Module 07: Efficiency

Topics:

- Basic introduction to run-time efficiency
- Analyzing recursive code
- Analyzing iterative code

Consider the following two ways to calculate the maximum of a nonempty list.
Why is one so much slower than the other?

```
def list_max(L):
    max_so_far = L[0]
    for v in L:
        if v > max_so_far:
            max_so_far = v
    return max_so_far
```

```
def list_max1(L):
    if len(L) == 1:
        return L[0]
    elif L[0] > list_max1(L[1:]):
        return L[0]
    else:
        return list_max1(L[1:])
```

Comparing Algorithms

Suppose you have two algorithms to solve a problem. How can we determine which one is better?

- Which is easier to understand? Implement? Accurate? More robust? Adaptable? Efficient?

- We will use *efficiency* to compare algorithms.

Efficiency

The most common measure of efficiency is ***time efficiency***, or how long it takes an algorithm to solve a problem.

- Depends on its implementation

Another measure of efficiency is ***space efficiency***, or how much space (memory) an algorithm requires to solve a problem.

Efficiency: measurement of Running Time of an algorithm

What is our unit of measurement? Seconds?

- Dependent on when statement made, what computer, how much RAM, what language used, what OS, etc.
- Do we consider the average time over all possible problems? Just one? Which one?

The actual time taken is not a great choice. Instead, we will count number of steps or basic operations performed.

Example

What is the number of operations executed when calling this function?

```
def sum_all(values):
    sum = 0
    ind = 0
    upper = len(values)
    while (ind < upper):
        sum = sum + values[ind]
        ind = ind + 1
    return sum
```

Input size

- Let n refer to the size of the problem
  - Length of list
  - Number of characters in a string
  - The number itself
  - Number of digits in a Nat
  - Meaning should be specified if not clear
- Running time is always stated as a function of n. We denote it by $T(n)$

Running time depends on data values, not just input size

- Assume $n = len(L)$
- How many steps are taken by the following code? What does it do?

```
ind = 0
length = len(L)
while (ind < length) and (L[ind] > 0):
        ind = ind + 1
```

Terminology

- We will be pessimistic, and determine the largest value of $T(n)$ possible for a fixed n
  - ***Worst case running time***
  - This is our default meaning of "run time"
- Sometimes we are interested in the ***best case***, i.e. the minimum value of $T(n)$ possible for a fixed value of n

Big O notation

- In practice, we are not concerned with the difference between the running times $6n + 6$ and $174n + 32$.
- We are interested in the ***order*** of a running time. The order is the *dominant* term without its coefficient.
- The dominant term in both $6n + 6$ and $174n + 32$ is n, so both are ***"Order n"***, denoted $\boldsymbol{O(n)}$
- This is called the *asymptotic* run time

Big O Examples

- $2016 = O(1)$
- $12 \log n + 45 = O(\log n)$
- $12 \log n + 45n = O(n)$
- $20\, n \log n + 3n + 27 = O(n \log n)$
- $3 + n + n^2 + 2^n = O(2^n)$

Important Big O information

- In this course, we will encounter only a few orders (arranged from smallest to largest): $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$
- Note that these relationships hold as $n \to \infty$
- When comparing algorithms, the most efficient algorithm is the one with the lowest order.
- If two algorithms have the same order, they are considered equivalent, but may not take exactly the same number of steps.

What is the running time of this code?

```
def list_max(values):
    max_so_far = values[0]
    for v in values:
        if v > max_so_far:
            max_so_far = v
    return max_so_far
```

Big O arithmetic

- When adding two orders, the result is the larger of the two orders.
  - $O(\log n) + O(n) = O(n)$
  - $O(1) + O(1) = O(1)$
- How can we use this result?
  - Break code into blocks that run one after the other
  - If you determine the asymptotic run times of the blocks independently, then just add them to get the overall run time.

Algorithm analysis

- An important skill in Computer Science is the ability to analyze a program and determine the order of its running time.
- In this course, you will not need to count operations exactly.
- Our goal is to give you experience and to work towards building your intuition.

```
sum=0
for x in lst:
    sum = sum + x
```

Each item in the list is retrieved once, so running time is $O(n)$

Basic Operations in Python

We will make the following assumptions

- Numerical operations :
  - **`+`**, **`-`**, **`*`**, **`/`**, **`=`** are $O(1)$
  - **`max(a,b)`**, **`min(a,b)`** are $O(1)$ for numbers **`a`** and **`b`**
  - **`a==b`** is $O(1)$ for numbers **`a`** and **`b`**

Basic Operations in Python

We will make the following assumptions

- String operations, where $n = len(s)$
  - **`len(s), s[k]`** are $O(1)$
  - **`s + t`** is $O(n + len(t))$
  - Most string methods (e.g. **`count, find, lower)`** are $O(n)$
- **`print`** and **`input`** are dependent on the length of what is being printed and read in

Basic List Operations, where $n = len(L)$

We will make the following *assumptions*:

- `len(L)`, `L[k]` are $O(1)$
- `L + M` is $O(n + len(M))$
  - `L + [x]` is $O(n)$
- `sum(L)`, `max(L)`, `min(L)` are $O(n)$
- `L[a:b]` is $O(b - a)$, so at most $O(n)$
  - `L[1:]` is $O(n)$
  - `L[3:5]` is $O(1)$
- `L.append(x)` is $O(1)$

More basic list operations, where $n = len(L)$

We will make the following *assumptions*:

- **`list(range(n))`** is $O(n)$
- **`[x]*n`** is $O(n)$
- Most other list methods on **`L`** (e.g. **`count`**, **`index`**, **`insert`**, **`pop`**, **`remove`**) are $O(n)$
- **`L.sort()`** is $O(n \log n)$
- **`L.extend(M)`** is $O(len(M))$
  - *Note that* ***`extend`****'s run-time is independent of* ***`n`***

Here are two ways to duplicate a list. Which is most efficient?

```
def duplicate1(L):
    extra = []
    for x in L:
        extra.append(x)
    return extra
```

```
def duplicate2(L):
    extra = []
    for x in L:
        extra = extra + [x]
    return extra
```

General Procedure for analyzing a loop

- Determine the number of iterations
- For each iteration, determine the running time of body of the loop
  - Each loop body may have the same running time, but that is not guaranteed
- Add together the running time of each loop body to get the overall running time

More Big O arithmetic

- When multiplying two orders, the result is the product of the two orders.
  - $O(\log n) * O(n) = O(n \log n)$
  - $O(n) * O(n) = O(n^2)$
- How can we use this result?
  - Determine the asymptotic run time of the number of iterations of a loop
  - Determine the asymptotic run time of the body of the loop
  - Multiply them to get the overall asymptotic run time

Warning: The following code fragments do NOT have the same runtime. Why?

```
diff = 0
for x in L:
    diff += abs(x - sum(L)/len(L))
```

```
diff = 0
mean = sum(L)/len(L)
for x in L:
    diff += abs(x-mean)
```

Be very careful about what steps are put inside the loop body. Try to move non-$O(1)$ *steps outside the loop body, when possible.*

What if there are nested loops?

- You can take different approaches:
  – Work from the innermost loop to the outermost
  – Work from the outermost loop to the innermost
- Nested loops can lead to nested sums

What is the running time of `mult_table(n)`?

```
def mult_table(n):
    table = [0]*n
    row = 0
    columns = list(range(n))
    while row < n:
        this_row = []
        for c in columns:
            this_row.append((row+1)*(c+1))
        table[row] = this_row
        row = row + 1
    return table
```

Useful summations

- $\sum_{i=1}^{n} 1 = O(n)$
- $\sum_{i=1}^{n} i = O(n^2)$
- $\sum_{i=1}^{n} n = O(n^2)$
- $\sum_{i=1}^{n} \sum_{j=1}^{n} 1 = O(n^2)$

How do we determine runtime of recursive code?

```
def list_max(L):
  if len(L) == 1:
     return L[0]
  else:
     return
       max(L[0],
        list_max(
          L[1:]))
```

- Count steps for:
  - Determine `len(L)`
  - Compare to 1
  - Calculate `L[0]`
  - Calculate `L[1:]`
  - Call `list_max` recursively on a list of length n-1
  - Determine `max` of two values
- T(n) = O(n) + T(n-1)

More generally …

- To help in analyzing recursive code, we will use basic recurrence relations.
- We will express the running time of a problem of size n in terms of
  - Running time of the code other than recursion
  - Running time of recursive call(s)
- For example:
  - $T(n) = O(n) + T(n-1)$

Helpful recurrence relations

- Once we have such a recurrence relation, use the following rules to determine the overall running time.
- $T(n) = O(1) + T(n-1) \rightarrow O(n)$
- $T(n) = O(n) + T(n-1) \rightarrow O(n^2)$
- $T(n) = O(1) + T({}^{n}/_{2}) \rightarrow O(\log n)$
- $T(n) = O(n) + 2T({}^{n}/_{2}) \rightarrow O(n \log n)$
  - $T(n) = O(n) + T({}^{n}/_{2}) \rightarrow O(n)$
- $T(n) = O(1) + T(n-1) + T(n-2) \rightarrow O(2^n)$
  - $T(n) = O(1) + 2T(n-1) \rightarrow O(2^n)$
  - $T(n) = O(n) + T(n-1) + T(n-2) \rightarrow O(2^n)$
  - $T(n) = O(n) + 2T(n-1) \rightarrow O(2^n)$

Here are two ways to find maximum in a list. Which is more efficient?

```
def list_max1(L):
    if len(L) == 1:
        return L[0]
    elif L[0] > list_max1(L[1:]):
        return L[0]
    else:
        return list_max1(L[1:])
```

```
def remember_max(m, L):
    if len(L)==0:
        return m
    elif m > L[0]:
        return remember_max(m, L[1:])
    else:
        return remember_max(L[0], L[1:])
def list_max2(L):
    return remember_max(L[0], L[1:])
```

Analysing abstract list functions

- **`map(f,L)`**, **`filter(f,L)`** are at least $O(n)$
- Actual running time depends on running time of **`f`**
- Hint: *Analyse the program as if it were a loop instead of* ***`map`*** *or* ***`filter`***

Determine the running times

```
def duplicate3(L):
    return list(map(lambda x:x, L))
```

```
def first_chars(words):
    return list(map(lambda t: t[0],
                    filter(lambda s:len(s)>0,
                           words)))
```

```
def list_of_lists(n):
    return list(map(lambda x:
                        list(range(n)),
                    range(n)))
```

Overall comments

- We've provided just a basic introduction to runtime analysis
  - Especially for recursive code
  - We have made some simplifications
- The topic is very important, though, and even an introduction can help you design better programs.
- Like this topic?
  - CS234 (non-majors)
  - CS240 (majors)

Summary of Common Runtimes

Functions grow more quickly

	Common Name
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log Linear
$O(n^2)$	Quadratic
$O(2^n)$	Exponential

Algorithms run more slowly

Goals of Module 07

- Understand how to analyze Python code to determine its running time, including
  - Recursion
  - Iteration
  - Abstract list functions
- Understand basic run time categories