RUNNING TIME & ASYMPTOTIC NOTATION

LECTURE 10-2

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

- **FRIDAY:** adventure game proposal due
- **MONDAY:** quiz on object classes
- **We continue...**
  - algorithm design and algorithm efficiency
  - consider running times
  - with some case studies:
    - searching a list: unsorted vs. sorted
    - computing a power
    - sorting a list
- **Also:**
  - We formalize algorithm efficiency and its notation.
    - asymptotic notation: “big theta” and “big oh”

RECALL: SEARCHING A LIST

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

- We scan through the list items from left to right.
  - Stop early if `value` is found.
  - Otherwise, we inspect them all.
- If the list has *n* items, we can inspect as many as *n* items.
  - We say that this is a ***linear time*** algorithm.

RECALL: SEARCHING A SORTED LIST

We can do better if the list is sorted…

- Suppose that

  `someList[0] ≤ someList[1] ≤ … ≤ someList[n-1]`
- Then we can just check `someList[n//2]`
  - If it's smaller than `value` we only need check the left side,
  - otherwise we need only check the right side.
- This led to a recursive algorithm called ***binary search***.

BINARY SEARCH AS A LOOP

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

BINARY SEARCH AS A LOOP

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

- With each `someList[middle]` check…
  - …we eliminate (about) half the list items from consideration.
- This means we inspect no more than $\log_2(n)$ items when the list has size n.
- This is a ***logarithmic time*** algorithm.

ANOTHER CASE STUDY: SORTING A LIST

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

- The total comparisons for potential swaps is

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

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

MEASURING RUNNING TIME

- Suppose you have two programs that compute the same result:
  - ➡ **program A** and **program B**.
  - **Q**: How do we determine which one is faster?
  - **A:** Run the code on typical inputs, measure the time it takes.

```
>>> import timeit
>>> i = 'import power_of2'
>>> s = 'power_of2.power_of2(20)'
>>> timeit.timeit(stmt=s,setup=i,number=100)
0.00022750999778509140
```

- This will time 100 evaluations of `power_of2(20)` then report the elapsed time in seconds.

MEASURING RUNNING TIME

- Suppose you have two programs that compute the same result:
  - ➡ **program A** and **program B**.
- But maybe...
  - You don't have an exact sense of the typical inputs.
  - The size of typical inputs increases over the lifetime of the algorithms' use.
  - The size and typicality of inputs might vary widely, depending on the application of the algorithm.
  - The computer might get upgraded in some near future. Or the programs might be rewritten for some unknown system.

MEASURING RUNNING TIME

- Suppose you have two programs that compute the same result:
  - ➡ **program A** and **program B**.
- But maybe...
  - You don't have an exact sense of the typical inputs.
  - The size of typical inputs increases over the lifetime of the algorithms' use.
  - The size and typicality of inputs might vary widely, depending on the application of the algorithm.
  - The computer might get upgraded in some near future. Or the programs might be rewritten for some unknown system.

- We then also work to ***estimate*** running times.
  - ➡ We use ***running time analysis***.

RUNNING TIME ANALYSIS

- Typical major concerns of running time estimation:

  - How does the running time scale (roughly) with input complexity?
    E.g. searching for an item in a list of size *n*
    - We will estimate "limiting" or ***asymptotic*** running time.

  - For a particular input sizer, what are the trickiest inputs the code will face?
    E.g. the search might have to scan the whole list.
    - We sometimes give ***bounds*** on the ***worst cases***.

RUNNING TIME ANALYSIS

- Typical lesser concerns of running time estimation:
  - E.g. Something that runs 11% faster on one machine over another.
  - E.g. **Program A** runs a little slower on small inputs well on large inputs (0.2sec versus 0.15sec for **Program B**), even though **Program A** runs must faster on large inputs (20sec versus 1000sec for **Program B**).

ASYMPTOTIC EQUIVALENCE

- Let's formalize some of these ideas:
  - ➡ Two algorithms' running times are ***asymptotically equal*** if, for large inputs, which algorithm is faster *depends on the relative speed of their executing computers*.

- Example scenario:
  - Suppose **algorithm A** takes $n^3 - 4n^2$ steps on an n-bit input.
  - Suppose **algorithm B** takes $10n^3+15$steps on an n-bit input.
    - ➡ If **A** and **B** run on the same computer, **A** runs faster.
    - ➡ If **B** runs on a 100x faster machine, it beats **A** on large inputs.

ASYMPTOTIC EFFICIENCY

- Let's formalize some of these ideas:
  - Two algorithms' running times are ***asymptotically equal*** if, for large inputs, which algorithm is faster depends on the relative speed of their executing computers.

- We define $\Theta(g(n))$, the set of functions asymptotically equal to g, with:

Definition: $f(n)$ is in the set $\Theta(g(n))$ whenever there exist positive constants L and U, and a positive constant m where

$$L\,g(n) \leq f(n) \leq U\,g(n)$$

for all $n \geq m$.

BIG THETA

Definition: $f(n)$ is in the class $\Theta(g(n))$ whenever there exist positive constants L and U, and a positive constant m where

$$L\,g(n) \leq f(n) \leq U\,g(n)$$

for all $n \geq m$.

Examples:

$n^3 - 4n^2$ is in the class $\Theta(10n^3+15)$

$10n^3+15$ is in the class $\Theta(n^3 - 4n^2)$

$n^3 - 4n^2$ is in the class $\Theta(n^3)$

$10n^3+15$ is in the class $\Theta(n^3)$

NOTE: All these functions grow as ***cubic functions*** of n.

BIG THETA

Definition: $f(n)$ is in the class $\boldsymbol{\Theta}(g(n))$ whenever there exist positive constants L and U, and a positive constant m where

$$L\,g(n) \leq f(n) \leq U\,g(n)$$

for all $n \geq m$.

Examples:

Searching… an entire list of length n takes $\boldsymbol{\Theta}(n)$ time.

…a sorted list of size n to discover that a value is missing is $\boldsymbol{\Theta}(\log_2(n))$ time.

Computing 2^n using repeated squaring takes $\boldsymbol{\Theta}(\log_2(n))$ time.

Computing 2^n by multiplying n times takes $\boldsymbol{\Theta}(n)$ time.

BIG OH

Definition: $f(n)$ is in the class $\mathbf{O}(g(n))$ whenever there are positive U and m such that

$$0 \leq f(n) \leq U\,g(n)$$

for all $n \geq m$.

Examples:

$n^3 - 4n^2$ is in the class $\mathbf{O}(10n^3+15)$

$10n^3+15$ is in the class $\mathbf{O}(n^3 - 4n^2)$

n^2 is in the class $\mathbf{O}(n^3)$

$100000n+987987987$ is in the class $\mathbf{O}(n)$

- We use "big Oh" to say “asymptotically grows no faster than…”

BIG OH

Definition: $f(n)$ is in the class $\mathbf{O}(g(n))$ whenever there are positive U and m such that

$$0 \leq f(n) \leq U\,g(n)$$

for all $n \geq m$.

Examples:

Searching a list of length n takes $\mathbf{O}(n)$ time.

Searching a sorted list of size n using binary search takes $\mathbf{O}(\log_2(n))$ time.

- We use "big Oh" to say "asymptotically grows no faster than…"

FACTS ABOUT BIG THETA AND BIG OH

- $\mathbf{O}(f(n) + g(n)) = \mathbf{O}(\max(f(n), g(n)))$
- $\mathbf{\Theta}(f(n) + g(n)) = \mathbf{\Theta}(\max(f(n), g(n)))$

```
def algorithm(n,…):
    algorithm_part_1(n,…) # runs in f(n) steps
    algorithm_part_2(n,…) # runs in g(n) steps
```

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?

- (Big diagram on the chalkboard.)

RUNNING TIME OF MERGESORT?

- Runs in $\mathbf{O}(n\log_2(n))$ time.