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

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

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

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

- Has to repeat several more times to bubble 2 down
  - Called Bubble Sort because elements "bubble up" the array
- What is its running time?
  - $O(N^2)$

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

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:
- $O(N^2) = \text{slow for large data}$
Selection Sort

- And so on...
  - Running time?
    - $O(\text{???})$

- Running time:
  - Also $O(N^2)$
  - For each array spot, we need to find the min of $N$ elements
  - $N$ times
  - 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

- 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

- 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

**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 $O(N^2)$

$O(N^2)$ sorting

- For $O(N^2)$ 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?

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?

- 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

• 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

ECE 590.01 (Hilton): Sorting

49

50

51

52

53

54
### Heap Sort

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

**Cons**
- Bad locality, especially on large data
- Bad cache 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 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

#### Merge Sort

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

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

#### Merge(a, b, end)

- Requires extra space to merge into
- Easy to compute “end2” = b - 1

```
[ ]   [ ]   [ ]
```

- 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

- **Merge(a,b,end)**
  - Requires extra space to merge into
  - Easy to compute \( "end2" = b - 1 \)
  - Pick smaller of \( \text{array}[a] \) and \( \text{array}[b] \)
  - Copy into \( \text{temp}[\text{next}] \)
  - Increment \( a \) or \( b \) (whichever was used)
  - Increment next

### Merge Sort

- **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

### Merge Sort

- **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);
Mergesort(0,4,7);

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

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

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

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

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

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

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

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

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

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

Mergesort(0,3) =
Mergesort(0,1);
Mergesort(2,3);
Mergesort(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); Mergesort(0,4.7);

- Merge

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

- Merge

\[
\begin{array}{cccccccc}
  & & & & & & & \\
 6 & 1 & 2 & \text{next} & & & & \\
\end{array}
\]

Mergesort(0,7) = Mergesort(0,3); Mergesort(4,7); Mergesort(0,4.7);

- Merge

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

- Merge

\[
\begin{array}{cccccccc}
  & & & & & & & \\
 0 & 1 & 2 & 3 & \text{next} & & & \\
\end{array}
\]

Mergesort(0,7) = Mergesort(0,3); Mergesort(4,7); Mergesort(0,4.7);

- Merge

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

- Merge

\[
\begin{array}{cccccccc}
  & & & & & & & \\
 0 & 1 & 2 & 3 & 5 & \text{next} & & \\
\end{array}
\]
Merge Sort

Merge Sort

Merge Sort

Merge Sort

Merge Sort

Merge Sort

Merge Sort

Quick Sort

Quick Sort

Quick Sort
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

- 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

Increment lo

Decrement hi

Continue

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

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

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?
  - 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 $O(N \cdot \log(N))$
  - 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...
- $O(N \cdot \log(N))$ 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