ALGORITHM EFFICIENCY

LECTURE 12-1

JIM FIX, REED COLLEGE CSCI 121

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.

▸ For both of these, a program will typically compute its answer faster.

• It runs faster.

▸ Today we'll focus on *running time*.

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 pow2'
>>> s = 'pow2.pow2(20)'
>>> timeit.timeit(stmt=s,setup=i,number=100)
0.0002275099977850914
```

▸ This will time 100 evaluations of `pow2(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 size, 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

▸ Other concerns of running time estimation

- Something that runs 11% faster on one machine over another.

- **Program A** runs a little <u>slower</u> on small inputs (0.2sec versus 0.15sec for **Program B**) but much <u>faster</u> on large inputs (20sec versus 1000sec for **Program B**).

ASYMPTOTIC RUNNING TIME

▸ Let's formalize some of these ideas:

→ We *asymptotically* compare running times of two algorithms: for large inputs, which algorithm is *faster*?

▸ Example scenario:

- Suppose **algorithm A** takes *$0.001n^3$* steps on an input of size *n*.
- Suppose **algorithm B** takes $100000n$ steps on an input of size *n*.

→ For *n=1* up to *10000*, **A** runs faster.

→ For *n=10001* onwards, **B** runs faster.

ASYMPTOTIC EQUIVALENCE

▸ More subtle comparison:

⤷ 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 speedier machine, it beats **A** on large inputs.

ASYMPTOTIC EFFICIENCY

▸ More subtle comparison:
　　⤷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 $\Theta(\,g(n)\,)$ 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 from the last lecture:

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

…a balanced BST of size n to discover that a key is missing is $\Theta(\log_2(n))$ time.

A nested pair of loops that sum the products $i*j$ takes $\Theta(n^2)$ time.

Computing `pow2(n)` using repeated squaring takes $\Theta(\log_2(n))$ time.

Computing `pow2(n)` by multiplying 2 of n times takes $\Theta(n)$ time.

Computing `pow2(n)` by summing 1s takes $\Theta(2^n)$ time.

BIG OH

Definition: $f(n)$ is in the class **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 **O**($10n^3 + 15$)

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

n^2 is in the class **O**(n^3)

$100000n + 987987987$ is in the class **O**(n)

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

BIG OH

Definition: $f(n)$ is in the class **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 **O**(n) time.

Searching a balanced BST of size n takes **O**$(\log_2(n))$ time.

Searching a BST of size n takes **O**(n) time.

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

A CASE STUDY: SEARCHING A LIST

SEARCHING A LIST

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

SEARCHING A SORTED LIST

Can we do better if a list is sorted?

▸ Suppose that
`someList[0]` ≤ `someList[1]` ≤ … ≤ `someList[n-1]`

SEARCHING A SORTED LIST

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

SEARCHING A SORTED LIST

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

▸ With each **someList[middle]** check, we eliminate half the undetermined list items from consideration.

▸ This means we inspect the list **O**($\log_2 (n)$) times.

SUMMARY

▸ In running time analysis use asymptotic notation to describe efficiency.

- We use **Big Theta** for asymptotic equivalence.

- We use **Big Oh** for asymptotic guarantees, i.e., *upper bounds*.

▸ Two classic searching and sorting algorithms:

- Binary search is a *logarithmic time* algorithm. It works on sorted lists.

- Bubble sort is a *quadratic time* algorithm. It sorts a list.

▸ Can we sort faster than in quadratic time?