

LAST DETAILS; SORTING

LECTURE 13-1

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

► **Project 4:**

→ completed project due April 28th at 11:59pm.

COURSE TOPICS

- ▶ scripting with `input` and `print`
- ▶ variables and assignment
- ▶ integer arithmetic, boolean connectives, integer comparisons
- ▶ strings and string operations
- ▶ integer division using `%` and `//`
- ▶ printing versus returning, the `None` value
- ▶ conditional statements and loops
- ▶ function definitions
- ▶ lists and dictionaries
- ▶ object-orientation and inheritance
- ▶ linked lists and binary search trees
- ▶ sorting and searching
- ▶ higher-order functions and `lambda`
- ▶ recursive functions

NEXT COURSES

► **CSCI 221 : CS Fundamentals II**

- low level computer details
 - ➡ digital logic and circuits
 - ➡ processor machine language
 - ➡ program memory layout: registers, stack, heap
 - ➡ pointers/addresses
- "industrial" level programming in C/C++
 - ➡ object-oriented language with "template" classes
 - ➡ sophisticated memory management
 - ➡ rich, complicated "standard template" library
- more coding: short programs and larger projects
- more experience using programmer tools: Unix, git, debuggers, profilers

NEXT COURSES (CONT'D)

- ▶ **MATH/CSCI 382 : Algorithms & Data Structures**
 - careful, mathematical treatment of coding
 - runtime analysis; revisit sorting and searching
 - lots of nifty data structures
 - lots of nifty algorithms and their applications:
 - ➡ network/graph analysis
- ▶ Requires **MATH 112: Intro to Analysis**
 - ➡ teaches you to make careful mathematical arguments
- ▶ Requires **MATH 113: Discrete Structures**
 - ➡ teaches you "computer science" mathematics
 - ➡ develops problem-solving skills
 - ➡ more mathematical proofs, different than MATH 112

RECALL: SELECTION SORT

CASE STUDY: BUBBLE 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(aList):  
    n = len(aList)  
    for scan in range(1,n):  
        i = 0  
        while i < n - scan:  
            if aList[i+1] < aList[i]: # Out of order? Swap!  
                aList[i],aList[i+1] = aList[i+1],aList[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(aList):  
    n = len(aList)  
    for scan in range(1,n):  
        i = 0  
        while i < n - scan:  
            if aList[i+1] < aList[i]: #swap?  
                aList[i],aList[i+1] = aList[i+1],aList[i]  
            i += 1
```

- ▶ This means we only need to 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(aList):  
    n = len(aList)  
    for scan in range(1,n):  
        i = 0  
        while i < n - scan:  
            if aList[i+1] < aList[i]: #swap?  
                aList[i],aList[i+1] = aList[i+1],aList[i]  
            i += 1
```

- ▶ This means we only need to make $n - 1$ scans.
- ▶ This means we can stop the scan earlier for later passes.

BUBBLE SORT ANALYSIS

- What is the running time of bubble sort?

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

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

- 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?

QUICKSORT

- ▶ A sorting algorithm that **partitions** then **recursively sorts**.

```
def quickSort(someList):  
    if len(someList) == 0:  
        # It's already sorted! BASE CASE.  
        return []  
    else:  
        smaller, pivot, larger = partition(someList)  
        smallerSorted = quickSort(smaller)  
        largerSorted = quickSort(larger)  
        return smallerSorted + [pivot] + largerSorted
```

PARTITIONING A LIST "AROUND" A PIVOT VALUE

- ▶ Here is the code for partitioning a list:

```
def partition(someList):  
    smaller = []  
    pivot = someList[0] # pick some value from the list  
    larger = []  
    for x in someList[1:]:  
        if x <= pivot:  
            smaller.append(x)  
        else:  
            larger.append(x)  
    return smaller, pivot, larger
```

PARTITIONING A LIST "AROUND" A PIVOT VALUE

- ▶ Here is the code for partitioning a list:

```
def partition(someList):  
    smaller = []  
    pivot = someList[0] # pick some value from the list  
    larger = []  
    for x in someList[1:]:  
        if x <= pivot:  
            smaller.append(x)  
        else:  
            larger.append(x)  
    return smaller, pivot, larger
```

- ▶ This always picks the left element as the pivot. Other pivot choices:
 - Find the median.
 - Pick a random element.
 - Choose the median of the left, middle, and right.

PARTITION

- ▶ Here is the code for partitioning a list:

```
def partition(someList):  
    smaller = []  
    pivot = someList[0] # pick some value from the list  
    larger = []  
    for x in someList[1:]:  
        if x <= pivot:  
            smaller.append(x)  
        else:  
            larger.append(x)  
    return smaller, pivot, larger
```

- ▶ This always picks the left element as the pivot. Other pivot choices:
 - Find the median. *Ideal, but expensive.*
 - Pick a random element. *Good, but has some overhead.*
 - Choose the median of the left, middle, and right. *Usually good enough.*

RUNNING TIME OF QUICKSORT?

BAD CASE FOR QUICKSORT

TYPICAL/RANDOM CASE FOR QUICKSORT

SORTING AND SEARCHING SUMMARY

- ▶ Sorting a list makes information retrieval faster:
 - can use binary search to check membership in $O(\log_2(n))$ time.
- ▶ "First try" sorting algorithms typically sort in quadratic time.
 - bubble sort, insertion sort, selection sort, etc.
 - They essentially (in the worst case) compare every item to every other.
 - This means they might perform $1 + 2 + 3 + \dots + (n-1)$ comparisons.
 - ➡ That sum is $n(n-1)/2$ and so that leads to $\Theta(n^2)$ comparisons.
- ▶ Faster sorts use recursion:
 - Merge sort sorts in $\Theta(n \log_2(n))$ time.
 - Quick sort typically sorts in $\Theta(n \log_2(n))$ time.
 - ✦ With bad pivot choices, can take $\Theta(n^2)$ time. Can be avoided with randomness.