How fast can we sort?

Heapsort, Mergesort, Heapsort, AVL sort all have $O(N \log N)$ worst case running time.

These algorithms, along with Quicksort, also have $O(N \log N)$ average case running time.

Can we do any better?

Permutations

- Suppose you are given $N$ elements
  - Assume no duplicates
- How many possible orderings can you get?
  - Example: $a, b, c$ ($N = 3$)

  - $6$ orderings = $3!$ (i.e., “3 factorial”)

- For $N$ elements
  - $N$ choices for the first position, $(N-1)$ choices for the second position, …, $(2)$ choices, $1$ choice
  - $N(N-1)(N-2)\cdots(2)(1) = N!$ possible orderings

Sorting Model

Recall our basic sorting assumption:

**We can only compare two elements at a time.**

These comparisons prune the space of possible orderings.

We can represent these concepts in a…

Decision Tree

The leaves contain all the possible orderings of $a, b, c$. 
Decision Trees

- A Decision Tree is a Binary Tree such that:
  - Each node = a set of orderings
    - i.e., the remaining solution space
  - Each edge = 1 comparison
  - Each leaf = 1 unique ordering
  - How many leaves for \( N \) distinct elements?

- Only 1 leaf has the ordering that is the desired correctly sorted arrangement

Decision Tree Example

<table>
<thead>
<tr>
<th>Possible Orders</th>
<th>Actual Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b &lt; c</td>
<td>a &lt; b &lt; c</td>
</tr>
<tr>
<td>a &lt; c &lt; b</td>
<td>a &lt; c &lt; b</td>
</tr>
<tr>
<td>b &lt; a &lt; c</td>
<td>b &lt; a &lt; c</td>
</tr>
<tr>
<td>b &lt; c &lt; a</td>
<td>b &lt; c &lt; a</td>
</tr>
<tr>
<td>c &lt; a &lt; b</td>
<td>c &lt; a &lt; b</td>
</tr>
<tr>
<td>c &lt; b &lt; a</td>
<td>c &lt; b &lt; a</td>
</tr>
</tbody>
</table>

Decision Trees and Sorting

- Every sorting algorithm corresponds to a decision tree
  - Finds correct leaf by choosing edges to follow
    - i.e., by making comparisons
- We will focus on worst case run time
- Observations:
  - Worst case run time \( \geq \) max number of comparisons
  - Max number of comparisons = length of the longest path in the decision tree = tree height

How many leaves on a tree?

Suppose you have a binary tree of height \( h \). How many leaves in a perfect tree?

We can prune a perfect tree to make any binary tree of same height. Can # of leaves increase?

Lower bound on Height

- A binary tree of height \( h \) has at most \( 2^h \) leaves
  - Can prove by induction
- A decision tree has \( N! \) leaves. What is its minimum height?

An Alternative Explanation

At each decision point, one branch has \( \leq \frac{1}{2} \) of the options remaining, the other has \( \geq \frac{1}{2} \) remaining.

Worst case: we always end up with \( \geq \frac{1}{2} \) remaining.

Best algorithm, in the worst case: we always end up with exactly \( \frac{1}{2} \) remaining.

Thus, in the worst case, the best we can hope for is halving the space \( d \) times (with \( d \) comparisons), until we have an answer, i.e., until the space is reduced to size = 1.

The space starts at \( N! \) in size, and halving \( d \) times means multiplying by \( 1/2^d \), giving us a lower bound on the worst case:

\[
\frac{N!}{2^d} = 1 \implies N! = 2^d \implies d = \log_2(N!)
\]
Lower Bound on $\log(N!)$

$\Omega(N \log N)$

**Worst case** run time of any comparison-based sorting algorithm is $\Omega(N \log N)$.

Can also show that **average case** run time is also $\Omega(N \log N)$.

Can we do better if we don’t use comparisons? (Huh?)

---

Can we sort in $O(n)$?

- Suppose keys are integers between 0 and 1000

**BucketSort (aka BinSort)**

If all values to be sorted are integers between 1 and $B$, create an array $\text{count}$ of size $B$, increment counts while traversing the input, and finally output the result.

**Example** $B=5$. Input = (5,1,3,4,3,2,1,1,5,4,5)

<table>
<thead>
<tr>
<th>count array</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

Running time to sort n items?

---

What about our $\Omega(n \log n)$ bound?

**Dependence on $B$**

What if $B$ is very large (e.g., $2^{64}$)?
Fixing impracticality: RadixSort

- RadixSort: generalization of BucketSort for large integer keys
- Origins go back to the 1890 census.
- Radix = “The base of a number system”
  - We’ll use 10 for convenience, but could be anything
- **Idea:**
  - BucketSort on one digit at a time
  - After k\(^{th}\) sort, the last k digits are sorted
  - Set number of buckets: \( B = \text{radix} \)

---

**Radix Sort Example**

**Input data**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>478</td>
<td>537</td>
<td>9</td>
<td>721</td>
<td>3</td>
<td>38</td>
<td>123</td>
<td>67</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**BucketSort on 1's**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>721</td>
<td>123</td>
<td>38</td>
<td>67</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**BucketSort on 10's**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**BucketSort on 100's**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Output:**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>478</td>
<td>67</td>
<td>123</td>
<td>38</td>
<td>3</td>
<td>9</td>
<td>721</td>
<td>537</td>
<td>478</td>
</tr>
</tbody>
</table>

---

**Radix Sort Example (1\(^{st}\) pass)**

**Input data**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>478</td>
<td>537</td>
<td>9</td>
<td>721</td>
<td>3</td>
<td>38</td>
<td>123</td>
<td>67</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bucket sort by 1's digit**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>721</td>
<td>123</td>
<td>122</td>
<td>38</td>
<td>67</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**After 1\(^{st}\) pass**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>3</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

---

**Radix Sort Example (2\(^{nd}\) pass)**

**Input data**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>478</td>
<td>537</td>
<td>9</td>
<td>721</td>
<td>3</td>
<td>38</td>
<td>123</td>
<td>67</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bucket sort by 10's digit**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**After 2\(^{nd}\) pass**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>3</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

---

**Radix Sort Example (3\(^{rd}\) pass)**

**Input data**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>478</td>
<td>537</td>
<td>9</td>
<td>721</td>
<td>3</td>
<td>38</td>
<td>123</td>
<td>67</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Bucket sort by 100's digit**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>721</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**After 3\(^{rd}\) pass**

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>721</td>
<td>3</td>
<td>123</td>
<td>537</td>
<td>67</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

---

**RadixSort: Complexity**

In our examples, we had:
- Input size, \( N \)
- Number of buckets, \( B = 10 \)
- Maximum value, \( M < 10^3 \)
- Number of passes, \( P = \)

How much work per pass?

Total time?
Choosing the Radix

Run time is roughly proportional to:

\[ P(B+N) = \log_B M(B+N) \]

Can show that this is minimized when:

\[ B \log_B N = N \]

In theory, then, the best base (radix) depends only on \( N \).

For fast computation, prefer \( B = 2^6 \). Then best \( b \) is:

\[ b + \log_2 b = \log_2 N \]

Example:

\[ N = 1 \text{ million (i.e., } \sim 2^{20} \text{) 64 bit numbers, } M = 2^{64} \]
\[ \log_2 N = 20 \rightarrow b = 16 \]
\[ B = 2^{16} = 65,536 \text{ and } P = \log_{2^{16}} 2^{64} = 4. \]

In practice, memory word sizes, space, other architectural considerations, are important in choosing the radix.

Big Data: External Sorting

Goal: **minimize disk/tape access** time:

- Quicksort and Heapsort both jump all over the array, leading to expensive random disk accesses
- Mergesort scans linearly through arrays, leading to (relatively) efficient sequential disk access

Basic Idea:

- Load chunk of data into Memory, sort, store this “run” on disk/tape
- Use the Merge routine from Mergesort to merge runs
- Repeat until you have only one run (one sorted chunk)
- Mergesort can leverage multiple disks
- Weiss gives some examples

Sorting Summary

- **\( O(N) \)** average, worst case:
  - Selection Sort, Bubblesort, Insertion Sort
- **\( O(N \log N) \)** average case:
  - Heapsort: in-place, not stable.
  - BST Sort: \( O(N) \) extra space (including tree pointers, possibly poor memory locality), stable.
  - Mergesort: \( O(N) \) extra space, stable.
- **\( \Omega(N \log N) \)** worst and average case:
  - Any comparison-based sorting algorithm
- **\( O(N) \)**