Concrete Models and Tight Upper and Lower Bounds

Elaine Shi

Concrete Models and Tight Upper and Lower Bounds

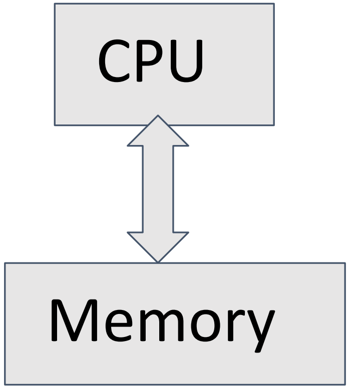
Why do we care about lower bounds?

Concrete Models and Tight Upper and Lower Bounds

What is “concrete models?”

Models of Computation

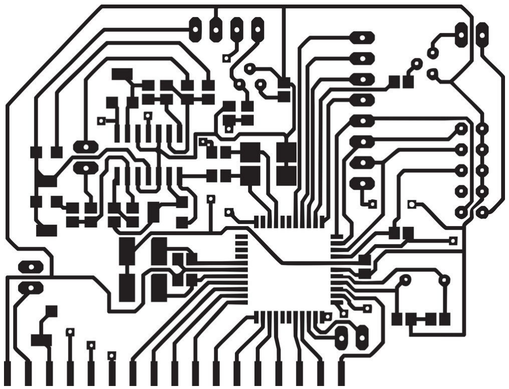
Random Access Machines (RAM)



most software today

Metrics: time, space

Circuits/Circuitry



hardware, cryptography

size, depth

Today: Concrete models of computation

focus on the cost of **a specific type of operation**

Today: Concrete models of computation

focus on the cost of **a specific type of operation**

- e.g., number of comparisons to sort an array? (recall last lecture)
- Number of probes into a graph needed to determine if the graph is connected?

Today: Concrete models of computation

focus on the cost of **a specific type of operation**

- e.g., number of comparisons to sort an array?

Recall last lecture:

- we used **# comparisons** as the cost metric

Today: Concrete models of computation

focus on the cost of **a specific type of operation**

- e.g., number of comparisons to sort an array?

Recall last lecture:

- we used **# comparisons** as the cost metric
- all ***asymptotic*** analysis also holds for the RAM model

Today: Concrete models of computation

focus on the cost of **a specific type of operation**

- e.g., number of comparisons to sort an array?

Recall last lecture:

- we used **# comparisons** as the cost metric
- all ***asymptotic*** analysis also holds for the RAM model
- **fun fact:** possible to compute median in a linear-sized circuit, but much more challenging [Lin-Shi, SODA'22]

Today: Concrete models of computation

focus on the cost of **a specific type of operation**

- don’t care whether the algorithm is actually implemented as a RAM program or circuit
- don’t care about other costs (e.g., space, cost of moving data round, etc)

Why do we care about costs in concrete models?

- Understand the limit of a class of algorithms
  - e.g., comparison-based sorting
- Understand information theoretic limits
  - e.g., number of probes into a graph to decide connectivity
- Focus on **heavy-weight** operations
  - e.g., in crypto, public-key ops > secret-key ops

Formal Model for Capturing Cost

- An upper bound of f(n) means the algorithm takes at most f(n) operations on **any input** of size n

- A lower bound of g(n) means for any algorithm there **exists an input** for which the algorithm takes at least g(n) operations on that input

Sorting in the Comparison Model

- In the comparison model, we have n items in some initial order
  An algorithm may compare two items (asking is $a_i > a_j$?) at a cost of 1
  - Moving the items is free

Sorting in the Comparison Model

- In the comparison model, we have n items in some initial order

  An algorithm may compare two items (asking is $a_i > a_j$?) at a cost of 1
  - Moving the items is free

- No other operations allowed, such as XORing, hashing, etc.

Sorting in the Comparison Model

- In the comparison model, we have n items in some initial order

  An algorithm may compare two items (asking is $a_i > a_j$?) at a cost of 1
  - Moving the items is free

- No other operations allowed, such as XORing, hashing, etc.

- Sorting: given an array a = $[a_1, \ldots, a_n]$, output a permutation π so that $[a_{\pi(1)}, \ldots, a_{\pi(n)}]$ in which the elements are in increasing order

Want $a_{\pi(1)}, a_{\pi(2)} \cdots a_{\pi(n)}$ to be sorted

Example for sorting

Array = [4, 6, 3, 1, 2] 4

π: 4, 5, 3, 1, 2

What is π?

- Sorting: given an array a = $[a_1, \ldots, a_n]$, output a permutation π so that $[a_{\pi(1)}, \ldots, a_{\pi(n)}]$ in which the elements are in increasing order

Sorting Lower Bound

- Theorem: Any deterministic comparison-based sorting algorithm must perform at least $\lg_2(n!)$ comparisons to sort n elements in the worst case

$$\lg_2(n!) = \lg(n \cdot (n-1)(n-2) \cdots 1) = \lg n + \lg(n-1) \cdots + \lg 1$$

$$= \Theta(n \lg n)$$

Sorting Lower Bound

- Theorem: Any deterministic comparison-based sorting algorithm must perform at least $\lg_2(n!)$ comparisons to sort n elements in the worst case

- I.e., for any sorting algorithm A and $n \geq 2$, there is an input I of size n so that A makes $\geq \lg(n!) = \Omega(n \log n)$ comparisons to sort I.

Sorting Lower Bound

- Without loss of generality, we may assume that the input contains numbers in [1, n], and all numbers are distinct
  - How many possible inputs are there?

$n!$

Sorting Lower Bound

- Without loss of generality, we may assume that the input contains numbers in [1, n], and all numbers are distinct
  - How many possible inputs are there?

Sorting Lower Bound

- Without loss of generality, we may assume that the input contains numbers in [1, n], and all numbers are distinct

Proving the Sorting Lower Bound

Think of an algorithm as a **decision tree**

```
procedure InsertionSort(A: list of items)
    n = length(A)
    for i = 2 to n do
        j = i
        while j > 1 and A[j-1] > A[j] do
            swap(A[j], A[j-1])
            j = j - 1
        end while
    end for
end procedure
```

Example: A = [3, 2, 1]

$a_1 \quad a_2 \quad a_3$

Example: A = [3, 2, 1]

a1 = 3, a2 = 2, a3 = 1 initial

cmp(a2, a1)

a2 = 2, a1 = 3, a3 = 1

cmp(a3, a1)

a2 = 2, a3 = 1, a1 = 3

cmp(a3, a2)

a3 = 1, a2 = 2, a1 = 3

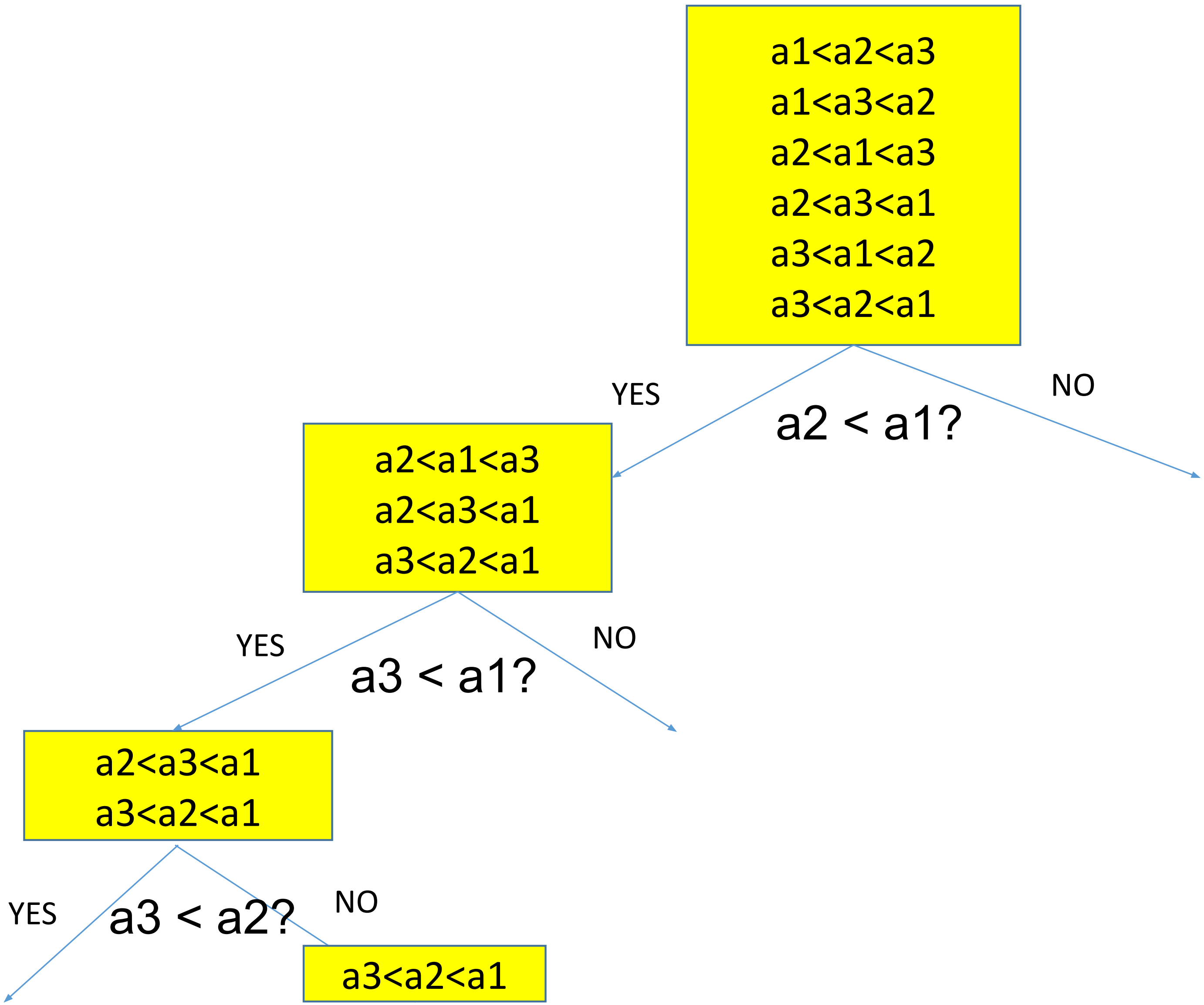
$a_1 < a_2 < a_3$
$a_1 < a_3 < a_2$
$a_2 < a_1 < a_3$
$a_2 < a_3 < a_1$
$a_3 < a_1 < a_2$
$a_3 < a_2 < a_1$

$a_2 < a_1$ cmp(a_2, a_1)

$a_2 < a_1 < a_3$
$a_2 < a_3 < a_1$
$a_3 < a_2 < a_1$

$a_3 < a_1$ cmp(a_3, a_1)

$a_2 < a_3 < a_1$
$a_3 < a_2 < a_1$


a1<a2<a3
a1<a3<a2
a2<a1<a3
a2<a3<a1
a3<a1<a2
a3<a2<a1
YES
a2 < a1?
NO
a2<a1<a3
a2<a3<a1
a3<a2<a1
YES
a3 < a1?
NO
a2<a3<a1
a3<a2<a1
YES
a3 < a2?
NO
a3<a2<a1

Example: $A = [1, 2, 3]$

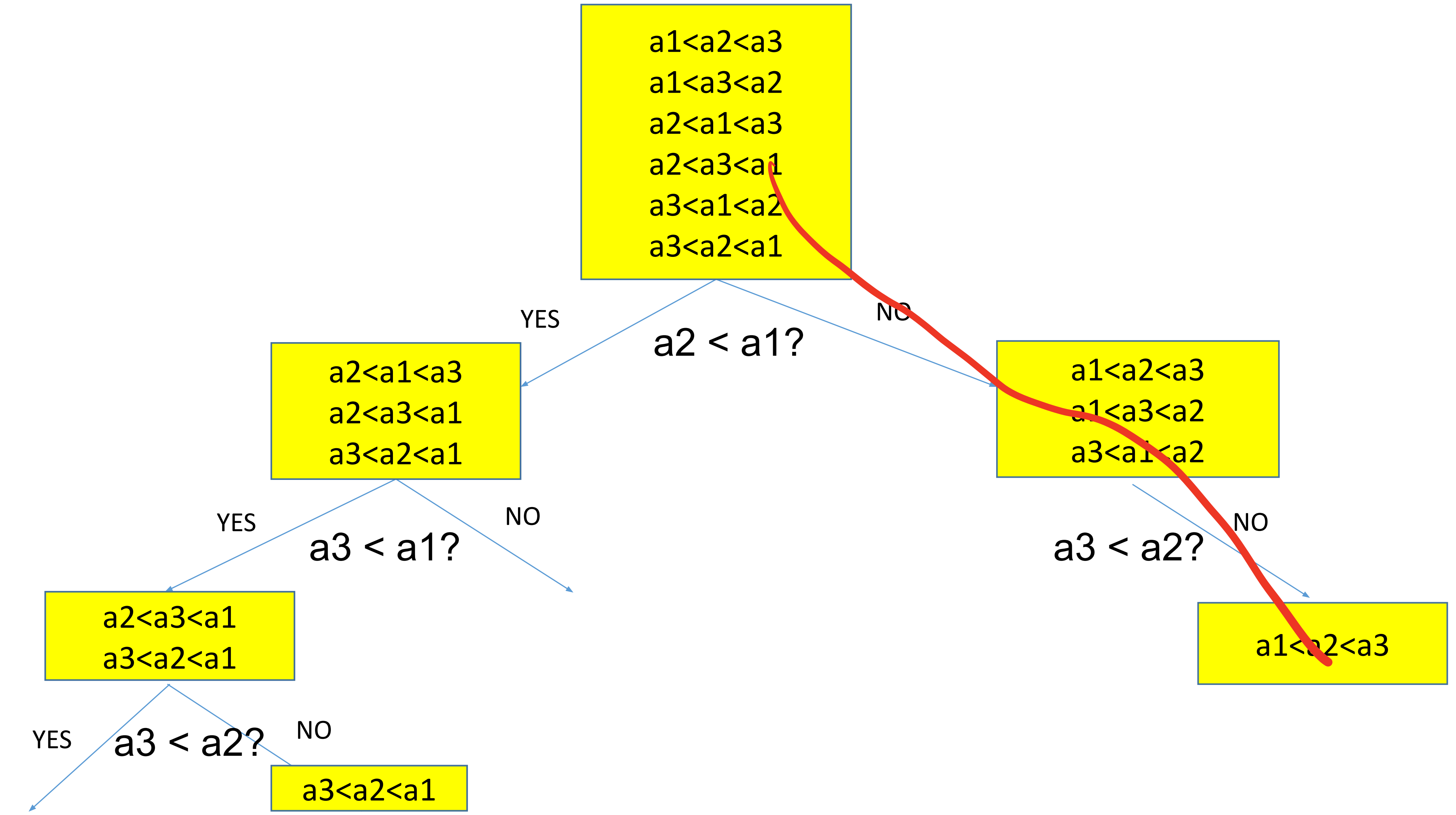
$a1 = 1, a2 = 2, a3 = 3$ initial

cmp(a2, a1)

$a1 = 1, a2 = 2, a3 = 3$

cmp(a3, a2)

$a1 = 1, a2 = 2, a3 = 3$

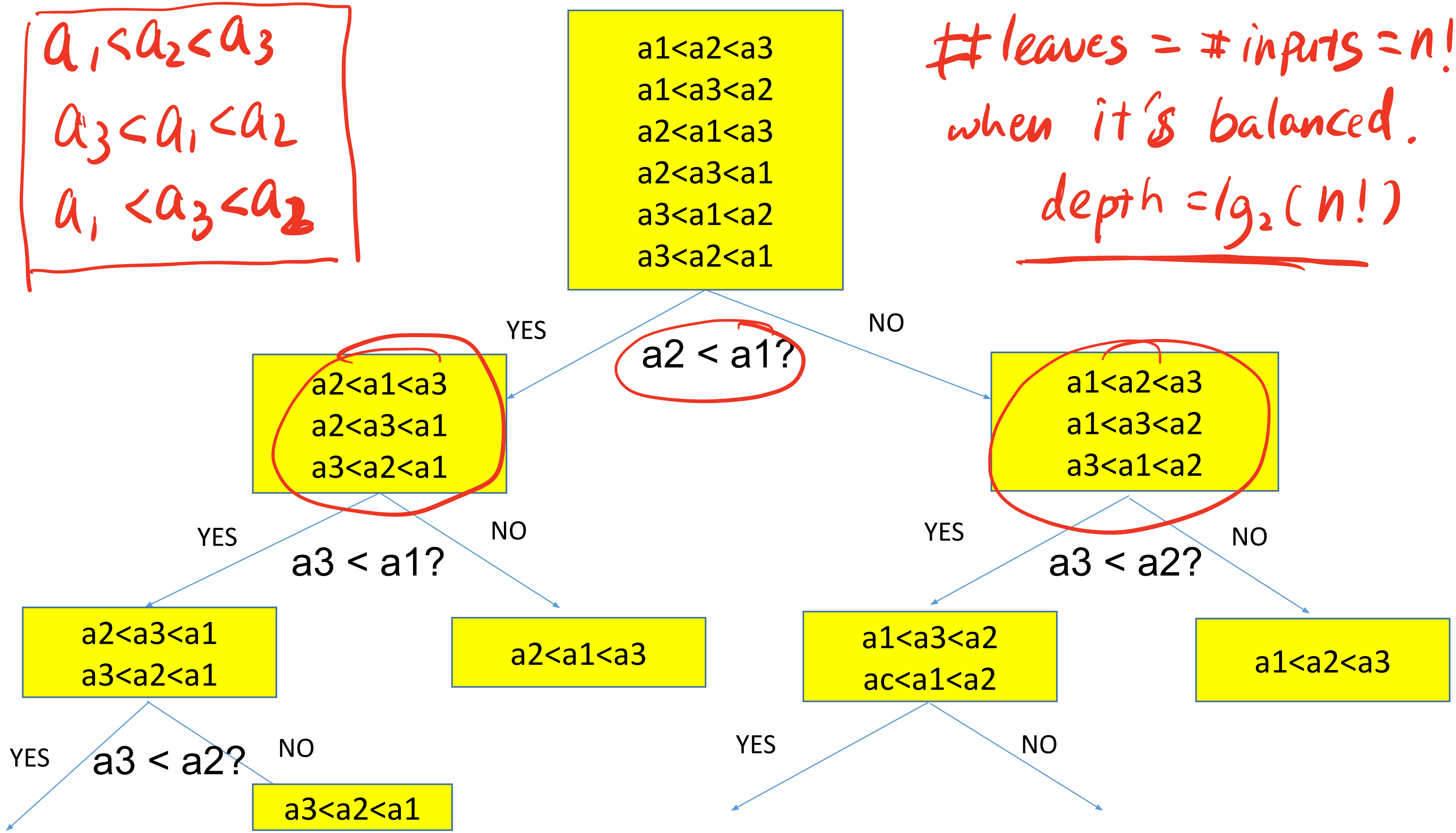


a1<a2<a3
a1<a3<a2
a2<a1<a3
a2<a3<a1
a3<a1<a2
a3<a2<a1
YES
a2 < a1?
NO
a2<a1<a3
a2<a3<a1
a3<a2<a1
a1<a2<a3
a1<a3<a2
a3<a1<a2
YES
a3 < a1?
NO
a3 < a2?
NO
a2<a3<a1
a3<a2<a1
a1<a2<a3
YES
a3 < a2?
NO
a3<a2<a1

$a_1<a_2<a_3$

$a_3<a_1<a_2$

$a_1<a_3<a_2$



leaves = # inputs = $n!$ when it's balanced.

depth = $\lg_2(n!)$

max depth of decision tree $\geq \log_2(n!)$

Sorting Lower Bound

- Information-theoretic: need $\lg(n!)$ bits of information about the input before we can correctly decide on the output

- $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ \ldots + \lg(1) < n \lg n$

Sorting Lower Bound

- Information-theoretic: need $\lg(n!)$ bits of information about the input before we can correctly decide on the output

- $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ \dots + \lg(1) < n \lg n$
- $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ \dots + \lg(1) > \left(\frac{n}{2}\right) \lg \left(\frac{n}{2}\right) = \Omega(n \lg n)$

Sorting Lower Bound

- Information-theoretic: need $\lg(n!)$ bits of information about the input before we can correctly decide on the output

- $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ \ldots + \lg(1) < n \lg n$
- $\lg(n!) = \lg(n) + \lg(n-1) + \lg(n-2) + \ \ldots + \lg(1) > \left(\frac{n}{2}\right) \lg\left(\frac{n}{2}\right) = \Omega(n \lg n)$

- $n! \in [\left(\frac{n}{e}\right)^n, n^n]$, $\quad$ so $n \lg n - n \lg e < \lg(n!) < n \lg n$
  $$n \lg n - 1.443n < \lg(n!) < n \lg n$$

- $\lg(n!) = (n \lg n)(1 - o(1))$

Sorting Upper Bounds

- Suppose for simplicity n is a power of 2
- Binary insertion sort: using binary search to insert each new element,

Sorting Upper Bounds

- Suppose for simplicity n is a power of 2
- Binary insertion sort: using binary search to insert each new element, the number of comparisons is $\sum_{k=2,\ldots,n} \lceil \lg k \rceil \leq n \lceil \lg n \rceil$

Sorting Upper Bounds

- Suppose for simplicity n is a power of 2
- Binary insertion sort: using binary search to insert each new element, the number of comparisons is $\sum_{k=2,\ldots,n} \lceil \lg k \rceil \leq n \lceil \lg n \rceil$

Why don't we often use binary insertion sort in practice?

Sorting Upper Bounds

- Suppose for simplicity n is a power of 2

- Binary insertion sort: using binary search to insert each new element,

- Mergesort: merging two sorted lists of n/2 elements requires at most n-1 comparisons

$$\underline{T(n) = 2T\left(\frac{n}{2}\right) + (n-1)} \qquad O(n \log n)$$

Sorting Upper Bounds

- Suppose for simplicity n is a power of 2
- Binary insertion sort: using binary search to insert each new element, the number of comparisons is $\sum_{k=2,\ldots,n} \lceil \lg k \rceil \leq n \lg n$
- Mergesort: merging two sorted lists of n/2 elements requires at most n-1 comparisons

What's the cost?

Sorting Upper Bounds

- Suppose for simplicity n is a power of 2

- Binary insertion sort: using binary search to insert each new element, the number of comparisons is $\sum_{k=2,\ldots,n} \lceil \lg k \rceil \leq n \lg n$

- Mergesort: merging two sorted lists of n/2 elements requires at most n-1 comparisons

Implication our the lower bound:

any **comparison-based** sorting algorithm must take Ω (n log n) time on a RAM

- no matter whether actual algorithm is implemented as a RAM program or circuit
- a useful guide and sanity check when we design algorithm

Implication our the lower bound:

any **comparison-based** sorting algorithm must take Ω (n log n) time on a RAM

Non-comparison-based algorithms can take o(n log n) time

e.g., bucket sort

ARTICLE

Deterministic sorting in *O*(*n*log log *n*) time and linear space

Author: Yijie Han Authors Info & Claims

STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing • May 2002 • Pages 602–608 • https://doi.org/10.1145/509907.509993

Online: 19 May 2002 Publication History

any **comparison-based** sorting algorithm must take Ω (n log n) time on a RAM

Cool fact about comparison-based sort (0-1 principle)

- any comparison-based sorting algorithm that can sort 0s and 1s can sort arbitrary numbers!
- Proof: see Knuth's textbook

An O(n)-time comparison based sorting algorithm

Go down the list and check if each element is bigger than the previous. If not, eliminate the element.

The result must be sorted

Elimination-based sorting

Selection in the Comparison Model

- How many comparisons are necessary and sufficient to find the maximum of n elements in the comparison model?

Selection in the Comparison Model

- How many comparisons are necessary and sufficient to find the maximum of n elements in the comparison model?

- Claim: n-1 comparisons are sufficient
- Proof: scan from left to right, keep track of the largest element so far

Selection in the Comparison Model

- How many comparisons are necessary and sufficient to find the maximum of n elements in the comparison model?

- Claim: n-1 comparisons are sufficient
- Proof: scan from left to right, keep track of the largest element so far

- For lower bounds, what does our earlier information-theoretic argument give?

Selection in the Comparison Model

- How many comparisons are necessary and sufficient to find the maximum of n elements in the comparison model?

- Claim: n-1 comparisons are sufficient
- Proof: scan from left to right, keep track of the largest element so far

- For lower bounds, what does our earlier information-theoretic argument give?
  - Only $\Omega(\log n)$, which is too weak

- Also, we have to look at all elements, otherwise we may have not looked at the largest, but that can be done with n/2 comparisons, also not tight

Lower Bound for Finding the Maximum

- Claim: n-1 comparisons are needed in the worst-case to find the maximum of n elements

Lower Bound for Finding the Maximum

- Claim: n-1 comparisons are needed in the worst-case to find the maximum of n elements

Adversarial argument

If output produced without having made enough comparisons, can construct **an adversarial input consistent with all the answers so far**, that **fools the algorithm to output incorrectly**

$A \to B$: $A < B$

Lower Bound for Finding the Maximum

- Claim: n-1 comparisons are needed in the worst-case to find the maximum of n elements

- Proof: suppose A is an algorithm which finds the maximum of n distinct elements using fewer than n-1 comparisons
- Construct a graph G in which we join two elements by an edge if they are compared by A

$M > 5$

Lower Bound for Finding the Maximum

- Claim: n-1 comparisons are needed in the worst-case to find the maximum of n elements

- Proof: suppose A is an algorithm which finds the maximum of n distinct elements using fewer than n-1 comparisons
- Construct a graph G in which we join two elements by an edge if they are compared by A
- G has at least 2 connected components C_1 and C_2

Lower Bound for Finding the Maximum

- Claim: n-1 comparisons are needed in the worst-case to find the maximum of n elements

- Proof: suppose A is an algorithm which finds the maximum of n distinct elements using fewer than n-1 comparisons
- Construct a graph G in which we join two elements by an edge if they are compared by A
- G has at least 2 connected components C_1 and C_2
- Suppose A outputs element u as the maximum, and $u \in C_1$

Lower Bound for Finding the Maximum

- Claim: n-1 comparisons are needed in the worst-case to find the maximum of n elements

- Proof: suppose A is an algorithm which finds the maximum of n distinct elements using fewer than n-1 comparisons
- Construct a graph G in which we join two elements by an edge if they are compared by A
- G has at least 2 connected components C_1 and C_2
- Suppose A outputs element u as the maximum, and u $\in C_1$
- Add a large positive number to each element in C_2
- Does not change any of the comparisons made by A, so will still output u
- But now u is not the maximum, so A is incorrect

Lower Bound for Finding the Maximum

- Recap: upper and lower bounds match at n-1
- Argument different from information-theoretic bound for sorting, use the **adversarial argument**

First and Second Largest of n Elements

- How many comparisons are necessary (lower bound) and sufficient (upper bound) to find the first and second largest of n distinct elements?

First and Second Largest of n Elements

- How many comparisons are necessary (lower bound) and sufficient (upper bound) to find the first and second largest of n distinct elements?

- Claim: n-1 comparisons are needed in the worst-case

- Proof: need to at least find the maximum

What about Upper Bounds?

What about Upper Bounds?

- Claim: 2n-3 comparisons are sufficient to find the first and second-largest of n elements
- Proof: find the largest using n-1 comparisons, then find the largest of the remainder using n-2 comparisons, so 2n-3 total

What about Upper Bounds?

- Claim: 2n-3 comparisons are sufficient to find the first and second-largest of n elements
- Proof: find the largest using n-1 comparisons, then find the largest of the remainder using n-2 comparisons, so 2n-3 total
- Upper bound is 2n-3, and lower bound n-1, both are $\Theta(n)$ but can we get tight bounds?

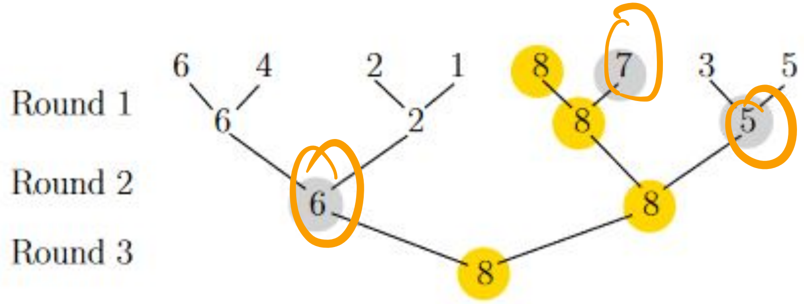
Second Largest of n Elements Upper Bound

- Claim: $n + \lg n - 2$ comparisons are sufficient to find the first and second-largest of n elements

Second Largest of n Elements Upper Bound

$n-1 + \lg n - 1$

- Claim: $n + \lg n - 2$ comparisons are sufficient to find the first and second-largest of n elements
- Proof: find the maximum element using n-1 comparisons by grouping elements into pairs, finding the maximum in each pair, and recursing

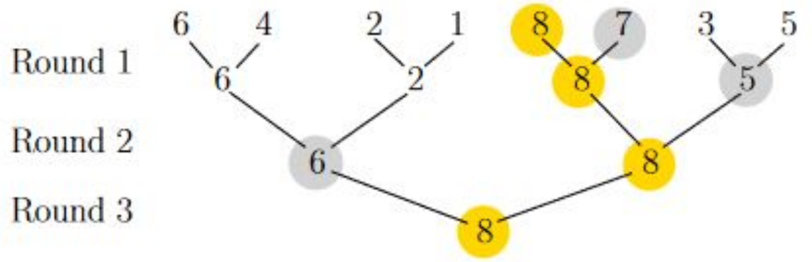


Grey nodes = lg n

the second max must be among the grey nodes

Second Largest of n Elements Upper Bound

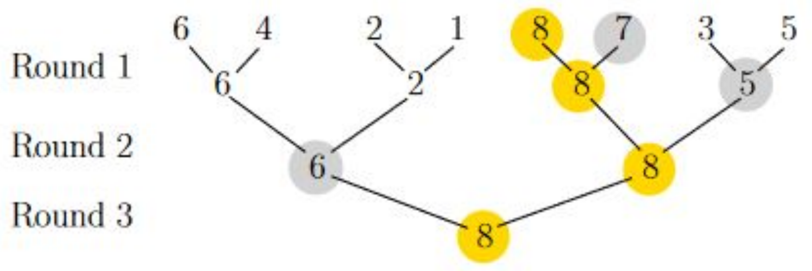
- Claim: $n + \lg n - 2$ comparisons are sufficient to find the first and second-largest of n elements
- Proof: find the maximum element using n-1 comparisons by grouping elements into pairs, finding the maximum in each pair, and recursing



- How do we find the second maximum?

Second Largest of n Elements Upper Bound

- Claim: $n + \lg n - 2$ comparisons are sufficient to find the first and second-largest of n elements
- Proof: find the maximum element using n-1 comparisons by grouping elements into pairs, finding the maximum in each pair, and recursing



- How do we find the second maximum?
  - Must have been directly compared to the maximum and lost, so lg(n)-1 additional comparisons suffice. Kislitsyn (1964) shows this is optimal

Sorting in the Exchange Model

- Consider a shelf containing n unordered books to be arranged alphabetically. How many swaps do we need to order them?

Sorting in the Exchange Model

- Consider a shelf containing n unordered books to be arranged alphabetically. How many swaps do we need to order them?

- In the exchange model, you have n items and the only operation allowed on the items is to swap a pair of them at a cost of 1 step
  - All other work is free, e.g., the items can be examined and compared

Sorting in the Exchange Model

- Claim: n-1 exchanges is sufficient

Sorting in the Exchange Model

- Claim: n-1 exchanges is sufficient
- Proof:
- 1st step: swap the smallest item with the item in the first location
- 2nd step: swap the second smallest item with the item in the second location
- k-th step: swap the k-th smallest item with item in the k-th location
  - If no swap is necessary, just skip a given step
- No swap ever undoes our previous work
- At the end, the last item must already be in the correct location

Lower Bound for Sorting in Exchange Model

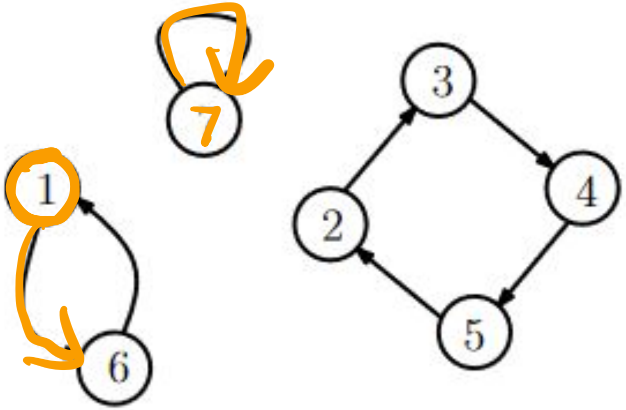
- Claim: n-1 exchanges are necessary in the worst case

Lower Bound for Sorting in Exchange Model

- Claim: n-1 exchanges are necessary in the worst case
- Proof: create a directed graph in which the edge (i,j) means the book in location i must end up in location j

initial

Claim: graph consists of cycles



end config:

n

Figure 1: Graph for input [f c d e b a g]

Lower Bound for Sorting in Exchange Model

- Claim: n-1 exchanges are necessary in the worst case
- Proof: create a directed graph in which the edge (i,j) means the book in location i must end up in location j

Graph is a set of cycles

Indegree and Outdegree of each node is 1

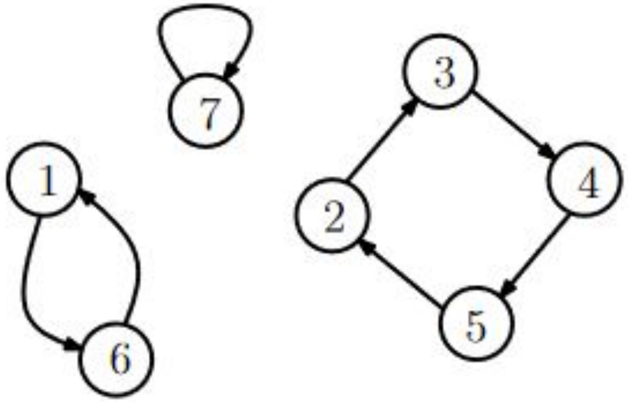
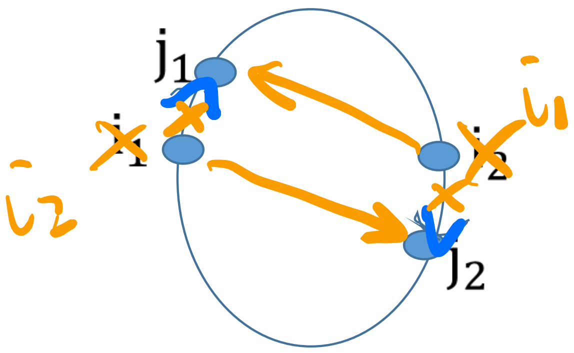


Figure 1: Graph for input [f c d e b a g]

Lower Bound for Sorting in Exchange Model

Case 1

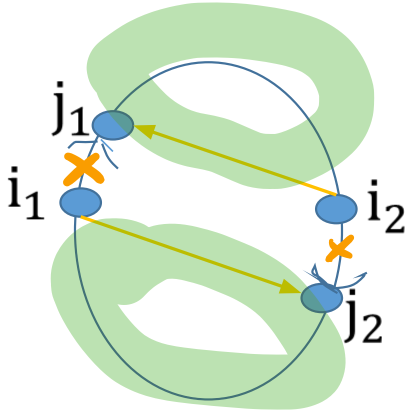
- What is the effect of exchanging any two elements in the same cycle?



Swap i1 and i2

Lower Bound for Sorting in Exchange Model

- What is the effect of exchanging any two elements in the same cycle?

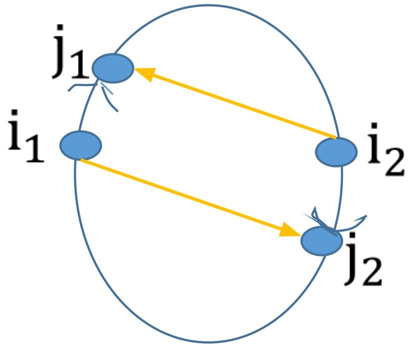


Swap i1 and i2

Lower Bound for Sorting in Exchange Model

- What is the effect of exchanging any two elements in the same cycle?

Break into 2 cycles



Lower Bound for Sorting in Exchange Model

Case 2

- What is the effect of exchanging any two elements in different cycles?

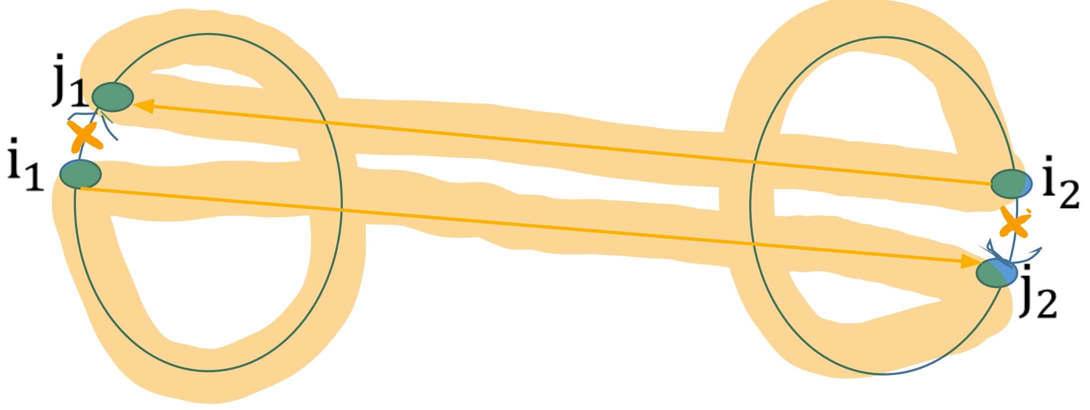
Swap i1 and i2

Lower Bound for Sorting in Exchange Model

- What is the effect of exchanging any two elements in different cycles?

merged into one cycle

special case : self-loop



Lower Bound for Sorting in Exchange Model

- Exchanging any two elements in the same cycle?
  - Get two disjoint cycles
- Exchanging any two elements in different cycles?
  - Merges two cycles into one cycle
- Corner cases also result in self loop and create two disjoint cycles

Lower Bound for Sorting in Exchange Model

- How many cycles are in the final sorted array?
  - n cycles
- Suppose we begin with an array [n, 1, 2, …, n-1] with one big cycle
- Each step increases the # cycles by at most 1, so need **n-1** steps

Query Models and Evasiveness

- Let G be the adjacency matrix of an n-node graph
  - G[i,j] = 1 if there is an edge between i and j, else G[i,j] = 0
- In 1 step, we can query any element of G. All other computation is free
- How many queries do we need to tell if G is connected?

Query Models and Evasiveness

- Let G be the adjacency matrix of an n-node graph
  - G[i,j] = 1 if there is an edge between i and j, else G[i,j] = 0
- In 1 step, we can query any element of G. All other computation is free
- How many queries do we need to tell if G is connected?
- Claim: $n(n-1)/2$ queries suffice

Query Models and Evasiveness

- Let G be the adjacency matrix of an n-node graph
  - G[i,j] = 1 if there is an edge between i and j, else G[i,j] = 0
- In 1 step, we can query any element of G. All other computation is free
- How many queries do we need to tell if G is connected?
- Claim: n(n-1)/2 queries suffice

- *What about lower bounds?*

Connectivity is an Evasive Graph Property

- Theorem: $n(n-1)/2$ queries are necessary to determine connectivity

Proof strategy: design an “evader” adversary

Algorithm A playing a game with an evader E

Each time, A asks about (i, j), E reveals whether there’s an edge
At some point, A outputs a decision

E wants to force A to query the entire graph to rule out any ambiguity

Example

Intuition: evader’s goal

- Maintain ambiguity
  among all graphs consistent with the revealed part, some are connected and some are not

- Make sure it’s not possible for A to decide without querying some unqueried edge

Example

Revealed edges form two connected components A and B.
A contains a1 and a2, B contains b1 and b2.
(a1, b1), (a2, b2) not queried.
Now, query (a1, b1). What should E answer?

Evader’s algorithm

Let T1, T2, … Tk be the current connected components among the edges revealed to exist.
Query (v, u).

- Suppose v $\in$ T1 and u $\in$ T2. Answer YES only if every other edge between T1 and T2 have been queried. Otherwise answer NO.

Evader’s algorithm

Let T1, T2, … Tk be the current connected components among the edges revealed to exist.

Query (v, u).

- Suppose v ∈ T1 and u ∈ T2. Answer YES only if every other edge between T1 and T2 have been queried. Otherwise answer NO.
- what if (v, u) in same connected component?

Evader’s algorithm

Let T1, T2, … Tk be the current connected components among the edges revealed to exist.

Query (v, u).

- Suppose v ∈ T1 and u ∈ T2. Answer YES only if every other edge between T1 and T2 have been queried. Otherwise answer NO.
- what if (v, u) in same connected component? **This will not happen.**

Claim: The evader algorithm maintains the following invariants:

1. All pairs inside a connected component have been queried

2. Between any two connected components T_i and T_j, some pair has not been queried.

Claim: The evader algorithm maintains the following invariants:

1. All pairs inside a connected component have been queried

2. Between any two connected components T_i and T_j, some pair has not been queried.

Proof: by induction

- Base case: it's true initially
- Induction hypothesis: suppose true now, true after next query

Claim: The evader algorithm maintains the following invariants:

1. All pairs inside a connected component have been queried

2. Between any two connected components T_i and T_j, some pair has not been queried.

Theorem: A must query all pairs when interacting with this evader

Claim: The evader algorithm maintains the following invariants:

1. All pairs inside a connected component have been queried
2. Between any two connected components T_i and T_j, some pair has not been queried.

Theorem: A must query all pairs when interacting with this evader
Proof: at the end of the algorithm, there cannot be more than 1 connected components left, since otherwise, by invariant 2, correctness is not guaranteed. The theorem follows due to invariant 1.


Happy Spring Festival!