Merge Sort & Quick Sort

Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $S$ in two disjoint subsets $S_1$ and $S_2$
  - Conquer:
    - Recur: solve the subproblems associated with $S_1$ and $S_2$
    - Combine: make the solutions for $S_1$ and $S_2$ into a solution for $S$
- The base case for the recursion are subproblems of size 0 or 1
Merge Sort

- Classical example of divide-and-conquer technique
- Problem: Given $n$ elements, sort elements into non-decreasing order
- Divide-and-Conquer:
  - If $n=1$ terminate (every one-element list is already sorted)
  - If $n>1$, partition elements into two or more sub-collections; sort each; combine into a single sorted list
- How do we partition?

Partitioning - Choice 1

- First $n-1$ elements into set $A$, last element into set $B$
- Sort $A$ using this partitioning scheme recursively
  - $B$ already sorted
- Combine $A$ and $B$ using method $\text{Insert()}$ (= insertion into sorted array)
- Leads to recursive version of $\text{InsertionSort()}$
  - Number of comparisons: $O(n^2)$
    - Best case = $n-1$
    - Worst case = $\left\lfloor \frac{n(n-1)}{2} \right\rfloor$
Partitioning - Choice 2

- Pick the element with largest key in B, remaining elements in A
- Sort A recursively
- To combine sorted A and B, append B to sorted A
  - Use Max() to find largest element → recursive SelectionSort()
  - Use bubbling process to find and move largest element to right-most position → recursive BubbleSort()
- All $O(n^2)$

Partitioning - Choice 3

- Let’s try to achieve balanced partitioning – that’s typical for divide-and-conquer.
- A gets $n/2$ elements, B gets the rest half
- Sort A and B recursively
- Combine sorted A and B using a process called *merge*, which combines two sorted lists into one
  - How? We will see soon
Merge-Sort

- **Merge-sort** is a sorting algorithm based on the divide-and-conquer paradigm.
- Like heap-sort:
  - It has $O(n \log n)$ running time.
- Unlike heap-sort:
  - It usually needs extra space in the merging process.
  - It accesses data in a sequential manner (suitable to sort data on a disk).

Merging

- The key to Merge Sort is merging two sorted lists into one, such that if you have two lists $X = (x_1 \leq x_2 \leq \cdots \leq x_m)$ and $Y = (y_1 \leq y_2 \leq \cdots \leq y_n)$, the resulting list is $Z = (z_1 \leq z_2 \leq \cdots \leq z_{m+n})$.
- Example:
  
  $L_1 = \{3, 8, 9\}$, $L_2 = \{1, 5, 7\}$
  
  $\text{merge}(L_1, L_2) = \{1, 3, 5, 7, 8, 9\}$
Merging

X: 3 10 23 54
Y: 1 5 25 75

Result:

Merging (cont.)

X: 3 10 23 54
Y: 5 25 75

Result: 1
Merging (cont.)

X: 10 23 54
Y: 5 25 75

Result: 1 3

Merging (cont.)

X: 10 23 54
Y: 25 75

Result: 1 3 5
Merging (cont.)

X: 23 54
Y: 25 75

Result: 1 3 5 10

Merging (cont.)

X: 54
Y: 25 75

Result: 1 3 5 10 23
Merging (cont.)

X: 1 3 5 10 23 25
Y: 54 75

Result: 1 3 5 10 23 25 54
Merging (cont.)

X: 10 23 25 54 75
Y: 1 3 5

Result: 1 3 5 10 23 25 54 75

Merging Two Sorted Sequences

- The combine step of merge-sort has to merge two sorted sequences $A$ and $B$ into a sorted sequence $S$ containing the union of the elements of $A$ and $B$.
- Merging two sorted sequences, each with $n$ elements, takes $O(2n)$ time.

Algorithm merge($S_1$, $S_2$, $S$):

Input: Two arrays, $S_1$ and $S_2$, of size $n_1$ and $n_2$, respectively, sorted in non-decreasing order, and an empty array, $S$, of size at least $n_1 + n_2$

Output: $S$, containing the elements from $S_1$ and $S_2$ in sorted order

1. $i \leftarrow 1$
2. $j \leftarrow 1$
3. while $i \leq n_1$ and $j \leq n_2$ do
   4. if $S_1[i] \leq S_2[j]$ then
      5. $S[i+j-1] \leftarrow S_1[i]$
      6. $i \leftarrow i+1$
   7. else
      8. $S[i+j-1] \leftarrow S_2[j]$
      9. $j \leftarrow j+1$
10. while $i \leq n_1$ do
    11. $S[i+j-1] \leftarrow S_1[i]$
    12. $i \leftarrow i+1$
13. while $j \leq n_2$ do
    14. $S[i+j-1] \leftarrow S_2[j]$
    15. $j \leftarrow j+1$
Implementing Merge Sort

- There are two basic ways to implement merge sort:
  - **In Place:** Merging is done with only the input array
    - Pro: Requires only the space needed to hold the array
    - Con: Takes longer to merge because if the next element is in the right side then all of the elements must be moved down.
  - **Double Storage:** Merging is done with a temporary array of the same size as the input array.
    - Pro: Faster than In Place since the temp array holds the resulting array until both left and right sides are merged into the temp array, then the temp array is appended over the input array.
    - Con: The memory requirement is doubled.

Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1
Execution Example

- Partition

Execution Example (cont.)

- Recursive call, partition
Execution Example (cont.)

- Recursive call, partition

```
7 2 9 4 | 3 8 6 1
```

Execution Example (cont.)

- Recursive call, base case

```
7 2 9 4 | 3 8 6 1
```
Execution Example (cont.)

- Recursive call, base case

Execution Example (cont.)

- Merge
Execution Example (cont.)

- Recursive call, ..., base case, merge

Execution Example (cont.)

- Merge
Execution Example (cont.)

- Recursive call, ..., merge, merge

Execution Example (cont.)

- Merge
The Merge-Sort Algorithm

- Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:
  - **Divide:** partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each.
  - **Recur:** recursively sort $S_1$ and $S_2$.
  - **Combine:** merge $S_1$ and $S_2$ into a unique sorted sequence.

Algorithm $mergeSort(S)$

**Input** sequence $S$ with $n$ elements

**Output** sequence $S$ sorted according to $C$

1. if $S.size() > 1$
   - $(S_1, S_2) \leftarrow partition(S, n/2)$
   - $mergeSort(S_1)$
   - $mergeSort(S_2)$
   - $S \leftarrow merge(S_1, S_2)$

Analysis of Merge-Sort

- The height $h$ of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence
- The overall amount or work done at the nodes of depth $i$ is $O(n)$
  - we partition and merge $2^i$ sequences of size $n/2^i$
  - we make $2^{i+1}$ recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$
Evaluation

- Recurrence equation:
- Assume \( n \) is a power of 2

\[
T(n) = \begin{cases} 
  c_1 & \text{if } n=1 \\
  2T(n/2) + c_2n & \text{if } n>1, \; n=2^k
\end{cases}
\]

Solution

By Substitution:

\[
T(n) = 2T(n/2) + c_2n
\]

\[
T(n/2) = 2T(n/4) + c_2n/2
\]

\[
\vdots
\]

\[
T(n) = 4T(n/4) + 2c_2n
\]

\[
T(n) = 8T(n/8) + 3c_2n
\]

\[
\vdots
\]

\[
T(n) = 2^iT(n/2^i) + ic_2n
\]

Assuming \( n = 2^k \), expansion halts when we get \( T(1) \) on right side; this happens when \( i=k \)  \( T(n) = 2^kT(1) + kc_2n \)

Since \( 2^k = n \), we know \( k = \log(n) \); since \( T(1) = c_1 \), we get

\[
T(n) = c_1n + c_2n\log n
\]

thus an upper bound for \( T_{\text{mergeSort}}(n) \) is \( O(n\log n) \)
Variants and Applications

- There are other variants of Merge Sorts including bottom-up merge sort, k-way merge sorting, natural merge sort.
- Bottom-up merge sort eliminates the divide process and assumes all subarrays have $2^k$ elements for some $k$, with exception of the last subarray.
- Natural merge sort is known to be the best for nearly sorted inputs. Sometimes, it takes only $O(n)$ for some inputs.
- Merge sort’s double memory demands makes it not very practical when main memory is in short supply.
- Merge sort is the major method for external sorting, parallel algorithms, and sorting circuits.

Natural Merge Sort

- Identify sorted (or reversely sorted) sub-lists in the input (each is called a run).
- Merge all runs into one.
- Example:
  - Input $A = [10, 6, 2, 3, 5, 7, 3, 8]$
  - Three (reversed) runs: $[(10, 6, 2), (3, 5, 7), (3, 8)]$
  - Reverse the reversed: $[(2, 6, 10), (3, 5, 7), (3, 8)]$
  - Merge them in one: $[2, 3, 4, 5, 6, 7, 8, 10 ]$
  - It takes $O(n)$ to sort $[n, n-1, ..., 3, 2, 1].$
- A good implementation of natural merge sort is called Timsort.
External Sorting

- Using secondary storage effectively
- General Wisdom:
  - I/O costs dominate
  - Design algorithms to reduce I/O

2-Way Sort: Requires 3 Buffers

- **Phase 1: PREPARE.**
  - Read a page, sort it, write it.
  - only one buffer page is used

- **Phase 2, 3, ..., etc.: MERGE:**
  - Three buffer pages used.
Two-Way External Merge Sort

- **Idea:** Divide and conquer: sort subfiles and merge into larger sorts
- **Run:** a list of sorted items

Pass 0 ➔ Only one memory block is needed

Pass i > 0 ➔ Only three memory blocks are needed

Notice: We ignored the CPU cost to sort a block in memory or merge two blocks
Two-Way External Merge Sort

- Each pass we read/write each page in file.
- N pages in file => 2N
- Number of passes
  \[ = \left\lceil \log_2 N \right\rceil + 1 \]
- So total cost is:
  \[ 2N\left(\left\lceil \log_2 N \right\rceil + 1\right) \]

External Merge Sort

- What if we had more buffer pages?
- How do we utilize them wisely?
  \[ \rightarrow \text{Two main ideas!} \]
Phase 1: Prepare

- Construct as large as possible starter lists.
- ➔ Will reduce the number of needed passes

Phase 2: Merge

- Merge as many sorted sublists into one long sorted list.
- Keep 1 buffer for the output
- Use B-1 buffers to read from B-1 lists
General External Merge Sort

How can we utilize more than 3 buffer pages?

To sort a file with \( N \) pages using \( B \) buffer pages:

- Pass 0: use \( B \) buffer pages.
  Produce \( \left\lceil \frac{N}{B} \right\rceil \) sorted runs of \( B \) pages each.
- Pass 1, 2, ..., etc.: merge \( B-1 \) runs.

Cost of External Merge Sort

- Number of passes: \( 1 + \left\lceil \log_{B-1} \left( \frac{N}{B} \right) \right\rceil \)
- Cost = \( 2N \times (\# \text{ of passes}) \)
Example
- Buffer: with 5 buffer pages
- File to sort: 108 pages

- Pass 0:
  - Size of each run?
  - Number of runs?

- Pass 1:
  - Size of each run?
  - Number of runs?

- Pass 2: ???

Example
- Buffer: with 5 buffer pages
- File to sort: 108 pages

- Pass 0: \( \left\lceil \frac{108}{5} \right\rceil = 22 \) sorted runs of 5 pages each (last run is only 3 pages)
- Pass 1: \( \left\lceil \frac{22}{4} \right\rceil = 6 \) sorted runs of 20 pages each (last run is only 8 pages)
- Pass 2: 2 sorted runs, 80 pages and 28 pages
- Pass 3: Sorted file of 108 pages

- Total I/O costs: ?
Summary of External Sorting

- External sorting is important;
- Choose best internal sorting in Pass 0.
- **External merge sort minimizes disk I/O costs:**
  - **Two-Way External Sorting**
    - Only 3 memory buffers are needed
  - **Multi-Way External Sorting**
    - Use large number of memory buffers available
  - **Cost:** $1 + \left\lceil \log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil$

Example: 2-Way Merge for 20 Pages

Number of passes = 5
Example: 5-Way Merge for 20 Pages

- gain of utilizing all available buffers
- importance of a high fan-in during merging

<table>
<thead>
<tr>
<th>#Pages in File</th>
<th>N</th>
<th>B=3</th>
<th>B=5</th>
<th>B=9</th>
<th>B=17</th>
<th>B=129</th>
<th>B=257</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1,000</td>
<td>1,000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>10,000</td>
<td>10,000</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100,000</td>
<td>100,000</td>
<td>17</td>
<td>9</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1,000,000</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10,000,000</td>
<td>10,000,000</td>
<td>23</td>
<td>12</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>100,000,000</td>
<td>100,000,000</td>
<td>26</td>
<td>14</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1,000,000,000</td>
<td>1,000,000,000</td>
<td>30</td>
<td>15</td>
<td>10</td>
<td>8</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>stable, in-place, for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>stable, in-place, for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>non-stable, in-place, for large data sets (1K — 1M)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>stable, sequential data access, for huge data sets (&gt; 1M)</td>
</tr>
</tbody>
</table>

Quick-Sort

- Quick-sort is also a sorting algorithm based on the divide-and-conquer paradigm:
  - **Divide**: pick a random element $x$ (called pivot) and partition $S$ into
    - $L$ elements less than $x$
    - $E$ elements equal $x$
    - $G$ elements greater than $x$
  - **Recur**: sort $L$ and $G$
  - **Combine**: join $L$, $E$ and $G$
- Alternative: merge $E$ into $L$ or $G$. 
Quicksort Algorithm

Given an array of $n$ elements (e.g., integers):
- If array only contains one element, return
- Else
  - pick one element to use as *pivot*.
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

Partitioning Array

Given a pivot, partition the elements of the array such that the resulting array consists of:
1. One sub-array that contains elements $\geq$ pivot
2. Another sub-array that contains elements $<$ pivot

The sub-arrays are stored in the original data array.

Partitioning loops through, swapping elements below/above pivot.
Partition using Lists

- We partition an input sequence as follows:
  - We remove, in turn, each element \( y \) from \( S \) and
  - We insert \( y \) into \( L \), \( E \) or \( G \) depending on the result of the comparison with the pivot \( x \).
- Each insertion is at the end of a sequence, and hence takes \( O(1) \) time.
- Thus, the partition step of quick-sort takes \( O(n) \) time.
- Pro: stable sort
- Cons: not in place

Algorithm \( \text{partition}(S, p) \)

<table>
<thead>
<tr>
<th>Input sequence ( S ), position ( p ) of pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output subsequences ( L, E, G ) of the elements of ( S ) less than, equal to, or greater than the pivot, resp.</td>
</tr>
<tr>
<td>( L, E, G ) ( \leftarrow ) empty sequences</td>
</tr>
<tr>
<td>( x ) ( \leftarrow ) ( S ).remove((p) )</td>
</tr>
<tr>
<td>while ( \sim ) ( S ).isEmpty()</td>
</tr>
<tr>
<td>( y ) ( \leftarrow ) ( S ).remove((S \text{.first()}) )</td>
</tr>
<tr>
<td>if ( y &lt; x )</td>
</tr>
<tr>
<td>( L ).addLast((y) )</td>
</tr>
<tr>
<td>else if ( y = x )</td>
</tr>
<tr>
<td>( E ).addLast((y) )</td>
</tr>
</tbody>
</table>
| else \{ \( y > x \} \)
| \( G \).addLast\((y) \) |
| return \( L, E, G \) |

In-Place Partitioning Example

We are given array of \( n \) integers to sort:

\[
\begin{array}{cccccccc}
40 & 20 & 10 & 80 & 60 & 50 & 7 & 30 & 100
\end{array}
\]
Pick Pivot Element

There are a number of ways to pick the pivot element. In this example, we will use the first element in the array:

```
[40, 20, 10, 80, 60, 50, 7, 30, 100]
```

```
[0] [1] [2] [3] [4] [5] [6] [7] [8]
```

```
too_big_index  too_small_index
```

```
pivot_index = 0
```

```
[0] [1] [2] [3] [4] [5] [6] [7] [8]
```

```
too_big_index  too_small_index
```

1. While $\text{data}[\text{too_big_index}] \leq \text{data}[\text{pivot}]$
   
   $++\text{too_big_index}$
1. While data[too_big_index] <= data[pivot]
   ++too_big_index

2. While data[too_small_index] > data[pivot]
   --too_small_index
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. If too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

Line 5 is optional
### Partition Result

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>20</td>
<td>10</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>80</td>
<td>100</td>
</tr>
<tr>
<td>[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>[4]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;=</td>
<td>data[pivot]</td>
<td></td>
<td></td>
<td>=&gt;</td>
<td>data[pivot]</td>
</tr>
</tbody>
</table>

Recursive calls on two sides to get a sorted array.

### Quick-Sort Tree

- An execution of quick-sort is depicted by a binary tree
- Each node represents a recursive call of quick-sort and stores:
  - Unsorted sequence before the execution and its pivot
  - Sorted sequence at the end of the execution
- The root is the initial call
- The leaves are calls on subsequences of size 0 or 1

```
7 4 9 6 2  \rightarrow 2 4 6 7 9
```

```
4 2  \rightarrow 2 4
```

```
7 9  \rightarrow 7 9
```

```
2  \rightarrow 2
```

```
9  \rightarrow 9
```
Execution Example

- Pivot selection

```
7 2 9 4 3 7 6 1
```

Execution Example (cont.)

- Partition, recursive call, pivot selection

```
7 2 9 4 3 7 6 1
2 4 3 1
```
Execution Example (cont.)

- Partition, recursive call, base case

```
7 2 9 4 3 7 6 1
```

Execution Example (cont.)

- Recursive call, ..., base case, join

```
7 2 9 4 3 7 6 1
```
Execution Example (cont.)

- Recursive call, pivot selection

- Partition, ..., recursive call, base case
Execution Example (cont.)

- **Join**

```
7 2 9 4 3 7 6 1 → 1 2 3 4 6 7 7 9
```

```plaintext
2 4 3 1 → 1 2 3 4
7 9 7 → 7 7 9
```

```
1 → 1
4 3 → 3 4
9 → 9
```

```
1 → 1
4 → 4
```

Quicksort Analysis

- In-place quick-sort is not stable.
- Stable quick-sort is not in-place.
- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree?
Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size $n/2$
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? $O(n)$

Best case running time: $O(n \log_2 n)$
Worst case running time?
Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: \(O(n \log_2 n)\)
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size \(n-1\)
    2. Quicksort each sub-array
  - Depth of recursion tree?

Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element
- One of \(L\) and \(G\) has size \(n - 1\) and the other has size 0
- The running time is proportional to the sum
  \[ n + (n - 1) + \ldots + 2 + 1 \]
- Thus, the worst-case running time of quick-sort is \(O(n^2)\)
QuickSort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size $n-1$
    2. QuickSort each sub-array
  - Depth of recursion tree? $O(n)$
  - Number of accesses per partition? $O(n)$

Worst case running time: $O(n^2)$!!!
Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time: $O(n^2)$!!!
- What can we do to avoid worst case?
  - Randomly pick a pivot

Quicksort Analysis

- Bad divide: $T(n) = T(1) + T(n-1)$  -- $O(n^2)$
- Good divide: $T(n) = T(n/2) + T(n/2)$  -- $O(n \log_2 n)$
- Random divide: Suppose on average one bad divide followed by one good divide.
  - $T(n) = T(1) + T(n-1) = T(1) + 2T((n-1)/2)$
  - $T(n) = c + 2T((n-1)/2)$ is still $O(n \log_2 n)$
Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size \( s \).
  - **Good call**: the sizes of \( L \) and \( G \) are each less than \( 3s/4 \).
  - **Bad call**: one of \( L \) and \( G \) has size greater than \( 3s/4 \).

- **Good call**
  - Example: 7 2 9 4 3 7 6 1
  - 2 4 3 1 \( \rightarrow \) 7 9 7

- **Bad call**
  - Example: 7 2 9 4 3 7 6 1
  - 7 2 9 4 3 7 6 \( \rightarrow \) 1

- A call is **good** with probability \( 1/2 \):
  - \( 1/2 \) of the possible pivots cause good calls:
    - 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    - Bad pivots: 7 2 9 4 3 7 6
    - Good pivots: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
    - Bad pivots: 7 2 9 4 3 7 6

Expected Running Time, Part 2

- **Probabilistic Fact**: The expected number of coin tosses required in order to get \( k \) heads is \( 2^k \).
- For a node of depth \( i \), we expect:
  - \( i/2 \) ancestors are good calls
  - The size of the input sequence for the current call is at most \( (3/4)^i n \)

- Therefore, we have:
  - For a node of depth \( 2\log_{4/3} n \), the expected input size is one
  - The expected height of the quick-sort tree is \( O(\log n) \)
- The amount or work done at the nodes of the same depth is \( O(n) \)
- Thus, the expected running time of quick-sort is \( O(n \log n) \)

- Total expected time: \( O(n \log n) \)
Randomized Guarantees

- Randomization is a very important and useful idea. By either picking a random pivot or scrambling the permutation before sorting it, we can say:
  - “With high probability, randomized quicksort runs in $O(n \log n)$ time.”
- Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity.
- The worst-case is still there, but we almost certainly won’t see it.

In-Place 3-Way Partitioning

- Dijkstra’s 3-way partitioning (Holland flag problem)

```
<p | =p | ?......................? | >p
```

- Bentley-McIlroy’s 3-way partitioning

```
=p | <p | ?......................? | >p | =p
```

```
<p | =p | >p
```
In-Place 3-Way Randomized Quick-Sort

- Quick-sort can be implemented to run in-place.
- In the partition step, we use replace operations to rearrange the elements of the input sequence such that:
  - the elements less than the pivot have rank less than $h$.
  - the elements equal to the pivot have rank between $h$ and $k$.
  - the elements greater than the pivot have rank greater than $k$.
- The recursive calls consider:
  - elements with rank less than $h$.
  - elements with rank greater than $k$.

Algorithm `inPlaceQuickSort(S, l, r)`

- **Input**: sequence $S$, ranks $l$ and $r$
- **Output**: sequence $S$ with the elements of rank between $l$ and $r$ rearranged in increasing order

  ```
  if $l \geq r$
  return
  
  i ← a random integer between $l$ and $r$
  p ← $S$.elemAtRank($i$)
  (h, k) ← inPlace3WayPartition($p$)
  inPlaceQuickSort($S$, $l$, $h - 1$)
  inPlaceQuickSort($S$, $k + 1$, $r$)
  ```

Improved Pivot Selection

Pick median value of three elements from data array:
- $data[0]$, $data[n/2]$, and $data[n-1]$.

Use this median value as pivot.

For large arrays, use the median of three medians from:
- $\{data[0], data[n/8], data[2n/8]\}$, $\{data[3n/8], data[4n/8], data[5n/8]\}$, and $\{data[6n/8], data[7n/8], data[n-1]\}$.
Improving Performance of Quicksort

- Improved selection of pivot.
- For sub-arrays of size 100 or less, apply brute force search, or insert-sort.
  - Sub-array of size 1: trivial
  - Sub-array of size 2:
    - if(data[first] > data[second]) swap them
  - Sub-array of size 100 or less: call insert-sort.

- Improved selection of pivot.
- For sub-arrays of size 100 or less, apply brute force search, e.g., insert-sort.
- Test if the sub-array is already sorted before recursive calls.
### Stability of sorting algorithms

A **STABLE** sort preserves relative order of records with equal keys.

**Sorted on first key:**

<table>
<thead>
<tr>
<th>Name</th>
<th>First Key</th>
<th>Last Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>1</td>
<td>346</td>
</tr>
<tr>
<td>Ben</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>Chad</td>
<td>3</td>
<td>123</td>
</tr>
<tr>
<td>Fox</td>
<td>0</td>
<td>123</td>
</tr>
<tr>
<td>Maria</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>Obama</td>
<td>1</td>
<td>123</td>
</tr>
<tr>
<td>Parker</td>
<td>1</td>
<td>123</td>
</tr>
<tr>
<td>Will</td>
<td>1</td>
<td>22</td>
</tr>
</tbody>
</table>

Records with key value 3 are not in order on first key!!

**Sort the first file on second key:**

<table>
<thead>
<tr>
<th>Name</th>
<th>First Key</th>
<th>Last Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>Ben</td>
<td>1</td>
<td>346</td>
</tr>
<tr>
<td>Chad</td>
<td>2</td>
<td>346</td>
</tr>
<tr>
<td>Fox</td>
<td>1</td>
<td>346</td>
</tr>
<tr>
<td>Maria</td>
<td>1</td>
<td>346</td>
</tr>
<tr>
<td>Obama</td>
<td>1</td>
<td>346</td>
</tr>
<tr>
<td>Parker</td>
<td>1</td>
<td>346</td>
</tr>
<tr>
<td>Will</td>
<td>1</td>
<td>346</td>
</tr>
</tbody>
</table>

### Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection-sort</td>
<td>$O(n^2)$</td>
<td>in-place, stable, slow (not good for any inputs)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>in-place, stable, slow (good for small inputs)</td>
</tr>
<tr>
<td>quick-sort</td>
<td>$O(n \log n)$ expected</td>
<td>in-place, not stable, fastest (good for large inputs)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>in-place, not stable, fast (good for large inputs)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>not in-place, stable, fast (good for huge inputs)</td>
</tr>
</tbody>
</table>
Divide and Conquer

<table>
<thead>
<tr>
<th></th>
<th>Simple Divide</th>
<th>Fancy Divide</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uneven Divide</td>
<td>Insert Sort</td>
<td>Selection Sort</td>
</tr>
<tr>
<td>1 vs n-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Even Divide</td>
<td>Merge Sort</td>
<td>Quick Sort</td>
</tr>
<tr>
<td>n/2 vs n/2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Comparison-Based Sorting

- Many sorting algorithms are comparison based.
  - They sort by making comparisons between pairs of objects
  - Examples: bubble-sort, selection-sort, insertion-sort, heap-sort, merge-sort, quick-sort, ...
- Let us therefore derive a lower bound on the running time of any algorithm that uses comparisons to sort $n$ elements, $x_1, x_2, \ldots, x_n$.

Is $x_i < x_j$?

- If yes, then...
- If no, then...
How Fast Can We Sort?

- Selection Sort, Bubble Sort, Insertion Sort: $O(n^2)$
- Heap Sort, Merge sort: $O(n \log n)$
- Quicksort: $O(n \log n)$ - average

What is common to all these algorithms?
- Make comparisons between input elements
  - $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$

Comparison-based Sorting

- **Comparison sort**
  - Only comparison of pairs of elements may be used to gain order information about a sequence.
  - Hence, a lower bound on the number of comparisons will be a lower bound on the complexity of any comparison-based sorting algorithm.
- All our sorts have been comparison sorts
- The best worst-case complexity so far is $\Theta(n \log n)$ (e.g., heapsort, merge sort).
- We prove a lower bound of $\Omega(n \log n)$ for any comparison sort: merge sort and heapsort are optimal.
- The idea is simple: there are $n!$ outcomes, so we need a tree with $n!$ leaves, and therefore $\log(n!) = n \log n$. 
Decision Tree

For insertion sort operating on three elements.

Contains $3! = 6$ leaves.

Decision Tree (Cont.)

- Execution of sorting algorithm corresponds to tracing a path from root to leaf.
- The tree models all possible execution traces.
- At each internal node, a comparison $a_i \leq a_j$ is made.
  - If $a_i \leq a_j$, follow left subtree, else follow right subtree.
  - View the tree as if the algorithm splits in two at each node, based on information it has determined up to that point.
- When we come to a leaf, ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$ is established.
- A correct sorting algorithm must be able to produce any permutation of its input.
  - Hence, each of the $n!$ permutations must appear at one or more of the leaves of the decision tree.
A Lower Bound for Worst Case

- Worst case no. of comparisons for a sorting algorithm is
  - Length of the longest path from root to any of the leaves in the decision tree for the algorithm.
    - Which is the height of its decision tree.
- A lower bound on the running time of any comparison sort is given by
  - A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf.

Optimal sorting for three elements

Any sort of six elements has 5 internal nodes.

There must be a worst-case path of length ≥ 3.
A Lower Bound for Worst Case

**Theorem:** Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

**Proof:**
- Suffices to determine the height of a decision tree.
- The number of leaves is at least $n!$ (# outputs)
- The number of internal nodes $\geq n! - 1$
- The height is at least $\log(n! - 1) = \Omega(n \lg n)$

Counting Comparisons
- Let us just count comparisons then.
- Each possible run of the algorithm corresponds to a root-to-leaf path in a decision tree
**Decision Tree Height**

- The height of the decision tree is a lower bound on the running time.
- Every input permutation must lead to a separate leaf output.
- If not, some input ...4...5... would have same output ordering as ...5...4..., which would be wrong.
- Since there are $n! = 1 \cdot 2 \cdot ... \cdot n$ leaves, the height is at least $\log (n!)$.

**The Lower Bound**

- Any comparison-based sorting algorithms takes at least $\log(n!)$ time.
- Therefore, any such algorithm takes worst-case time at least

$$\log (n!) \geq \log \left( \frac{n}{2} \right)^{\frac{n}{2}} = \left( \frac{n}{2} \right) \log (n/2).$$

- That is, any comparison-based sorting algorithm must run in $\Omega(n \log n)$ time in the worst case.