Last Lecture ...

- Recursive Functions
  - Natural expression for many algorithms

- Dynamic Programming
  - Automatic strategy for generating efficient versions of recursive algorithms
Today ...

- Properties of Sorting Algorithms
- Elementary Sorting Algorithms
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
Applications of Sorting

- Facilitate searching
  - Building indices

- Identify quantiles of a distribution

- Identify unique values

- Browsing data
Elementary Methods

- Suitable for
  - Small datasets
  - Specialized applications

- Prelude to more complex methods
  - Illustrate ideas
  - Introduce terminology
  - Sometimes useful complement
... but beware!

- Elementary sorts are very inefficient
  - Typically, time requirements are $O(N^2)$

- Probably, most common inefficiency in scientific computing
  - Make programs “break” with large datasets
Aim

- Rearrange a set of keys
  - Using some predefined order
    - Integers
    - Doubles
    - Indices for records in a database

- Keys stored as array in memory
  - More complex sorts when we can only load part of the data
Basic Building Blocks

- An type for each element
  ```c
  #define Item int
  ```
- Compare two elements
- Exchange two elements
- Compare and exchange two elements
Comparing Two Elements

- Using a function

```c
bool isLess(Item a, Item b) {
    return a < b;
}
```

- Or a macro

```c
#define isLess(a,b) ((a)<(b))
```
Exchanging Two Elements

- Using a function

```c
void Exchange(Item * a, Item * b)
{
    Item temp = *a; *a = *b; *b = temp;
}
```

- Or a C++ function

```cpp
void Exchange(Item & a, Item & b)
{
    Item temp = a; a = b; b = temp;
}
```

- Or a macro

```c
#define Exchange(a,b) \ 
    { Item tmp = (a); (a) = (b); (b) = tmp; }
```
Comparing And Exchange

- Using a function

```c
Item CompExch(Item * a, Item * b)
    { if (isLess(*b, *a) Exchange(a, b); }
```

- Or a C++ function

```c
Item CompExch(Item & a, Item & b)
    { if (isLess(b, a) Exchange(a, b); }
```

- Or a macro

```c
#define CompExch(a,b) \    
    if (isLess((b),(a))) Exchange((a),(b));
```
Basic Building Blocks (in R)

- Variables are not passed by reference
- For sorting, create and return new array
- For exchanging elements
  - Insert code where appropriate
  - Can you think of on an alternative?
A Simple Sort

- Gradually sort the array by:
  - Sorting the first 2 elements
  - Sorting the first 3 elements
  - ...
  - Sort all N elements
A Simple Sort Routine

```c
void sort(Item *a, int start, int stop) {
    int i, j;

    for (i = start + 1; i <= stop; i++)
        for (j = i; j > start; j--)
            CompExch(a[j-1], a[j]);
}
```
A Simple Sort Routine in R

```r
sort <- function(a, start, stop)
{
  for (i in (start + 1):stop)
    for (j in i:(start+1))
      if (a[j] < a[j - 1])
        {
          temp <- a[j];
          a[j] <- a[j - 1];
          a[j - 1] <- a[j];
        }
  return (a)
}
```
Properties of this Simple Sort

- **Non-adaptive**
  - Comparisons do not depend on data

- **Stable**
  - Preserves relative order for duplicates

- Requires $O(N^2)$ running time
Sorts We Will Examine Today

- Selection Sort
- Insertion Sort
- Bubble Sort
Recipe: Selection Sort

• Find the smallest element
  • Place it at beginning of array

• Find the next smallest element
  • Place it in the second slot

...
C Code: Selection Sort

```c
void sort(Item *a, int start, int stop)
{
    int i, j;

    for (i = start; i < stop; i++)
    {
        int min = i;
        for (j = i + 1; j < stop; j++)
        {
            if (isLess(a[j], a[min]))
            {
                min = j;
                Exchange(a[i], a[min]);
            }
        }
    }
}
```
Selection Sort

Notice:

Each exchange moves element into final position.

Right portion of array looks random.
Properties of Selection Sort

- Running time does not depend on input
  - Random data
  - Sorted data
  - Reverse ordered data…

- Performs exactly N-1 exchanges

- Most time spent on comparisons
Recipe: Insertion Sort

- The “Simple Sort” we first considered

- Consider one element at a time
  - Place it among previously considered elements
  - Must move several elements to “make room”

- Can be improved, by “adapting to data”
Improvement I

- Decide when further comparisons are futile
- Stop comparisons when we reach a smaller element
- What speed improvement do you expect?
Insertion Sort (I)

```c
void sort(Item *a, int start, int stop)
{
    int i, j;

    for (i = start + 1; i <= stop; i++)
        for (j = i; j > start; j--)
            if (isLess(a[j], a[j-1]))
                Exchange(a[j-1], a[j]);
            else
                break;
}
```
Improvement II

- Notice that inner loop continues until:
  - First element reached, or
  - Smaller element reached

- If smallest element is at the beginning...
  - Only one condition to check
void sort(Item *a,
    int start, int stop)
{
    int i, j;

    for (i = stop; i > start; i--)
        CompExch(a[i-1], a[i]);

    for (i = start + 2; i <= stop; i++)
    {
        int j = i;
        while (isLess(a[j], a[j-1]))
        {
            Exchange(a[j], a[j-1]); j--; }
    }
}
Improvement III

- The basic approach requires many exchanges involving each element
- Instead of carrying out exchanges …
- Shift large elements to the right
**Insertion Sort (III)**

```c
void sort(Item *a, int start, int stop)
{
    int i, j;

    for (i = stop; i > start; i--)
        CompExch(a[i-1], a[i]);

    for (i = start + 2; i <= stop; i++)
    {
        int j = i; Item val = a[j];
        while (isLess(val, a[j-1]))
        {
            a[j] = a[j-1]; j--; }
        a[j] = val;
    }
}
```
Insertion Sort

Notice:

Elements in left portion of array can still change position.

Right remains untouched.
Properties of Insertion Sort

- Adaptive version running time depends on input
  - About 2x faster on random data
  - Improvement even greater on sorted data
  - Similar speed on reverse ordered data

- Stable sort
Recipe: Bubble Sort

- Pass through the array
  - Exchange elements that are out of order

- Repeat until done…

- Very “popular”
  - Very inefficient too!
C Code: Bubble Sort

```c
void sort(Item *a, int start, int stop)
{
    int i, j;

    for (i = start; i <= stop; i++)
        for (j = stop; j > i; j--)
            CompExch(a[j-1], a[j]);
}
```
Bubble Sort

Notice:

Each pass moves one element into position.

Right portion of array is partially sorted
Shaker Sort

Notice:

Things improve slightly if bubble sort alternates directions…
Notes on Bubble Sort

- Similar to non-adaptive Insertion Sort
  - Moves through unsorted portion of array

- Similar to Selection Sort
  - Does more exchanges per element

- Stop when no exchanges performed
  - Adaptive, but not as effective as Insertion Sort
Performance Characteristics

- Selection, Insertion, Bubble Sorts

- All quadratic
  - Running time differs by a constant

- Which sorts do you think are stable?
Selection Sort

- Exchanges
  - N – 1

- Comparisons
  - N * (N – 1) / 2

- Requires about $N^2 / 2$ operations
- Ignoring updates to min variable
Adaptive Insertion Sort

- **Half - Exchanges**
  - About $N^2 / 4$ on average (random data)
  - $N \times (N - 1) / 2$ (worst case)

- **Comparisons**
  - About $N^2 / 4$ on average (random data)
  - $N \times (N - 1) / 2$ (worst case)

- Requires about $N^2 / 4$ operations
- Requires nearly linear time on sorted data
Bubble Sort

• Exchanges
  • \( N \cdot (N - 1) / 2 \)

• Comparisons
  • \( N \cdot (N - 1) / 2 \)

• Average case and worst case very similar, even for adaptive method
## Empirical Comparison

<table>
<thead>
<tr>
<th>N</th>
<th>Selection</th>
<th>Insertion</th>
<th>Insertion (adaptive)</th>
<th>Bubble</th>
<th>Shaker</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>2000</td>
<td>21</td>
<td>29</td>
<td>15</td>
<td>45</td>
<td>34</td>
</tr>
<tr>
<td>4000</td>
<td>85</td>
<td>119</td>
<td>62</td>
<td>182</td>
<td>138</td>
</tr>
</tbody>
</table>

(Running times in seconds)
Reading

- Sedgewick, Chapter 6