Reminder: Quiz Thursday evening

Want to move on to talk about efficiency. Often an important consideration in designing a program.

Goal of next couple of lectures is not to make you an expert in this.

Do want to give you some intuition about how to approach questions of efficiency. Help you to understand why some programs complete before you know it, others you let run overnight, and still others that are not going to finish until you are old and gray.

Talk about importance of efficiency
   Computers are fast
   But problems can be very large
    E.g., I’m involved in a research where we are analyzing over a billion heart beats and we run computations that take weeks to complete.

When problems are large, brute force usually doesn’t get the job done

Efficiency mostly about algorithm choice, not about clever coding.

Clever algorithms hard to invent. A successful computer scientist might invent maybe one important algorithm during their whole career—if they are lucky. I haven’t invented any.

Therefore important to learn to reduce problem to previously solved problems.

We will spend time in 6.00 looking at how to do this.

Let’s start by asking, How should we think about efficiency?

Two dimensions, space and time. As we will see later in the term, one can often trade one for the other. We will focus on time for now.

Suppose I ask you “how long does the algorithm implemented by this program take to run?” How would you go about answering this question?

Could run the program on some input and time it?

Not a very good answer. Depends upon too many things.

   Speed of machine
   Cleverness of Python information
   And, most importantly, input on which you test it.

Typically, we ask what is the number of "basic steps" taken by an algorithm,
as a function of the size of its input:  \( T: \mathbb{N} \rightarrow \mathbb{N} \)

This is good because:

- It lets us compare algorithms
- Let’s us talk about how the running time grows with respect to the size of the input

“Input size” can be defined in terms of the number of bits, nodes, elements, integers, and so on. When talking about efficiency always important to be careful to specify what it is in terms of.

A "step" is an operation that takes constant time, such as a variable assignment, a comparison, an array access, an arithmetic function, and so on.

For simplicity we will use random access machines (RAM) as our model of computation. In RAM instructions are executed one after another (sequentially), i.e., no concurrency (mention PRAM and human brain). Also assume that time required to access an object is constant.

These are pretty standard and good assumptions.

There are three broad cases to think about:

- **best case** running time is the running time the algorithm has when the input is as favorable as possible. In other words, it is the minimum running time over all the possible inputs of a given input size.

Similarly, the **worst case** running time is the worst (maximum) over all possible inputs of a given size, and

- the **average case** (or expected case) running time is the average over all the inputs of a given size.

People usually focus on the worst case. Some good reasons for this;

Human nature. All engineers believe in Murphy’s Law. We may be Christian or Muslin or Jewish or Hindu or atheist. But there is one article of faith on which we can all agree, “If something can go wrong, it will go wrong.”

**Worst case**

1. Provides an upper bound. No unpleasant surprises when we use program.
2. In a lot of situations the worst case occurs often. For example, in many search problems worst case occurs when search fails.
3. Expected case can be hard to characterize, since it depends on expected distribution of inputs.
Returning to the issue of computational efficiency, consider the following code:

```python
def f(n):
    assert n >= 0
    answer = 1
    while n > 1:
        answer *= i
        n -= 1
    return answer
```

What mathematical function is `f` computing? Factorial.

How many steps does it take to run this function? 2 + 3*n + 1

So if n is 3000, it takes 9003 steps.

As we look at large size inputs, additive constant factors tend to become irrelevant. We usually ignore these. We don’t care if a program takes 9000 steps or 9003 and steps.

As inputs get truly enormous, even multiplicative constnats lose relevance. Do we really care whether a program will run for 3000 or 9000 years?

For this reason, we focus on describing the rate of growth of a program relative to its input.

Asymptotic notation is a way of describing functions without having to deal with distracting details.

The most well-known symbol in asymptotic notation is Big Oh notation.

Why Oh? Oh my god, will this program ever finish!

Not really. Introduced with this usage by the computer scientist Donald Knuth. He chose the Greek letter Omicron because it was used as early as the late 19th century for similar notions in calculus.

It is used to give an upper bound for the asymptotic growth of a function. For example, if we write

\[ f(x) \in O(n^2) \]

this means that the function f grows no faster than the quadratic polynomial \( n^2 \), in an asymptotic sense. I.e., this is an upper bound on the complexity of f.

Some of the most common instances of big O:
\(O(1)\): constant
\(O(\log n)\): logarithmic
\(O(n)\): linear
\(O(n \log n)\): loglinear
\(O(n^c)\): polynomial
\(O(c^n)\): exponential

To give you an idea of what these classes mean, let’s look at some plots. In a few weeks, we’ll talk about how to produce such plots.

Run `showGrowth.py`

For large problems even quadratic growth may make things impractical

Exponential growth quickly becomes unmanageable

10**256 is an unimaginably large number

Need to use a log y axis to even see what’s going

Switch to semilogy for last plot

Back to basics

In characterizing an algorithm we are usually interested in the smallest class to which it belongs. When we ask what is the complexity of this algorithm, we are looking for a tight bound. Formally speaking, people use Big Theta \(\Theta\), for this. However, when most people write \(O(f(x)) = g(x)\) they take it to mean that \(g(x)\) is a reasonably tight approximation to the worst case running time of \(f\). That is the way we will use it here.

So, what is the complexity of \(f\)? \(O(n)\)

How about a recursive factorial?

```python
def fact(n):
    # assume n >= 0
    if n <= 1: return 1
    else return n*fact(n - 1)
```

Key thing here is number of times fact gets called. Each time we call it, we call it on a number that is 1 smaller than the previous number, so there are \(O(n)\) calls. I.e., it is \(O(n)\)
Will this run more slowly than the iterative version? Yes, because a function call takes a bit of time. Do we care? No. The two versions are of the same order of complexity, and the multiplicative constant is small.

How about:

```python
def g(n):
    x = 0
    for i in range(n):
        for j in range(n):
            x += 1
    return x
```

This is $O(n^2)$
g computes $n^2$ in a clumsy way

The number of times through a loop (or number of recursive calls) are the key things to look at. Everything else is additive.

How about the following (where x is an integer)

```python
def h(x):
    assert type(x) == int and x >= 0
    answer = 0
    s = str(x)
    for c in s:
        answer += int(c)
    return answer
```

What is its order of growth? Order log base 10 of x

Lesson: Very important to focus on what size of input means. In one case size is the magnitude of a number in the other the number of digits in number. Need to say something like $O(n)$, where $n$ is the number of elements in the list.

Reviewing, asymptotic notation is designed to let us ignore things that don't matter when we look at the big picture. For example, if we want to compare the running times of two algorithms, we're mainly interested in how fast the time grows when the problem size grows.

We're not really interested, for example, in which machine each algorithm is run on, or, for example, any fixed start-up times that may differ between the algorithms. What we want is a description of the running time (for example, or the memory use) that is independent of such things.

A notation that gives us only the core of the function; the part that describes its growth.

Typically, people talk about one algorithm as being more efficient than
another if its worst case running time has a lower order of growth.

Now, let’s look at two algorithms for searching a list that is known to be sorted:

```python
def search(L, e):
    for elem in L:
        if elem == e:
            return True
        if elem > e:
            return False
    return False
```

```python
L = range(100)
print search(L, 3)
print search(L, 50)
print search(L, 100)
```

This program is $O(n)$, where $n$ is len(L)*\text{test}(L, i, e),

Where \text{test}(L, i, e) is the time it takes to test whether the ith element of L is equal to e.

As it happens $t(L, i, e)$ is $O(1)$. I.e., it does not depend on properties of L or i. It runs in constant time. We will show you why this is true shortly.

So, search is $O(\text{len}(L))$

Discuss version with while loop. Different code, but same algorithm

```python
def search(L, e):
    i = 0
    while i < len(L):
        if L[i] == e:
            return True
        if L[i] > e:
            return False
        i += 1
    return False
```

Is this the best we can do?

No. Think of our old friend bisection search.

Pick the mid point
Ask if this is what we are looking for
If not
    Find a smaller problem
    Solve that problem

Given this structure, it should not surprise you I have chosen to write a recursive implementation. I have a nice base case and an obvious recursive
def bSearch(L, e, low, high):
global numCalls
    numCalls += 1
    if high - low < 2:
        return L[low] == e or L[high] == e
    mid = low + int((high - low)/2)
    if L[mid] == e:
        return True
    if L[mid] > e:
        return bSearch(L, e, low, mid - 1)
    else:
        return bSearch(L, e, mid + 1, high)
def search(L, e):
    return bSearch(L, e, 0, len(L) - 1)

*** Explain
Use of search function to kick things off. User of search should not care that it is bsearch. Needs to be the same interface as the other search function.

Use of global variable

Run tests

L = range(100)
numCalls = 0
search(L, 100)
msg = 'Number of calls when length = '
print msg + str(len(L)) + ' is', numCalls
L = range(200)
numCalls = 0
search(L, 200)
print msg + str(len(L)) + ' is', numCalls
L = range(400)
numCalls = 0
search(L, 400)
print msg + str(len(L)) + ' is', numCalls
L = range(800)
numCalls = 0
search(L, 800)
print msg + str(len(L)) + ' is', numCalls
L = range(1600)
numCalls = 0
search(L, 1600)
print msg + str(len(L)) + ' is', numCalls
L = range(10000000) #ten million
numCalls = 0
search(L, 10000000)
print msg + str(len(L)) + ' is', numCalls

Double size of array, note that it took exactly one more step. Beauty of log algorithms. Time grows very slowly.

Now, let's go back to analyzing bSearch
  First line, constant
  Second line raises the question about test we deferred earlier, and we will defer once again

So, let's assume that second line can be executed in constant time.

Back to binary search

What's constant + constant? Constant time

Lines 3 and 4
  mid = first + (last - first)/2
  if s[mid] == e: return True
also constant time.

Now the fun starts:
  if L[mid] > e:
    return bSearch(L, e, low, mid - 1)
  else:
    return bSearch(L, e, mid + 1, high)

These lines are not constant time because each causes a call to bsearch which may in turn cause more calls to bsearch.

In an algorithms class I would show you a formal way to analyze this using something called a recurrence relation. But here I want to be much less formal.

Start with the key question. How do we know that this program doesn’t run until it runs out of memory? I.e., why do you believe that bsearch gets called only a finite number of times?

Have to pull back and look at the entire function and ask when there is a recursive call:

```python
def bSearch(L, e, low, high):
  global numCalls
```
numCalls += 1
if high - low < 2:
    return L[low] == e or L[high] == e
mid = low + int((high - low)/2)
if L[mid] == e:
    return True
if L[mid] > e:
    return bSearch(L, e, low, mid - 1)
else:
    return bSearch(L, e, mid + 1, high)

We call bsearch only if last - first >= 2

In each call we reduce the value of last - first by approximately (last-first)/2

How many times can I divide a number ,n,  by 2 before it is <= 2?

log(n)

---Ended here---

Now, back to the question of testing whether an element of a list is equal to the element for which we are searching.

Complexity of this depends upon time required to locate the ith element of a list. This depends upon how lists are implemented in Python. Let’s look at that for a few minutes.

In this case, we have a list in which each element is the same size, say, for simplicity that an int occupies 4 units of memory. How do I find where the ith elements of the list is in memory?

    start of list + 4*I    -- Constant time

I.e., our random access model of being able to fetch any location in constant time holds

But we know that lists in Python don’t have to be homogeneous. Suppose I have a list with ints, floats, strings, lists, etc.?

Could lay it out in memory as what is called a linked list

    <ptr to next elem, val>

In this case the complexity of accessing an element of the list is O(len(list)), since I have to visit every element to get to the last one.

But this is not the best way to implement a list of this kind

    <ptr to first>, <ptr to second>, ...<None>
Now we can again fetch any element in constant time. This is the usual way to implement lists in object-oriented languages, and what I believe is done in most Python implementations.

This example illustrates one of the most important implementation techniques that we know, *indirection*.

My dictionary defines “indirection” as “lack of straightforwardness and openness: deceitfulness”

Until about 1950 it had a pejorative implication. Than the computer scientists got a hold of it.

In computing, indirection is the ability to access something using a name or reference instead of the value itself.

*All problems in computer science can be solved by another level of indirection*

*Except having too many levels of indirection*