A brief diversion: What's on the midterm

- Same format as last time
  - 5 questions, 20 points each, 100 points total
  - 15% of your grade
- 2 topics:
  - trees, trees, and more trees
  - graphs

Binary trees: What to know
- Characteristics
- Searching and traversing
- Inserting and deleting
- Balancing a tree (definition of “balanced”)
- AVL trees
- Self-adjusting trees/splaying
- Implementations
- Complexity

Heaps: what to know
- What is a heap
- Relationship between heap and array
- Making heap from array and array from heap
- Insertion and deletion (reorganizing a heap)
- Applications: what heaps are used for (and how)
  - e.g., priority queue
B-trees and tries: what to know

- Characteristics
- In general, how to insert/delete from a B-tree
- Applications

Graphs: what to know

- Characteristics (digraphs and simple graphs) and definitions
- Spanning tree: how to find, what it is
- Shortest path (Dijkstra and Bellman-Ford)
- Max flow (Ford-Fulkerson method)

Sorting algorithms

- Insertion sort
- Selection sort
- Bubble sort
- Shell sort
- Heap sort
- Quick sort
- Merge sort
- Radix sort

Complexity

- Number of comparisons
- Number of swaps
- Storage requirements
**Decision tree**

- A generic way to express any sorting algorithm
  - “lower bound” of number of comparisons
- Binary tree
- Non-leaf nodes are possible comparisons
- Arcs are Y/N (results of comparisons)
- Number of leaves is $\geq n!$
- Number of comparisons (worst and avg cases) = $O(n \ lg \ n)$
  - best we can do for sorting an array, comparison-wise

**Simple (inefficient) sorting algorithms**

- **Insertion sort**
- **Selection sort**
- **Bubble sort**
- **Shell sort**
- **Heap sort**
- **Quick sort**
- **Merge sort**
- **Radix sort**

**Insertion sort**

- Compare first two elements, put in proper order
- Compare first three elements, put in proper order
- ....

```java
public void insertionSort(Comparable[] data) {
    Comparable temp;
    for (int i=1; i<data.length; i++) {
        temp = data[i]; // store the element in question
        for (int j=i; j>0 && temp.compareTo(data[j-1])<0; j--)
            data[j] = data[j-1]; // move data elements up
        data[j] = temp; // move temp into place
    }
}
```
Insertion sort: complexity

- Best case: data already in order
  - $n-1$ comparisons, $2(n-1)$ moves
- Worst case: data in reverse order
  - $i$ comparisons in each iteration, $n-1$ iterations, ...
    - $n(n-1)/2 = O(n^2)$
    - moves: temp is loaded/unloaded $2(n-1)$ times, $i$
      moves in iteration $i$, ... $(n^2+3n-4)/2 = O(n^2)$
- Average case: data in random order

Insertion sort: average complexity

- In iteration $i$, there are up to $j+1$ comparisons
  - average: $(i+1)/2$ (if equal probability cell occupation)
  - average # comparisons = sum of the averages =
    $(n^2+n-2)/4 = O(n^2)$
- In iteration $i$, there are up to $i-1$ movements
  - average: $(i-1)/2$
  - plus, 2 movements (into and out of temp)
  - average # movements = sum of the averages =
    $(n^2+5n-6)/4 = O(n^2)$

Selection sort

- Start at the first element in the array
- Find the minimum element in the array
  - save its index
- Swap the first element with the minimum element in the array
- Repeat for the remaining elements in the array
- In each pass, at least one element will be moved to its proper position in the array

Selection sort: implementation

```java
public void selectionSort(Comparable[] data) {
    int min;
    for (int i=0; i<data.length-1; i++) {
        min = i;
        for (int j=i+1; j<data.length; j++) {
            if (data[j].compareTo(data[min]) < 0) {
                min = j;
            }
        }
        swap(data, min, i);
    }
}
```
Implementation of `swap`

```java
private void swap(Comparable[] data, int pos1, int pos2) {
    Comparable temp = data[pos1];
    data[pos1] = data[pos2];
    data[pos2] = temp;
}
```

Complexity of selection sort: comparisons

- Same for all three cases
- Outer loop executes \(n-1\) times
- Inner loop executes \((n-1) - i\) times on \(i^{th}\) iteration
- \(n(n-1)/2 = O(n^2)\)

Complexity of selection sort: movements

- Best case: data sorted
  - 0 swaps
- Worst case: largest element in first position, rest of data ordered
  - swap largest with next element \(n-1\) times
  - each swap involves three steps
  - \(O(n)\)

Bubble sort

- Idea: make more data swaps each pass through the array
- During each pass, compare \(pairs\) of array elements, and swap if they are out of order
- In each pass, at least one element will be moved to its proper position in the array
Bubble sort: implementation

```java
public void bubbleSort(Comparable[] data) {
    for (int i=0; i<data.length-1; i++) {
        for (int j=1; j<data.length-i; j++) {
            if (data[j].compareTo(data[j-1]) < 0) {
                swap(data, j, j-1);
            }
        }
    }
}
```

Complexity of bubble sort:
comparisons

- Same for all three cases
- Outer loop executes \(n-1\) times
- Inner loop executes \((n-1) - i\) times on \(i^{th}\) iteration
- \(n(n-1)/2 = O(n^2)\)

Complexity of bubble sort:
movements

- Best case: data sorted
  - 0 swaps
- Worst case: data in reverse order
  - \(3\times(\text{number of comparisons}) = O(n^2)\)
- Average case
  - up to \((n-1-i)\) swaps per subarray: average is \((n-1-i)/2\)
  - \(n-1\) iterations....
  - \(3n(n-1)/4 = O(n^2)\)

Efficient sorting algorithms

- Insertion sort
- Selection sort
- Bubble sort
- Shell sort
- Heap sort
- Quick sort
- Merge sort
- Radix sort
Shell sort

- Idea: keep sorting subarrays until the entire array is sorted
  - split the array into \( n_1 \) subarrays, sort
  - split the array into \( n_2 > n_1 \) subarrays, sort
  - repeat until no subdivisions can be made
- array is sorted
- Any sorting method can be used for the subarrays
  - typically insertion
  - some implementations use a different sort for the last pass

Choosing subarrays

- Take every \( h^\text{th} \) element from array
- Decrease \( h \) each time so that you get different subarrays each time
- Difficult problem: picking the \( h \)'s!
  - original algorithm: use powers of 2
  - experimental results: select \( h \) values so that
    - \( h_1 = 1 \)
    - \( h_{i+1} = 3h_i + 1 \)
    - stop with \( h_i \) for which \( h_{i+2} \geq n \)

Complexity

- Depends on how the increments are chosen
- Better than \( O(n^2) \) but worse than \( O(n \log n) \)
  - Some experimental measurements:
    - \( O(n^{1.25}) \)
    - \( O(n^{5/3}) \)
    - \( O(n \log^2 n) \)

Heap sort

- Idea:
  - start with a heap and an empty array
  - “remove” the largest element and place it at the end of the array
  - restructure the heap
  - repeat until all elements have been removed from the heap and the array is full (and in order)
Complexity of heap sort

- Creating the heap: $O(n)$
- Sorting the heap: worst case
  - Swap root with element to be removed: $n-1$ times
  - Restore heap: occurs $n-1$ times, each iteration takes $\lg i$ steps --> $O(n \lg n)$
  - Total complexity = heap + swap + restore = $O(n \lg n)$
- Sorting the heap: best case
  - all elements are identical: $O(n)$
  - distinct elements already in order: $O(n \lg n) - O(n)$

Quicksort

- Very fast, recursive sorting algorithm
- Idea:
  - divide array into 2 subarrays, using a pivot
    - first half contains elements less than pivot
    - second half contains elements greater than pivot
  - repeat this division/partitioning until size of subarrays = 1
  - by preparing to sort, you have sorted the array

Key concept: selecting the pivot

- Need to pick a pivot that will approximately split our current array
- Common strategies:
  - pick the first element
  - pick the middle element
  - pick a random element
  - preprocess the array

Complexity of quicksort

- Worst case: select pivot that's the largest or smallest element in the subarray
  - $O(n^2)$
- Best case: select pivot so that subarrays are roughly equal
  - $O(n \lg n)$
- Average case is closer to the best case than the worst case --> $O(n \lg n)$
Merge sort

- Idea:
  - divide array in half
  - divide these arrays in half
  - ... repeat until subarrays are size 1
  - start merging subarrays
  - repeat until array is restored and sorted

Complexity of merge sort

- Worst case: last element of one half is less than only the last element of the other half (in any iteration)
  - Swaps: \( O(n \lg n) \)
    - Number of swaps into and out of temp array = the length of the subarray (for each subarray)
    - Number of swaps for subarray of size 1 = 0
  - Comparisons: \( O(n \lg n) \)
    - Number of comparisons = the length of the subarray (for each subarray)
    - Number of comparisons for subarray of size 1 = 0

Radix sort

- Idea: Sort by looking at one letter or digit at a time, then move to the next letter or digit, etc.
- Words: put elements into “bins” by first letter. Recombine. Put elements into bins by second letter....
  - bins = letters of the alphabet
- Numbers: put elements into “bins” by least significant digit. Recombine. Put elements into bins by second least significant digit....
  - bins = digits 0-9
Notes on radix sort

- No sorting necessary!
  - data gets sorted via the binning process
- Bins can be implemented as queues
  - retain ordering with each pass

Complexity of radix sort

- No data comparison necessary
- 2 operations per data element:
  - divide by factor (ignore less significant digits)
  - modulo division (ignore more significant digits)
  - $O(n)$
- Number of swaps = $2n = O(n)$
- Queue operations: $O(n)$

Another implementation

- What if we handle the elements on a bit level?
  - only two “bins” per pass
  - no division required!
    - do bitwise-and instead (apply mask to element)
    - less storage (only 2 queues)
- Drawback: more iterations of loop!
  - bitwise radix sort turns out to be much slower

Sorting in Java

- Arrays: method sort in Arrays class
  - algorithm is quicksort
- Lists: method sort in Collections interface
  - algorithm is merge sort
# Summary of sorting algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comparisons</th>
<th>Swaps</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insertion sort</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>Selection sort</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>Bubble sort</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>Shell sort</td>
<td>O(n lg n)</td>
<td>O(n lg n)</td>
<td>Between O(n) and O(n lg n)</td>
</tr>
<tr>
<td>Heap sort</td>
<td>O(n lg n)</td>
<td>O(n lg n)</td>
<td>O(n lg n)</td>
</tr>
<tr>
<td>Quicksort</td>
<td>O(n lg n)</td>
<td>O(n lg n)</td>
<td>Can be O(n) in worst case</td>
</tr>
<tr>
<td>Merge sort</td>
<td>O(n lg n)</td>
<td>O(n lg n)</td>
<td></td>
</tr>
<tr>
<td>Radix sort</td>
<td>O(n)</td>
<td></td>
<td>O(n) divisions, O(n) queue operations</td>
</tr>
</tbody>
</table>