CS 253: Algorithms

Chapter 8

Sorting in Linear Time
How Fast Can We Sort?

I won’t quit until I find a $\Theta(n)$ sorting algorithm.
How Fast Can We Sort?

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

What is common to all these algorithms?
- They all sort by making comparisons between the input elements.
Comparison Sorts

- Comparison sorts use comparisons between elements to gain information about an input sequence \( \langle a_1, a_2, \ldots, a_n \rangle \)

- Perform tests:
  
  \[ a_i < a_j, \quad a_i \leq a_j, \quad a_i = a_j, \quad a_i \geq a_j, \quad \text{or} \quad a_i > a_j \]

  to determine the relative order of \( a_i \) and \( a_j \)

- For simplicity, assume that all the elements are distinct
Lower-Bound for Sorting

Theorem:

To sort \( n \) elements, comparison sorts must make \( \Omega(n \log n) \) comparisons in the worst case.
Decision Tree Model

- Represents the comparisons made by a sorting algorithm on an input of a given size.
  - Models all possible execution traces
  - Control, data movement, other operations are ignored
  - Count only the comparisons

![Decision Tree Diagram]

Worst-case number of comparisons depends on:

- the length of the longest path from the root to a leaf
  (i.e., the height of the decision tree)
Lemma

Any binary tree of height $h$ has at most $2^h$ leaves

Proof: by induction on $h$

Basis: $h = 0 \implies$ tree has one node, which is a leaf

$\# \text{ of Leaves} = 1 \leq 2^0$ (TRUE)

Inductive step: assume true for $h-1$ (i.e. $\# \text{Leaves} \leq 2^{h-1}$)

- Extend the height of the tree with one more level
- Each leaf becomes parent to two new leaves

No. of leaves at level $h = 2 \times (\text{no. of leaves at level } h-1)$

$\leq 2 \times 2^{h-1}$

$\leq 2^h$
What is the least number of leaves in a Decision Tree Model?

- All permutations on n elements must appear as one of the leaves in the decision tree:
  
  \( n! \) permutations

- At least \( n! \) leaves
**Theorem:** Any comparison sort algorithm requires \( \Omega(n \lg n) \) comparisons in the worst case.

**Proof:**

- How many leaves does the tree have?
  - At least \( n! \) (each of the \( n! \) permutations must appear as a leaf)
  - There are at most \( 2^h \) leaves (by the previous Lemma)

\[ n! \leq 2^h \]

\[ h \geq \lg(n!) = \Theta(n \lg n) \]

(see next slide)

---

**Lower Bound for Comparison Sorts**

We can beat the \( \Omega(n \lg n) \) running time if we use other operations than just comparing elements with each other!
\[ \lg(n!) = \Theta(n \lg n) \]

1. \( n! \leq n^n \quad \Rightarrow \quad \lg(n!) \leq n \lg n \quad \Rightarrow \quad \lg(n!) = O(n \lg n) \)

2. \( n! \geq 2^n \quad \Rightarrow \quad \lg(n!) \geq n \lg 2 = n \quad \Rightarrow \quad \lg(n!) = \Omega(n) \)

\[ \Rightarrow \Rightarrow \Rightarrow \quad n \leq \lg(n!) \leq n \lg n \]

- We need a tighter lower bound!
- Use Stirling’s approximation (3.18):
  \[ n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \]

\[ \log_e(n!) = \log_e \sqrt{2\pi n} + \log_e \left(\frac{n}{e}\right)^n + \log_e \left(1 + \Theta\left(\frac{1}{n}\right)\right) \]

\[ \geq n \log_e \left(\frac{n}{e}\right) \geq cn \log_e n \quad \text{for } c = 0.5 \quad \text{and } n > n_0 = e^2 \]

\[ \log_e(n!) = \Omega(n \log n) \]
Counting Sort

- Assumptions:
  - Sort $n$ integers which are in the range $[0 \ldots r]$.
  - $r$ is in the order of $n$, that is, $r = O(n)$.

- Idea:
  - For each element $x$, find the number of elements $\leq x$.
  - Place $x$ into its correct position in the output array.

\[
\text{input array } \quad \begin{array}{cccccccc}
A: & 3 & 6 & 4 & 2 & 5 & 8 & 10 \\
\end{array}
\]

\[
x = 5, \frac{\text{number of elements } \leq 5}{\text{(number of elements } \leq 5)} = 4 \quad \{3, 4, 2, 5\}
\]

\[
\text{output array } \quad \begin{array}{cccccccc}
B: & & & & & & & \\
\end{array}
\]

- put 5 here !!!
**Step 1**

Find the number of times $A[i]$ appears in $A$

<table>
<thead>
<tr>
<th>input array $A$:</th>
<th>3</th>
<th>6</th>
<th>4</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate $C$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$i=1, A[1]=3$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=2, A[2]=6$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=3, A[3]=4$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i=8, A[8]=4$</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$C[i] = \text{number of times element } i \text{ appears in } A$

Allocate $C[1..r]$ (histogram)

For $1 \leq i \leq n$, $++C[A[i]]$

(i.e., frequencies/histogram)
Step 2

Find the number of elements $\leq A[i]$ (i.e. cumulative sums)

Input array $A$: [3, 6, 4, 1, 3, 4, 1, 4]

Old C array

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

New C array

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

\[ C[0] = C[0] \]
\[ C[i] = C[i-1] + C[i] \]

\[ C[i] \] = # elements $\leq i$
Algorithm

- Start from the last element of $A$
- Place $A[i]$ at its correct place in the output array
- Decrease $C[A[i]]$ by one
Example

(frequencies)

(cumulative sums)
Example (cont.)

A

1 2 3 4 5 6 7 8

2 5 3 0 2 3 0 3

B

1 2 3 4 5 6 7 8

0 0 2 3 3 3

C

0 2 3 5 7 8

B

1 2 3 4 5 6 7 8

0 0 2 3 3 3 3 5

C

0 2 3 4 7 8

B

1 2 3 4 5 6 7 8

0 0 2 3 3 3 3 5

C

0 2 3 4 7 8

B

1 2 3 4 5 6 7 8

0 0 2 3 3 3 3 5
Alg.: COUNTING-SORT(A, B, n, k)

1. for i ← 0 to r
2. do C[i] ← 0
3. for j ← 1 to n
4. do C[A[j]] ← C[A[j]] + 1
   % C[i] contains the number of elements = i ; frequencies
5. for i ← 1 to r
6. do C[i] ← C[i] + C[i - 1]
   % C[i] contains the number of elements ≤ i ; cumulative sum
7. for j ← n downto 1
   % B[.] contains sorted array
Algorithm: COUNTING-SORT(A, B, n, k)

1. for i ← 0 to r
2. do C[i] ← 0
3. for j ← 1 to n
4. do C[A[j]] ← C[A[j]] + 1
5. for i ← 1 to r
6. do C[i] ← C[i] + C[i - 1]
7. for j ← n downto 1

Overall time: $\Theta(n + r)$
Analysis of Counting Sort

- Overall time: $\Theta(n + r)$
- In practice we use COUNTING sort when $r = O(n)$
  \[ \Rightarrow \text{running time is } \Theta(n) \]
- Counting sort is **stable**
- Counting sort is **not in place** sort
Radix Sort

- Represents keys as $d$-digit numbers in some base-$k$
  
  e.g. $\text{key} = x_1x_2...x_d$ where $0 \leq x_i \leq k-1$

- Example: key=15

  $\text{key}_{10} = 15$, $d=2$, $k=10$ where $0 \leq x_i \leq 9$
  
  $\text{key}_2 = 1111$, $d=4$, $k=2$ where $0 \leq x_i \leq 1$
Radix Sort

- Assumptions: \( d=\Theta(1) \) and \( k = O(n) \)

- Sorting looks at one column at a time
  - For a \( d \) digit number, sort the least significant digit first
  - Continue sorting on the next least significant digit,
    until all digits have been sorted
  - Requires only \( d \) passes through the list

326
453
608
835
751
435
704
690
Radix Sort

**Algorithm:** RADIUS-SORT(A, d)

for $i \leftarrow 1$ to $d$

  do use a **stable** sort to sort array $A$ on digit $i$

- $1$ is the lowest order digit, $d$ is the highest-order digit

How do things go wrong if an **unstable** sorting alg. is used?
Analysis of Radix Sort

- Given $n$ numbers of $d$ digits each, where each digit may take up to $k$ possible values, RADIX-SORT correctly sorts the numbers in $\Theta(d(n+k))$

  - One pass of sorting per digit takes $\Theta(n+k)$ assuming that we use counting sort

  - There are $d$ passes (for each digit) $\Rightarrow \Theta(d(n+k))$
Bucket Sort

• **Assumption:**
  ◦ the input is generated by a random process that distributes elements **uniformly** over [0, 1)

• **Idea:**
  ◦ Divide [0, 1) into $n$ equal-sized buckets
  ◦ Distribute the $n$ input values into the buckets
  ◦ Sort each bucket (e.g., using **quicksort**)
  ◦ Go through the buckets in order, listing elements in each one

• **Input:** $A[1 \ldots n]$, where $0 \leq A[i] < 1$ for all $i$
• **Output:** elements $A[i]$ sorted
• **Auxiliary array:** $B[0 \ldots n - 1]$ of **linked lists**, each list initially empty
Example - Bucket Sort

A

1 | .78
2 | .17
3 | .39
4 | .26
5 | .72
6 | .94
7 | .21
8 | .12
9 | .23
10 | .68

B

0 | /
1 | .17 -> .12 /
2 | .26 -> .21 -> .23 /
3 | .39 /
4 | /
5 | /
6 | .68 /
7 | .78 -> .72 /
8 | /
9 | .94 /
Analysis of Bucket Sort

*Alg.: BUCKET-SORT(A, n)*

for i ← 1 to n
do insert A[i] into list B[⌊nA[i]⌋]

for i ← 0 to n - 1
do sort list B[i] with quicksort

concatenate lists B[0], B[1], ..., B[n -1] together in order

return the concatenated lists

\[ \theta(n) \]

\[ O(n) \]

\[ \Omega(n) \]
Conclusions

- Any *comparison sort* will take at least $n \log n$ to sort an array of $n$ numbers.

- We can achieve a $O(n)$ running time for sorting if we can make certain assumptions on the input data:
  - **Counting sort**: each of the $n$ input elements is an integer in the range $[0 \ldots r]$ and $r = O(n)$
  - **Radix sort**: the elements in the input are integers represented with $d$ digits in base-$k$, where $d = \Theta(1)$ and $k = O(n)$
  - **Bucket sort**: the numbers in the input are *uniformly distributed* over the interval $[0, 1)$
Problem

You are given 5 distinct numbers to sort. Describe an algorithm which sorts them using at most 6 comparisons, or argue that no such algorithm exists.

Solution:
Total # of leaves in the comparison tree = 5!
If the height of the tree is h, then \((\text{total # of leaves} \leq 2^h)\)

\[ 2^h \geq 5! \]

\[ h \geq \log_2(5!) \]

\[ \geq \log_2 120 \]

\[ h > 6 \]

\( \Rightarrow \) There is at least one input permutation which will require at least 7 comparisons to sort. Therefore, no such algorithm exists.