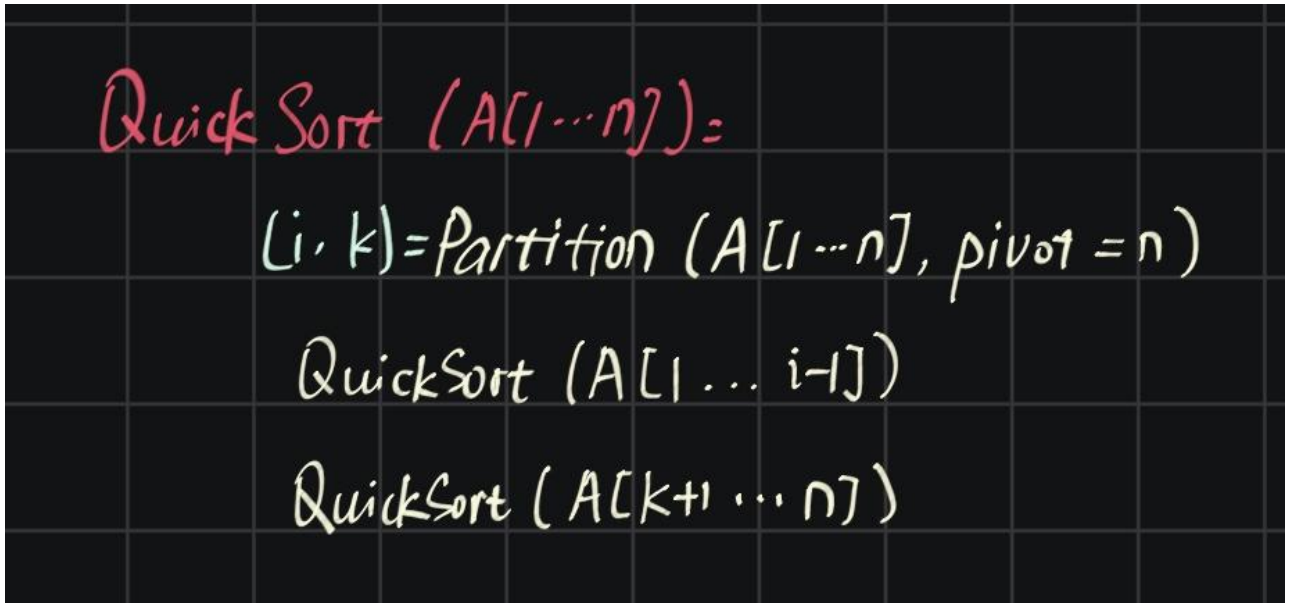
(b) Write down the pseudo-code for this modified version of the PARTITION function. Note that your algorithm must be an in-place algorithm (no auxiliary array should be used) with $\Theta(n)$ time complexity.

Hint: Define three indices: i as the end of the left part, k as the end of the middle part, and j denoting the current index which is used in the for-loop. Modify the PARTITION function discussed in the lecture for building both the left part and the middle part iteratively.



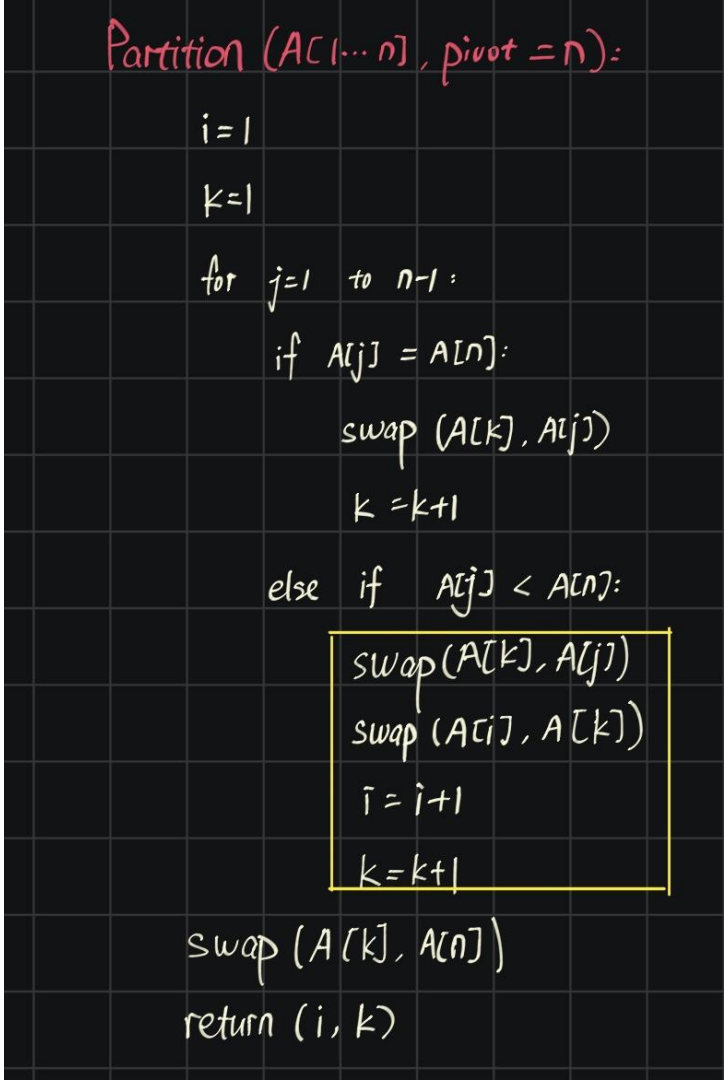
Problem 2

(b)


QuickSort (A[1...n]):
(i, k) = Partition (A[1...n], pivot = n)
QuickSort (A[1 ... i-1])
QuickSort (A[k+1 ... n])

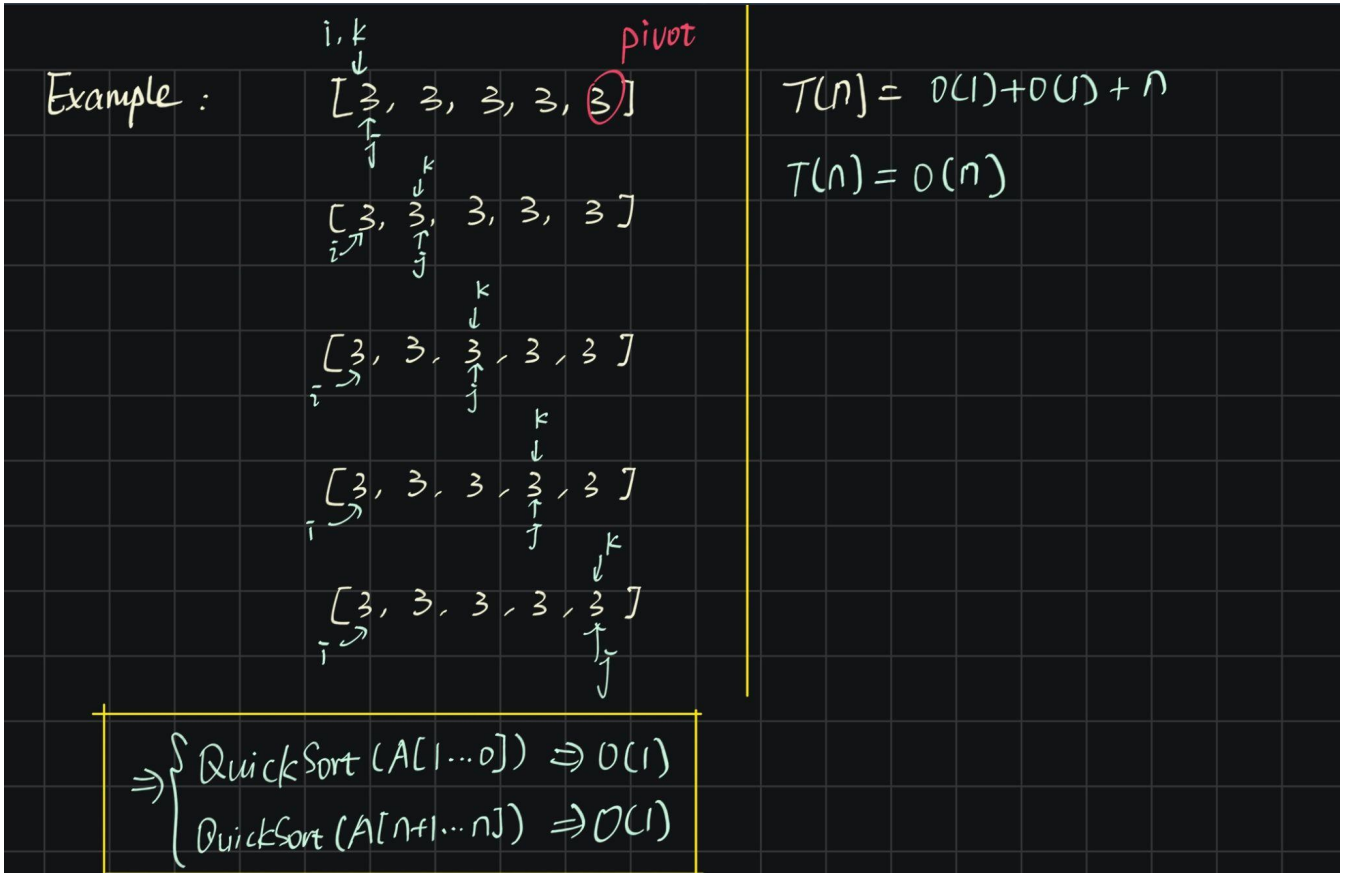
Problem 2

(b)


Partition (A[1⋯n], pivot = n):
i = 1
k = 1
for j = 1 to n-1 :
if A[j] = A[n]:
swap (A[k], A[j])
k = k+1
else if A[j] < A[n]:
swap (A[k], A[j])
swap (A[i], A[k])
i = i+1
k = k+1
swap (A[k], A[n])
return (i, k)

Problem 2

(c)


Example :
pivot
[3, 3, 3, 3, 3]
[3, 3, 3, 3, 3]
[3, 3, 3, 3, 3]
[3, 3, 3, 3, 3]
[3, 3, 3, 3, 3]
T(n) = O(1)+O(1)+n
T(n) = O(n)
⇒ { QuickSort (A[1…0]) ⇒ O(1)
QuickSort (A[n+1…n]) ⇒ O(1)

Problem 3

Problem 3 (20 points)

Let A be an array of n integers, where n is divisible by 4. Describe an algorithm with $O(n)$ time complexity to decide whether there exists an element that occurs more than $n/4$ times in A.

(a) Show that if such an element exists in A, it must be either the $(n/4)$-th, or the $(2n/4)$-th, or the $(3n/4)$-th smallest elements.

(b) Use part (a) to derive your algorithm.

Problem 3

(a) Show that if such an element exists in A, it must be either the $(n/4)$-th, or the $(2n/4)$-th, or the $(3n/4)$-th smallest elements.

Problem 3

(a) Show that if such an element exists in A, it must be either the $(n/4)$-th, or the $(2n/4)$-th, or the $(3n/4)$-th smallest elements.

We prove by **contradiction**:

If such element exists but neither of (n/4)-th, (2n/4)-th, (3n/4)-th smallest element is this element,

Problem 3

(a) Show that if such an element exists in A, it must be either the $(n/4)$-th, or the $(2n/4)$-th, or the $(3n/4)$-th smallest elements.

We prove by **contradiction**:

If such element exists but none of (n/4)-th, (2n/4)-th, (3n/4)-th smallest element is this element,

Then if we sort A, the index interval of this element must inside either [1,(n/4)-1], [(n/4)+1, (2n/4)-1], [(2n/4)+1, (3n/4)-1], [(3n/4)+1, n].

However, the length of four intervals are not larger than n/4.

Problem 3

(b) Use part (a) to derive your algorithm.

If such an element exists in A, it must be either the $(n/4)$-th, or the $(2n/4)$-th, or the $(3n/4)$-th smallest elements.

Problem 3

(b) Use part (a) to derive your algorithm.

If such an element exists in A, it must be either the $(n/4)$-th, or the $(2n/4)$-th, or the $(3n/4)$-th smallest elements.

Thus, we can verify three elements separately.

If one of the elements satisfy the condition, return **TRUE**

Problem 3

(b) Use part (a) to derive your algorithm.

Given array A is not guaranteed to be sorted, how to find the three elements?

Problem 3

(b) Use part (a) to derive your algorithm.

Given array A is not guaranteed to be sorted, how to find the three elements?

Recall the **Quick-Select** algorithm, which finds the k smallest element in O(n).

Problem 3

(b) Use part (a) to derive your algorithm.

Given array A is not guaranteed to be sorted, how to find the three elements?

Recall the **Quick-Select** algorithm, which finds the k smallest element in O(n).

For each element, we need a loop to verify.

Thus, the running time is 3*(O(n) + O(n)) = O(n)

Problem 3

(b) Use part (a) to derive your algorithm.

```
1  x = Quick_select(n/4)    // O(n), following same
2  if count(x) > n/4    // a loop, O(n), following same
3      return TRUE
4
5  x = Quick_select(2n/4)
6  if count(x) > n/4
7      return TRUE
8
9  x = Quick_select(3n/4)
10 if count(x) > n/4
11     return TRUE
12
13 return FALSE
```

Problem 4

Problem 4 (13 points)

Consider the recursion $T(n) = 2T(n/2) + n$ for $n \geq 2$, and $T(1) = 1$ that was solved in the lecture. What is the answer to $T(n)$?
Below, you are provided a reasoning showing that $T(n) = O(n)$. Of course, it is not correct (as seen in the lecture). Find the mistake and justify why the following reasoning is not correct. For simplicity, you may assume that both n and k (the variable used in the inductive step) are powers of 2.

We use strong induction to show that $T(n) = O(n)$, where n is a power of 2:

- Base case: One can easily check that the statement holds for the base case $n = 1$; We have $T(1) = 1$, thus, we get $T(1) = O(1)$.
- Inductive step:
  1. Assumption: Assume that the statement holds for $n = k/2$, i.e., we have $T(k/2) = O(k/2)$.
  2. Conclusion: Based on the assumption, we prove that the statement holds for $n = k$, i.e., we show that $T(k) = O(k)$:

$$
\begin{aligned}
T(k) &= 2T(k/2) + k && (1)\\
&= 2O(k/2) + k && (2)\\
&= 2O(k) + k && (3)\\
&= O(k) + k && (4)\\
&= O(k). && (5)
\end{aligned}
$$

For full credit, you have to provide a full justification for your arguments.

Problem 4

We use strong induction to show that $T(n) = O(n)$, where n is a power of 2:

- Base case: One can easily check that the statement holds for the base case $n = 1$; We have $T(1) = 1$, thus, we get $T(1) = O(1)$.
- Inductive step:
  1. Assumption: Assume that the statement holds for $n = k/2$, i.e., we have $T(k/2) = O(k/2)$.
  2. Conclusion: Based on the assumption, we prove that the statement holds for $n = k$, i.e., we show that $T(k) = O(k)$:

$$
\begin{aligned}
T(k) &= 2T(k/2) + k && (1)\\
&= 2O(k/2) + k && (2)\\
&= 2O(k) + k && (3)\\
&= O(k) + k && (4)\\
&= O(k). && (5)
\end{aligned}
$$

Problem 4

$T(n) = O(n) \Rightarrow T(n) \le c \cdot n \quad , c > 0$

Base Case:

$n = 2, \quad T(2) = 2T(1) + 2 = 4$

$4 \le c \cdot 2$

$\Rightarrow c \ge 2$

Inductive Step:

- Assume $T(k/2) = O(k/2)$

  $\Rightarrow T(k/2) \le c \cdot k/2 \quad , c > 0$

- prove $T(k) = O(k)$

  $\Rightarrow T(k) \le c \cdot k$

$T(k) = 2T(k/2) + k$

$\le 2 \cdot c \cdot k/2 + k$

$\le (c+1)k$

$\Rightarrow \boxed{T(k) \le (c+1)k}$

We cannot conclude $T(k) \le ck$

Problem 5

Problem 5 (20 points)

Given a positive integer n, we can easily compute 2^n by using $\Theta(n)$ multiplications. Explain an algorithm that computes 2^n using $\Theta(\log n)$ multiplications. Justify why your algorithm uses $\Theta(\log n)$ multiplications.

(a) Consider the case where n is a power of 2, i.e., $n = 2^k$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k} = \left(2^{2^{k-1}}\right)^2$.

(b) Consider the case where $n = 2^k + 2^{k-1}$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k+2^{k-1}} = 2^{2^k} \times 2^{2^{k-1}}$.

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

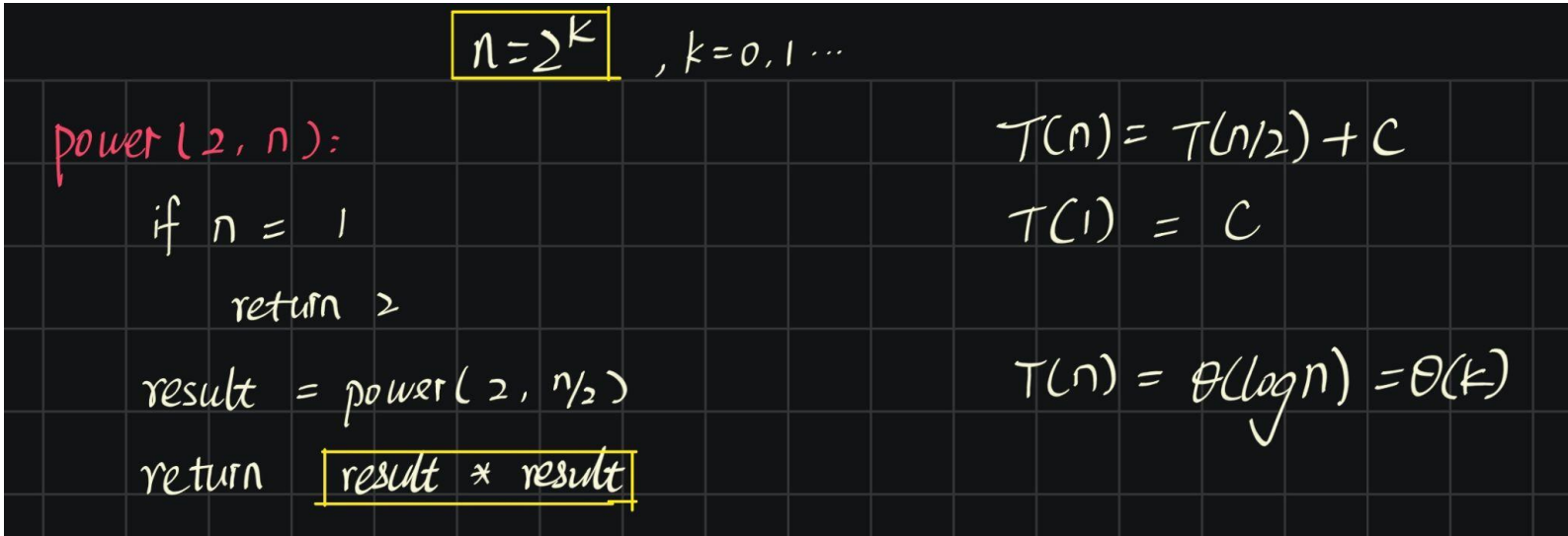
Problem 5

(a) Consider the case where n is a power of 2, i.e., $n = 2^k$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k} = \left(2^{2^{k-1}}\right)^2$.

Problem 5

(a) Consider the case where n is a power of 2, i.e., $n = 2^k$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k} = \left(2^{2^{k-1}}\right)^2$.

2^n = **2^(n/2) * 2^(n/2)**



Problem 5

(a) Consider the case where n is a power of 2, i.e., $n = 2^k$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k} = \left(2^{2^{k-1}}\right)^2$.

Define f(k) = **2^(2^k)**. Then f(0) = 2^(2^0) = 2

From the **Hint**, f(k) = [f(k-1)]^2

f(0) = 2^(2^0) = 2

f(k) = [f(k-1)]^2

Problem 5

(a) Consider the case where n is a power of 2, i.e., $n = 2^k$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k} = \left(2^{2^{k-1}}\right)^2$.

Define f(k) = **2^(2^k)**. Then f(0) = 2^(2^0) = 2

From the **Hint**, f(k) = [f(k-1)]^2

f(0) = 2^(2^0) = 2

f(k) = [f(k-1)]^2

```
1 compute(k):
2     result = 2
3     for i from 1 to k:
4         result = result * result
5     return result
```

Problem 5

(b) Consider the case where $n = 2^k + 2^{k-1}$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k+2^{k-1}} = 2^{2^k} \times 2^{2^{k-1}}$.

Problem 5

(b) Consider the case where $n = 2^k + 2^{k-1}$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k+2^{k-1}} = 2^{2^k} \times 2^{2^{k-1}}$.

Define f(k) = **2^(2^k)**. Then f(0) = 2^(2^0) = 2, f(k) = [f(k-1)]^2

The final result is

f(k) * f(k-1)

Problem 5

(b) Consider the case where $n = 2^k + 2^{k-1}$. Use $\Theta(k)$ multiplications to obtain 2^n.
Hint: Note that $2^{2^k+2^{k-1}} = 2^{2^k} \times 2^{2^{k-1}}$.

Define f(k) = **2^(2^k)**. Then f(0) = 2^(2^0) = 2, f(1) = 4

f(k) = [f(k-1)]^2

The final result is

f(k) * f(k-1)

```
1 compute(k):
2     result = 2
3     pre_result = 1
4     for i from 1 to k:
5         pre_result = result
6         result = result * result
7     return result * pre_result
```

Problem 5

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

Problem 5

(c) Generalize the ideas in integer n.

$$n = \begin{cases} \text{even}, \geq 2 \Rightarrow 2^n = 2^{n/2} \cdot 2^{n/2} \\ \text{odd}, \geq 3 \Rightarrow 2^n = 2 \cdot 2^{\frac{n-1}{2}} \cdot 2^{\frac{n-1}{2}} \end{cases}$$

```
power(2, n):
    if n = 0
        return 1
    if n = 1
        return 2
    if n % 2 = 0          // even
        result = power(2, n/2)
        return result * result
    else                  // odd
        result = power(2, (n-1)/2)
        return result * result * 2
```

$$T(n) = T(n/2) + C$$

$$T(1) = C$$

$$T(n) = \theta(\log n)$$

Problem 5

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

Part(a), n = 2^k, result is f(k)

Part(b), n = 2^k + 2^(k-1), result is f(k) * f(k-1)

Problem 5

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

Part(a), n = 2^k, result is f(k)

Part(b), n = 2^k + 2^(k-1), result is f(k) * f(k-1)

Recall: Any non-negative integer n can be decomposed into sum of power of 2.

What can we infer?

Problem 5

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

Generally, assume n = 2^x_1 + 2^x_2 + … + 2 ^ x_m.

Then the result is f(x_1) * f(x_2) * … * f(x_m).

Recall: f(k) = **2^(2^k)**

Problem 5

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

More generally, let n = t_0 * 2^0 + t_1 * 2^1 + … + t_m * 2^m.

Here, t_i = 0 or 1 for any i.

Thus, the result is the product of f(k) where t_k = 1

Problem 5

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

These t_i make up of the binary representation of n.

Example: 11 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0

Its binary representation is 1011

The result of 2^11 = f(3) * f(1) * f(0)

Problem 5

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

Thus, Compute each binary digit of n and multiply f(k) if the binary digit is 1.

At the same time, compute f(k) from f(k-1).

Problem 5

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

```
1  compute(n):
2      answer = 1                              // the result
3      power = 2                               // to compute f(k)
4      while(n > 0):
5          current_digit = n % 2               // get current lowest digit
6          if(current_dight == 1)              // current lowest digit is 1
7              answer = answer * power         // multiply f(k)
8          n = n / 2 (integer division)        // remove current lowset digit
9          power = power * power               // update f(k)
10     return answer
```

Problem 5

(c) Generalize the ideas in parts (a) and (b) to compute 2^n using $\Theta(\log n)$ multiplications for any positive integer n.

This algorithm is ‘Quick-Power’.

Bonus Problem 1

Bonus problem

The following problems are optional. They will NOT be graded and they will NOT appear on any of the exams. Work on them only if you are interested. They will help you to develop problem solving skills for your future endeavors.
You are highly encouraged to think on them and discuss them with your peers on a separate thread on the course page on Campuswire.

Bonus Problem 1

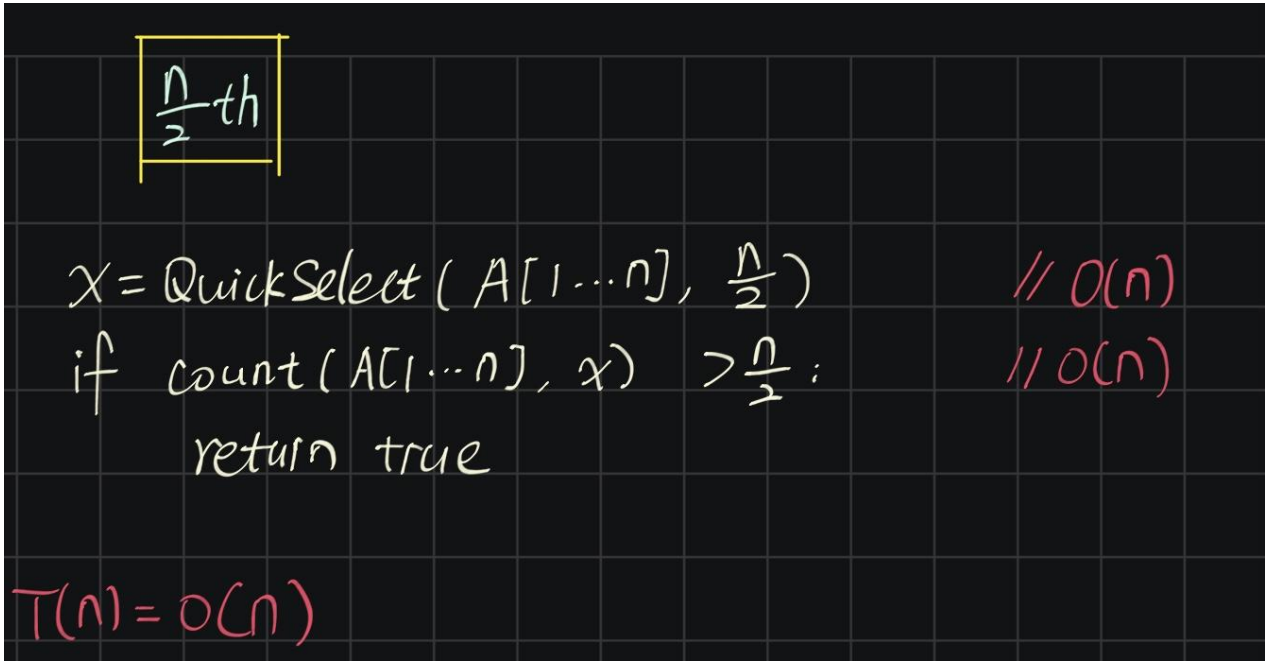
(a) Let A be an array of n integers, where n is divisible by 2. Use an idea similar to Problem 3 of this homework to develop an algorithm with $O(n)$ time complexity to decide whether there exists an element that occurs more than $n/2$ times in A.

(b) Develop the algorithm for part (a), i.e. deciding the existence of an element occurring more than $n/2$ times in A, but do not use the idea of Problem 3. In other words, your algorithm must NOT deal with the smallest k elements of the array. Your algorithm must run in $O(n)$ time with $O(1)$ space complexity.

(c) Generalize your idea in part (b) to solve Problem 3 of this homework, i.e. deciding the existence of an element occurring more than $n/4$ times in A, without using the smallest k elements of the array. Your algorithm must, again, run in $O(n)$ time with $O(1)$ space complexity.

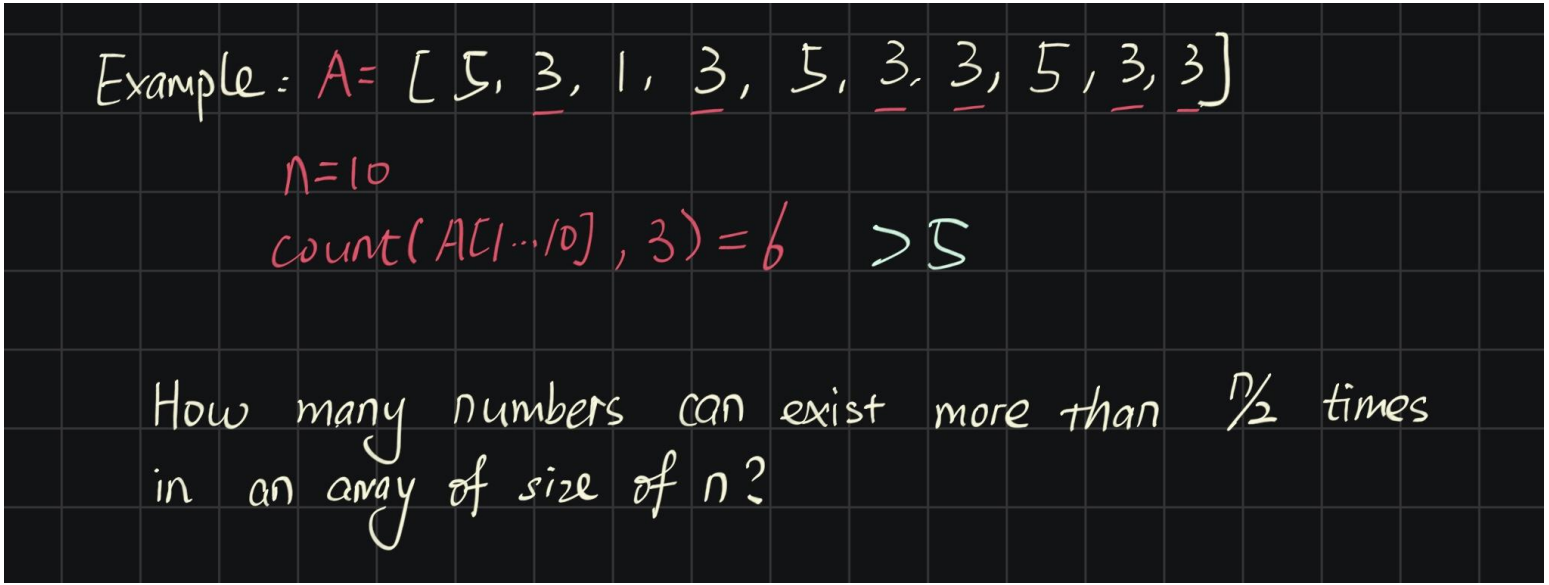
Bonus Problem 1

(a) Let A be an array of n integers, where n is divisible by 2. Use an idea similar to Problem 3 of this homework to develop an algorithm with O(n) time complexity to decide whether there exists an element that occurs more than n/2 times in A.



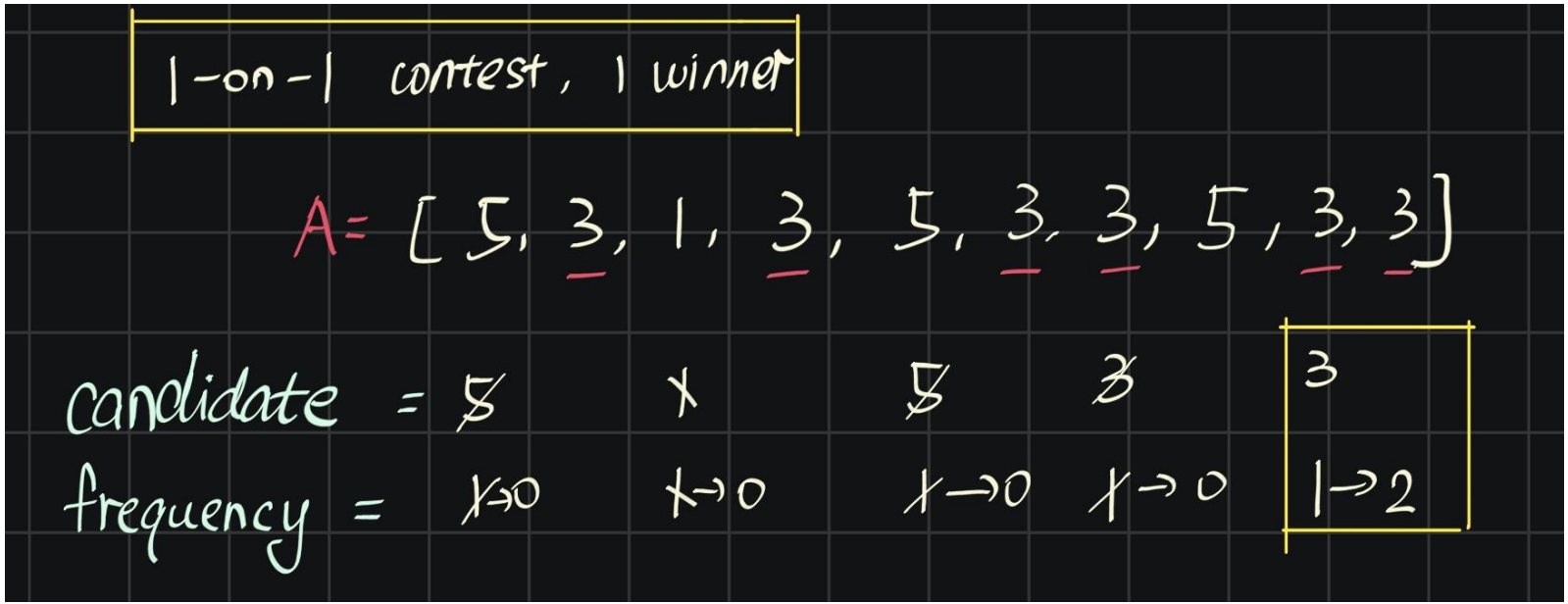
Bonus Problem 1

(b) Develop the algorithm for part (a), i.e. deciding the existence of an element occurring more than n/2 times in A, but do not use the idea of Problem 3. In other words, your algorithm must NOT deal with the smallest k elements of the array. Your algorithm must run in O(n) time with O(1) space complexity.



Bonus Problem 1

(b) Develop the algorithm for part (a), i.e. deciding the existence of an element occurring more than n/2 times in A, but do not use the idea of Problem 3. In other words, your algorithm must NOT deal with the smallest k elements of the array. Your algorithm must run in O(n) time with O(1) space complexity.



Bonus Problem 1

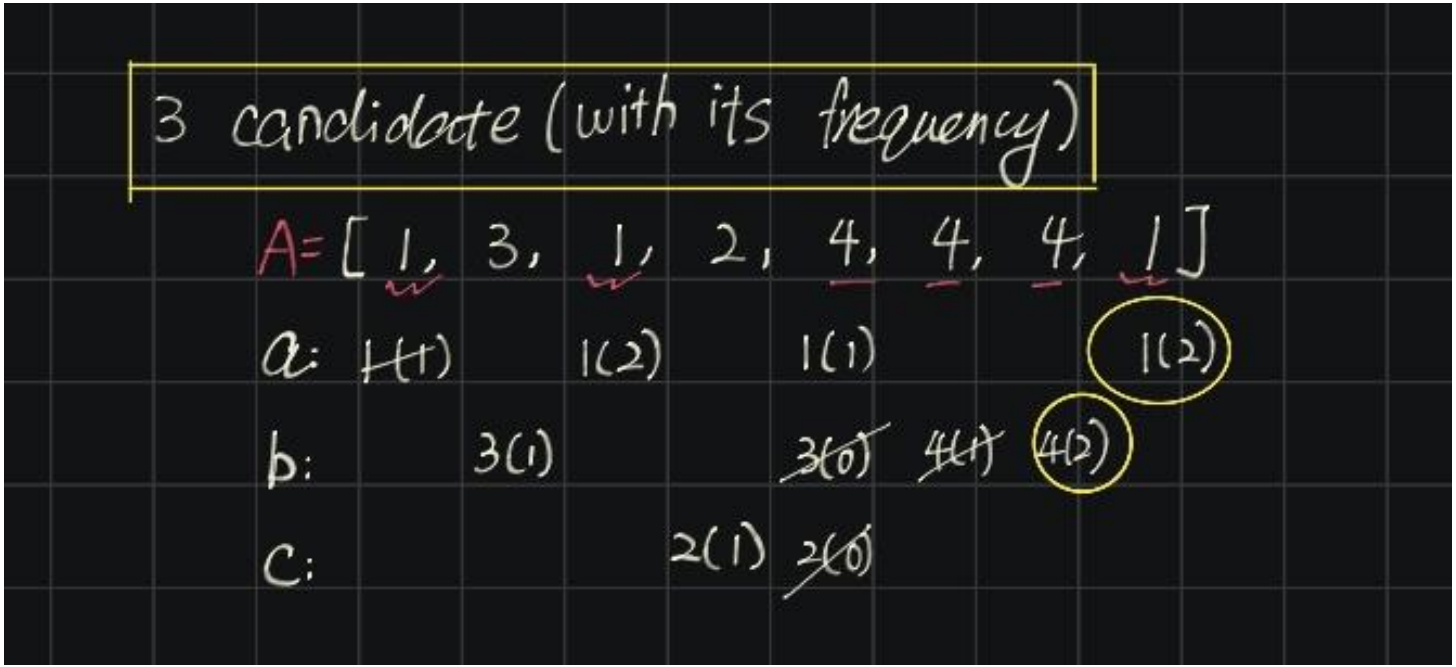
(c) Generalize your idea in part (b) to solve Problem 3 of this homework, i.e. deciding the existence of an element occurring more than n/4 times in A, without using the smallest k elements of the array. Your algorithm must, again, run in O(n) time with O(1) space complexity.

Think:

How many numbers can exist more than n/4 times in an array of size of n?

Bonus Problem 1

(c) Generalize your idea in part (b) to solve Problem 3 of this homework, i.e. deciding the existence of an element occurring more than n/4 times in A, without using the smallest k elements of the array. Your algorithm must, again, run in O(n) time with O(1) space complexity.



Q & A

Thank you