1. (★ 5')The special matrix

Let's define a special matrix as H_k, and these matrix satisfy the follow properties:

1. $H_0 = [1]$
2. For $k > 0$, H_k is a $2^k \times 2^k$ matrix.

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

(a) $H_k\vec{v} = \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} \vec{v}_1 \\ \vec{v}_2 \end{bmatrix}$

$= \begin{bmatrix} H_{k-1}\vec{v}_1 + H_{k-1}\vec{v}_2 \\ H_{k-1}\vec{v}_1 - H_{k-1}\vec{v}_2 \end{bmatrix} = \begin{bmatrix} H_{k-1}(\vec{v}_1 + \vec{v}_2) \\ H_{k-1}(\vec{v}_1 - \vec{v}_2) \end{bmatrix}$

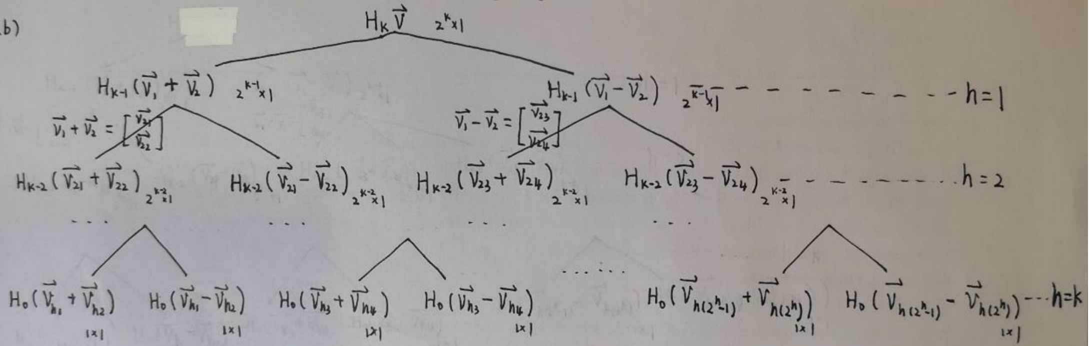
(a)Suppose that

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

is a column vector of length $n = 2^k$. v_1 and v_2 are the top and bottom half of the vector, respectively. Therefore, they are each vectors of length $\frac{n}{2} = 2^{k-1}$. Write the matrix-vector product $H_k v$ in terms of H_{k-1}, v_1, and v_2 (note that H_{k-1} is a matrix of dimension $\frac{n}{2} \times \frac{n}{2}$, or $2^{k-1} \times 2^{k-1}$). Since H_k is a $n \times n$ matrix, and v is a vector of length n, the result will be a vector of length n.

(b) Use your results from (a) to come up with a divide-and-conquer algorithm to calculate the matrix-vector product $H_k v$, and show that it can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time. **You do not need to use the four part proof.**

(b)



1. After calculating $(\vec{v}_1+\vec{v}_2)$ and $(\vec{v}_1-\vec{v}_2)$, we can get $H_k\vec{v}$ by computing $H_{k-1}(\vec{v}_1+\vec{v}_2)$ and $H_{k-1}(\vec{v}_1-\vec{v}_2)$ and recursively,
2. To get 1, we can just compute $H_{k-1}\vec{v}_1$ and $H_{k-1}\vec{v}_2$. ('+' and '-' — αn)

Then we have divided the problem into small issues, after that we need to conquer them, and the solution is $\begin{bmatrix} H_0(\vec{V}_{h_1}+\vec{V}_{h_2}) \\ H_0(\vec{V}_{h_1}-\vec{V}_{h_2}) \\ \vdots \\ H_0(\vec{V}_{h(2^h-1)}+\vec{V}_{h(2^h)}) \\ H_0(\vec{V}_{h(2^h-1)}-\vec{V}_{h(2^h)}) \end{bmatrix}$.

$T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n\log n)$ (Master theorem)

or we can view the division as a binary tree, and the height of it is $\log n$. We can get the running time $O(n\log n)$.

2. (★★★ 10')Majority Elements

An array $A[1 \ldots n]$ is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and if so to find that element. The elements of the array are not necessarily from some ordered domain like the integers, so there can be no comparisons of the form "is $A[i] > A[j]$?".(For example, sort is not allowed.) The elements are also not hashable, i.e., you are not allowed to use any form of sets or maps with constant time insertion and lookups. However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time. **Four part proof are required for each part below.**

(a) Show how to solve this problem in $O(n \log n)$ time.

(b) Can you give a linear time algorithm whose running time is $O(n)$? (You should not reuse the algorithm to answer part a)

(a) Main idea: Divide-and-conquer. Divide array A into two arrays: A_1 and A_2, each of size $\frac{n}{2}$:
If both A_1 and A_2 have majority elements and they are equal, then it is A's majority element;
If both A_1 and A_2 don't have a majority element, then A doesn't have a majority element;
If only A_1 or only A_2 has a majority element $A[i]$, we can compare it with every other element in A by using "is $A[i] = A[j]$", and count the number of it; And if the number is more than $\frac{n}{2}$, it is the majority element of A. Otherwise, A doesn't have a majority element.
If both A_1 and A_2 have majority elements and they are not equal, compare them with other elements in A in turn (the same way with above). If one of them occurs more than $\frac{n}{2}$ times, it is the majority element of A.

Pseudocode:

```
Algorithm  GetMajorityElement (A[1...n]
if ( n==1 ) then
   return A[1]
end if
k ← n/2
l ← GetMajorityElement ( A[1...k])
r ← GetMajorityElement( A[k+1...n])
if ( l and r don't exist) then
    return NoMajorityElement
else if ( l and r exist ) then
    if ( l==r) then
        return l
    else
        num_l ← the number that l occurs in A[1...n]
        num_r ← the number that r occurs in A[1...n]
        if ( num_l > k+1) then
            return num_l
        else if (num_r > k+1) then
            return num_r
        else
            return NoMajorityElement
        end if
    end if
else //only one of l and r exist
    num_m ← the number of l or r occurs in A[1...n]
    if ( num_m > k+1) then
        return num_m
    else
        return NoMajorityElement
    end if
end if
```

Proof of Correctness:
If m is a majority element of A, then it must be a majority element of one half of A, and recursively. (1) True before loop (2) true before ith iteration of the loop (3) True after the last iteration of the loop.

Running time analysis:
By divide-and-conquer, we need $O(n)$ comparisons.
Then $T(n) = 2T(\frac{n}{2}) + O(n)$
$\Rightarrow T(n) = O(n \log n)$

2(b) Main idea:

See two arbitrary elements of A as a pair, then we can get $\frac{n}{2}$ pairs. In the $\frac{n}{2}$ pairs, if the 2 elements are different, discard the pair. If they are the same, record the number. If A has odd number of elements, just ignore the last one. After that, check when the majority element is what we output, whether the number of it is more than $\frac{n}{2}$ because A may have no majority element.

Pseudocode:

```
Algorithm  GetMajorityElement2 (A[1...n])
---------------------------------------------
if (n==2) then
    if (A[1] == A[2]) then
        return A[1]
    else
        return NoMajorityElement
    end if
end if
create an array temp
i ← 1
while i < n do
    if (A[i] == A[i+1]) then
        Push A[i] into temp
    end if
    i ← i+2
end while
return GetMajorityElement2 (temp)
```

```
Algorithm  Check (A[1...n])
---------------------------------------------
m ← GetMajorityElement2 (A[1...n])
num ← the number that m occurs in A
if (num > n/2) then
    return m
else
    return NoMajorityElement
end if
```

Proof of Correctness:

Suppose A has a majority element m, then the number of m is greater than n/2, so that at least one pair is (m, m). Hence after GetMajorityElement2 (A[1...n]), at least m will be left in temp. If (first = m, second ≠ m), we remove first and second and one m is removed, while another is also removed, and m maintains majority. Recursively, we can get m and at last we need to check whether A has a majority element. If it has, then m must be the majority element.

Running time analysis:

When pair the elements in A, we divide it into two halves, and cut one half. The 2 functions is done in O(n) time. Then we can get $T(n) = T(\frac{n}{2}) + O(n) \Rightarrow T(n) = O(n)$

3. (★★★ 5')Find the missing integer

An array A of length N contains all the integers from 0 to N except one (in some random order). In this problem, we cannot access an entire integer in A with a single operation. The elements of A are represented in binary, and the only operation we can use to access them is "fetch the jth bit of $A[i]$". Using only this operation to access A, give an algorithm that determines the missing integer by looking at only $O(N)$ bits. (Note that there are $O(NlogN)$ bits total in A, so we can't even look at all the bits). Assume the numbers are in bit representation with leading 0s. **Four part proof is required.**

Main idea: (Not need to separate in operation, just count the numbers)

Fetch the least significant bits (LSB) of A[i]. Separate the N numbers into two sets, and one with LSB '0' while the other with LSB '1'. Discard the numbers in the larger set, and the missing number must be in the smaller set if it had not been missed. Hence the last bit of the missing number must be the LSB of the smaller set. Apply this algorithm recursively so that we can build the missing number from most right bit to the most left bit.

Pseudocode:

```
Algorithm   FindMissingInteger (A, m)      // m stands for the missing integer
  if (length[A]==0) then
     return m
  else
     B ← elements of A with LSB '0'
     C ← elements of B with LSB '1'
     if ( length[B] ≤ length[C] ) then
         B ← B with all the elements' LSB removed
         m ← m with 0 prepended to LSB
         return FindMissingInteger (B, m)
     else
         C ← C with all the elements' LSB removed
         m ← m with 1 prepended to LSB
         return FindMissingInteger (C, m)
     end if
  end if
```

Running time analysis:

Every time, the worst case is to cut the problem into half, and we need to create new arrays, which cost $O(N)$. Then we can get: $B(N) = B(\frac{N}{2}) + O(N)$

$\Rightarrow B(N) = O(N)$. Hence we just look at $O(N)$ bits (Master Theorem)

When creating arrays and counting the number of '0' and '1', we cost $O(N\log N)$ time. Everytime we divide the problem into halves and discard half of the numbers.

Then $T(N) = T(\frac{N}{2}) + O(N\log N)$

$\Rightarrow T(N) = O(N\log N)$.

Then the run time is $O(N\log N)$.

Proof of Correctness:

If N is odd, the number of LSB '0' and '1' should equal.

If N is even, the number of LSB '0' should be 1 more than '1'.

In general, number of 0 ≥ number of 1

When number of 0 < number of 1, we can conclude that the missing number m must have LSB with 0.

When number of 0 > number of 1, m cannot have LSB with 1 for in that case the number of 0 ≤ number of 1. Hence m has LSB with 1.

The method above can be applied to find all the bits of m from right to left. If we remove the LSB every time, then the bit before LSB would become the new LSB so that the method can be applied. When there are no bits to become LSB, the algorithm is finished.

4. (★★ 10') Median of Medians

The *Quickselect*(A, k) algorithm for finding the kth smallest element in an unsorted array A picks an arbitrary pivot, then partitions the array into three pieces: the elements less than the pivot, the elements equal to the pivot, and the elements that are greater than the pivot. It is then recursively called on the piece of the array that still contains the kth smallest element.

(a) Consider the array $A = [1, 2, ..., n]$ shuffled into some arbitrary order. What is the worst-case runtime of $Quickselect(A, \lfloor n/2 \rfloor)$ in terms of n? Construct the sequence of pivots which have the worst run-time.

(b) Let's define a new algorithm Better-Quickselect that deterministically picks a better pivot. This pivot-selection strategy is called 'Median of Medians', so that the worst-case runtime of $Better-Quickselect(A, k)$ is O(n).

Median of Medians

1. Group the array into $\lfloor n/5 \rfloor$ groups of 5 elements each (ignore any leftover elements)
2. Find the median of each group of 5 elements (as each group has a constant 5 elements, finding each individual median is O(1))
3. Create a new array with only the $\lfloor n/5 \rfloor$ medians, and find the true median of this array using Better-Quickselect.
4. Return this median as the chosen pivot

Let p be the chosen pivot. Show that for least $3n/10$ elements x we have that $p \geq x$, and that for at least $3n/10$ elements we have that $p \leq x$.

(c) Show that the worst-case runtime of Better-Quickselect(A. k) using the 'Median of Medians' strategy is $O(n)$. **Hint**: Using the Master theorem will likely not work here. Find a recurrence relation for $T(n)$, and try to use induction to show that $T(n) \leq c \cdot n$ for some $c > 0$

(a) $O(n^2)$

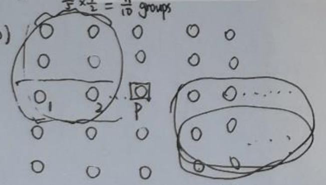
① first half of A is shuffled into the same order:

pivot: $1, 2, \cdots, n/2$

② first half of A is shuffled into the reverse order:

pivot: $n, n-1, \cdots, n/2+1$

(b) $\frac{n}{5} \times \frac{1}{2} = \frac{n}{10}$ groups



for all the medians before p, they are less or equal to p. And in each group of them, 2 elements are smaller or equal to the according median. As a result, for at least $\underbrace{\frac{n}{5} \times \frac{1}{2}}_{\text{number of groups before the group of p}} \times 3 = \frac{3}{10}n$ elements. we have that $p \geq x$.

The same method, at least $\frac{n}{5} \times \frac{1}{2}$ medians are larger or equal to p, and in their groups, there are 2 elements larger than the medians, so that they also $\geq p$. As a result, for at least $\frac{n}{5} \times \frac{1}{2} \times 3 = \frac{3}{10}n$ elements we have that $p \leq x$.

(c) in (b) we have moved $\frac{3}{10}n$ possible elements that must have that $x \leq p$, then use p as a pivot to partition: $T(\frac{7}{10}n)$

Group the array into $\lfloor n/5 \rfloor$ groups of 5 elements each: $T(\frac{n}{5})$

partition: n

$T(n) = T(\frac{n}{5}) + T(\frac{7}{10}n) + n$ $\quad T(n) = O(n)$

$\exists\, c > 0$. s.t. $T(n) \leq cn$

$T(n) \leq \frac{c}{5}n + \frac{7}{10}cn + n = \frac{9}{10}cn + n = (\frac{9}{10}c + 1)n$

$= cn + (1 - \frac{1}{10}c)n$

$\exists\, c \geq 10$. s.t. $T(n) \leq cn$

$\therefore T(n) \overset{?}{=} O(n)$.

5.(★★★★★ 10') Merged Median

Given k sorted arrays of length l, design a deterministic algorithm (i.e. an algorithm that uses no randomness) to find the median element of all the $n = kl$ elements. Your algorithm should run asymptotically faster than O(n). **Four part proof is required.**

Main idea:

Firstly, compare the first element of each array to get the minimum element, and compare the last element of each array to get the maximum element. Then we can use binary search on the numbers ranging from the minimum and the maximum (Notice that the order of the arrays is fixed) so that we can get the mid of the min and the max. Then we get the count of the elements less than the mid. Change the min or max. At last, for every number we count the elements less than it in each array, if it is less than the required count, the median must be larger than it. Otherwise, the median is less than or equal to it.

Proof of Correctness:

There must be $kl/2$ elements smaller than the median. By comparing the count of elements less than the specific element with the count needed, we can find the midian finally.

Running time analysis:

To find the count of elements smaller than mid: $O(\log l)$, for each array: k

Then $\Rightarrow$ $O(k \log l)$

$n = kl$ $\log l < l$

Hence the running time is $O(n)$

Pseudocode:

```
Algorithm  MergedMedian ( matrix[][size], A1, A2, ..., Ak, k, l )
Store the k sorted arrays into a k×l matrix
min ← matrix[1][1]            // Suppose the index start from 1
max ← matrix[k][l]
for i from 2 to k, do
    if ( matrix[i][1] < min ) then
        min ← matrix[i][0]
    end if
    if ( matrix[i][l] > max ) then
        max ← matrix[i][l]
    end if
end for
need ← (kl + 1) / 2
while ( min < max ) do
    mid = min + (max - min) / 2
    place = 0
    for i from 1 to r, do
        place ← place + count of elements smaller than mid
    end for
    if ( place < need ) then
        min ← mid + 1
    else
        max ← mid
    end if
end while
return min
```