What is sorting?

• Input: sequence of $n$ comparable values

• Reorder the input to be non-descending.

• Items we wish to sort are called “keys”

• Not here: retain associated information

Why discuss sorting?

• Standard library has sorting

• Why not use that and move on?

In this class, sorting is:

• a good intro for techniques

• a good intro to comparative algorithms

SelectionSort

Idea: Swap min into first spot, second-min to second, etc.

\[
\text{for } i \leftarrow 1 \text{ to } n - 1 \text{ do} \\
\quad \text{min} \leftarrow i \\
\quad \text{for } j \leftarrow i + 1 \text{ to } n \text{ do} \\
\quad \quad \text{if } A[j] < A[\text{min}] \text{ then} \\
\quad \quad \quad \text{min} \leftarrow j \\
\quad \quad \text{Swap } A[i] \text{ and } A[\text{min}]
\]

Let’s talk about SelectionSort.

• Does it waste memory?

• Does it only work for numbers?

• What other info do we need?

• Are there inputs that are sorted faster?

• Is there a lot of data movement?
Bubble Sort

**Idea:** Think globally act locally

\[
\text{for } i \leftarrow 1 \text{ to } n - 1 \text{ do } \\
\quad \text{for } j \leftarrow 1 \text{ to } n - i \text{ do } \\
\quad \quad \text{if } A[j + 1] < A[j] \text{ then } \\
\quad \quad \quad \text{Swap } A[j] \text{ and } A[j + 1]
\]

\[
\begin{array}{cccccccc}
85 & 24 & 63 & 45 & 17 & 31 & 96 & 50 \\
\end{array}
\]

InsertionSort

\[
\text{for } j \leftarrow 2 \text{ to } n \text{ do } \\
\quad \text{key } \leftarrow A[j] \\
\quad i \leftarrow j - 1 \\
\quad \text{while } i > 0 \text{ and } A[i] > \text{key} \text{ do } \\
\quad \quad A[i + 1] \leftarrow A[i] \\
\quad \quad i \leftarrow i - 1 \\
\quad A[i + 1] \leftarrow \text{key}
\]

- What is the worst-case running time of InsertionSort?

- Why is InsertionSort correct?

- What is true *every time* we check the for loop? (including the time we find $j > n$ and stop)
\( O \) notation

We say that \( f(n) \) is \( O(g(n)) \) (read: \( f \) of \( n \) is big-oh of \( g \) of \( n \)) if and only if:

For some constants \( c \) and \( n_0 \), for all \( n > n_0 \), \( f(n) \leq cg(n) \).

Alternatively, \( \lim_{n \to \infty} \frac{f(n)}{g(n)} \leq c \)

(This is sometimes written as \( f(n) = O(g(n)) \))

These images were created with the graphing utility at [www.desmos.com/calculator]. I encourage students to create such graphs in order to better visualize what \( O \) notation and related concepts mean rather than memorizing the mechanics of them. Graph the function you want to find the asymptotic notation for, then also find appropriate values for \( n_0 \) and \( c \) and graph \( cg(n) \).

In these graphs:

Red represents the function whose running time we are bounding. The exact value is \( y = 3n^2 + 8n + 1 \)

The blue represents \( y = 12n^2 \)

The green function, appearing only in the right-hand graph, is \( y = n^2 \)
Heaps

Using the following 1-based array:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>14</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

- What is that array, interpreted as a binary tree (for heap purposes)?
- Where is the parent of node $i$?
- Where is left child of node $i$?
- Where is right child of node $i$?
- What is a complete binary tree?

- What is the max heap property?

- How tall is a heap?

Here is the same heap, drawn twice, as you will want to “reset” during the lecture at one point:
HeapSort

**Idea 1:** Insert all \(n\) elements into an (initially empty) max heap. Remove the max element, placing it in the original array’s position \(n\). Then remove the new max, placing it in position \(n-1\). Continue in this fashion. How long does this take?

**Idea 2:** Bottom-up heap construction. We know which locations will be leaf nodes.

| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |

**Question:** Once we have the array turned into a max-heap, what do we do? Where do you place the result of a **remove-max** operation?

Lower Bound for Sorting

We have seen algorithms that take \(O(n^2)\) time. We saw HeapSort which takes \(O(n \log n)\) time. You might also be familiar with TreeSort, MergeSort, and QuickSort, which take (or can take) \(O(n \log n)\) time. Are there algorithms which are strictly better than \(O(n \log n)\) time for a general comparison-based sort?

To answer this question, we will build a **decision tree** that represents any comparison based sorting algorithm. Such an algorithm will ask questions of the form “is \(x_i < x_j\)?”

- What are leaf nodes of decision tree?

- What are internal nodes?

- What is height of the tree?

- What does this tell us about any such algorithm?