Module 10: Efficiency

Exercise

If you have not already, get prepared for class by downloading the start code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/module-10-start.ipynb
```

Comparing Algorithms

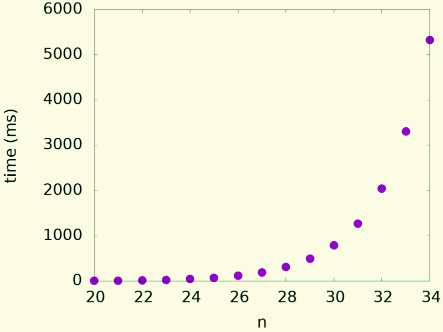
- Given two algorithms which both solve a problem, how can we tell which is better?
- Which is easier to understand? Implement? Accurate? More robust? Adaptable? Efficient?
- We define efficiency as how much of something the algorithm requires.
- The **something** is usually time, but sometimes space (memory).
- Faster is better.

In Jupyter, look at the file you get by running:

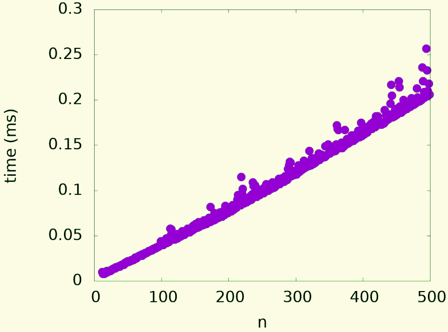
```
!wget https://student.cs.uwaterloo.ca/~cs114/src/efficiency_measurement.ipynb
```

Example: measuring time for Fibonacci

```python
def fibr(n: int) -> int:
    """Return the n-th number in the
    Fibonacci sequence starting 0 1."""
    if n < 2: return n
    else: return fibr(n-1) + fibr(n-2)
```




```python
def fibt(n: int, f0: int, f1: int) -> int:
    """Return the n-th number in the
    Fibonacci sequence starting f0 f1."""
    if n == 0: return f0
    else: return fibt(n-1, f1, f0+f1)
```



`fibt` is **vastly** faster.... We are not concerned about small improvements.

Measuring time

Time in seconds depends on the exact computer (a faster processor runs faster), what language, compiler settings,...

- Instead of counting seconds,we will measure the **number of steps** or basic operations performed.
  For example, the number of additions or multiplications.
- Sometimes different inputs will cause different running times. We could consider best case, average case, or worst case.
- We will consider the **worst case**: assume data are organized as badly as possible.
- We will be informal; take CS234/240 for details.

Problem size

We use n to refer to the size of the problem. But this depends on the context.

Running time is a function of n, denoted $T(n)$.

```python
def sum_all(values: list[int]) -> int:
    total = 0
    i = 0
    upper = len(values)
    while (i < upper):
        total = total + values[i]
        i = i + 1
    return total
```

Let n be the length of the list.
How many steps?
Something like $6n+6$. But we don't really care about the constants.

Exercise

Pick a small `int` for `n`, different from the person beside you.

Count how many times `+` is used by the following program:

```
total = 0
for i in range(0, n):
    for j in range(0, n):
        total = total + j
```

In general, n^2 additions are done.

We are not concerned about small improvements:

- Removing a constant amount does not really matter; $6n+6$ is not much worse than $6n$.
- The coefficient does not matter; $6n$ is not much worse than n.
- We are interested in the **order** of the running time: the **dominant term without its coefficients.**
- So $6n+6$, $6n$, n, and $174n+32$ are all considered to be more or less the same. They are "order n", which we denote $O(n)$.
- This is the **asymptotic** running time; what $T(n)$ approaches as n gets large.

A few Examples

- $24601 = O(1)$
- $12\sqrt{n} + 45 = O(\sqrt{n})$
- $20n^2 + 3n + 27 = O(n^2)$
- $3 + n + n^2 + 2^n = O(2^n)$

Best Case/Worst Case

Consider the following. (Let n be the length of L.)

```
def has10(L: list[any]) -> bool:
    i = 0
    while i < len(L):
        if L[i] == 10:
            return True
        i = i + 1
    return False
```

How many steps take place if $L = [10, 0, 0, 0, \ldots, 0]$?

How many steps take place if $L = [0, 0, 0, \ldots, 0, 10]$?

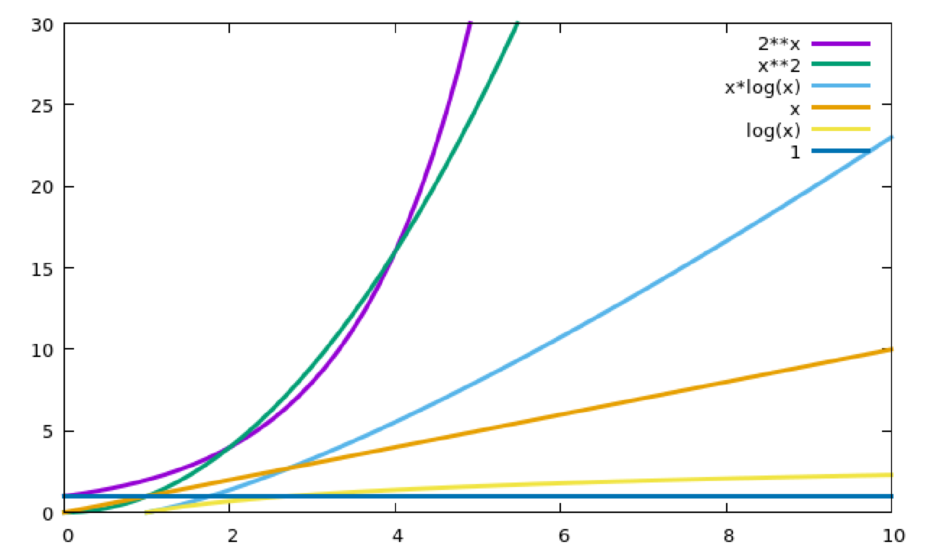
Important Big O information

- In this course we will encounter only a few orders:

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

- Note that these relationships hold as $n \to \infty$
- When comparing algorithms, the most efficient is the one with the lowest order.
- If two algorithms have the same order, they are considered equivalent, even if they do not take exactly the same number of steps.

Important Orders



Big O arithmetic

When adding two orders, the result is the larger of the two orders.

- $O(\sqrt{n}) + O(n) = O(n)$
- $O(1) + O(1) = O(1)$

How can we use this result?

- Break code into blocks that run one after the other.
- Determine the asymptotic running times of the blocks independently.
  Add them to get the overall running time.

- Working with `float` and `int` values:
  - `+`, `-`, `*`, `/`, `//`, `%`, `=`, `==`, `>`, `<` etc are $O(1)$
  - `max(a,b)`, `min(a,b)` are $O(1)$

- Working with `str` values, where `n=len(s)`
  - `len(s)`, `s[i]` are $O(1)$
  - `s + t` is $O(n + \texttt{len}(\texttt{t}))$
  - Most methods (e.g. `split`, `join`, `count`) are $O(n)$.

- `print` depends on the length of what is being printed.

Basic operations on lists

Working with lists, where `n=len(L)`:

- `len(L)`, `L[k]` are $O(1)$.
- `L + M` is $O(n + \texttt{len(M)})$.
- `sum(L)`, `max(L)`, `min(L)` are $O(n)$
- `L.append(x)` is $O(1)$.
- `L.pop(0)` is $O(n)$, but `L.pop()` is $O(1)$.
- `L.sort()` and `sorted(L)` are $O(n \log n)$.
- Slicing: `L[a:b]` costs the length of the slice: `L[:]` is $O(n)$, `L[1:]` is $O(n-1)$, but this is also $O(n)$.
- Most other methods are $O(n)$.

General strategy

- Count iterations.
- For each iteration, determine the running time of the body.
- Multiply the iterations by the cost of each.
- Add totals and simplify.

```
s = []                  # cost is O(1)
for j in range(0, n):   # count iterations: O(n)
    s.append(i*j)       # body running time: O(1)
## "for j" runs n times, O(1) each time, so O(1) * O(n) ⇒ O(n)
```

```
t = list(range(n))
for j in range(0, n):      # count iterations: O(n)
    t = [t[-1]] + t[:-1]   # body running time: O(n)
## "for j" runs n times, O(n) each time, so O(n) * O(n) ⇒ O(n**2)
```

More Big O arithmetic

When multiplying two orders, the result is the product of the two orders.
$O(A) \cdot O(B) = O(A \cdot B)$

Important examples:

- $O(\log n) \cdot O(n) = O(n \log n)$
- $O(n) \cdot O(n) = O(n^2)$

So we can multiply the running time of the number of loop iterations by the running time of the body of the loop to get the overall running time.

Exercise

Which of these is more efficient?

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

$O(n^2)$

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

$O(n)$

Avoid re-computing things, and move non-$O(1)$ steps outside the loop when possible.

What is the worst case running time?

```python
def sum_odd(L: list[int]) -> int:
    M = L[:]
    total = 0
    while M != []:
        if M[0] % 2 == 1:
            total = total + M[0]
        M = M[1:]
    return total
```

```python
def sum_odd(L: list[int]) -> int:
    M = L[:]
    total = 0
    while M != []:       # M shortens by 1 each loop, so n iterations
        if M[0] % 2 == 1:
            total = total + M[0]
        M = M[1:]        # this line takes len(M) each time
    return total
```

Exercise

Rewrite `sum_odd` in $O(n)$. Hint: use a `for` loop, or use `pop()`.

```python
def sum_odd(L: list[int]) -> int:
    M = L[:]
    total = 0
    while M != []:       # M shortens by 1 each loop, so n iterations
```

What if there are nested loops?

The body of a loop may contain another loop. This changes nothing; we still:

- Count iterations.
- For each iteration, determine the running time of the body.
- Add them up.

If the body contains another loop, do this again to compute the running time of the body.

```
s = []
for i in range(0, n):     # count iterations: O(n)
    for j in range(0, n): # count iterations: O(n)
        s.append(i*j)     # body running time: O(1)
    ## "for j" runs n times, O(1) each time, so O(n) total
## "for i" runs n times, n times, O(n) each time, so O(n**2) total.
```

```
t = list(range(n))
for i in range(0, n):       # count iterations: O(n)
    for j in range(0, n):   # count iterations: O(n)
        t = [t[-1]] + t[:-1] # body running time: O(n)
    ## "for j" runs n times, O(n) each time, so O(n**2) total
## "for i" runs n times, O(n**2) each time, so O(n**3) total.
```

Hidden $O(n^2)$ summations

There are several ways we can get $O(n^2)$ unexpectedly:

```python
L = []
for i in range(0, n): ## O(n) iterations
  L = [i] + L               # 1, 2, 3, ..., (n-2), (n-1), n
```

$$\sum_{i=1}^{n} i = \underbrace{1 + 2 + \cdots + (n-1) + n}_{n} = \frac{n(n+1)}{2} = O(n^2)$$

```python
## Let n be the length of L
while L != []:          ## O(n) iterations
    L.pop(0)                # (n-1), (n-2), (n-3), ..., 2, 1, 0
```

$$\sum_{i=1}^{n} i - 1 = \underbrace{(n-1) + (n-2) + \ldots 2 + 1 + 0} = \frac{n(n-1)}{2} = O(n^2)$$

Key message: when you add up n items, and the cost per item grows (or shrinks) linearly, you will end up with $O(n^2)$ total cost.

Exercise

For each function,

1. determine the running time of each line, then
2. add these up to determine the total.

```
def sum(n: int) -> int:
    total = 0
    while n > 0:
        total = total + n
        n = n - 1
    return total
```

```
def list_sum(n: int) -> list[int]:
    total = []
    while n > 0:
        total = total + [n]
        n = n - 1
    return total
```

How do we determine runtime of recursive code?

```python
def list_max(L): # T(n): cost of whole function
    if (len(L)                 # O(1): determine len(L)
           == 1):              # O(1): compare to 1: O(1)
        return L[0]
    else:
        return (
          max(                 # O(1): max of two values
              L[0],            # O(1): calculate L[0]
              list_max(        # T(n-1): recursive call on 1 smaller
                  L[1:])))     # O(n): calculate L[1:]
```

Let $T(n)$ be the running time of the function. Then:

$$T(n) = O(1) + O(1) + O(1) + O(1) + T(n-1) + O(n)$$

Simplifying:

$$T(n) = O(n) + T(n-1)$$

Recurrence Relations

Analysis of recursive code gives a **recurrence relation** such as $T(n) = O(n) + T(n-1)$.

The running time of a problem is the sum of

- running time of the non-recursive code
- running time of the recursive call(s).

We don't want a recurrence relation, we want a running time.

We need to solve the recurrence relation. There are many techniques to do this. We will only consider simple cases which can be solved by drawing a tree.

We can **reason** through a recurrence relation to figure out its total cost, using **area** as an analogy.

Here are solutions to a bunch of important relations.

Don't memorize this. If you ever need these, I will give you this table.
(Outside of exams, you can look it up here).

1. $T(n) = O(n) + T(n-1) \rightarrow O(n^2)$
2. $T(n) = O(1) + T(n-1) \rightarrow O(n)$
3. $T(n) = O(1) + T(n/2) \rightarrow O(\log n)$
4. $T(n) = O(n) + 2T(n/2) \rightarrow O(n \log n)$
5. $T(n) = O(n) + T(n/2) \rightarrow O(n)$
6. $T(n) = O(1) + 2T(n-1) \rightarrow O(2^n)$
7. $T(n) = O(1) + T(n-1) + T(n-2) \rightarrow O(2^n)$ (or...?)
8. $T(n) = O(n) + 2T(n-1) \rightarrow O(2^n)$

Two ways to find maximum in a list

Exercise

For each algorithm:

1. Add a comment to each line, indicating the big-O running time of that line. **Use $T(...)$ to represent the running time of a recursive call.**
2. Add up the costs, simplify the recurrence, and look up its solution in the table.

```
def list_max1(x: list[float]) -> float:
    if len(x) == 1:
        return x[0]
    elif x[0] > list_max1(x[1:]):
        return x[0]
    else:
        return list_max1(x[1:])
```

```
def list_max2(y: list[float],
              m: float|None=None) -> float:
    if m == None:
        m = y[0]

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

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 a introduction can help you design better programs.

Like this topic? See CS 234 (non-CS-majors), CS 240 (CS-majors).

Solving recurrences also features in Math 229/239/249 and courses in C&O.

Write a function `is_prime(n)` that returns **`True`** if `n` is prime, and **`False`** otherwise.

What is the running time of your algorithm? Can you improve it?

Exercise

Suppose we wanted to find all the prime numbers between `0` and `n`. Write a function to do this.

Can you make your function run faster than running `is_prime(n)` repeatedly?

Module summary

- Understand how to analyse Python code to determine its running time, including code which is recursive or iterative.
- Recognize basic run time categories, from $O(1)$ to $O(2^n)$.

Read and complete the exercises in module 10 of the online textbook, at https://online.cs.uwaterloo.ca

Options for continuing CS

If you want to take more CS courses, enrol in **CS 115.**

In it you will:

- Use no loops, and change no variables, in a wacky language that has evolved since 1958,
- do some pretty abstract stuff, and
- recurse until you curse.

Talk to the CS advisors if you're interested in a CS major, computing minor, or whatever.

Participate in course selection!