CSE 326 Lecture 16: All sorts of sorts

✦ What’s on our plate today?
  ➤ Sorting Algorithms: The Best of the Fastest…
    ♦ Heapsort
    ♦ Mergesort
    ♦ Quicksort
  ➤ Covered in Chapter 7 of the textbook

Review of Sorting Algorithms

✦ “Simple” sorts
  ➤ Bubblesort, Selection Sort, and Insertion Sort
  ➤ Run Time = O(N^2)
✦ Insertion Sort: O(N) if elements already sorted
✦ Shellsort
  ➤ Works by running Insertion sort on subsets of elements over several passes
  ➤ O(N^{1.5}) using Hibbard’s increment sequence
Review of Sorting Algorithms

✦ “Simple” sorts
  ✤ Bubblesort, Selection Sort, and Insertion Sort
  ✤ Run Time = O(N^2)
  ✤ Insertion Sort: O(N) if elements already sorted

✦ Shellsort
  ✤ Works by running Insertion sort on subsets of elements
  ✤ O(N^{1.5}) using Hibbard’s increment sequence

Canya beat O(N^{1.5}) usin’ a Binary Search Tree to sort?

Using Binary Search Trees for Sorting

✦ Can we beat O(N^{1.5}) using a BST to sort N elements?
  ✤ Yes!!
  ✤ Insert each element into an initially empty BST
  ✤ Do an In-Order traversal to get sorted output

✦ Running time = ?
Using Binary Search Trees for Sorting

✦ Can we beat $O(N^{1.5})$ using a BST to sort $N$ elements?
  ➔ Yes!!
  ➔ Insert each element into an initially empty BST
  ➔ Do an In-Order traversal to get sorted output

✦ Running time = $N$ Inserts, each takes $O(\log N)$ time, plus $O(N)$ for In-Order traversal = $O(N \log N) = o(N^{1.5})$

✦ Any Drawbacks?

Using Binary Search Trees for Sorting

✦ Drawback: Uses Extra Space
  ➔ Need to allocate space for tree nodes and pointers
  ➔ $O(N)$ extra space needed, not in place sorting

Waftaminute…what if the tree is complete, and we use an array representation – can we sort in place?

Recall your favorite data structure with the initials B. H.
Using a Binary Heap for Sorting

✦ Main Idea:
  ➤ Build a max-heap
  ➤ Do N \text{DeleteMax} operations and store each Max element in the unused end of array

$$\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
8 & 2 & 9 & 4 & 5 & & & \\
9 & 5 & 8 & 4 & 2 & & & \\
8 & 5 & 2 & 4 & 9 & & & \\
\end{array}$$

Heapsort: Analysis

✦ Heapsort is in-place…is it also stable?
✦ Running time = time needed for building max-heap + time for N \text{DeleteMax} operations = ?
Heapsort: Analysis

- Running time = time to build max-heap +
  time for N DeleteMax operations
  = O(N) + N O(log N) = O(N \log N)

- Can also show that running time is \Omega(N \log N) for some inputs, so worst case is \Theta(N \log N)

- Average case running time is also O(N \log N) (see text for proof)

How about a “Divide and Conquer” strategy?

- Very important strategy in computer science:
  1. Divide problem into smaller parts
  2. Independently solve the parts
  3. Combine these solutions to get overall solution
How about a “Divide and Conquer” strategy?

- **Idea**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves
  - Known as **Mergesort**

- Example: Mergesort this input array:

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>
```

---

**Mergesort Example**

```
8 2 9 4 5 3 1 6
---
Divide
Divide
Divide
1 element
Merge
Merge
Merge
```

```
2 8 2 4 8 9
9 4 9 3 5
5 3 1 6
1 2 3 4 5 6 8 9
```
Mergesort Analysis

✦ Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array (see textbook for code).

✦ Is Mergesort stable? In-place?

✦ Let T(N) be the running time for an array of N elements

✦ Recurrence relation for run time = ?

Mergesort Analysis

✦ Let T(N) be the running time for an array of N elements

✦ Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array (see textbook for code).

✦ Each recursive call takes T(N/2) and merging takes O(N)

✦ Therefore, the recurrence relation for T(N) is:

  ⇒ T(1) = O(1) (Base case: 1 element array = constant time)
  ⇒ T(N) = 2T(N/2) + N

✦ What is T(N) as a big-oh function of N?
Squeezing the big-oh out of our recurrence…

- Can solve the recurrence by expanding the terms:
  \[ T(N) = 2 \cdot T(N/2) + N \]
  \[ T(N/2) = 2 \cdot T(N/4) + N/2, \text{ etc. Therefore:} \]
  \[ T(N) = 2 \cdot [2 \cdot T(N/4) + N/2] + N \]
  \[ = 2^2 \cdot T(N/2^2) + 2 \cdot N \]
  \[ = 2^2 \cdot [2 \cdot T(N/8) + N/4] + 2 \cdot N \]
  \[ = 2^3 \cdot T(N/2^3) + 3 \cdot N \]
  \[ \vdots \quad (\text{recall that } 2^{\log N} = N) \]
  \[ = 2^{\log N} \cdot T(N/2^{\log N}) + (\log N) \cdot N \]
  \[ = N \cdot T(1) + N \log N \]
  \[ = N \cdot O(1) + N \log N = O(N \log N) \]
  \[ \Rightarrow T(N) = O(N \log N) \]

Being Quick without taking up Space…

- Mergesort requires temporary array for merging = \( O(N) \)
  extra space – can we do in place sorting without extra space?
  \[ \Rightarrow \text{ Want a divide and conquer strategy} \] that does not use the
  \( O(N) \) extra space

- Enter…“Quicksort”:
  Idea:
  Partition the array such that Elements in left sub-array <
  elements in right sub-array.
  Recursively sort left and right sub-arrays
How do we **partition** the array?

✦ Choose an element from the array as the **pivot**
✦ Move all elements < pivot into left sub-array and all elements > pivot into right sub-array
  ➤ If element = pivot, can be handled in several ways
    
    \[
    7 \quad 18 \quad 2 \quad 15 \quad 9 \quad 11
    \]

➤ Suppose pivot = 7  
➤ Left subarray = 2  
➤ Right sub-array = 18 \hspace{1em} 15 \hspace{1em} 9 \hspace{1em} 11

Now we are ready to Quicksort

✦ **Quicksort Algorithm:**
  1. Partition array into left and right sub-arrays such that:
     Elements in left sub-array < elements in right sub-array
  2. Recursively sort left and right sub-arrays
  3. Concatenate left and right sub-arrays with pivot in middle

✦ **How to Partition the Array:**
  1. Choose an element from the array as the **pivot**
  2. Move all elements < pivot into left sub-array and all elements > pivot into right sub-array

✦ **Pivot?** One choice: use **first element** in array
QuickSort Example

✦ Sort the array containing:

\[ 9 \ 16 \ 4 \ 15 \ 2 \ 5 \ 17 \ 1 \]

\[ \text{pivot} \]

\[ \text{Partition} \]
\[ 4 \ 2 \ 5 \ 1 < \ 9 < 16 \ 15 \ 17 \]

\[ \text{Partition} \]
\[ 2 \ 1 \ 4 \ 5 \]
\[ 15 \ 16 \ 17 \]

\[ \text{Concatenate} \]
\[ 1 \ 2 \ 5 \]
\[ 15 \]
\[ 16 \]
\[ 17 \]

\[ \text{Concatenate} \]
\[ 1 \ 2 \ 4 \ 5 \]
\[ 15 \]
\[ 16 \]
\[ 17 \]

\[ \text{Concatenate} \]
\[ 1 \ 2 \ 4 \ 5 \]
\[ 9 \]
\[ 15 \]
\[ 16 \]
\[ 17 \]

Partitioning In Place

✦ Hmm…seems like we need an extra array for partitioning and concatenating left/right sub-arrays

⇒ No!

✦ Algorithm for In Place Partitioning:
2. Set pointers i and j to beginning and end of array
3. Increment i until you hit an element A[i] > pivot
4. Decrement j until you hit an element A[j] < pivot
5. Swap A[i] and A[j]
6. Repeat until i and j cross (i exceeds or equals j)
7. Swap pivot and A[i]

✦ Example: Partition in place:
\[ 2 \ 16 \ 4 \ 15 \ 2 \ 5 \ 17 \ 1 \] (pivot = A[0] = 9)
The Pivotal Role of Pivots

✦ How do we pick the pivot for each partition?
  ➤ Pivot choice can make a big difference in run time

✦ First Idea: Pick the first element in (sub-)array as pivot
  ➤ What if it is the smallest or largest?
  ➤ What if the array is sorted? How many recursive calls does quicksort make?

<table>
<thead>
<tr>
<th>2</th>
<th>4</th>
<th>6</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
</table>

Choosing the Right Pivot

✦ 2nd Idea: Pick a random element
  ➤ Gets rid of asymmetry in left/right sizes
  ➤ But…requires calls to pseudo-random number generator – expensive/error-prone

<table>
<thead>
<tr>
<th>9</th>
<th>16</th>
<th>4</th>
<th>15</th>
<th>2</th>
</tr>
</thead>
</table>

✦ Third idea: Pick median (N/2th largest element)
  ➤ Ideal but hard to compute without sorting!
  ➤ Compromise: Pick median of three elements
Median-of-Three Pivot

- Find the median of the first, middle and last element
- Takes only O(1) time and not error-prone like the pseudo-random pivot choice
- Less chance of poor performance as compared to looking at only 1 element
- For sorted inputs, splits array nicely in half each recursion
  - Good performance

Quicksort Performance Analysis

- **Best Case Performance:** Algorithm always chooses best pivot and keeps splitting sub-arrays in half at each recursion
  - $T(0) = T(1) = O(1)$ (constant time if 0 or 1 element)
  - For $N > 1$, 2 recursive calls + linear time for partitioning
  - Recurrence Relation for $T(N) = ?$
  - Big-Oh function for $T(N) = ?$
Quicksort Performance Analysis

✦ **Best Case Performance**: Algorithm always chooses best pivot and keeps splitting sub-arrays in half at each recursion
  ➤ $T(0) = T(1) = O(1)$ (constant time if 0 or 1 element)
  ➤ For $N > 1$, 2 recursive calls + linear time for partitioning
  ➤ $T(N) = 2T(N/2) + O(N)$ (Same as Mergesort)
  ➤ $T(N) = O(N \log N)$

✦ **Worst Case Performance**: What is the worst case?

Quicksort Performance Analysis

✦ **Worst Case Performance**: Algorithm keeps picking the worst pivot – one sub-array empty at each recursion
  ➤ $T(0) = T(1) = O(1)$
  ➤ Recurrence relation for $T(N) = ?$
  ➤ Big-Oh function for $T(N) = ?$
Quicksort Performance Analysis

✦ **Worst Case Performance**: Algorithm keeps picking the worst pivot – one sub-array empty at each recursion

\[ T(0) = T(1) = O(1) \]

\[ T(N) = T(N-1) + O(N) = T(N-2) + O(N-1) + O(N) = \ldots \]

\[ = T(0) + O(1) + \ldots + O(N) \]

\[ T(N) = O(N^2) \]

✦ Fortunately, *average case performance* is \( O(N \log N) \) (see text for proof)

---

Can We Sort Any Faster?

✦ Heapsort, Mergesort, and Quicksort all run in \( O(N \log N) \) best case running time

✦ Can we do any better?

✦ Can Joey Sortiepants from Hackersville, USA come up with an \( O(N) \) sorting algorithm?
Questions to ponder over the Weekend

How fast can one sort?
Can I find time to read Chapter 7?
What was the meaning of the midterm?
What is the meaning of life? (extra credit)

Have a great weekend!