CMPSC 465: LECTURE V

Information Theory Lower Bound
QuickSort

Ke Chen

September 08, 2025

Recall from last week...

The time complexity of MergeSort satisfies
$T(n) = 2T(n/2) + \Theta(n)$.

We can solve this recurrence relation by

- substitution (guess and induction),
- unrolling,
- recursion tree,
- master theorem,

to get $T(n) = \Theta(n \log n)$.

Recall from last week...

The time complexity of MergeSort satisfies
$T(n) = 2T(n/2) + \Theta(n)$.

We can solve this recurrence relation by

- substitution (guess and induction),
- unrolling,
- recursion tree,
- master theorem,

to get $T(n) = \Theta(n \log n)$.

Can we do better?

Recall from last week...

The time complexity of MergeSort satisfies
$T(n) = 2T(n/2) + \Theta(n)$.

We can solve this recurrence relation by

- substitution (guess and induction),
- unrolling,
- recursion tree,
- master theorem,

to get $T(n) = \Theta(n \log n)$.

Can we do better?

No. Any comparison-based sorting algorithm needs $\Omega(n \log n)$ comparisons in the worst case.

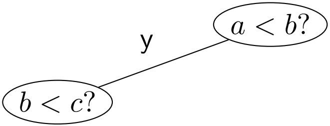
Decision-tree algorithms

Consider sorting three distinct numbers a, b, and c:

$$a < b?$$

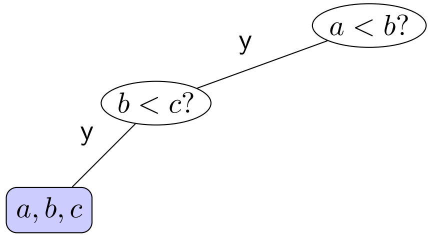
Decision-tree algorithms

Consider sorting three distinct numbers a, b, and c:



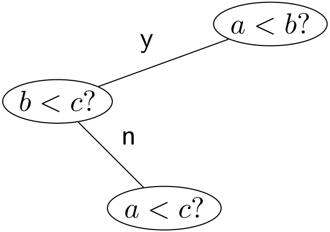
Decision-tree algorithms

Consider sorting three distinct numbers a, b, and c:



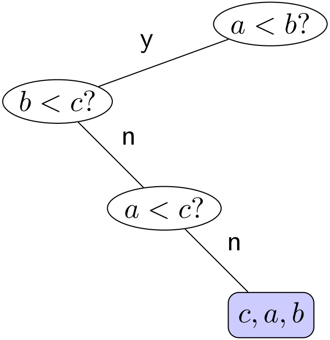
Decision-tree algorithms

Consider sorting three distinct numbers a, b, and c:



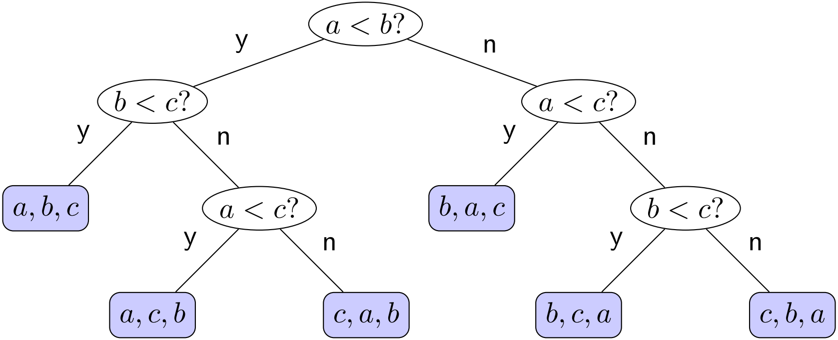
Decision-tree algorithms

Consider sorting three distinct numbers a, b, and c:



Decision-tree algorithms

Consider sorting three distinct numbers a, b, and c:



The information theory lower bound (ITLB)

Observations:

- Comparison-based sorting algorithms can be modeled by decision trees.
- Each leaf node corresponds to a possible output.
- The execution of the algorithm corresponds to a path from root down to a leaf.
- A longest path (i.e., height of the tree) gives the worst-case number of comparisons needed.

The information theory lower bound (ITLB)

Observations:

- Comparison-based sorting algorithms can be modeled by decision trees.
- Each leaf node corresponds to a possible output.
- The execution of the algorithm corresponds to a path from root down to a leaf.
- A longest path (i.e., height of the tree) gives the worst-case number of comparisons needed.

A leaf-counting argument:

- Each possible output must appear in the tree. Sorting n distinct numbers, there are ? possible outputs.

The information theory lower bound (ITLB)

Observations:

- Comparison-based sorting algorithms can be modeled by decision trees.
- Each leaf node corresponds to a possible output.
- The execution of the algorithm corresponds to a path from root down to a leaf.
- A longest path (i.e., height of the tree) gives the worst-case number of comparisons needed.

A leaf-counting argument:

- Each possible output must appear in the tree. Sorting n distinct numbers, there are $n!$ possible outputs.

The information theory lower bound (ITLB)

Observations:

- Comparison-based sorting algorithms can be modeled by decision trees.
- Each leaf node corresponds to a possible output.
- The execution of the algorithm corresponds to a path from root down to a leaf.
- A longest path (i.e., height of the tree) gives the worst-case number of comparisons needed.

A leaf-counting argument:

- Each possible output must appear in the tree. Sorting n distinct numbers, there are $n!$ possible outputs.
- A binary tree of height h can have at most ? leaves.

The information theory lower bound (ITLB)

Observations:

- Comparison-based sorting algorithms can be modeled by decision trees.
- Each leaf node corresponds to a possible output.
- The execution of the algorithm corresponds to a path from root down to a leaf.
- A longest path (i.e., height of the tree) gives the worst-case number of comparisons needed.

A leaf-counting argument:

- Each possible output must appear in the tree. Sorting n distinct numbers, there are $n!$ possible outputs.
- A binary tree of height h can have at most 2^h leaves.

The information theory lower bound (ITLB)

Observations:

- Comparison-based sorting algorithms can be modeled by decision trees.
- Each leaf node corresponds to a possible output.
- The execution of the algorithm corresponds to a path from root down to a leaf.
- A longest path (i.e., height of the tree) gives the worst-case number of comparisons needed.

A leaf-counting argument:

- Each possible output must appear in the tree. Sorting n distinct numbers, there are $n!$ possible outputs.
- A binary tree of height h can have at most 2^h leaves.
- So $2^h \geq n!$, or $h \geq \log(n!) = \Omega(n \log n)$.

Lower bounds on sorting

Theorem: Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Lower bounds on sorting

Theorem: Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Corollary: MergeSort is an asymptotically optimal comparison-based sorting algorithm.

Lower bounds on sorting

Theorem: Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Corollary: MergeSort is an asymptotically optimal comparison-based sorting algorithm.

Notes:

- Not all sorting algorithms are comparison-based: e.g., CountingSort, RadixSort, and BucketSort run in $O(n)$ time (with constraints on inputs).

Lower bounds on sorting

Theorem: Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons in the worst case.

Corollary: MergeSort is an asymptotically optimal comparison-based sorting algorithm.

Notes:

- Not all sorting algorithms are comparison-based: e.g., CountingSort, RadixSort, and BucketSort run in $O(n)$ time (with constraints on inputs).
- ITLB can also be used to obtain lower bounds for other comparison-based problems.

ITLB for insertion

Insertion problem: Given a sorted set of n distinct numbers, insert a new number into the proper position.

ITLB for insertion

Insertion problem: Given a sorted set of n distinct numbers, insert a new number into the proper position.

- There are ? possible outputs, so ITLB requires any algorithm make at least ? comparisons.

ITLB for insertion

Insertion problem: Given a sorted set of n distinct numbers, insert a new number into the proper position.

- There are $n + 1$ possible outputs, so ITLB requires any algorithm make at least $\log(n + 1)$ comparisons.

ITLB for insertion

Insertion problem: Given a sorted set of n distinct numbers, insert a new number into the proper position.

- There are $n+1$ possible outputs, so ITLB requires any algorithm make at least $\log(n+1)$ comparisons.
- It turns out that this number suffices: binary search can solve the problem with $\lceil \log(n+1) \rceil$ comparisons.

ITLB for insertion

Insertion problem: Given a sorted set of n distinct numbers, insert a new number into the proper position.

- There are $n + 1$ possible outputs, so ITLB requires any algorithm make at least $\log(n + 1)$ comparisons.
- It turns out that this number suffices: binary search can solve the problem with $\lceil \log(n + 1) \rceil$ comparisons.
- Note that ITLB only considers comparisons.

ITLB for insertion

Insertion problem: Given a sorted set of n distinct numbers, insert a new number into the proper position.

- There are $n + 1$ possible outputs, so ITLB requires any algorithm make at least $\log(n + 1)$ comparisons.
- It turns out that this number suffices: binary search can solve the problem with $\lceil \log(n + 1) \rceil$ comparisons.
- Note that ITLB only considers comparisons.
  How many other operations are needed for insertion?

ITLB for insertion

Insertion problem: Given a sorted set of n distinct numbers, insert a new number into the proper position.

- There are $n + 1$ possible outputs, so ITLB requires any algorithm make at least $\log(n + 1)$ comparisons.
- It turns out that this number suffices: binary search can solve the problem with $\lceil \log(n + 1) \rceil$ comparisons.
- Note that ITLB only considers comparisons.
  How many other operations are needed for insertion?
  Can insertion be done in $O(\log n)$ time?

ITLB for selection

Selection problem: Given a list of n numbers and a target k, select the k-th smallest number.

ITLB for selection

Selection problem: Given a list of n numbers and a target k, select the k-th smallest number.

- There are ? possible outputs, so ITLB requires any algorithm make at least ? comparisons.

ITLB for selection

Selection problem: Given a list of n numbers and a target k, select the k-th smallest number.

- There are n possible outputs, so ITLB requires any algorithm make at least $\log n$ comparisons.

ITLB for selection

Selection problem: Given a list of n numbers and a target k, select the k-th smallest number.

- There are n possible outputs, so ITLB requires any algorithm make at least $\log n$ comparisons.
- This is quite a poor lower bound! Can you argue for $\Omega(n)$ comparisons?

ITLB for selection

Selection problem: Given a list of n numbers and a target k, select the k-th smallest number.

- There are n possible outputs, so ITLB requires any algorithm make at least $\log n$ comparisons.
- This is quite a poor lower bound! Can you argue for $\Omega(n)$ comparisons?
- There is an algorithm using $O(n)$ comparisons in the worst case. Thus the complexity of selection is $\Theta(n)$.

ITLB for selection

Selection problem: Given a list of n numbers and a target k, select the k-th smallest number.

- There are n possible outputs, so ITLB requires any algorithm make at least $\log n$ comparisons.
- This is quite a poor lower bound! Can you argue for $\Omega(n)$ comparisons?
- There is an algorithm using $O(n)$ comparisons in the worst case. Thus the complexity of selection is $\Theta(n)$.

Morale: The ITLB does not always give a tight bound, it can be very far off.

Back to MergeSort

Time complexity?

- $\Theta(n \log n)$, asymptotic optimal.

Space complexity?

- $O(n)$ extra space for merging.
- Can be reduced to $O(1)$, but is complicated and not practical.

Back to MergeSort

Time complexity?

- $\Theta(n \log n)$, asymptotic optimal.

Space complexity?

- $O(n)$ extra space for merging.
- Can be reduced to $O(1)$, but is complicated and not practical.

Can we do better?

QuickSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer

1. Split input by a pivot $\quad$ $\overset{<}{1,2,4}$ $\overset{\text{pivot}}{5}$ $\overset{>}{8,9,6,8}$

QuickSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer

		$<$	pivot	$>$
1	Split input by a pivot	1,2,4	5	8,9,6,8
2	Sort each part	1,2,4	5	6,8,8,9

QuickSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer

		$<$	pivot	$>$
1	Split input by a pivot	1,2,4	5	8,9,6,8
2	Sort each part	1,2,4	5	6,8,8,9
3	TADA!	1,2,4,5,6,8,8,9		

QuickSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer

		<	pivot	>
1	Split input by a pivot	1,2,4	5	8,9,6,8
2	Sort each part	1,2,4	5	6,8,8,9
3	TADA!	1,2,4,5,6,8,8,9		

QuickSort(A, st, ed)

```
if st < ed then
    p = Partition(A, st, ed)
    QuickSort(A, st, p − 1)
    QuickSort(A, p + 1, ed)
```

QuickSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer	<	pivot	>
1 Split input by a pivot	1,2,4	5	8,9,6,8
2 Sort each part	1,2,4	5	6,8,8,9
3 TADA!	1,2,4,5,6,8,8,9		

QuickSort(A, st, ed)

- **if** $st < ed$ **then**
  - $p =$ Partition(A, st, ed)
  - QuickSort(A, st, $p - 1$)
  - QuickSort(A, $p + 1$, ed)

How to do Partition?

QuickSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer

		<	pivot	>
1	Split input by a pivot	1,2,4	5	8,9,6,8
2	Sort each part	1,2,4	5	6,8,8,9
3	TADA!	1,2,4,5,6,8,8,9		

QuickSort(A, st, ed)

if $st < ed$ **then**
- $p =$ Partition(A, st, ed)
- QuickSort(A, st, $p - 1$)
- QuickSort(A, $p + 1$, ed)

How to do Partition? Easy!

QuickSort

Input: 8,1,9,2,8,4,6,5

Idea: divide and conquer

		<	pivot	>
1	Split input by a pivot	1,2,4	5	8,9,6,8
2	Sort each part	1,2,4	5	6,8,8,9
3	TADA!	1,2,4,5,6,8,8,9		

QuickSort(A, st, ed)

if $st < ed$ **then**
- $p =$ Partition(A, st, ed)
- QuickSort(A, st, $p - 1$)
- QuickSort(A, $p + 1$, ed)

How to do Partition in place, namely, with $O(1)$ extra space?

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

$$8 \quad 1 \quad 9 \quad 2 \quad 8 \quad 4 \quad 6 \quad 5$$

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

last element in the $<$ pivot region

8	1	9	2	8	4	6	5
st							ed

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

$\downarrow$

8	1	9	2	8	4	6	5
st							ed

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

$\downarrow$

8	1	9	2	8	4	6	5
st							ed

Partition around pivot

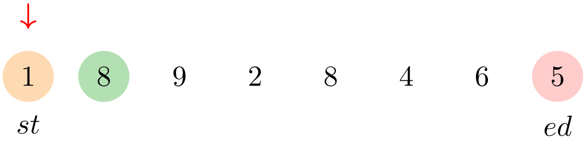
Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

$\downarrow$

8	1	9	2	8	4	6	5
st							ed

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region



Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

$$\begin{array}{cccccccc} \downarrow & & & & & & & \\ 1 & 8 & 9 & 2 & 8 & 4 & 6 & 5 \\ st & & & & & & & ed \end{array}$$

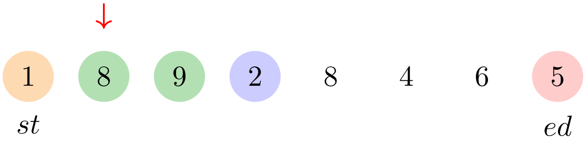
Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

st							ed
1	8	9	2	8	4	6	5

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region



Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

	$\downarrow$						
1	2	9	8	8	4	6	5
st							ed

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

	$\downarrow$						
1	2	9	8	8	4	6	5
st							ed

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

$$\downarrow$$

1	2	9	8	8	4	6	5
st							ed

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

$$\begin{array}{cccccccc} & & \downarrow & & & & & \\ 1 & 2 & 9 & 8 & 8 & 4 & 6 & 5 \\ st & & & & & & & ed \end{array}$$

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

		$\downarrow$					
1	2	4	8	8	9	6	5
st							ed

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

		$\downarrow$					
1	2	4	8	8	9	6	5
st							ed

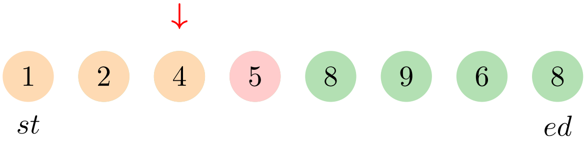
Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

1	2	4	8	8	9	6	5
st							ed

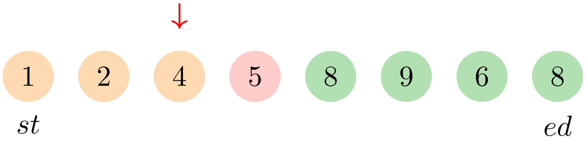
Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region



Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region



Partition(A, st, ed)

$pivot = A[ed]$
$bd = st - 1$
for $cur = st$ **to** $ed - 1$ **do**
  if $A[cur] < pivot$ **then**
    $bd = bd + 1$
    swap $A[bd]$ with $A[cur]$
swap $A[bd + 1]$ with $A[ed]$
return *bd+1*

Partition around pivot

Idea: check elements one by one against the pivot and maintain a $<$ pivot region and a $\geq$ pivot region

$\downarrow$

1	2	4	5	8	9	6	8
st							ed

Partition(A, st, ed)

- $pivot = A[ed]$
- $bd = st - 1$
- **for** $cur = st$ **to** $ed - 1$ **do**
  - **if** $A[cur] < pivot$ **then**
    - $bd = bd + 1$
    - swap $A[bd]$ with $A[cur]$
- swap $A[bd + 1]$ with $A[ed]$
- **return** *bd+1*

Time complexity? $\Theta(n)$

Space complexity? $\Theta(1)$

QuickSort

```
QuickSort(A, st, ed)
  if st < ed then
    p = Partition(A, st, ed)
    QuickSort(A, st, p − 1)
    QuickSort(A, p + 1, ed)
```

Time complexity?

- Worst-case is when each partition results in sizes $(0, 1, n-1)$.

  $T(n) \leq T(n-1) + cn$

QuickSort

QuickSort(A, st, ed)

if $st < ed$ **then**
- $p =$ Partition(A, st, ed)
- QuickSort(A, st, $p-1$)
- QuickSort(A, $p+1$, ed)

Time complexity?

- Worst-case is when each partition results in sizes $(0, 1, n-1)$.
  $T(n) \leq T(n-1) + cn$ $\rightsquigarrow$ $T(n) = O(n^2)$.

QuickSort

QuickSort(A, st, ed)

if $st < ed$ **then**

  $p =$ Partition(A, st, ed)

  QuickSort(A, st, $p - 1$)

  QuickSort(A, $p + 1$, ed)

Time complexity?

- Worst-case is when each partition results in sizes $(0, 1, n - 1)$.
  $T(n) \leq T(n-1) + cn \rightsquigarrow T(n) = O(n^2)$.
- Can you find an input that achieves this worst-case performance?

QuickSort

QuickSort(A, st, ed)

if $st < ed$ **then**
- $p =$ Partition(A, st, ed)
- QuickSort(A, st, $p - 1$)
- QuickSort(A, $p + 1$, ed)

Time complexity?

- Worst-case is when each partition results in sizes $(0, 1, n-1)$.
  $T(n) \leq T(n-1) + cn$ $\rightsquigarrow$ $T(n) = O(n^2)$.
- Can you find an input that achieves this worst-case performance?
- Why is it called QuickSort then?