Lecture 08: Sorting II

COMP526: Efficient Algorithms

Will Rosenbaum
University of Liverpool

Updated: October 29, 2024

Announcements

1. Fourth Quiz, due Friday
   - Similar format to before
   - Covers (Balanced) Binary Search Trees (Lectures 6–7)
   - Quiz is **closed resource**
     - No books, notes, internet, etc.
     - Do not discuss until after submission deadline (Friday night, after midnight)
2. Programming Assignment (Draft) Posted
   - Due Wednesday, 13 November
3. Attendance Code:

Meeting Goals

- Discuss Divide and Conquer approaches to sorting
  - MergeSort
  - QuickSort
- Demonstrate lower bounds for comparison-based sorting

From Last Time

We recalled the **Sorting Task**:

7	1	2	5	3	4	8	6

$\longmapsto$

1	2	3	4	5	6	7	8

We discussed four sorting algorithms:

1. SELECTIONSORT: find the (next) smallest element and put it in place
2. BUBBLESORT: "pull" the largest values toward the end of the array
3. INSERTIONSORT: sort prefixes of the array by inserting the "next" element into sorted place
4. HEAPSORT: make a (max) heap, then repeated call REMOVEMAX, placing elements at the end of the array

Sorting by Divide & Conquer

The Divide & Conquer Strategy

Generic Strategy

Given an algorithmic task:

1. Break the input into smaller instances of the task
2. Solve the smaller instances
   - this is typically recursive!
3. Combine smaller solutions to a solution to the whole task

Divide & Conquer Sorting

MergeSort: Divide by *index*

- divide array into left and right halves
- recursively sort halves
- merge halves

QuickSort: Divide by *value*

- pick a *pivot value* p
- split array according to p
  - $\leq p$ on left, $> p$ on right
- recursively sort sub-arrays

Merging Sorted Arrays

Question

Suppose we are given two **sorted arrays**, *a* and *b*. How can we merge them into a single sorted array that contains all the values from both arrays?

0	1	2	3	4
2	3	6	7	8

0	1	2	3
1	4	5	9

0	1	2	3	4	5	6	7	8

Merging Code

Merging *sorted* arrays a (size m) and b (size n) into array c starting at index s

```
procedure MERGE(a, b, c, s, m, n)    ▷ Merge arrays a and b into array c starting at index s. a has size m and b has size n
    i, j ← 0, k ← s
    while k < s + m + n do
        if j = n or a[i] < b[j] then
            c[k] ← a[i]
            i ← i + 1
        else
            c[k] ← b[j]
            j ← j + 1
        end if
        k ← k + 1
    end while
end procedure
```

Merging Code

PollEverywhere

What is the running time of MERGE?

1. $\Theta(m+n)$
2. $\Theta(m \cdot n)$
3. $\Theta(\log(m+n))$
4. $\Theta(\log mn)$

pollev.com/comp526

```
procedure MERGE(a, b, c, s, m, n)    ▷ Merge arrays a and b into array c starting at index s. a has size m and b has size n
    i, j ← 0, k ← s
    while k < s + m + n do
        if j = n or a[i] < b[j] then
            c[k] ← a[i]
            i ← i + 1
        else
            c[k] ← b[j]
            j ← j + 1
        end if
        k ← k + 1
    end while
end procedure
```

Sorting by Merging

MergeSortStrategy:

- To sort $a[i \dots k]$:
  - If $i = k$, then we're done
  - Otherwise split (sub)interval in half
  - Recursively sort halves
  - Merge sorted halves
    - copy values to new arrays for this

Sorting by Merging

MergeSortStrategy:

- To sort $a[i \ldots k]$:
  - If $i = k$, then we're done
  - Otherwise split (sub)interval in half
  - Recursively sort halves
  - Merge sorted halves
    - copy values to new arrays for this

procedure MergeSort(a, i, k)
    if $i < k$ **then**
        $j \leftarrow \lfloor (i + k)/2 \rfloor$
        MergeSort(a, i, j)
        MergeSort$(a, j + 1, k)$
        $b \leftarrow$ Copy(a, i, j)
        $c \leftarrow$ Copy$(a, j + 1, k)$
        Merge(b, c, a, i)
    end if
end procedure

Sorting by Merging

PollEverywhere

Consider an execution of MergeSort$(a, 0, 3)$ where $a = [4, 2, 1, 3]$. How many total calls to MergeSort are executed (including the initial call)?

pollev.com/comp526

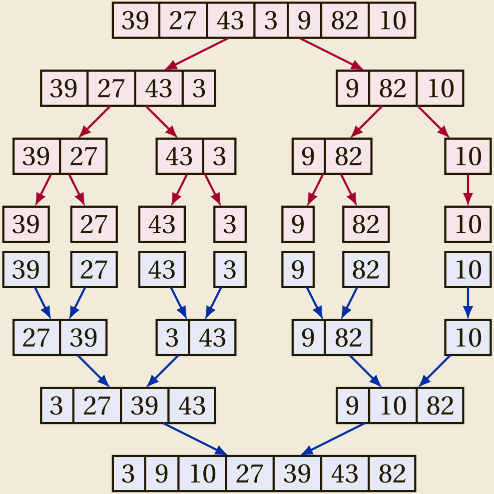
```
procedure MergeSort(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MergeSort(a, i, j)
        MergeSort(a, j + 1, k)
        b ← Copy(a, i, j)
        c ← Copy(a, j + 1, k)
        Merge(b, c, a, i)
    end if
end procedure
```

Sorting by Merging

Tracing the Recursive Calls

```
procedure MergeSort(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MergeSort(a, i, j)
        MergeSort(a, j + 1, k)
        b ← Copy(a, i, j)
        c ← Copy(a, j + 1, k)
        Merge(b, c, a, i)
    end if
end procedure
```

A Larger Example




```
procedure MergeSort(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MergeSort(a, i, j)
        MergeSort(a, j + 1, k)
        b ← Copy(a, i, j)
        c ← Copy(a, j + 1, k)
        Merge(b, c, a, i)
    end if
end procedure
```

tikz code courtesy of SebGlav on `tex.stackexchange.com`

MergeSort Analysis

Question. What is the running time of MERGESORT?

PollEverywhere

What is the running time of MERGESORT?

1. $\Theta(n)$
2. $\Theta(n \log n)$
3. $\Theta(n^{3/2})$
4. $\Theta(n^2)$

pollev.com/comp526

```
procedure MERGESORT(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MERGESORT(a, i, j)
        MERGESORT(a, j + 1, k)
        b ← COPY(a, i, j)
        c ← COPY(a, j + 1, k)
        MERGE(b, c, a, i)
    end if
end procedure
```

Running Time of Recursive Functions

Question. How do we analyze the running time of recursively defined functions?

Running Time of Recursive Functions

Question. How do we analyze the running time of recursively defined functions?

General Approach. Write (and solve) a *recursive formula* for the running time:

- Define $T(n)$ to be the worst case running time of all instances of size n
- Find a (recursive) relationship between $T(n)$ and $T(n')$ with $n' < n$
- Solve the recursive function for T.

A Recursive Formula for MergeSort

General Approach. Write (and solve) a *recursive formula* for the running time

- Define $T(n)$ to be the worst case running time of all instances of size n
- How is $T(n)$ related to $T(n')$ for smaller values of n?

```
procedure MergeSort(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MergeSort(a, i, j)
        MergeSort(a, j + 1, k)
        b ← Copy(a, i, j)
        c ← Copy(a, j + 1, k)
        Merge(b, c, a, i)
    end if
end procedure
```

A Recursive Formula for MergeSort

General Approach. Write (and solve) a *recursive formula* for the running time

- Define $T(n)$ to be the worst case running time of all instances of size n
- How is $T(n)$ related to $T(n')$ for smaller values of n?
  - $T(n) = 2T(n/2) + cn$

```
procedure MergeSort(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MergeSort(a, i, j)
        MergeSort(a, j + 1, k)
        b ← Copy(a, i, j)
        c ← Copy(a, j + 1, k)
        Merge(b, c, a, i)
    end if
end procedure
```

A Recursive Formula for MergeSort

General Approach. Write (and solve) a *recursive formula* for the running time

- Define $T(n)$ to be the worst case running time of all instances of size n
- How is $T(n)$ related to $T(n')$ for smaller values of n?
  - $T(n) = 2T(n/2) + cn$
- How do we solve this **recursive formula**?

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&= 2(2T(n/4) + c(n/2)) + cn \\
&= 4T(n/4) + 2cn \\
&= \cdots
\end{aligned}
$$

```
procedure MergeSort(a, i, k)
    if i < k then
        j ← ⌊(i + k)/2⌋
        MergeSort(a, i, j)
        MergeSort(a, j + 1, k)
        b ← Copy(a, i, j)
        c ← Copy(a, j + 1, k)
        Merge(b, c, a, i)
    end if
end procedure
```

Inductive Argument

Proposition

Suppose that for all n, $T(n)$ satisfies $T(n) \leq 2T(n/2) + cn$ and $T(1) = O(1)$. Then $T(n) = O(n \log n)$.

Inductive Argument

Proposition

Suppose that for all n, $T(n)$ satisfies $T(n) \leq 2T(n/2) + cn$ and $T(1) = O(1)$. Then $T(n) = O(n \log n)$.

Proof.

We claim that for all k, $T(n) = 2^k T(n/2^k) + kcn$.

- The base case $k = 1$ is the hypothesis of the proposition.
- For the inductive step, apply inductive hypothesis along with the base case for $n' = n/2^k$.

□

Inductive Argument

Proposition

Suppose that for all n, $T(n)$ satisfies $T(n) \le 2T(n/2) + cn$ and $T(1) = O(1)$. Then $T(n) = O(n \log n)$.

Proof.

We claim that for all k, $T(n) = 2^k T(n/2^k) + kcn$.

- The base case $k = 1$ is the hypothesis of the proposition.
- For the inductive step, apply inductive hypothesis along with the base case for $n' = n/2^k$.

Now apply the claim for $k = \log n$:

- $T(n) \le 2^{\log n} T(n/2^{\log n}) + (\log n)cn = O(n \log n)$

□

Inductive Argument

Proposition

Suppose that for all n, $T(n)$ satisfies $T(n) \leq 2T(n/2) + cn$ and $T(1) = O(1)$. Then $T(n) = O(n \log n)$.

Consequence

The running time of MERGESORT is $O(n \log n)$

Inductive Argument

Proposition

Suppose that for all n, $T(n)$ satisfies $T(n) \leq 2T(n/2) + cn$ and $T(1) = O(1)$. Then $T(n) = O(n \log n)$.

Consequence

The running time of MERGESORT is $O(n \log n)$

Also, MERGESORT performs reasonably well on large arrays in practice:

- Good locality of reference in MERGE operations

Inductive Argument

Proposition

Suppose that for all n, $T(n)$ satisfies $T(n) \leq 2T(n/2) + cn$ and $T(1) = O(1)$. Then $T(n) = O(n \log n)$.

Consequence

The running time of MERGESORT is $O(n \log n)$

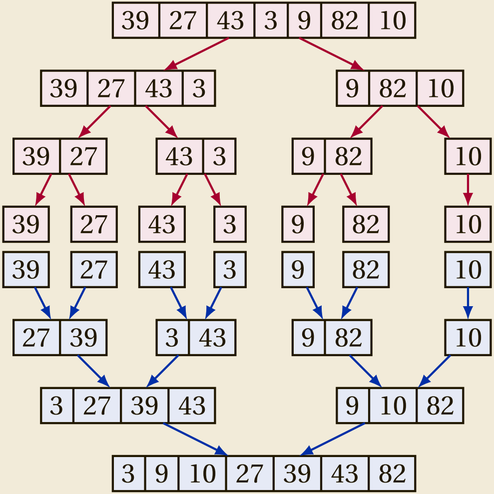
Also, MERGESORT performs reasonably well on large arrays in practice:

- Good locality of reference in MERGE operations

But MERGESORT operation requires $\Theta(m)$ additional space

- MERGE operation copies values

Visualizing the Argument



tikz code courtesy of SebGlav on tex.stackexchange.com

QuickSort

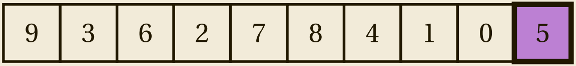
QuickSort: Dividing by Value

- The MergeSort algorithm divided arrays by **index**
- QuickSort divides arrays by **value**
    1. pick a **pivot value** p from the array
    2. **split** the array into sub-arrays
        - $a[1 \ldots j-1]$ stores values $\leq p$
        - $a[j \ldots n-1]$ stores values $> p$
    3. recursively sort $a[1 \ldots j-1]$ and $a[j \ldots n-1]$

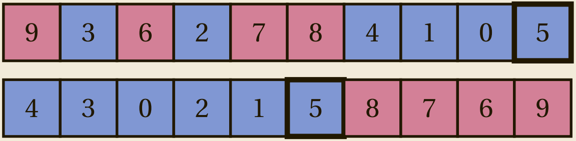
```
procedure QuickSort(a, min, max)
    p ← SelectPivot(a, min, max)
    j ← Split(a, min, max, p)
    QuickSort(a, min, j)
    QuickSort(a, j + 1, max)
end procedure
```

Visualizing QuickSort

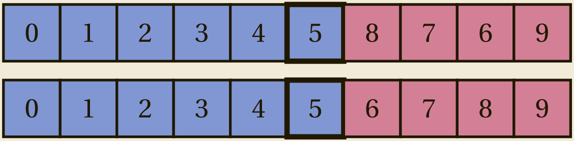
Select a pivot:



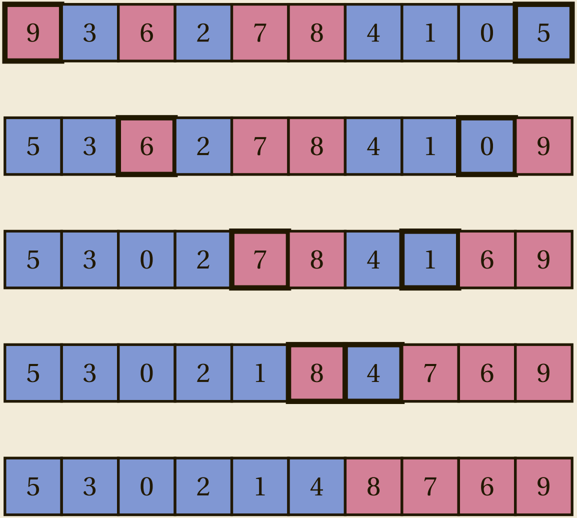
Split by pivot value:



Recursively sort left and right sides:



Hoare's Splitting Method



Splitting in Pseudocode

```
procedure SPLIT(a, min, max, p)
    i ← min
    j ← max
    while i < j do
        while a[i] ≤ p do
            i ← i + 1
        end while
        while a[j] > p do
            j ← j − 1
        end while
        SWAP(a, i, j)
    end while
    swap p into index i − 1
    return i − 1
end procedure
```

Splitting in Pseudocode

PollEverywhere

What is the running time of SPLIT$(a, \min, \max, p)$?

pollev.com/comp526

procedure SPLIT$(a, \min, \max, p)$
  $i \leftarrow \min$
  $j \leftarrow \max$
  while $i < j$ **do**
    while $a[i] \leq p$ **do**
      $i \leftarrow i + 1$
    end while
    while $a[j] > p$ **do**
      $j \leftarrow j - 1$
    end while
    SWAP(a, i, j)
  end while
  swap p into index $i - 1$
  return $i - 1$
end procedure

Splitting in Pseudocode

What is the running time of SPLIT(a, min, max, p)**?**

```
procedure SPLIT(a, min, max, p)
    i ← min
    j ← max
    while i < j do
        while a[i] ≤ p do
            i ← i + 1
        end while
        while a[j] > p do
            j ← j − 1
        end while
        SWAP(a, i, j)
    end while
    swap p into index i − 1
    return i − 1
end procedure
```

Running time of QuickSort?

PollEverywhere

What is the worst-case running time of QUICKSORT?

pollev.com/comp526

```
procedure QuickSort(a, min, max)
    p ← SelectPivot(a, min, max)
    j ← Split(a, min, max, p)
    QuickSort(a, min, j)
    QuickSort(a, j + 1, max)
end procedure
```

Running time of QuickSort?

The Worst Case:

- the pivot is the largest or smallest element in $a[\min \ldots \max]$.
- Then one of the recursive calls has size $\max - \min - 1$.
- The overall running time is then $\Omega(n^2)$.

```
procedure QuickSort(a, min, max)
    p ← SelectPivot(a, min, max)
    j ← Split(a, min, max, p)
    QuickSort(a, min, j)
    QuickSort(a, j + 1, max)
end procedure
```

No matter what:

- Each call to Split sorts at least one element (the pivot)
- Each call to QuickSort takes time $O(n)$
- $\implies$ Running time is $O(n^2)$

So the overall running time is $\Theta(n^2)$

Running time of QuickSort?

PollEverywhere

What is the **best-case** running time of QuickSort?

pollev.com/comp526

procedure QuickSort(a, min, max)
  $p \leftarrow$ SelectPivot(a, min, max)
  $j \leftarrow$ Split(a, min, max, p)
  QuickSort(a, min, j)
  QuickSort(a, $j+1$, max)
end procedure

Running time of QuickSort?

The Best Case Scenario:

- Each SPLIT partitions a perfectly in half
- Analysis as in MERGESORT
- $\Longrightarrow$ running time is $\Theta(n \log n)$

Bonus: QUICKSORT sorts *in-place*

- No extra arrays!

```
procedure QUICKSORT(a, min, max)
    p ← SELECTPIVOT(a, min, max)
    j ← SPLIT(a, min, max, p)
    QUICKSORT(a, min, j)
    QUICKSORT(a, j + 1, max)
end procedure
```

Random Pivot Selection

Suppose we choose each pivot **randomly**:

- SELECTPIVOT$(a, \min, \max)$ returns $a[i]$ where i is chosen *uniformly* from $\{\min, \min + 1, \ldots, \max\}$

Random Pivot Selection

Suppose we choose each pivot **randomly**:

- $\textsc{SelectPivot}(a, \min, \max)$ returns $a[i]$ where i is chosen *uniformly* from $\{\min, \min+1, \ldots, \max\}$

Intuition:

- A randomly chosen pivot is "reasonably likely" to be "close" to the **median** value
  - with probability 1/2 p will be in the middle half of the values
- Perhaps this is enough to get a *typical* running time of $O(n \log n)$?

Random Pivot Selection

Suppose we choose each pivot **randomly**:

- SELECTPIVOT$(a, \min, \max)$ returns $a[i]$ where i is chosen *uniformly* from $\{\min, \min + 1, \ldots, \max\}$

Theorem

The **expected** running time of QUICKSORT with random pivot selection is $O(n \log n)$.

- This expectation is over the **randomness of the algorithm**, not the input

$\Longrightarrow$ (Expected) guarantee holds for *all* arrays

Random Pivot Selection

Theorem

The **expected** running time of QUICKSORT with random pivot selection is $O(n \log n)$.

Proof.

Analyze the comparisons made by QUICKSORT:

- Write the values in a as $a_1 \le a_2 \le \cdots \le a_n$
- Define $X_{ij} = 1$ if a_i and a_j are compared in an execution

□

Random Pivot Selection

Theorem

The **expected** running time of QUICKSORT with random pivot selection is $O(n \log n)$.

Proof.

Analyze the comparisons made by QUICKSORT:

- Write the values in a as $a_1 \leq a_2 \leq \cdots \leq a_n$
- Define $X_{ij} = 1$ if a_i and a_j are compared in an execution
- $X_{ij} = 1$ only if a_i or a_j is chosen in pivot in SPLIT that separates a_i and a_j
- This happens with probability $p_{ij} = 2/(j - i + 1)$

□

Random Pivot Selection

Theorem

The **expected** running time of QUICKSORT with random pivot selection is $O(n \log n)$.

Proof.

Analyze the comparisons made by QUICKSORT:

- Write the values in a as $a_1 \leq a_2 \leq \cdots \leq a_n$
- Define $X_{ij} = 1$ if a_i and a_j are compared in an execution
- $X_{ij} = 1$ only if a_i or a_j is chosen in pivot in SPLIT that separates a_i and a_j
- This happens with probability $p_{ij} = 2/(j - i + 1)$
- This contributes $\mathbf{E}(X_{ij}) = p_{ij}$ comparisons in expectation
- Summing over all i and j we get the expected number of comparisons to be $\mathbf{E}\left(\sum_{j=1}^{n} \sum_{i<j} p_{ij}\right) = O(n \log n)$ (Use $\sum_{k=1}^{n} 1/k = \Theta(\log n)$)

□

Sorting So Far

Elementary Sorting

$\Theta(n^2)$ worst case

- SELECTIONSORT
- BUBBLESORT
- INSERTIONSORT

Faster Sorting

$\Theta(n \log n)$ worst case

- HEAPSORT
- MERGESORT

Good in Practice?

$\Theta(n^2)$ worst case

$\Theta(n \log n)$ in expectation

- QUICKSORT

Question

Can we sort in time $o(n \log n)$?

Comparison Based Sorting

High-level view of (sorting) algorithms (. . . so far)

- Access input, an array a
- *Compare* values of a:
  - if $a[i] \leq a[j]$ do something
  - otherwise do something else
- These are **comparison based algorithms**

Comparison Based Sorting

High-level view of (sorting) algorithms (…so far)

- Access input, an array a
- *Compare* values of a:
  - if $a[i] \leq a[j]$ do something
  - otherwise do something else
- These are **comparison based algorithms**

Consider

- **any** comparison based sorting algorithm A
- **every** possible input a to A where a stores distinct values between 1 and n.
  - $P_n = \{a \mid a \text{ contains distinct elements from 1 to } n\}$
  - $|P_n| = n! = n \cdot (n-1) \cdot (n-2) \cdots 1$

Question. How does A distinguish between $a, b \in P_n$?

Decision Trees

For a comparison based algorithm A a binary tree T_A:

- vertices labelled with
  - a comparison $a[i] <= a[j]$ performed by A
  - a subset of inputs
- root labels are (1) first comparison made by A, and (2) P_n
- each child corresponds to an **outcome** of comparison at parent node
  - left child labelled with TRUE inputs & next comparison
  - right child labelled with FALSE inputs & next comparison
- leaf vertices correspond to completed computations

Example: InsertionSort

procedure INSERTIONSORT(a, n)
  for $i = 1, 2, \ldots, n-1$ **do**
    $j \leftarrow i$
    while $j > 0$ and $a[j] < a[j-1]$ **do**
      SWAP($a, j, j-1$)
      $j \leftarrow j-1$
    end while
  end for
end procedure

Example: InsertionSort

Unwrapping the Loops for $n = 3$

1. $a[2] < a[1]$
2. $a[3] < a[2]$
   2.1 if yes, check $a[2] < a[1]$ (after SWAP)

```
procedure InsertionSort(a, n)
    for i = 1, 2, ..., n − 1 do
        j ← i
        while j > 0 and a[j] < a[j − 1] do
            Swap(a, j, j − 1)
            j ← j − 1
        end while
    end for
end procedure
```

Example: InsertionSort

Unwrapping the Loops for $n = 3$

1. $a[2] < a[1]$
2. $a[3] < a[2]$
   2.1 if yes, check $a[2] < a[1]$ (after SWAP)

```
procedure InsertionSort(a, n)
    for i = 1, 2, ..., n − 1 do
        j ← i
        while j > 0 and a[j] < a[j − 1] do
            Swap(a, j, j − 1)
            j ← j − 1
        end while
    end for
end procedure
```

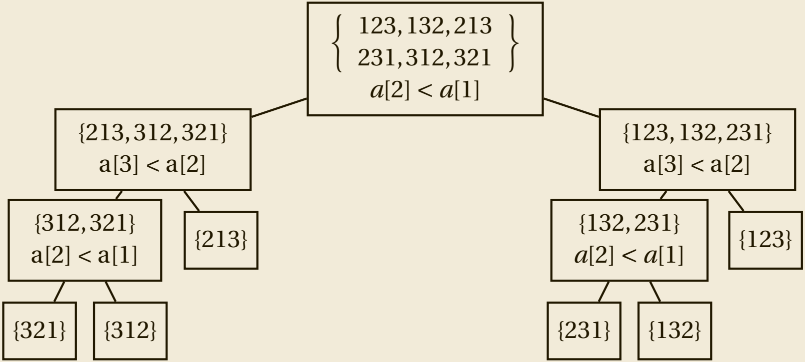
Decision tree structure

- Start with all inputs $S = \{123, 132, 213, 231, 312, 321\}$
- Apply comparison 1:
  - $S_T = \{213, 312, 321\} \mapsto \{123, 132, 231\}$, then apply comparison 2
    - $S_{TT} = \{312, 321\} \mapsto \{123, 213\}$
    - $S_{TF} = \{213\} \mapsto \{123\}$
  - $S_F = \{123, 132, 231\}$, then apply comparison 2
    - $S_{FT} = \{132, 231\} \mapsto \{123, 213\}$
    - $S_{FF} = \{123\}$

InsertionSort Decision Tree

Note the set labels are sets of **inputs**

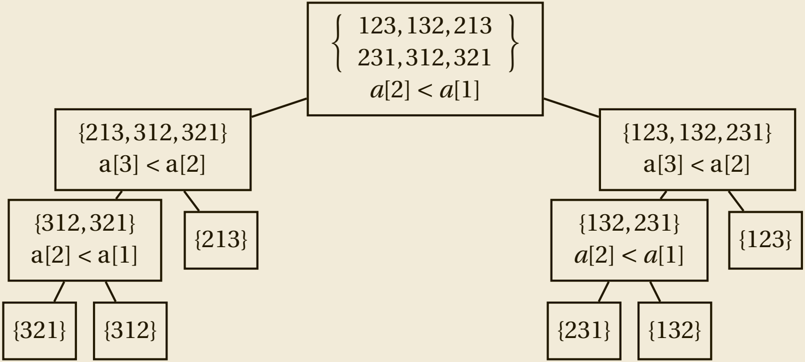
- InsertionSort **updates** the arrays as it executes the decision tree
- The comparisons are applied to the **updated** arrays



InsertionSort Decision Tree

Note the set labels are sets of **inputs**

- InsertionSort **updates** the arrays as it executes the decision tree
- The comparisons are applied to the **updated** arrays



Observation. Every *leaf* has corresponds to a unique input. ***Why?***

Comparison Based Lower Bounds

Obsevation 1. If arrays a and b are in the same label at a vertex v at depth d in T_A then:

- first d comparisons in a and b had same results
- A performed same operations on a and b

Comparison Based Lower Bounds

Obsevation 1. If arrays a and b are in the same label at a vertex v at depth d in T_A then:

- first d comparisons in a and b had same results
- A performed same operations on a and b

Observation 2. If $a \neq b$ and a *leaf* of T_A is labelled with both a and b then A did not sort *both* a and b.

Comparison Based Lower Bounds

Obsevation 1. If arrays a and b are in the same label at a vertex v at depth d in T_A then:

- first d comparisons in a and b had same results
- A performed same operations on a and b

Observation 2. If $a \neq b$ and a *leaf* of T_A is labelled with both a and b then A did not sort *both* a and b.

Consequence. If A sorts all arrays in P_A, then T_A must have at least $|P_A| = n!$ leaves.

Comparison Based Lower Bounds

Obsevation 1. If arrays a and b are in the same label at a vertex v at depth d in T_A then:

- first d comparisons in a and b had same results
- A performed same operations on a and b

Observation 2. If $a \neq b$ and a *leaf* of T_A is labelled with both a and b then A did not sort *both* a and b.

Consequence. If A sorts all arrays in P_A, then T_A must have at least $|P_A| = n!$ leaves.

Observation 3. A tree of depth d has at most 2^d leaves.

Comparison Based Lower Bounds

Obsevation 1. If arrays a and b are in the same label at a vertex v at depth d in T_A then:

- first d comparisons in a and b had same results
- A performed same operations on a and b

Observation 2. If $a \neq b$ and a *leaf* of T_A is labelled with both a and b then A did not sort *both* a and b.

Consequence. If A sorts all arrays in P_A, then T_A must have at least $|P_A| = n!$ leaves.

Observation 3. A tree of depth d has at most 2^d leaves.

Computation. Must have $2^n \geq n!$:

$$\implies \quad n \geq \log(n!) = \log(n) + \log(n-1) + \cdots + \log(2) + \log(1) = \Omega(n \log n)$$

Comparison Based Lower Bounds

Obsevation 1. If arrays a and b are in the same label at a vertex v at depth d in T_A then:

- first d comparisons in a and b had same results
- A performed same operations on a and b

Observation 2. If $a \neq b$ and a *leaf* of T_A is labelled with both a and b then A did not sort *both* a and b.

Consequence. If A sorts all arrays in P_A, then T_A must have at least $|P_A| = n!$ leaves.

Observation 3. A tree of depth d has at most 2^d leaves.

Computation. Must have $2^n \geq n!$:

$$\implies \quad n \geq \log(n!) = \log(n) + \log(n-1) + \cdots + \log(2) + \log(1) = \Omega(n \log n)$$

Theorem

Any comparison-based sorting algorithm requires $\Omega(n \log n)$ comparisons to sort arrays of length n in the worst case.

Next Time

- Non-comparison-based Sorting
  - Can we sort in $o(n \log n)$ time?
- Text Searching

Scratch Notes