Lecture 12

Sorting II

- Sorting in $O(n \log(n))$-time
  - Merge Sort
  - Quick Sort
  - Heap Sort
- Comparison Based Sorts
- Bucket Sort

Note: $\log n$ and $\log(n)$ are used interchangeably and both refer to $\log_2(n)$ unless otherwise specified.
Sorting Videos
Warning: Flashing Screen (especially for the first 8 seconds).
Retro Video Game Lab
Friday 11am – 12pm
Schow Library 014
entrance is near the bean bag chairs
Sorting in $O(n \log(n))$-time

Where will the log(n) come from?
Merge Sort
Merge Sort

Split the array into two halves. Sort each half recursively and merge them together.

- The base case is 1 element (or 2 elements shown below).

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

Some example steps of merge sort.

Analysis of Merge Sort

- It runs in $O(n \log n)$-time in the worst-case.
  - There are $\log n$ levels in the recursion and each level takes a total of $O(n)$-time.
  - Illustration on the next slide.

- It is difficult to implement this in-place.
  - The merge step uses additional space.

- It is stable.
  - Why?
Merge sort on an array $A$, where each node is a recursive call with a range of indices to be sorted.

- We divide the range in half when going down, so the height of this tree is $\log(n)$.
- The merge step is $O(k)$-time where $k$ is the number of values being merged.
  Therefore, we need to sum all of the ranges to get the overall run-time.
- Notice that each layer contains each element of $A$ exactly once.
  So each layer requires a total of $O(n)$-time.

There are $\log(n)$ layers and each requires $O(n)$-time. Hence, the overall run-time is $O(n \log(n))$-time.
Live Coding: MergeSort

Let’s implement merge sort!

- The function `MergeSort(int[] A)` modifies `A`.
  It will call the recursive function `MergeSortRec` that does all the work.

- The function `MergeSortRec` will use the following arguments:
  - `int A[]` is the array.
  - `int left` and `int right` specify that `A[left...right]` is to be sorted by this recursive call.
    Otherwise, we could copy subarrays during each recursive call, but this would be wasteful.
  - `int temp[]` will be used for temporary space during merging. It must have size at least `A.length() = n`.
    This temporary space will be created by the `MergeSort` function.
  - Therefore, the signature is `MergeSortRec(int[] A, int left, int right, int[] temp)`.

We also need to implement a helper function `merge`.

- Conceptually, its input is two sorted arrays, and it combines them into a single sorted array.
- In practice, merge sort on needs to merge two subarrays that are next to each other.
  - Thus, we can use `merge(int[] A, int first, int mid, int last, int[] temp)`.
- It will merge the values into the `temp` array and then copy these values back to `A`.

Note: There are many different ways to implement merge sort.
Implementing `MergeSort` (and `Merge`) in the file `Sorting.java`.

```java
public class Sorting {

    public static void MergeSort(int[] A) {
        int n, left, right, temp[];
    
        // Get the array length and return if the array is empty.
        n = A.length;
        if (n == 0) return;
    
        // Create a temporary array used during merges.
        temp = new int[n];
    
        // Call the recursive function on the full range of A.
        left = 0;
        right = n-1;
        MergeSortRec(A, left, right, temp);
    }

    protected static void MergeSortRec(int[] A, int left, int right, int[] temp) {
        // Base Case: A range of length at most 1 is already sorted.
        if (left >= right) return;
    
        // Compute the mid point.
        int mid = left + (right - left) / 2;
    
        // Sort both sides of the array.
        MergeSortRec(A, left, mid, temp);
        MergeSortRec(A, mid+1, right, temp);
    
        // Merge the two sorted sides of the array.
        Merge(A, left, mid, right, temp);
    }

    protected static void Merge(int[] A, int left, int mid, int right, int[] temp) {
        int finger1, finger2, finger;
        boolean more1, more2;

        // Set "finger" indices to the start of the subarrays and temp array.
        finger = left; // An index into temp.
        finger1 = left; // An index into the first subarray.
        finger2 = mid+1; // An index into the second subarray.
    
        // The more flags track if there are more values in each subarray.
        more1 = (finger1 <= mid);
        more2 = (finger2 <= right);
    
        // Repeatedly move the smaller of the two values into temp.
        while (more1 || more2) {
            // Move the smaller of the two values into temp.
            if ((more2 || (more1 && A[finger1] <= A[finger2]))) {
                temp[finger++] = A[finger1++];
            } else {
                temp[finger++] = A[finger2++];
            }
    
            // Update the more flags.
            more1 = (finger1 <= mid);
            more2 = (finger2 <= right);
        }
    
        // Copy the sorted values in temp into A.
        for (int i = left; i <= right; i++) {
            A[i] = temp[i];
        }
    }
```
Quick Sort
Quicksort

Pick a random pivot and separate the items based on being smaller or larger.

- Use up to 2 swaps to separate each item and recursively sort the two subarrays.

```
1 -5 4 6 -7 4 -3
1 -5 4 6 -7 4 -3
-5 1 4 6 -7 4 -3
-5 1 4 6 -7 4 -3
-5 -7 1 6 4 4 -3
-5 -7 1 6 4 4 -3
-5 -7 -3 1 4 4 6
```

Analysis of Quicksort

- It runs in $O(n \log n)$-time in average cases.
  - Why?
- It is $O(n^2)$-time in the worst case.
  - When does this occur?
- One of the most efficient algorithms in practice.
- It is in-place.
- It is not stable.

Notice that the last step swaps 6 with -3, and then -3 with 1.
Heap Sort

(preview)
Binary Heaps

A *binary heap* is a binary tree whose node values satisfy two rules:

1. The value of a node is less than or equal to the values of its children.
2. The binary tree is *full* meaning that all levels except the bottom are full and the bottom level has all of the nodes as far to the left as is possible.

We’ll implement a binary heap with the following:

(a) Insert a new value in \(\log(n)\)-time. We preliminarily add it to the bottom and then fix the structure.
(b) Remove the minimum value in \(\log(n)\)-time. Deleting the top element and fixing the structure.

How can we use these points as part of an \(O(n \log(n))\)-time sorting algorithm?
Heap Sort

Add all n items into a heap and then remove the minimum one at a time.

- The result is very similar to selection sort.
- More generally, any priority queue can be used to sort in this way.

Note: We’ll discuss binary trees, binary heaps, and priority queues later in the course.

Analysis of Heap Sort

- It runs in $O(n \log n)$-time in the worst-case.
  - Each insert takes $O(\log n)$-time.
  - Each remove takes $O(\log n)$-time.

- It is not in-place since we create a new array for the heap.

- It is not stable.
Comparison Based Sorting
Sorting Limitations

We have provide several algorithms for sorting in $O(n \log n)$-time. Is this the best possible result?

All of these algorithms have been *comparison-based* meaning that they only use the relative order of pairs of items via comparisons ($<$, $\leq$, $=$, $\geq$, $>$) to make decisions.

- The algorithms are focused on the *relative order* of the values, rather than any specific values.
  - For example, if you multiply every value by 10, then the algorithm works in exactly the same way.

This leads to two questions:

1. Is $O(n \log n)$-time the best possible for a comparison-based sorting algorithm?
2. Are there sorting algorithms that are not comparison-based? And could they run faster than $O(n \log n)$-time?
Limitation of Comparison-Based Sorting Algorithms

A list of n distinct items has n! different permutations.

- Only one of the n! permutations is the correct sorted order. The algorithm must determine it.
- Each comparison $x \ ? y$ divides the number of possibilities by two since $x < y$ or $x > y$.
- $\log(n!) = \log(1 \cdot 2 \cdots n) = \log(1) + \log(2) + \cdots + \log(n) \leq \log(n) + \cdots + \log(n) = n \log(n)$ by replacing each term with $\log(n)$.

The last point implies that shortest binary tree with n! leaves has height $O(n \log n)$.

**Theorem:** Any comparison-based sort requires $\Omega(n \log n)$-time in the worst-case.
Bucket Sort
Bucket Sort

Suppose that we know that the minimum value is $\geq m$ and the maximum value is $\leq M$.
- We may know these bounds in advance, or we can determine them exactly in $O(n)$-time.

Let’s create an array of $b = M - m + 1$ “buckets” to hold each of the possible values.

1. Scan through the $n$ values and put each value into its bucket. This takes $O(n)$-time.
   - Note: Multiple items can go in the same bucket by using frequencies or an array of linked lists.
2. Scan through the buckets and put the values into a sorted array. This takes $O(b+n)$-time.

This algorithm does not use comparisons and it runs in $O(n+b)$-time.

Notice that $O(n+b)$-time is equal to $O(n)$-time whenever $b \leq c \cdot n$ for some constant $c$.
In other words, if $b$ is $O(n)$ (i.e., the range is at most proportional to the number of values),
then bucket sort is $O(n)$-time, which is faster than any comparison-based sorting algorithm.
## Summary on Sorting

<table>
<thead>
<tr>
<th></th>
<th>Bubble</th>
<th>Selection</th>
<th>Merge</th>
<th>Quick</th>
<th>Heap</th>
<th>Bucket</th>
</tr>
</thead>
<tbody>
<tr>
<td>comparison?</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>worst-case</td>
<td>$O(n^2)$-time</td>
<td>$O(n^2)$-time</td>
<td>$O(n \log n)$-time</td>
<td>$O(n^2)$-time</td>
<td>$O(n \log n)$-time</td>
<td>$O(n+b)$-time</td>
</tr>
<tr>
<td>expected</td>
<td>$O(n^2)$-time</td>
<td>$O(n^2)$-time</td>
<td>$O(n \log n)$-time</td>
<td>$O(n \log n)$-time</td>
<td>$O(n \log n)$-time</td>
<td>$O(n+b)$-time</td>
</tr>
<tr>
<td>in-place?</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>stable?</td>
<td>yes</td>
<td>yes*</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

- Use library functions unless you have a special situation.
- Bucket sort is useful when we know there is a small range of possible values.
- Quicksort is usually the fastest in practice.
- Merge sort is an example of a divide-and-conquer algorithm.
- Heap sort is a data structure driven algorithm.
- Merge, Quick, Heap are asymptotically optimal in terms of total comparisons.
- Some of the above points require further explanation (e.g. Selection stability).

---

Selection sort can be implemented as a **stable sort** if, rather than swapping in step 2, the minimum value is **inserted** into the first position and the intervening values shifted up. However, this modification either requires a data structure that supports efficient insertions or deletions, such as a linked list, or it leads to performing $\Theta(n^2)$ writes.