Unit 14

Sorting
Sorting

• Sorting requires us to move data around within an array
• Allows users to see and organize data more efficiently
• Behind the scenes it allows more effective searching of data
• There are MANY sorting algorithms out there, we will focus on two simple ones
Bubble Sort

• Main Idea: Keep comparing neighbors, moving larger item up and smaller item down until largest item is at the top. Repeat on list of size n-1
• Have one loop to count each pass, (a.k.a. i) to identify which index we need to stop at
• Have an inner loop start at the lowest index and count up to the stopping location comparing neighboring elements and advancing the larger of the neighbors
Bubble Sort Algorithm

```c
void bsort(int mylist[], int size)
{
    int i, j;
    for(i=...){
        for(j=...){
            if(mylist[j] > mylist[j+1]) {
                swap(mylist[j], mylist[j+1])
            }
        }
    }
}
```
Selection Sort

• Selection sort does away with the many swaps and just records where the min or max value is and performs one swap at the end
• The list/array can again be thought of in two parts
  – Sorted
  – Unsorted
• The problem starts with the whole array unsorted and slowly the sorted portion grows
• We could find the max and put it at the end of the list or we could find the min and put it at the start of the list
  – Just for variation let's choose the min approach
Selection Sort Algorithm

```c
void ssort(int mylist[], int size)
{
    for(i=...){
        int min = i;
        for(j=... ){
            if(mylist[j] < mylist[min]) {
                min = j
            }
        }
        swap(mylist[i], mylist[min])
    }
}
```

<table>
<thead>
<tr>
<th>Pass 1</th>
<th>min=0</th>
<th>Pass 2</th>
<th>min=1</th>
<th>Pass n-2</th>
<th>min=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 3 8 6 5 1</td>
<td></td>
<td>1 3 8 6 5 7</td>
<td></td>
<td>1 3 5 6 7 8</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>i</td>
<td>j</td>
<td>i</td>
<td>j</td>
</tr>
<tr>
<td>7 3 8 6 5 1</td>
<td>min=1</td>
<td>1 3 8 6 5 7</td>
<td>min=1</td>
<td>1 3 5 6 7 8</td>
<td>min=4</td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>i</td>
<td>j</td>
<td>i</td>
<td>j</td>
</tr>
<tr>
<td>7 3 8 6 5 1</td>
<td>min=1</td>
<td>1 3 8 6 5 7</td>
<td>min=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 3 8 6 5 1</td>
<td>min=5</td>
<td>1 3 8 6 5 7</td>
<td>swap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>j</td>
<td>i</td>
<td>j</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 3 8 6 5 7</td>
<td>swap</td>
<td>1 3 8 6 5 7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Selection Sort

Value

List Index

Courtesy of wikipedia.org
OPERATIONS ON A SORTED ARRAY
Insertion to a Sorted Array

• Another option rather than sorting an unordered array is to always insert new data into the correct location of the array.
• See example below.
• To insert, we must
  – Iterate until we find the appropriate location to place the new value.
  – Make room for the new value by shifting the remaining items back a spot.

```
0 1 2 3
insert(7) 7 0 1
insert(3) 3 7 0 1
insert(8) 3 7 8 0 1
insert(6) 3 6 7 8
```
Removing from a Sorted Array

• Erasing / removing item at any location other than the very last item requires us to copy all items behind the removed item to the previous slot

To delete/remove the item at location 2 requires us to move everyone else up
COMPLEXITY & RUNTIME
Time Complexity

• Coming up with AN algorithm to solve a problem is often not TOO hard
• Coming up with a GOOD algorithm to solve a problem can be a bit harder
• We need a way to judge how "GOOD" an algorithm is
  – For us "GOOD" will mean how long the algorithm takes to solve the problem
  – We will count steps of work and come up with an answer in terms of N, where N is the size of the input/problem
Bubble Sorting

• Recall the bubble sort
• How much work do our nested loops require us to do
  – Think of each step/iteration as 1 unit of time/work

Original List is length N
(N=6 for this example)
Complexity of Sort Algorithms

- **Bubble Sort & Selection Sort**
  - 2 Nested Loops
  - Execute outer loop $n$ times
  - For each outer loop iteration, inner loop runs $i$ times.
  - Time complexity is proportional to $n^2$

- Other sort algorithms can run in time proportional to $n \cdot \log_2 n$
Importance of Time Complexity

- It makes the difference between effective and impossible
- Many important problems currently can only be solved with exponential run-time algorithms (e.g. \(O(2^n)\) time)
- Usually algorithms are only practical if they run in polynomial time (e.g. \(O(n)\) or \(O(n^2)\) etc.)

<table>
<thead>
<tr>
<th>N</th>
<th>(O(1))</th>
<th>(O(\log_2 n))</th>
<th>(O(n))</th>
<th>(O(n^*\log_2 n))</th>
<th>(O(n^2))</th>
<th>(O(2^n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>4.3</td>
<td>20</td>
<td>86.4</td>
<td>400</td>
<td>1,048,576</td>
</tr>
<tr>
<td>200</td>
<td>1</td>
<td>7.6</td>
<td>200</td>
<td>1,528.8</td>
<td>40,000</td>
<td>1.60694E+60</td>
</tr>
<tr>
<td>2000</td>
<td>1</td>
<td>11.0</td>
<td>2000</td>
<td>21,931.6</td>
<td>4,000,000</td>
<td>#NUM!</td>
</tr>
</tbody>
</table>