Sorting Lower Bounds

Abhiram Ranade

January 13, 2016

Fundamental Goal of Algorithm Design

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

We would like to prove that:

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

We would like to prove that:

- "X can be solved in time $O(f)$ using algorithm A."

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

We would like to prove that:

- "X can be solved in time $O(f)$ using algorithm A."
- "Every algorithm must take time at least $\Omega(g)$ for solving X."

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

We would like to prove that:

- "X can be solved in time $O(f)$ using algorithm A."
- "Every algorithm must take time at least $\Omega(g)$ for solving X."

"Problem lower bound $= \Omega(g)$"

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

We would like to prove that:

- "X can be solved in time $O(f)$ using algorithm A."
- "Every algorithm must take time at least $\Omega(g)$ for solving X."

"Problem lower bound $= \Omega(g)$"

This is usually difficult.

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

We would like to prove that:

- "X can be solved in time $O(f)$ using algorithm A."
- "Every algorithm must take time at least $\Omega(g)$ for solving X."

"Problem lower bound $= \Omega(g)$"

This is usually difficult.

- $f = g$

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

We would like to prove that:

- "X can be solved in time $O(f)$ using algorithm A."
- "Every algorithm must take time at least $\Omega(g)$ for solving X."

  "Problem lower bound $= \Omega(g)$"

  This is usually difficult.
- $f = g$

  This shows that A is the best possible algorithm for X.

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

We would like to prove that:

- "X can be solved in time $O(f)$ using algorithm A."
- "Every algorithm must take time at least $\Omega(g)$ for solving X."

"Problem lower bound $= \Omega(g)$"

This is usually difficult.

- $f = g$

This shows that A is the best possible algorithm for X.

Even more difficult..

Fundamental Goal of Algorithm Design

Design the best possible algorithm for solving a problem X.

Best possible: Worst case time is as small as possible.

We would like to prove that:

- "X can be solved in time $O(f)$ using algorithm A."
- "Every algorithm must take time at least $\Omega(g)$ for solving X."

  "Problem lower bound $= \Omega(g)$"

  This is usually difficult.

- $f = g$

  This shows that A is the best possible algorithm for X.

  Even more difficult..

Today: A non-trivial lower bound for sorting algorithms of a certain type.

Trivial lower bound: I/O time

Trivial lower bound: I/O time

Algorithm time $\geq$ time to read all inputs.

Trivial lower bound: I/O time

Algorithm time $\geq$ time to read all inputs.

Time for sorting n keys $= \Omega(n)$.

Trivial lower bound: I/O time

Algorithm time $\geq$ time to read all inputs.

Time for sorting n keys $= \Omega(n)$.

Time for multiplying $n \times n$ matrices $= \Omega(n^2)$.

Trivial lower bound: I/O time

Algorithm time $\geq$ time to read all inputs.

Time for sorting n keys $= \Omega(n)$.

Time for multiplying $n \times n$ matrices $= \Omega(n^2)$.

Key question: Can we prove $\omega(n^2)$ lower bounds for matrix multiplication, or $\omega(n)$ lower bound for sorting?

Trivial lower bound: I/O time

Algorithm time $\geq$ time to read all inputs.

Time for sorting n keys $= \Omega(n)$.

Time for multiplying $n \times n$ matrices $= \Omega(n^2)$.

Key question: Can we prove $\omega(n^2)$ lower bounds for matrix multiplication, or $\omega(n)$ lower bound for sorting?

Note: Sometimes we may assume that data is already in memory. In such case the input size is not a lower bound.

Trivial lower bound: I/O time

Algorithm time $\geq$ time to read all inputs.

Time for sorting n keys $= \Omega(n)$.

Time for multiplying $n \times n$ matrices $= \Omega(n^2)$.

Key question: Can we prove $\omega(n^2)$ lower bounds for matrix multiplication, or $\omega(n)$ lower bound for sorting?

Note: Sometimes we may assume that data is already in memory. In such case the input size is not a lower bound.

Example: determining if some x is present in a sorted array.

Trivial lower bound: I/O time

Algorithm time $\geq$ time to read all inputs.

Time for sorting n keys $= \Omega(n)$.

Time for multiplying $n \times n$ matrices $= \Omega(n^2)$.

Key question: Can we prove $\omega(n^2)$ lower bounds for matrix multiplication, or $\omega(n)$ lower bound for sorting?

Note: Sometimes we may assume that data is already in memory. In such case the input size is not a lower bound.

Example: determining if some x is present in a sorted array.

$O(\log n)$ suffices, $\Omega(n)$ not needed.

A lower bound for comparison based sorting algorithms

A lower bound for comparison based sorting algorithms

Bubble sort, insertion sort, selection sort, merge sort, heap sort, quick sort only use comparisons to get information about the keys.

A lower bound for comparison based sorting algorithms

Bubble sort, insertion sort, selection sort, merge sort, heap sort, quick sort only use comparisons to get information about the keys.

No arithmetic done on keys.

A lower bound for comparison based sorting algorithms

Bubble sort, insertion sort, selection sort, merge sort, heap sort, quick sort only use comparisons to get information about the keys.

No arithmetic done on keys.

Question: Is it possible to organize comparisons more cleverly so that we can get time $o(n \log n)$?

A lower bound for comparison based sorting algorithms

Bubble sort, insertion sort, selection sort, merge sort, heap sort, quick sort only use comparisons to get information about the keys.

No arithmetic done on keys.

Question: Is it possible to organize comparisons more cleverly so that we can get time $o(n \log n)$?

Informal answer: No, $\theta(n \log n)$ is best to within constant factors.

A lower bound for comparison based sorting algorithms

Bubble sort, insertion sort, selection sort, merge sort, heap sort, quick sort only use comparisons to get information about the keys.

No arithmetic done on keys.

Question: Is it possible to organize comparisons more cleverly so that we can get time $o(n \log n)$?

Informal answer: No, $\theta(n \log n)$ is best to within constant factors.

Formal statement: Define an abstract machine model called decision tree such that

A lower bound for comparison based sorting algorithms

Bubble sort, insertion sort, selection sort, merge sort, heap sort, quick sort only use comparisons to get information about the keys.

No arithmetic done on keys.

Question: Is it possible to organize comparisons more cleverly so that we can get time $o(n \log n)$?

Informal answer: No, $\theta(n \log n)$ is best to within constant factors.

Formal statement: Define an abstract machine model called decision tree such that

- "Comparison based" sorting algorithms will take no more time on a decision tree than on the RAM.

A lower bound for comparison based sorting algorithms

Bubble sort, insertion sort, selection sort, merge sort, heap sort, quick sort only use comparisons to get information about the keys.

No arithmetic done on keys.

Question: Is it possible to organize comparisons more cleverly so that we can get time $o(n \log n)$?

Informal answer: No, $\theta(n \log n)$ is best to within constant factors.

Formal statement: Define an abstract machine model called decision tree such that

- "Comparison based" sorting algorithms will take no more time on a decision tree than on the RAM.
- We will prove that every decision tree algorithm must take time at least $\Omega(n \log n)$.

Decision tree for sorting $x_1, \ldots, x_n$

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.
- Each leaf is labelled with a permutation of $1, \ldots, n$.

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.
- Each leaf is labelled with a permutation of $1, \ldots, n$.

Execution model

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.
- Each leaf is labelled with a permutation of $1, \ldots, n$.

Execution model

- Execution starts from the root.

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.
- Each leaf is labelled with a permutation of $1, \ldots, n$.

Execution model

- Execution starts from the root.
- If non-leaf node with label $i : j$ is visited, then x_i, x_j are compared 1 time step

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.
- Each leaf is labelled with a permutation of $1, \ldots, n$.

Execution model

- Execution starts from the root.
- If non-leaf node with label $i : j$ is visited, then x_i, x_j are compared 1 time step

  If $x_i \leq x_j$, left child visited next. Else right child.

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.
- Each leaf is labelled with a permutation of $1, \ldots, n$.

Execution model

- Execution starts from the root.
- If non-leaf node with label $i : j$ is visited, then x_i, x_j are compared 1 time step
  If $x_i \leq x_j$, left child visited next. Else right child.
- If leaf is visited, label is output. Label = final index order.

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.
- Each leaf is labelled with a permutation of $1, \ldots, n$.

Execution model

- Execution starts from the root.
- If non-leaf node with label $i : j$ is visited, then x_i, x_j are compared 1 time step
  If $x_i \leq x_j$, left child visited next. Else right child.
- If leaf is visited, label is output. Label $=$ final index order.

Note: Different decision tree for each n.

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.
- Each leaf is labelled with a permutation of $1, \ldots, n$.

Execution model

- Execution starts from the root.
- If non-leaf node with label $i : j$ is visited, then x_i, x_j are compared 1 time step
  If $x_i \leq x_j$, left child visited next. Else right child.
- If leaf is visited, label is output. Label $=$ final index order.

Note: Different decision tree for each n.

Theorem 1: Bubble sort, ... heap sort are decision tree algorithms.

Decision tree for sorting $x_1, \ldots, x_n$

Binary tree with labels on nodes.

- Every non-leaf node has a label "$i : j$", where i, j are integers.
- Each leaf is labelled with a permutation of $1, \ldots, n$.

Execution model

- Execution starts from the root.
- If non-leaf node with label $i : j$ is visited, then x_i, x_j are compared — 1 time step
  If $x_i \leq x_j$, left child visited next. Else right child.
- If leaf is visited, label is output. Label = final index order.

Note: Different decision tree for each n.

Theorem 1: Bubble sort, ... heap sort are decision tree algorithms.

Theorem 2: Any decision tree takes time $\Omega(n \log n)$.

Proof of Theorem 1 for bubble sort, $n = 3$

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label =

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child:

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.

Proof of Theorem 1 for bubble sort, $n = 3$

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next?

Proof of Theorem 1 for bubble sort, $n = 3$

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Proof of Theorem 1 for bubble sort, $n = 3$

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2

Label of left child: Left child visited if $x[1] \leq x[2]$.

Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node:

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3].

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2

Label of left child: Left child visited if $x[1] \leq x[2]$.

Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.

In this case x[1],x[2] are exchanged.

Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2

Label of left child: Left child visited if $x[1] \leq x[2]$.

Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.

In this case x[1],x[2] are exchanged.

Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Other labels: LL=

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if x[1] $\leq$ x[2].
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if x[1] > x[2].
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Other labels: LL=1:2,

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2

Label of left child: Left child visited if $x[1] \leq x[2]$.

Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.

In this case x[1],x[2] are exchanged.

Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Other labels: LL=1:2, LR=

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Other labels: LL=1:2, LR=1:3,

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2

Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Other labels: LL=1:2, LR=1:3, RL=

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Other labels: LL=1:2, LR=1:3, RL=2:1,

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Other labels: LL=1:2, LR=1:3, RL=2:1, RR=

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2
Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Other labels: LL=1:2, LR=1:3, RL=2:1, RR=2:3

Proof of Theorem 1 for bubble sort, n = 3

```
for last=n-1 to 1
  for j=1 to last
     if x[j] > x[j+1] then exchange x[j], x[j+1]
```

Root node label = 1:2

Label of left child: Left child visited if $x[1] \leq x[2]$.
Which keys are compared next? x[2],x[3]

Label = 2:3

Label of right node: Right child visited if $x[1] > x[2]$.
In this case x[1],x[2] are exchanged.
Next comparison: x[2],x[3]. But x[2] holds key originally in x[1].

Label = 1:3

When deciding the label of a node we know how keys have moved so far, so we know which keys are compared as per original order.

Other labels: LL=1:2, LR=1:3, RL=2:1, RR=2:3

Leaf labels: 123, 123, 132, 312, 213, 213, 231, 321

Remarks

Remarks

In any comparison based algorithm key movement, and what gets compared next depends only upon how previous comparisons turned out.

Remarks

In any comparison based algorithm key movement, and what gets compared next depends only upon how previous comparisons turned out.
Thus we know what keys will get compared next, as per initial key order.

Remarks

In any comparison based algorithm key movement, and what gets compared next depends only upon how previous comparisons turned out.

Thus we know what keys will get compared next, as per initial key order.

We also know order of outputting keys at end.

Remarks

In any comparison based algorithm key movement, and what gets compared next depends only upon how previous comparisons turned out.
Thus we know what keys will get compared next, as per initial key order.
We also know order of outputting keys at end.
So we can construct decision tree.

Theorem 2: Time $= \Omega(n \log n)$

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

We could always set $x_1, \ldots, x_n$ so that final order is $x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$
Proof:
Consider a permutation π of $1, \ldots, n$.
We could always set $x_1, \ldots, x_n$ so that final order is
$x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

We could always set $x_1, \ldots, x_n$ so that final order is

$x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$
Proof:
Consider a permutation π of $1, \ldots, n$.
We could always set $x_1, \ldots, x_n$ so that final order is
$x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.
Applies to every permutation π.

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

We could always set $x_1, \ldots, x_n$ so that final order is $x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.

Applies to every permutation π.

But there are $n!$ permutations. $\square$

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

We could always set $x_1, \ldots, x_n$ so that final order is $x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.

Applies to every permutation π.

But there are $n!$ permutations. $\square$

Theorem proof:

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$
Proof:
Consider a permutation π of $1, \ldots, n$.
We could always set $x_1, \ldots, x_n$ so that final order is
$x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.
Applies to every permutation π.
But there are $n!$ permutations. $\square$

Theorem proof:
Binary tree of height h can have at most 2^h leaves.

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

We could always set $x_1, \ldots, x_n$ so that final order is $x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.

Applies to every permutation π.

But there are $n!$ permutations. $\square$

Theorem proof:

Binary tree of height h can have at most 2^h leaves.

Binary tree with L leaves must have height at least $\log L$.

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

We could always set $x_1, \ldots, x_n$ so that final order is

$x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.

Applies to every permutation π.

But there are $n!$ permutations. $\square$

Theorem proof:

Binary tree of height h can have at most 2^h leaves.

Binary tree with L leaves must have height at least $\log L$.

Decision tree height $= \log n!$

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$
Proof:
Consider a permutation π of $1, \ldots, n$.
We could always set $x_1, \ldots, x_n$ so that final order is
$x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.
Applies to every permutation π.
But there are $n!$ permutations. $\square$

Theorem proof:
Binary tree of height h can have at most 2^h leaves.
Binary tree with L leaves must have height at least $\log L$.
Decision tree height $= \log n! = \sum_i \log i$

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

We could always set $x_1, \ldots, x_n$ so that final order is $x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.

Applies to every permutation π.

But there are $n!$ permutations. $\square$

Theorem proof:

Binary tree of height h can have at most 2^h leaves.

Binary tree with L leaves must have height at least $\log L$.

Decision tree height $= \log n! = \sum_i \log i$

$n/2$ terms are at least $(\log(n/2)$

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

We could always set $x_1, \ldots, x_n$ so that final order is

$x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.

Applies to every permutation π.

But there are $n!$ permutations. $\square$

Theorem proof:

Binary tree of height h can have at most 2^h leaves.

Binary tree with L leaves must have height at least $\log L$.

Decision tree height $= \log n! = \sum_i \log i$

$n/2$ terms are at least $(\log(n/2) \geq (\log n)/4$.

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$

Proof:

Consider a permutation π of $1, \ldots, n$.

We could always set $x_1, \ldots, x_n$ so that final order is

$x_{\pi[1]}, x_{\pi[2]}, \ldots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.

Applies to every permutation π.

But there are $n!$ permutations. $\square$

Theorem proof:

Binary tree of height h can have at most 2^h leaves.

Binary tree with L leaves must have height at least $\log L$.

Decision tree height $= \log n! = \sum_i \log i$

$n/2$ terms are at least $(\log(n/2) \geq (\log n)/4$.

Hence height $= \Omega(n \log n)$

Theorem 2: Time $= \Omega(n \log n)$

Lemma: Number of leaves in decision tree $\geq n!$
Proof:
Consider a permutation π of $1, \dots, n$.
We could always set $x_1, \dots, x_n$ so that final order is
$x_{\pi[1]}, x_{\pi[2]}, \dots, x_{\pi[n]}$.

Make $x_{\pi[1]}$ smallest, ...

π must appear as a leaf.
Applies to every permutation π.
But there are $n!$ permutations. $\square$

Theorem proof:
Binary tree of height h can have at most 2^h leaves.
Binary tree with L leaves must have height at least $\log L$.
Decision tree height $= \log n! = \sum_i \log i$
$n/2$ terms are at least $(\log(n/2) \geq (\log n)/4$.
Hence height $= \Omega(n \log n)$
Worst case time $=$ height. $\square$