ECE 590.01
C++ Programming, Data structures, and Algorithms

Sorting
Admin

• Reading
  • Chapter 7
  • After the break: Chapter 9
    • Skipping 8 (optional reading if you want)

• Homework 3 due Friday
What have we been talking about?

- What did we talk about last time?
What have we been talking about?

- What did we talk about last time?
  - Heaps
  - Priority Queues
  - Huffman coding
    - Compression
    - Application of Priority Queues
Today: Sorting

- Today is all about sorting
  - Many sorting algorithms
  - Generally $O(N \log(N))$ to $O(N^2)$ in complexity
  - No hats involved
Sorting: Big picture

- Have a bunch of data, want to put it in order
  - Ascending: smallest to largest
  - Descending: largest to smallest
- Doesn’t really matter which way, same algorithms work

- Why would we want to do this?
  - Makes finding things faster/easier
    - Frequently more efficient to spend up front time to sort first
    - Think about handing back your exams
  - May make other queries easier
    - Median, Quartiles, etc...
- Our focus: sorting ints
  - Can sort anything that is totally ordered
Bubble Sort

- **Simple algorithm:**
  - Compare element $i$ to element $i+1$
  - Out of order? Swap them
  - Repeat for N-1 of the N elements
  - Repeat all of that until no change
    - Max of N times
Bubble Sort

- Simple algorithm:
  - Compare element $i$ to element $i+1$
  - Out of order? Swap them
  - Repeat for $N-1$ of the $N$ elements
  - Repeat all of that until no change
    - Max of $N$ times

```
3  7  1  5  9  2
```
Bubble Sort

- Simple algorithm:
  - Compare element i to element i+1
  - Out of order? Swap them
  - Repeat for N-1 of the N elements
  - Repeat all of that until no change
    - Max of N times

3 1 7 5 9 2
Bubble Sort

• Simple algorithm:
  • Compare element $i$ to element $i+1$
  • Out of order? Swap them
  • Repeat for N-1 of the N elements
  • Repeat all of that until no change
    • Max of N times

\[\begin{array}{cccccc}
  3 & 1 & 5 & 7 & 9 & 2 \\
\end{array}\]
Bubble Sort

• Simple algorithm:
  • Compare element i to element i+1
  • Out of order? Swap them
  • Repeat for N-1 of the N elements
  • Repeat all of that until no change
    • Max of N times
Bubble Sort

Simple algorithm:
- Compare element \( i \) to element \( i+1 \)
- Out of order? Swap them
- Repeat for \( N-1 \) of the \( N \) elements
- Repeat all of that until no change
  - Max of \( N \) times
Bubble Sort

- Has to repeat several more times to bubble 2 down
  - Called Bubble Sort because elements “bubble up” the array

- What is its running time?
  - $O(??)$
Bubble Sort

- What is its running time?
  - $O(N^2)$ generally
  - Can “get lucky” and do better:
    - Input array is already sorted? $O(N)$
Bubble Sort

- **Pros:**
  - Quick and easy to implement
  - Simple to understand
  - Works well on arrays or LLs (sort the data, keep the nodes in place)
  - No space overhead

- **Cons:**
  - $O(N^2)$ = slow for large data
Selection Sort

- Also a simple algorithm:
  - Find the smallest element
  - If it's not the first element, swap with first element
  - Recursively selection sort the remaining elements
Selection Sort

- Also a simple algorithm:
  - Find the smallest element
  - If it's not the first element, swap with first element
  - Recursively selection sort the remaining elements
Selection Sort

- Also a simple algorithm:
  - Find the smallest element
  - If it's not the first element, swap with first element
  - Recursively selection sort the remaining elements
Selection Sort

• And so on...
  • Running time?
    • $O(???)$
Selection Sort

- Running time:
  - Also $O(N^2)$
  - For each array spot, we need to find the min of $N$ elements
    - $N$ times of
    - Finding min of $N$ things
- Similar pros and cons to Bubble Sort
Insertion Sort

- Divide Array into two regions
  - Sorted region (starts out with one element)
    - Invariant: sorted region is sorted
  - Unsorted region (everything else)
  - Make progress: move next item from unsorted region to sorted
    - Insert into region by finding correct place in ordering
    - Moves boundary between regions
Insertion Sort

- Divide Array into two regions
  - Sorted region (starts out with one element)
    - Invariant: sorted region is sorted
  - Unsorted region (everything else)
  - Make progress: move next item from unsorted region to sorted
    - Insert into region by finding correct place in ordering
    - Moves boundary between regions
Insertion Sort

- Divide Array into two regions
  - Sorted region (starts out with one element)
    - Invariant: sorted region is sorted
  - Unsorted region (everything else)
  - Make progress: move next item from unsorted region to sorted
    - Insert into region by finding correct place in ordering
    - Moves boundary between regions
Insertion Sort

- Divide Array into two regions
  - Sorted region (starts out with one element)
  - Invariant: sorted region is sorted
  - Unsorted region (everything else)
  - Make progress: move next item from unsorted region to sorted
    - Insert into region by finding correct place in ordering
    - Moves boundary between regions
Insertion Sort

- Divide Array into two regions
  - Sorted region (starts out with one element)
    - Invariant: sorted region is sorted
  - Unsorted region (everything else)
  - Make progress: move next item from unsorted region to sorted
    - Insert into region by finding correct place in ordering
    - Moves boundary between regions

1 3 5 7 9 2
Insertion Sort

- Divide Array into two regions
  - Sorted region (starts out with one element)
    - Invariant: sorted region is sorted
  - Unsorted region (everything else)
  - Make progress: move next item from unsorted region to sorted
    - Insert into region by finding correct place in ordering
    - Moves boundary between regions
Insertion Sort

- Kind of in-efficient on arrays
  - Insert = copy all greater elements over
    - Doesn’t affect $O$ though—still $O(N^2)$
  - Great for LLs, especially if you want to make a new one

1 2 3 5 7 9
Insertion Sort

- **Linked Lists:**
  - Iterate down original list
  - sorted_insert onto new list
**Insertion Sort**

- **Linked Lists:**
  - Iterate down original list
  - `sorted_insert` onto new list

---

```plaintext
unsorted\n 3 → 7 → 1 → 5 → 9 → 2\ncurr
```

```plaintext
sorted\n 3
```
Insertion Sort

- Linked Lists:
  - Iterate down original list
  - sorted_insert onto new list
Insertion Sort

- Linked Lists:
  - Iterate down original list
  - sorted_insert onto new list

- Still $O(N^2)$
O(N*lg(N)) sorting

- For O(N*lg(N)) let’s start by revisiting selection sort
  - N times of
  - Find min of N things
- Can do better if we can find the min faster than O(N).

- Sound familiar?
O(N*lg(N)) sorting

- For O(N*lg(N)) let’s start by revisiting selection sort
  - N times of
  - Find min/max of N things
- Can do better if we can find the min faster than O(N)

- Sound familiar?
  - Heaps let us findMin/findMax in O(1) time
    - And delete it in O(lg(N)) time
- Concept:
  - Put everything in a heap
  - Repeatedly extract min/max, put into array
  - Profit!
Heap Sort

• A quick note on ordering
  • A min heap (smallest on top) gives descending order
  • A max heap (largest on top) gives ascending order

• We will do a min heap/descending order
Heap Sort

- Conceptually putting into Heap
  - But...
  - How are heaps actually implemented?
Heap Sort

• Conceptually putting into Heap
  • But...
  • How are heaps actually implemented?
    • Arrays
    • So kind of silly to use a separate one
    • Just use original (constant space)
Heap Sort

- Start by turning input into a heap
  - Boundary between
    - Heap (left)
    - Not-a-heap (right)
  - Moving boundary
    - Just like adding to heap
      - Bubble up as needed
Heap Sort

- Start by turning input into a heap
  - Boundary between
    - Heap (left)
    - Not-a-heap (right)
  - Moving boundary
    - Just like adding to heap
    - Bubble up as needed
Heap Sort

- Start by turning input into a heap
  - Boundary between
    - Heap (left)
    - Not-a-heap (right)
  - Moving boundary
    - Just like adding to heap
      - Bubble up as needed
Heap Sort

- Start by turning input into a heap
  - Boundary between
    - Heap (left)
    - Not-a-heap (right)
  - Moving boundary
    - Just like adding to heap
      - Bubble up as needed
Heap Sort

- Start by turning input into a heap
  - Boundary between
    - Heap (left)
    - Not-a-heap (right)
  - Moving boundary
    - Just like adding to heap
      - Bubble up as needed
Heap Sort

- Start by turning input into a heap
  - Boundary between
    - Heap (left)
    - Not-a-heap (right)
  - Moving boundary
    - Just like adding to heap
      - Bubble up as needed
Heap Sort

- Start by turning input into a heap
  - Boundary between
    - Heap (left)
    - Not-a-heap (right)
  - Moving boundary
    - Just like adding to heap
      - Bubble up as needed
Heap Sort

- Now, go backwards through array
  - Boundary now
  - Heap (left)
  - Sorted array (right)
- Move boundary:
  - Delete min
  - Put into newly vacated spot
Heap Sort

- Now, go backwards through array
  - Boundary now
    - Heap (left)
    - Sorted array (right)
  - Move boundary:
    - Delete min
    - Put into newly vacated spot
Heap Sort

- May need to bubble down
Heap Sort

- May need to bubble down
  - Then repeat
Heap Sort

- May need to bubble down
  - Then repeat
Heap Sort

- May need to bubble down
  - Then repeat
Heap Sort

- May need to bubble down
  - Then repeat
Heap Sort

• May need to bubble down
  • Then repeat
Heap Sort

- May need to bubble down
  - Then repeat
Heap Sort

- May need to bubble down
  - Then repeat
Heap Sort

- May need to bubble down
  - Then repeat

- Etc

9 7 5 3 2 1
Heap Sort

- **Pros**
  - In place
  - $O(N \times \lg(N))$ — as good as it gets in the general case
    - Average AND worst case

- **Cons**
  - Bad locality, especially on large data
    - Bad cache behavior
    - Bad paging behavior
  - Index 10,000 and its children (20,000 and 20,001) are 10K items apart
    - Almost certainly on different pages
  - Only in place for arrays
    - Want to use on an LL? Make an explicit separate heap
Merge Sort

- Split array in half
  - Sort each half
    - Recursively merge sort
    - Small array (< 4 or 8 elements): use some other sort
  - Merge the results
- Splitting done by indices, not copying

Mergesort(0,7) =
  Mergesort(0,3);
  Mergesort(4,7);
  Merge(0,4,7);

\[
\begin{array}{cccccccc}
8 & 3 & 7 & 1 & 5 & 9 & 2 & 0 \\
\end{array}
\]
Merge Sort

- Split array in half
  - Sort each half
    - Recursively merge sort
    - Small array (< 4 or 8 elements): use some other sort
  - Merge the results
- Splitting done by indices, not copying

\[
\begin{align*}
\text{Mergesort}(0,7) &= \\
&= \text{Mergesort}(0,3); \\
&\quad \text{Mergesort}(4,7); \\
&\quad \text{Merge}(0,4,7); \\
\text{Mergesort}(0,3) &= \\
&= \text{Mergesort}(0,1); \\
&\quad \text{Mergesort}(2,3); \\
&\quad \text{Merge}(0,2,3);
\end{align*}
\]
Merge Sort

Mergesort(0,7) =
  Mergesort(0,3);
  Mergesort(4,7);
  Merge(0,4,7);
Mergesort(0,3) =
  Mergesort(0,1);
  Mergesort(2,3);
  Merge(0,2,3);

- Assume Mergesort(0,1) and Mergesort(2,3) use a trivial sort
  - Sorting 2 elements is quite easy
  - Now need to merge
Merge Sort

Mergesort(0,7) =
Mergesort(0,3);
Mergesort(4,7);
Merge(0,4,7);

Mergesort(0,3) =
Mergesort(0,1);
Mergesort(2,3);
Merge(0,2,3);

- Merge(a,b,end)
  - Requires extra space to merge into
  - Easy to compute “end2” = b - 1
Merge Sort

\[
\begin{align*}
\text{Mergesort}(0,7) &= \\
&= \text{Mergesort}(0,3); \\
&= \text{Mergesort}(4,7); \\
&= \text{Merge}(0,4,7); \\
\text{Mergesort}(0,3) &= \\
&= \text{Mergesort}(0,1); \\
&= \text{Mergesort}(2,3); \\
&= \text{Merge}(0,2,3);
\end{align*}
\]

- Merge(a, b, end)
  - Requires extra space to merge into
  - Easy to compute “end2” = b – 1
  - Pick smaller of array[a] and array[b]
    - Copy into temp[next]
    - Increment a or b (whichever was used)
    - Increment next
Merge Sort

Mergesort(0,7) =
  Mergesort(0,3);
  Mergesort(4,7);
  Merge(0,4,7);

Mergesort(0,3) =
  Mergesort(0,1);
  Mergesort(2,3);
  Merge(0,2,3);

- Merge(a,b,end)
  - Requires extra space to merge into
  - Easy to compute “end2” = b – 1
  - Pick smaller of array[a] and array[b]
    - Copy into temp[next]
    - Increment a or b (whichever was used)
    - Increment next
Merge Sort

Mergesort(0,7) =
  Mergesort(0,3);
  Mergesort(4,7);
  Merge(0,4,7);

Mergesort(0,3) =
  Mergesort(0,1);
  Mergesort(2,3);
  Merge(0,2,3);

- Merge(a,b,end)
  - Requires extra space to merge into
  - Easy to compute “end2” = b – 1
  - Pick smaller of array[a] and array[b]
    - Copy into temp[next]
    - Increment a or b (whichever was used)
    - Increment next
Merge Sort

Mergesort(0,7) =
   Mergesort(0,3);
   Mergesort(4,7);
   Merge(0,4,7);

Mergesort(0,3) =
   Mergesort(0,1);
   Mergesort(2,3);
   Merge(0,2,3);

- Merge(a, b, end)
  - Whenever b goes past end
    - Or a goes past end2
  - Done with that half, just copy from other half into temp

ECE 590.01 (Hilton): Sorting
Merge Sort

- Mergesort(0,7) =
  - Mergesort(0,3);
  - Mergesort(4,7);
  - Merge(0,4,7);

- Mergesort(0,3) =
  - Mergesort(0,1);
  - Mergesort(2,3);
  - Merge(0,2,3);

- Merge(a,b,end)
  - Whenever b goes past end
    - Or a goes past end2
  - Done with that half, just copy from other half into temp

ECE 590.01 (Hilton): Sorting
Merge Sort

Mergesort(0,7) =
  Mergesort(0,3);
  Mergesort(4,7);
  Merge(0,4,7);

Mergesort(0,3) =
  Mergesort(0,1);
  Mergesort(2,3);
  Merge(0,2,3);

• Merge(a,b,end)
  • Whenever b goes past end
    • Or a goes past end2
  • Done with that half, just copy from other half into temp
  • Now copy temp back into main array

ECE 590.01 (Hilton): Sorting
### Merge Sort

- **Mergesort(0,7) =**
  - Mergesort(0,3);
  - Mergesort(4,7);
  - Merge(0,4,7);

- **Mergesort(0,3) =**
  - Mergesort(0,1);
  - Mergesort(2,3);
  - Merge(0,2,3);

- **Merge(a,b,end)**
  - Whenever b goes past end
    - Or a goes past end2
  - Done with that half, just copy from other half into temp
  - Now copy temp back into main array
Merge Sort

Mergesort(0,7) =
Mergesort(0,3);
Mergesort(4,7);
Merge(0,4,7);

Mergesort(0,3) =
Mergesort(0,1);
Mergesort(2,3);
Merge(0,2,3);

- Merge(a,b,end)
  - Whenever b goes past end
    - Or a goes past end2
  - Done with that half, just copy from other half into temp
  - Now copy temp back into main array
  - That concludes the merge
Merge Sort

Mergesort(0,7) =
    Mergesort(0,3);
    Mergesort(4,7);
    Merge(0,4,7);

Mergesort(0,3) =
    Mergesort(0,1);
    Mergesort(2,3);
    Merge(0,2,3);

• Merge(a,b,end)
  • Whenever b goes past end
    • Or a goes past end2
  • Done with that half, just copy from other half into temp
  • Now copy temp back into main array
  • That concludes the merge
    • Which also concludes the Mergesort(0,3) call (0—3 are sorted)
Merge Sort

Mergesort(0,7) =
  Mergesort(0,3);
  Mergesort(4,7);
  Merge(0,4,7);

• The recursion returns back to Mergesort(0,7)
  • Next thing here is Mergesort(4,7)
  • Recursively go do that
Merge Sort

Mergesort(0, 7) =
   Mergesort(0, 3);
   Mergesort(4, 7);
   Merge(0, 4, 7);

- The recursion returns back to Mergesort(0, 7)
  - Next thing here is Mergesort(4, 7)
  - Recursively go do that
  - Not shown:
    - But you trust recursion right?
  - So now we just need to merge
Merge Sort

\[
\text{Mergesort}(0,7) = \\
\text{Mergesort}(0,3); \\
\text{Mergesort}(4,7); \\
\text{Merge}(0,4,7);
\]

- Merge
Merge Sort

Mergesort(0,7) =
    Mergesort(0,3);
    Mergesort(4,7);
    Merge(0,4,7);

- Merge

<table>
<thead>
<tr>
<th>1</th>
<th>3</th>
<th>7</th>
<th>8</th>
<th>0</th>
<th>2</th>
<th>5</th>
<th>9</th>
</tr>
</thead>
</table>

next
Merge Sort

Mergesort(0,7) =
Mergesort(0,3);
Mergesort(4,7);
Merge(0,4,7);

- Merge
Merge Sort

Mergesort(0,7) =
Mergesort(0,3);
Mergesort(4,7);
Merge(0,4,7);

• Merge

ECE 590.01 (Hilton): Sorting
Merge Sort

\[ \text{Mergesort}(0,7) = \text{Mergesort}(0,3); \]
\[ \text{Mergesort}(4,7); \]
\[ \text{Merge}(0,4,7); \]

- Merge
Merge Sort

Mergesort(0,7) =
  Mergesort(0,3);
  Mergesort(4,7);
  Merge(0,4,7);

• Merge
Merge Sort

Mergesort(0,7) =
  Mergesort(0,3);
  Mergesort(4,7);
  Merge(0,4,7);

- Merge
Merge Sort

Mergesort(0,7) =
  Mergesort(0,3);
  Mergesort(4,7);
  Merge(0,4,7);

• Merge
  • Note we took from left side twice in a row
  • That’s fine, happens all the time (may take many from one side)
Merge Sort

\[ \text{Mergesort}(0,7) = \]
\[ \text{Mergesort}(0,3); \]
\[ \text{Mergesort}(4,7); \]
\[ \text{Merge}(0,4,7); \]

- **Merge**
  - a is past the end of its half now, so copy in from right side (b’s side)
Merge Sort

Mergesort(0,7) =
Mergesort(0,3);
Mergesort(4,7);
Merge(0,4,7);

- Merge
  - Copy back into original array, and done with merge
Merge Sort

\[
\text{Mergesort}(0,7) = \\
\text{Mergesort}(0,3); \\
\text{Mergesort}(4,7); \\
\text{Merge}(0,4,7);
\]

- Now done with Mergesort(0,7), so whole array is sorted
Merge Sort: Pros and Cons

• Pros:
  • Good locality
  • $O(N \times \lg(N))$: average and worst case
  • Easy to parallelize:
    • Great for having TAs sort tests by last name

• Cons:
  • Requires extra space (not done in places) for arrays
  • Linked Lists require work to find halfway through
    • But can be merged with no extra space
Quick Sort

- Step 1: pick a pivot
  - Always pick last element?
    - Leads to bad performance if array is already (almost) sorted
  - Good choice: random index
Quick Sort

- Step 1: pick a pivot
  - Always pick last element?
    - Leads to bad performance if array is already (almost) sorted
  - Good choice: random index
    - I picked index 4 (value = 5)
Quick Sort

- Step 1: pick a pivot
  - Always pick last element?
    - Leads to bad performance if array is already (almost) sorted
  - Good choice: random index
    - I picked index 4 (value = 5)
- Step 2:
  - Get elements less than pivot (5) on left, greater on right
Quick Sort

• Step 2:
  • Get elements less than pivot (5) on left, greater on right
  • Swap pivot into last slot
Quick Sort

- Step 2:
  - Get elements less than pivot (5) on left, greater on right
  - Swap pivot into last slot
  - Track two indices
    - Lo starts at 0, moves right
      - Scan for element larger than pivot
    - Hi starts at last spot before pivot, moves left
      - Scan for element smaller than pivot
Quick Sort

- When array[lo] < pivot and array[hi] >= pivot
  - Swap(lo, hi)
  - Increment lo
  - Decrement hi
  - Keep going
Quick Sort

- Now array[lo] < pivot, so we keep scanning
  - Incrementing low and re-checking
Quick Sort

Now array[lo] >= pivot, so we stop, and work on hi
Array[hi] >= pivot, so we need to scan right
Quick Sort

- Now array[hi] < pivot, so we stop, and swap
Quick Sort

- Now array[hi] < pivot, so we stop, and swap
  - Increment lo
  - Decrement hi
  - Continue
Quick Sort

- Lo scans right
Quick Sort

- Lo scans right
- Hi scans left (doesn’t go anywhere)

- Now, since lo > hi (the pointers have crossed)
  - We don’t swap
  - Instead, we swap(lo,pivot)
    - Pivot’s new index at the end
Quick Sort

- Lo scans right
- Hi scans left (doesn’t go anywhere)

- Now, since lo > hi (the pointers have crossed)
  - We don’t swap
  - Instead, we swap(lo,pivot)
    - Pivot’s new index at the end
Quick Sort

- At this point
  - All items smaller than 5 are to its left
  - All items larger than 5 are to its right
  - 5 is in the correct place
- Now we make two recursive quick sort calls
  - One sorts the left
  - The other sorts the right
Quick Sort Performance

- What is the Big-O of quick sort?
Quick Sort Performance

- What is the Big-O of quick sort?
  - Worst Case: $O(N^2)$
    - Always pick largest/smallest (or close) element?
    - Will effectively sort 1 element each recursion
  - Average Case: $O(N \cdot \log(N))$
    - Pick elements somewhere near the middle mostly?
    - Will cut array roughly in half each time
Quick Sort Pros and Cons

• Pros
  • Usually $O(N \times \lg (N))$
  • Good locality (cache behavior)
  • Sorts array in place

• Cons
  • Can be $O(N^2)$
  • Requires extra space for recursive call stack
Other sorting things

• Some sorts are **stable**, some are not
  • Stable sort =equal elements stay in the same order
  • Some sorts are stable, some are not
  • Which of these are stable? I’ll leave that to you...

• $O(N^*\log(N))$ is provable best $O$ for sorting without special prior knowledge
  • Can do better if you know some constraints on the data
    • Limited ranges
    • The input array will already be sorted...
Sorting Summary

- Sorting
  - So many ways to do it
    - We didn’t even start to cover them all
  - Did cover a bunch
    - Bubble
    - Insertion
    - Selection
    - Heap
    - Quick
  - These are the most common ones
    - Really good to know for interviews
      - Along with their Big-O, and other pros and cons