Today …

- Homework
  - Homework 3 Due
  - Homework 4 Assigned
- More on complexity analysis [Sect. 9.1]
- Binary search
- Bubble, selection, and insertion sort [Sect. 9.2]

Next week:
- Midterm (Thurs.)
Order-of-Magnitude Analysis

Big O notation focuses algorithm growth rates
– an upper bound (running time will not be worse)
– independent of the particular implementation or computer used to execute the algorithm

Given algorithm A, we say A requires time proportional to a function \( f(n) \) …
– \( n \) is the size of the input
– we say that “A is order \( f(n) \)”
– … which we write as \( O(f(n)) \)

Order-of-Magnitude Analysis

\( f(n) \) represents the algorithm’s growth rate as a function over input size \( n \)

Examples:
– If A requires time directly proportional to \( n \), then \( f(n) = n \)
  … which we write \( O(n) \)
– If A requires time proportional to \( n^2 \), then \( f(n) = n^2 \)
  … which we write \( O(n^2) \)
Order-of-Magnitude Analysis

Notice that we don’t include *constant* terms

e.g., instead of $6n + 5$, we write $O(n)$

… and instead of $6n^2 + n – 8$, we write $O(n^2)$

– We are interested in the algorithm’s *general* rate of growth
– This makes it easier to compare different solutions

Order-of-Magnitude Analysis

• We’re looking at input sizes *large enough* to make only the *growth* of the running time relevant

• We’re studying the “*asymptotic*” efficiency of algorithms
  – We’re concerned with how the running time of an algorithm increases with the size of the input *in the limit*
  – … as the size of the input increases *without bound*
Order-of-Magnitude Analysis

Definition of the Order of an Algorithm (textbook):

Algorithm $A$ is $O(f(n))$ if constants $k$ and $n_0$ exist such that $A$ requires no more than $k \cdot f(n)$ time units to solve a problem of size $n \geq n_0$

Alternative definition of big $O$ analysis [Cormen et al.]

For a given function $g(n)$, we denote by $O(g(n))$ the set of functions

$O(g(n)) = \{ f(n) \mid \text{there exist positive constants } k \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq k \cdot g(n) \text{ for all } n \geq n_0 \} $

• Thus, big $O$ gives an upper bound on a function $f(n)$ to within a constant factor

Growth Rates

Fairly typical growth-rate functions

• Ordered from least to most expensive

<table>
<thead>
<tr>
<th>Function</th>
<th>$10$</th>
<th>$100$</th>
<th>$1,000$</th>
<th>$10,000$</th>
<th>$100,000$</th>
<th>$1,000,000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\log_{10} n$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
<tr>
<td>$n$</td>
<td>$3$</td>
<td>$6$</td>
<td>$9$</td>
<td>$13$</td>
<td>$16$</td>
<td>$19$</td>
</tr>
<tr>
<td>$n \cdot \log_{10} n$</td>
<td>$30$</td>
<td>$664$</td>
<td>$9,966$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^4$</td>
</tr>
<tr>
<td>$n^2$</td>
<td>$10^2$</td>
<td>$10^4$</td>
<td>$10^4$</td>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^6$</td>
</tr>
<tr>
<td>$n^3$</td>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^6$</td>
<td>$10^6$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$10^3$</td>
<td>$10^{12}$</td>
<td>$10^{18}$</td>
<td>$10^{24}$</td>
<td>$10^{30}$</td>
<td>$10^{36}$</td>
</tr>
</tbody>
</table>

[CPSC 223, 2009]
Asymptotic Upper Bounds

Big O notation can sometimes be misleading

– We are considering asymptotic upper bounds

Any algorithm that is $O(n)$ is also $O(n^2)$, since every function $f(n)$ in $O(n)$ is also in $O(n^2)$ … ($n \leq k_1 \cdot n \leq k_2 \cdot n^2$)

– Are we analyzing the algorithm or the problem?

The algorithm is a particular computational approach for solving the problem … e.g., selection sort is $O(n^2)$

The problem is the general task … e.g., what is the smallest upper bound for the complexity of sorting

Binary Search
Recurring Themes ...

• What are common List ADT operations?
  – **Insert** items into the collection (add)
  – **Remove** items from the collection (delete)
  – **Search** for elements of a collection (retrieve, lookup)
  – … plus checking if empty, printing, etc.

• When analyzing data structures, we often are comparing the cost of each of these operations

• So how does “sorting” fit in?

Why Sorting …

• We often want to output elements in order

• Lookup (search) is more efficient if items are sorted
  – for an unordered list $O(n)$ … why?
  – for a sorted list $O(\log n)$ … we’ll discuss this more later

• But, sorting has a cost!
  – we either have to sort as part of search
  – sort after insert or remove
  – or somehow split the cost (like in a binary search tree)
Binary Search

- Given a sorted list, *binary search* finds an element in $O(\log n)$ time

- Recursively search for an element:
  1. Pick *middle element* of the list
  2. If *middle element* == *key*, then found match
  3. If *middle element* > *key*, search *left half* of list
  4. If *middle element* < *key*, search *right half* of the list

### Binary Search (Review)

**Is C in the List?**

<table>
<thead>
<tr>
<th>Input:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pick middle:</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td>C &lt; D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New input: A B C

<table>
<thead>
<tr>
<th>Pick middle:</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>B &lt; C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New input: C

<table>
<thead>
<tr>
<th>Pick middle:</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found it!</td>
<td></td>
</tr>
</tbody>
</table>
Average-Case Time Complexity

• Many sorting algorithms have $O(n^2)$ worst-case and $O(n\log n)$ or $O(n)$ best-case costs
  – worst-case is the most “pessimistic” view
  – … the algorithm on any problem won’t be slower

• Average-case
  – An algorithm is $O(f(n))$ if the average amount of time it requires to solve a problem of size $n$ is no more than $k\cdot f(n)$ for $n > n_0$
  – Often hard to determine … e.g., what’s the average case?

Three Sorting Algorithms:
Selection, Bubble, and Insertion Sort
Selection Sort

The basic idea

– Select (i.e., “find”) the largest item
– Swap it with the last item in the list
– Repeat with items 0 to \( n - 2 \)
– Stop when only one item left

<table>
<thead>
<tr>
<th>Initial list: ( n=4 )</th>
<th>Pass 1</th>
<th>Pass 2</th>
<th>Pass 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>29 10 14 13</td>
<td>13 10 14 29</td>
<td>13 10 14 29</td>
</tr>
<tr>
<td></td>
<td>13 10 14 29</td>
<td>13 10 14 29</td>
<td>10 13 14 29</td>
</tr>
</tbody>
</table>

Selection Sort (based on textbook)

```c
void selectionSort(Entry theArray[], int n) {
    for(int i = n - 1; i >= 1; i--) {   // passes 1..n-1
        int index = 0;
        for(int j = 1; j < i + 1; j++) {    // passes 1..i
            if(theArray[j] >= theArray[index])
                index = j;                           // 1 move
        } // end inner for
        swap(theArray[index], theArray[i]);     // 3 moves
    } // end outer for
}
```
Evaluating Sorting Algorithms

Number of comparisons
– How many pairs of items are compared within a pass

Number of moves
– How many times do we move list items
– Within a list or to a temporary variable

• We often do “swaps”

```c
void swap(int& x, int& y)     // here just integers
{
    int tmp = x;               How many moves in a swap?
    x = y;
    y = tmp;                   3
}
```

Selection Sort

What is the best and worst case?
– They are the same for selection sort!

What is the cost?
– The outer loop executes $n - 1$ times (passes)
– Each pass $p$ requires $n - p$ comparisons

\[
\sum_{p=1}^{n-1} (n-p) = (n-1) + (n-2) + \ldots + 1 = n(n-1)/2
\]
– Thus, insertion sort is $O(n^2)$!

Check that we would get the same result if we analyzed moves instead of comparisons
Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Best Case</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insertion Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mergesort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quicksort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heapsort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Treesort</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Bubble Sort

The basic idea:

- Compare adjacent items ...
- Exchange items if they are out of order
- Repeat for $n - 1$ passes
- Each pass $p$ requires $n - p$ comparisons

Initial list: $(n=4)$

Pass 1

```
29 10 14 13
10 29 14 13
10 14 29 13
10 14 13 29
```

Pass 2

```
10 14 13 29
10 14 13 29
10 13 14 29
10 13 14 29
```

Pass 3

```
10 13 14 29
10 13 14 29
10 13 14 29
10 13 14 29
```
**Bubble Sort (based on textbook)**

```cpp
void bubbleSort(Entry theArray[], int n) {
    bool sorted = false;
    for(int i = 1; i < n && !sorted; i++) {
        sorted = true;
        for(int j = 0; j < (n - i); j++) {
            if(theArray[j] > theArray[j+1]) {
                swap(theArray[j], theArray[j+1]);
                sorted = false;
            }
        } // end inner for
    } // end outer for
} // end outer for
```

**Bubble Sort**

**What is the best and worst case?**

- Can check if array is sorted before next pass
  - If no items require exchanging then don’t proceed

- So the best case is when the list is *already sorted*
  - It takes one pass of $n - 1$ comparisons … $O(n)$

- Worst case when the list is sorted from highest to lowest
  - Requires $n - 1$ passes
  - Each pass $p$ requires $n - p$ comparisons
    \[
    \sum_{i=1}^{n-1} (n - i) = (n - 1) + (n - 2) + \ldots + 1 = n^2 (n - 1)/2
    \]

- Thus, in the worst case for bubble sort is $O(n^2)$!
### Comparison of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Best Case</th>
<th>Average Case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Mergesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Quicksort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Heapsort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>Treesort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
</tbody>
</table>

### Insertion Sort

The basic idea:

- Partition list into sorted and unsorted regions
- Select first item in unsorted region
- Insert item into the right location of sorted region
- Shift larger items one location forward in the list

<table>
<thead>
<tr>
<th>Initial list: $(n=4)$</th>
<th>Pass 1</th>
<th>Pass 2</th>
<th>Pass 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 10 14 13</td>
<td>10 29 14 13</td>
<td>10 14 29 13</td>
<td>10 14 29 13</td>
</tr>
<tr>
<td>29 10 14 13</td>
<td>10 29 14 13</td>
<td>10 14 29 13</td>
<td>10 13 14 29</td>
</tr>
</tbody>
</table>
**Insertion Sort (based on textbook)**

```c
void insertionSort(Entry theArray[], int n) {
    for(int i = 1; i < n; i++) {
        Entry next = theArray[i];
        int j = i; // insertion index
        while(j > 0 && theArray[j-1] > next) {
            theArray[j] = theArray[j-1]; // shift
            j = j - 1;
        } // end inner for
        theArray[j] = next;
    } // end outer for
}
```

---

**Insertion Sort**

What is the best and worst case?

- **Worst case** is when inner loop has to shift \((n - 1)\) times
  - … which happens when list is in **reverse order**

- **Best case** is when list is already in order
  - … outer loop executes \(n - 1\) times doing nothing in inner loop

What is the worst-case cost?

- The outer loop executes \(n - 1\) times (passes)
- Each pass \(p\) requires \(p\) comparisons
  \[
  \sum_{p=1}^{n-1} p = 1 + 2 + \ldots + (n - 1) = n*(n - 1)/2
  \]
- Each pass \(p\) also requires \(p\) data-item shifts
- Thus, insertion sort is \(O(n^2)!\)
Evaluating Sorting Algorithms

An example of why counting comparisons separately from moves is useful …

– A variant of insertion sort is “binary” insertion sort
– Here we find the insert location using a binary search
– Comparisons go from $O(n)$ on each pass to $O(\log n)$!
– The algorithm is still $O(n^2)$ … why?

• Many of the algorithms we discuss have variants