Question:

- You are interested in analyzing some hard-to-obtain data from two separate databases:
- Each database contains n numerical values --so there are

  2n values total-- and you may assume that no two values are the same.

Question:

- You are interested in analyzing some hard-to-obtain data from two separate databases:
- Each database contains n numerical values --so there are

  2n values total-- and you may assume that no two values are the same.
- *determine the median of this set of 2n values.*

Question:

- You are interested in analyzing some hard-to-obtain data from two separate databases:
- Each database contains n numerical values --so there are

  2n values total-- and you may assume that no two values are the same.
- *determine the median of this set of 2n values.*
- Only way you can access these values is through queries to the databases: In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains.

Question:

- You are interested in analyzing some hard-to-obtain data from two separate databases:
- Each database contains n numerical values --so there are

  2n values total-- and you may assume that no two values are the same.
- *determine the median of this set of 2n values.*
- Only way you can access these values is through queries to the databases: In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains.
- Since queries are expensive, you would like to compute the median using as few queries as possible.
- *Give an algorithm that finds the median value using at most O(log n)*

  queries.

Answer :

- A(i), B(i): i^{th}smallest element of database A/B
- A(k), B(k): medians of database A/B (k = $\left\lceil \frac{1}{2}\, n \right\rceil$)

Answer :

- A(i), B(i): i^{th}smallest element of database A/B
- A(k), B(k): medians of database A/B (k = $\left\lceil \frac{1}{2}\ n \right\rceil$)

1.1 A(k) < B (k) $\rightarrow$

Answer :

- A(i), B(i): i^{th}smallest element of database A/B
- A(k), B(k): medians of database A/B (k = $\left\lceil \frac{1}{2}\, n \right\rceil$)

1.1 $A(k) < B(k) \rightarrow$ B (k) > A(1) …… A(k) (because $A(i) < A(k)$ for all $i < k$)

B(k) > B(1) …… B (k-1) (by definition)

Answer :

- A(i), B(i): i^{th}smallest element of database A/B
- A(k), B(k): medians of database A/B (k = $\left\lceil \frac{1}{2}\, n \right\rceil$)

1.1 A(k) < B (k) → B (k) > A(1) …… A(k) (because A(i) < A(k) for all i < k)

B(k) > B(1) …… B (k-1) (by definition)

→ in merged database C, B (k) > (at least!) C (1) …….. C(2k-1)

Answer :

- A(i), B(i): i^{th}smallest element of database A/B
- A(k), B(k): medians of database A/B (k = $\left\lceil \frac{1}{2}\, n \right\rceil$)

1.1 A(k) < B (k) → B (k) > A(1) …… A(k) (because A(i) < A(k) for all i < k)

B(k) > B(1) …… B (k-1) (by definition)

→ in merged database C, B (k) > (at least!) C (1) …….. C(2k-1)

→ $B(k) > C\ (n)$: median of merged db C ! Remenber $2k \geq n$

meaning: B (k) and other larger elements of B: B (k+1) …… B(n) are greater than median of the merged database => no need to look at that part of DB B !

Answer :

- A(i), B(i): i^{th} smallest element of database A/B
- A(k), B(k): medians of database A/B (k = $\left\lceil \frac{1}{2}\, n \right\rceil$)

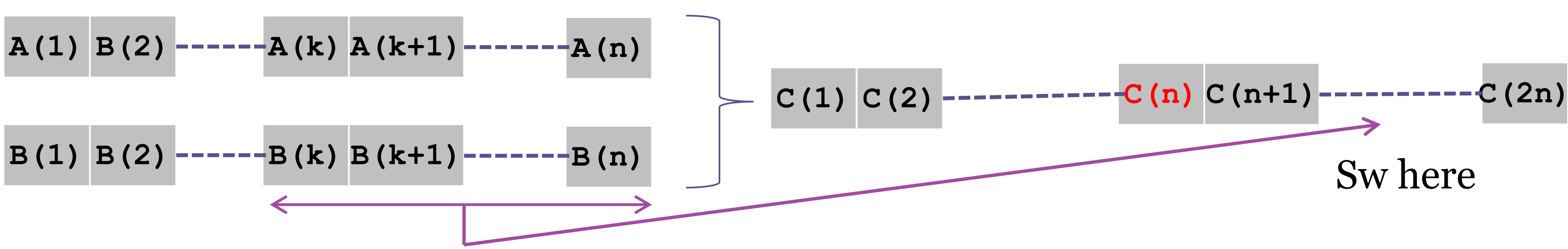
1.1 A(k) < B (k) → B (k) > A(1) …… A(k) (because A(i) < A(k) for all i < k)

B(k) > B(1) …… B (k-1) (by definition)

→ in merged database C, B (k) > (at least!) C (1) …….. C(2k-1)

→ $B(k) > C\ (n)$: median of merged db C ! Remenber $2k \geq n$

meaning: B (k) and other larger elements of B: B (k+1) …… B(n) are greater than median of the merged database => no need to look at that part of DB B !



Answer :

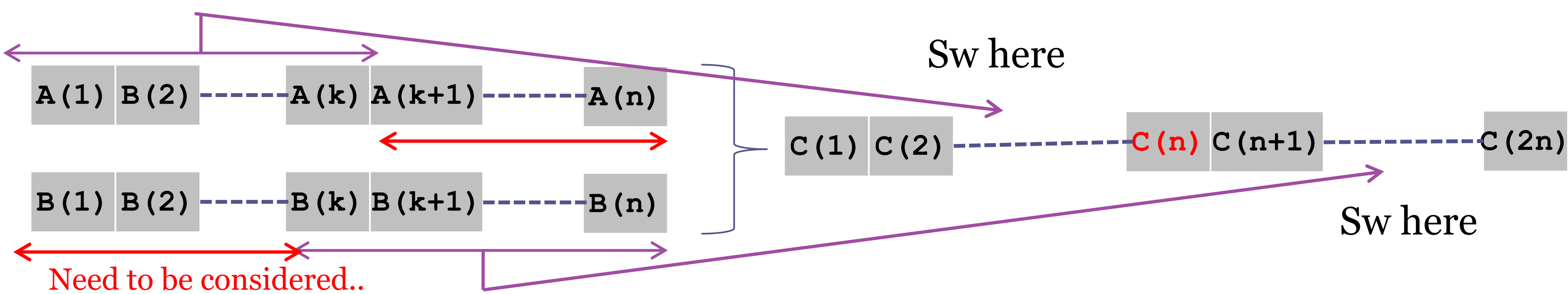
1.2 A(k) < B (k) → A(1) …… A(k) < A (k+1)…… A (n) (because A(i) < A(k) for all i < k)

A(1) …… A(k) < B (k+1) ….. B (n)

→ in merged database C, A(1)……(k) block < at least! (n – k -1 + 1) + $\left\lceil \frac{1}{2}\, n \right\rceil$ = n + 1 , elements!

→ $A(1)\ \ldots\ldots A(k) < C\ (n)$: median of merged db C !

meaning: A (k) and other smaller elements of A: A (1) …… A(k-1) are smaller than median of the merged database => no need to look at that part of DB A !



Answer :

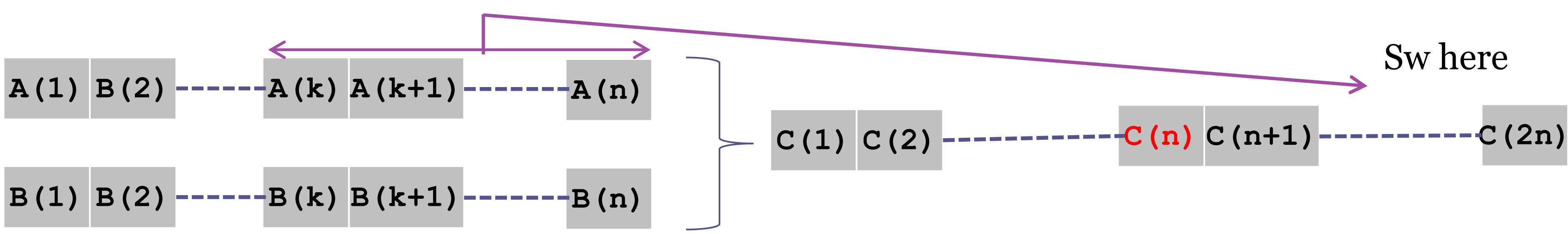
2.1 A(k) > B (k) → A (k) > B(1) …… B(k) (because B(i) < B(k) for all i < k)

A(k) > A(1) …… A(k-1) (by definition)

→ in merged database C, A (k) > (at least!) C (1) …….. C(2k-1)

→ $A(k) > C\ (n)$: median of merged db C !

meaning: A (k) and other larger elements of A: (A (k+1) …… A(n) are greater than median of the merged database => no need to look at that part of DB A !



Answer :

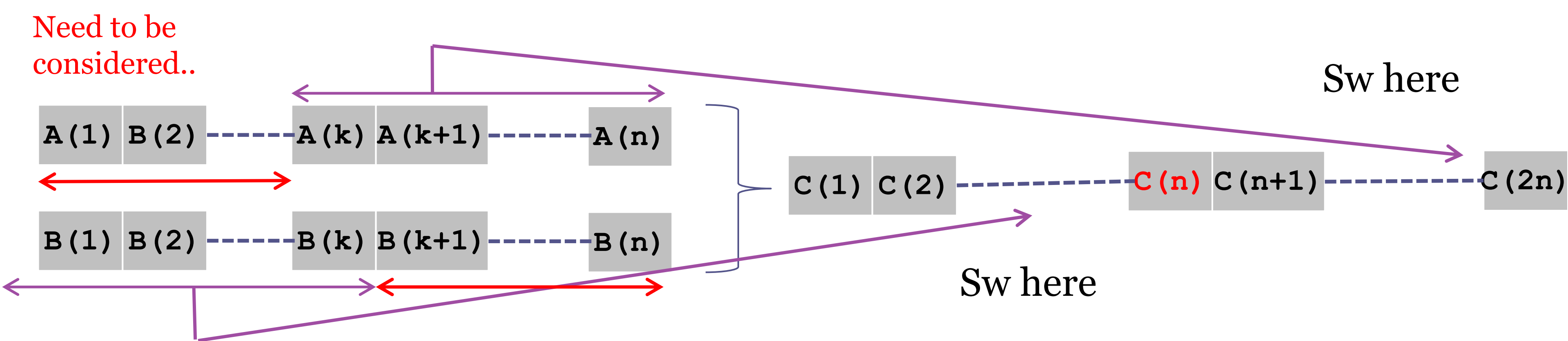
2.2 B(k) < A (k) → B(1) …… B(k) < B (k+1)…… B (n) (because B(i) < B(k) for all i < k)

B(1) …… B(k) < A (k+1) ….. A (n)

→ in merged database C, B(1)……B(k) block < at least! (n – k -1 + 1) + $\left\lceil \frac{1}{2} n \right\rceil$ = n + 1 , elements!

→ $B(1)\ \ldots\ldots B(k) < C\ (n)$: median of merged db C !

meaning: B (k) and other smaller elements of B: B (1) …… B(k-) are smaller than median of the merged database => no need to look at that part of DB B !



Answer :

∴ remaining part to be considered: half of the original DBs!

Algorithm?

```
median (n, a, b)
    if (n = 1)
            return min ( A(a+k), B(b+k) )    // base case..
    k = ⌈½ n⌉
    if ( A (a+k) < B (b+k) )
            return median ( k, a + ⌊½ n⌋, b )
    else
            return median ( k, a, b + ⌊½ n⌋ )
```

Call? median (n,0,0)

Complexity?

Answer :

∴ remaining part to be considered: half of the original DBs!

Algorithm?

median (n, a, b)
    if (n = 1)
        return min (A(a+k), B(b+k)) // base case..
    k = $\left\lceil \frac{1}{2} n \right\rceil$
    if (A (a+k) < B (b+k))
        return median (k, a + $\left\lfloor \frac{1}{2} n \right\rfloor$, b)
    else
        return median (k, a, b + $\left\lfloor \frac{1}{2} n \right\rfloor$)

Call? median (n,0,0)

Complexity? $Q(n) = Q(\left\lceil \frac{1}{2} n \right\rceil + 2)$ => $Q(n) = 2 \lceil \log n \rceil$

Question:

- Suppose you’re consulting for a bank about fraud detection,
- They have a collection of n bank cards suspecting of being used in fraud.
- Each card containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards

Question:

- Suppose you're consulting for a bank about fraud detection,
- They have a collection of n bank cards suspecting of being used in fraud.
- Each card containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards
- 2 bank cards are equivalent if they correspond to the same account.
- Bank has a high-tech "equivalence tester" that takes two bank cards and, determines whether they are equivalent.

Question:

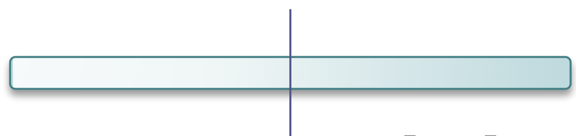
- Suppose you’re consulting for a bank about fraud detection,
- They have a collection of *n* bank cards suspecting of being used in fraud.
- Each card containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards
- 2 bank cards are equivalent if they correspond to the same account.
- Bank has a high-tech "equivalence tester" that takes two bank cards and, determines whether they are equivalent.
- QUESTION: the collection of n cards, is there a set of more than n/2 of them that are all equivalent to one another?
  - Only feasible operation: equivalence tester.
  - Show how to decide the answer to their question with only O(n log n) invocations of the equivalence tester.

Answer:

- *Equivalance classes:* e_1 …… e_n *(cards are equivalant if their classes are same:* $e_{i\ =}\ e_j$ *)*
- *Question:* whether there exists any equivalance class with more than n/2 members : i.e.: more than n/2 cards have $e_{i\ =}$ x.

Answer:

- *Equivalance classes:* e_1 …… e_n *(cards are equivalant if their classes are same: $e_{i} = e_{j}$)*
- *Question:* whether there exists any equivalance class with more than n/2 members : i.e.: more than n/2 cards have $e_{i} =$ x.
- Division?

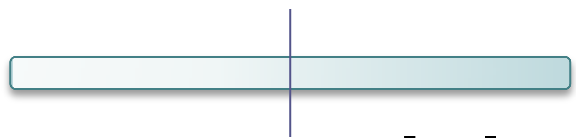


$\left\lfloor \frac{1}{2} n \right\rfloor$ cards $\left\lceil \frac{1}{2} n \right\rceil$cards

- Look for? Equivalance class containing more than half of currently examined cards ….
- Base case: 2 cards …

Answer:

- *Equivalance classes:* e_1 …… e_n *(cards are equivalant if their classes are same:* $e_{i} = e_{j}$ *)*
- *Question:* whether there exists any equivalance class with more than n/2 members : i.e.: more than n/2 cards have $e_i = \text{x}$.
- Division?



$\left\lfloor \frac{1}{2}\, n \right\rfloor$ cards $\left\lceil \frac{1}{2}\, n \right\rceil$cards

- Look for? Equivalance class containing more than half of currently examined cards ….
- Base case: 2 cards …
- OBSERVATION: If we have more than n/2 cards of same equivalance class –x in whole set. Then at least one of the halves will have more than half of its cards equivalant to x.
- BUT: reverse is not always true => test other cards …

Algorithm:

- Running time??

İf |S| = 1 return one card

İf |S| = 2

test if equivalent

return either card if equivalant

S_1 : first half containing $\left\lfloor \frac{1}{2} \text{n} \right\rfloor$ card$s$

S_2 : second half containing remaining cards

Call algorithm recursively for S_1

If a card is returned:

test against all other cards

If no card with majority equivalance has yet been found

then call algorithm recursively for S_2

If a card is returned:

test against all other cards

Return a card from majority equivalance class if one is found

Algorithm:

- Running time??
- 2 recursive calls,
- At most 2n tests

outside recursive calls.

$T(n) \le 2T(n/2) + 2n$

⬇

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

İf $|S| = 1$ return one card

İf $|S| = 2$

test if equivalent

return either card if equivalant

S_1 : first half containing $\left\lfloor \frac{1}{2} n \right\rfloor$ cards

S_2 : second half containing remaining cards

Call algorithm recursively for S_1

If a card is returned:

test against all other cards

If no card with majority equivalance has yet been found

then call algorithm recursively for S_2

If a card is returned:

test against all other cards

Return a card from majority equivalance class if one is found

Question:

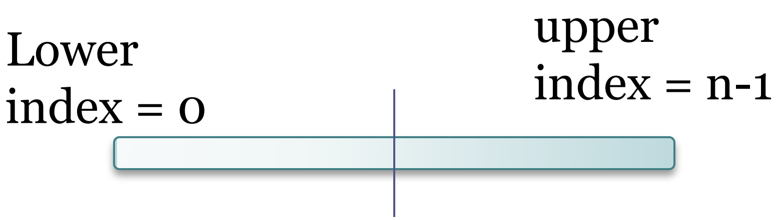
- A: positive or negative integers of size n,

    where A[1] < A[2] < A[3]…< A[n]

- Write an algorithm to find an i such that A[i] = i provided that such i exists
- Make sure that its complexity is not O(n) !

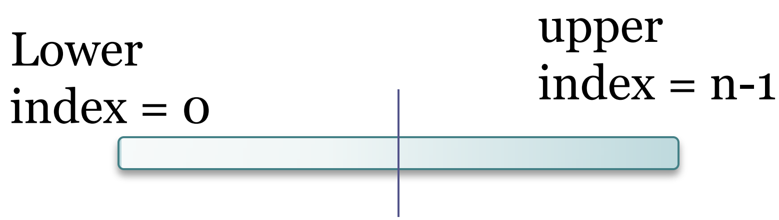
Answer:

- Division?
- Again from middle,

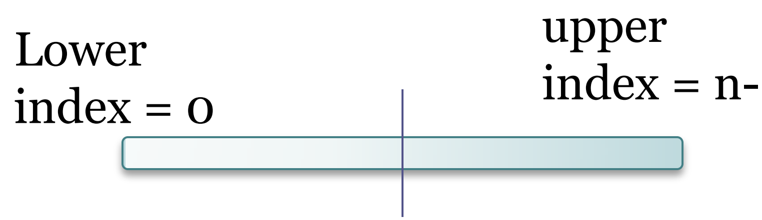


Answer:

- Division?
- Again from middle,
- If A[i] < i =>



Answer:



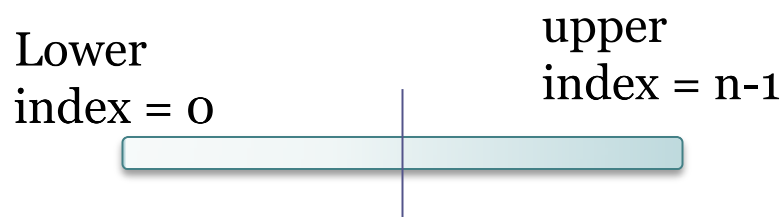
- Division?
- Again from middle,
- If A[i] < i =>

  since A is sorted, all A[j] < j for j < i

  why? Need to subtract at least 1 from i (value) for each previous element, than its index (exactly one less than the latter one) still bigger than its value...= > search for i in right portion!

- If A[i] > i =>

Answer:



- Division?
- Again from middle,
- If A[i] < i =>

  since A is sorted, all A[j] < j for j < i

  why? Need to subtract at least 1 from i (value) for each previous element, than its index (exactly one less than the latter one) still bigger than its value…= > search for i in right portion!

- If A[i] > i =>

  since A is sorted, all A[j] > j for j > i

  why? Need to add 1 to each index for each next element, also at least 1 bigger because A is sorted, than its index is still smaller than its value…= > search for i in left portion!

Answer:

- Eample
- If A[i] < i =>

A_o	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
-1	0	1	2	3	5	7	9	10

search for i in right portion

Answer:

- Eample
- If A[i] < i =>

A_o	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
-1	0	1	2	3	5	7	9	10

search for i in right portion

A_5	A_6	A_7	A_8
5	7	9	10

- If A[i] > i =>

Answer:

- Eample
- If A[i] < i =>

A_o	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
-1	0	1	2	3	5	7	9	10

search for i in right portion

A_5	A_6	A_7	A_8
5	7	9	10

- If A[i] > i =>

A_5	A_6	A_7	A_8
5	7	9	10

search for i in left portion, found ..

Algorithm:

- Running time??

```
lower ← 0
upper  ← n
notFound  ← true

while notFound
        i ← ⌈(lower+upper)/2⌉
        if  A[i] = i
                notFound ← false
        else
                if A[i] < i
                                lower  ← i + 1
                else
                                upper  ← i − 1
                endif
        endif
endwhile
print (i)
```

Algorithm:

- Running time??
- At each iteration,

divide current array by 2

And continue with one of them ..!

$\Downarrow$

T(n) = O (logn)

```
lower ← 0
upper ← n
notFound ← true

while notFound
        i ← ⌈(lower+upper)/2⌉
        if A[i] = i
                notFound ← false
        else
                if A[i] < i
                        lower ← i + 1
                else
                        upper ← i − 1
                endif
        endif
endwhile
print (i)
```