ALGORITHM DESIGN & ALGORITHM EFFICIENCY

LECTURE 9–2

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

- **Next Monday:** quiz on recursion
- **Today:**
  - look at `delete` for class `LinkedList`
  - consider algorithm design and algorithm efficiency
  - case studies:
    - searching a list
    - sorting a list
- **Next week:**
  - formalize algorithm efficiency and its notation
  - look at other algorithms

WRITING BETTER CODE

- As you become a more sophisticated programmer, you'll be driven to write good code. Some measures of "goodness":
  - Is it ***correct***?
  - Is it ***readable***?
  - Is it ***maintainable***?
  - And sometimes writing ***efficient*** code is important, too.

EFFICIENT CODE

- Efficient code is code that uses fewer resources when run. Examples:
  - ➡ It makes fewer calculations and/or takes fewer steps.
  - ➡ It uses less memory with its data structures.

- We'll focus on the the time that a program takes to compute its answer.
  - ➡ We seek algorithms that run faster.
  - ➡ Focus on ***running time***.

A CASE STUDY: SEARCHING A LIST

SEARCHING A LIST

```
def search(v, xs):
    i, n = 0, len(xs)
    while i < n:
        if xs[i] == v:
            return True
        i += 1
    return False
```

SEARCHING A SORTED LIST

Can we do better if a list is sorted?

- Suppose that

  `xs[0] ≤ xs[1] ≤ … ≤ xs[n-1]`

SEARCHING A SORTED LIST

```
def binarySearch(v,xs):
    left, right = 0, len(someList)-1
    while left <= right:
        middle = (left + right) // 2
        if v == xs[middle]:
            return True
    elif v < xs[middle]:
        right = middle-1
    else:
        left = middle+1
return False
```

SEARCHING A SORTED LIST

```
def binarySearch(v,xs):
    left, right = 0, len(xs)-1
    while left <= right:
        middle = (left + right) // 2
        if v == xs[middle]:
            return True
        elif v < xs[middle]:
            right = middle-1
        else:
            left = middle+1
    return False
```

- With each `xs[middle]` check, we eliminate half the undetermined list items from consideration.
- This means we inspect the list about $\log_2(n)$ times.

ANOTHER CASE STUDY: SORTING A LIST

DESIGN: SELECTION SORT

DESIGN: BUBBLE SORT

DESIGN: INSERTION SORT

BUBBLE SORT

- With bubble sort we make several left-to-right scans over the list.
  - We swap out-of-order values at neighboring locations
  - This "bubbles up" larger values so they "rise" to the right.

```
def bubbleSort(xs):
    n = len(xs)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if xs[i+1] < xs[i]:                  # out of order? swap!
                xs[i],xs[i+1] = xs[i+1],xs[i]
            i += 1
```

BUBBLE SORT

- With bubble sort we make several left-to-right scans over the list.
  - We swap out-of-order values at neighboring locations
  - This "bubbles up" larger values so they "rise" to the right.

```
def bubbleSort(xs):
    n = len(xs)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if xs[i+1] < xs[i]:                    # swap?
                xs[i],xs[i+1] = xs[i+1],xs[i]
            i += 1
```

- This means we make *n* -1 scans.

BUBBLE SORT

- With bubble sort we make several left-to-right scans over the list.
  - We swap out-of-order values at neighboring locations
  - This "bubbles up" larger values so they "rise" to the right.

```
def bubbleSort(xs):
    n = len(xs)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if xs[i+1] < xs[i]:                    # swap?
                xs[i],xs[i+1] = xs[i+1],xs[i]
            i += 1
```

- This means we make *n* -1 scans.
- We can stop the scan earlier for later passes.

BUBBLE SORT ANALYSIS

- What is the running time of bubble sort?

```
def bubbleSort(xs):
    n = len(xs)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if xs[i+1] < xs[i]:
                xs[i],xs[i+1] = xs[i+1],xs[i]
            i += 1
```

The if statement runs $n - 1$ times on the first scan, then $n - 2$ times on the second scan, then $n - 3$ times on the third scan, …

BUBBLE SORT ANALYSIS

- What is the running time of bubble sort?

```
def bubbleSort(xs):
    n = len(xs)
    for scan in range(1,n):
        i = 0
        while i < n - scan:
            if xs[i+1] < xs[i]:
                xs[i],xs[i+1] = xs[i+1],xs[i]
            i += 1
```

The if statement runs $n - 1$ times on the first scan, then $n - 2$ times on the second scan, then $n - 3$ times on the third scan, …

- ➡ The total number of swaps is

$$n(n-1)/2 = (n-1) + (n-2) + \ldots + 3 + 2 + 1$$

- Its running time scales ***quadratically*** with n.

MERGING SORTED LISTS

- Suppose we have two sorted lists, how do we combine their data into one?

MERGE

- Here is a procedure that "merges" two sorted lists into one:

```
def merge(list1, list2):
    list = []
    index1 = 0
    index2 = 0
    n = len(list1) + len(list2)
for index in range(n):
    if list1[index1] <= list2[index2]:
        list.append(list1[index1])
        index1 += 1
    else:
        list.append(list2[index2])
        index2 += 1
return list
```

BAD MERGE

- Here is a procedure that "merges" two sorted lists into one:

```
def merge(list1, list2):
    list = []
    index1 = 0
    index2 = 0
    n = len(list1) + len(list2)
    for index in range(n):
        if list1[index1] <= list2[index2]:
            list.append(list1[index1])
            index1 += 1
        else:
            list.append(list2[index2])
            index2 += 1
    return list
```

- ***WHOOPS!*** we might have *exhausted* `list1` or `list2`
  - ➡ `index1` could be `len(list1)` or `index2` could be `len(list2)`

    ...***This leads to a list indexing error!***

MERGE (FIXED)

- Here is a procedure that "merges" two sorted lists into one:

```
def merge(list1, list2):
    list = []
    index1 = 0
    index2 = 0
    n = len(list1) + len(list2)
    for index in range(n):
        if index2 >= len(list2):
            list.append(list1[index1])
            index1 += 1
        elif index1 >= len(list1):
            list.append(list2[index2])
            index2 += 1
        elif list1[index1] <= list2[index2]:
            list.append(list1[index1])
            index1 += 1
        else:
            list.append(list2[index2])
            index2 += 1
    return list
```

A RECURSIVE SORTING ALGORITHM

- Can we use this as part of a sorting algorithm?

MERGESORT

- A recursive sorting algorithm that uses **`merge`**.

```
def mergeSort(someList):
    if len(someList) <= 1:
        # It's already sorted! BASE CASE.
        return someList
    else:
        # It's larger and needs more work. RECURSIVE CASE.
        n = len(someList)
        # Split into two halves.
        list1 = someList[:n//2]
        list2 = someList[n//2:]
        # Sort each half.
        sorted1 = mergeSort(list1)
        sorted2 = mergeSort(list2)
        # Combine them with merge.
        return merge(sorted1, sorted2)
```

MERGESORT

- A **recursive** sorting algorithm that uses `merge`.

```
def mergeSort(someList):
    if len(someList) <= 1:
        # It's already sorted! BASE CASE.
        return someList
    else:
        # It's larger and needs more work. RECURSIVE CASE.
        n = len(someList)
        # Split into two halves.
        list1 = someList[:n//2]
        list2 = someList[n//2:]
        # Sort each half.
        sorted1 = mergeSort(list1)
        sorted2 = mergeSort(list2)
        # Combine them with merge.
        return merge(sorted1, sorted2)
```

RUNNING TIME OF MERGESORT?