Why efficiency of algorithm matters?

- An example of growth of functions:

<table>
<thead>
<tr>
<th>$n$</th>
<th>log $n$</th>
<th>$n$</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$3 \times 10^{-11}$ s</td>
<td>$10^{-10}$ s</td>
<td>$3 \times 10^{-10}$ s</td>
<td>$10^{-9}$ s</td>
<td>$10^{-8}$ s</td>
<td>$3 \times 10^{-7}$ s</td>
</tr>
<tr>
<td>$10^2$</td>
<td>$7 \times 10^{-11}$ s</td>
<td>$10^{-9}$ s</td>
<td>$7 \times 10^{-9}$ s</td>
<td>$10^{-7}$ s</td>
<td>$4 \times 10^{11}$ yr</td>
<td></td>
</tr>
<tr>
<td>$10^3$</td>
<td>$1.0 \times 10^{-10}$ s</td>
<td>$10^{-8}$ s</td>
<td>$1 \times 10^{-7}$ s</td>
<td>$10^{-5}$ s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$10^4$</td>
<td>$1.3 \times 10^{-10}$ s</td>
<td>$10^{-7}$ s</td>
<td>$1 \times 10^{-6}$ s</td>
<td>$10^{-3}$ s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$10^5$</td>
<td>$1.7 \times 10^{-10}$ s</td>
<td>$10^{-6}$ s</td>
<td>$2 \times 10^{-5}$ s</td>
<td>$0.1$ s</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>$10^6$</td>
<td>$2 \times 10^{-10}$ s</td>
<td>$10^{-5}$ s</td>
<td>$2 \times 10^{-4}$ s</td>
<td>$0.17$ min</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>
Why efficiency of algorithm matters?

- Another example of growth of functions:
Comparison of growth of functions

- When $n$ is arbitrarily big, growth of functions highly depends on the dominant term in the function:
  - $n+5$
  - $n+1000000$
  - $n^2+n+5$
  - $n^2+1000000n+5$
  - $2n^2 + n^3$
  - $n + \log n + n \log n$
  - $n + (\log n)^5 + n \log n$
  - $2^n + n^2$
  - $2^n + n^{200}$
Comparison of growth of functions

- Ignore coefficients as well:
  - $20n + 5$
  - $200n + 1000000$
  - $600n^2 + n + 5$
  - $200n^2 + 1000000n + 5$
  - $2n^2 + 50n^3$
  - $n + 5000 \log n + 300 n \log n$
  - $n + (\log n)^5 + 300 n \log n$
  - $2^n + 1000 n^2$
  - $10002^n + 2000 n^{200}$
Comparison of growth of functions

- Notation:
  - \(20n + 5\) \(O(n)\)
  - \(200n + 1000000\) \(O(n)\)
  - \(600n^2 + n + 5\) \(O(n^2)\)
  - \(200n^2 + 1000000n + 5\) \(O(n^2)\)
  - \(2n^2 + 50n^3\) \(O(n^3)\)
  - \(n + 5000 \log n + 300 \, n \log n\) \(O(n \log n)\)
  - \(n + (\log n)^5 + 300 \, n \log n\) \(O(n \log n)\)
  - \(2^n + 1000 \, n^2\) \(O(2^n)\)
  - \(10002^n + 2000 \, n^{200}\) \(O(2^n)\)
Ordering functions by big_O

- **Ordering:**

  - $20n + 5$ \( O(n) \) 1
  - $200n + 1000000$ \( O(n) \) 1
  - $600n^2 + n + 5$ \( O(n^2) \) 3
  - $200n^2 + 1000000n + 5$ \( O(n^2) \) 3
  - $2n^2 + 50n^3$ \( O(n^3) \) 4
  - $n + 5000 \log n + 300 n \log n$ \( O(n \log n) \) 2
  - $n + (\log n)^5 + 300 n \log n$ \( O(n \log n) \) 2
  - $2^n + 1000 n^2$ \( O(2^n) \) 5
  - $10002^n + 2000 n^{200}$ \( O(2^n) \) 5
Ordering functions by their growth

- Ordering:

  - $f_1(n) = (1.5)^n$
  - $f_2(n) = 8n^3 + 17n^2 + 111$
  - $f_3(n) = (\log n)^2$
  - $f_4(n) = 2^n$
  - $f_5(n) = \log (\log n)$
  - $f_6(n) = n^2 (\log n)^3$
  - $f_7(n) = 2^n (n^2 + 1)$
  - $f_8(n) = n^3 + n(\log n)^2$
  - $f_9(n) = 10000$
  - $f_{10}(n) = n!$
Time complexity of algorithms

- How time efficient is an algorithm given input size of $n$.

- The **worst-case time complexity**:
  - an upper bound on the number of operations an algorithm conducts to solve a problem with input size of $n$.

- We measure time complexity in the order of number of operations an algorithm uses in its worst-case and will demonstrate it using $\text{big}_O$.
  - ignore implementation details
def max(list):
    max = list[0]
    for i in range(len(list)):
        if max < list[i]:
            max = list[i]
    return max

Exact counting:
Count the number of comparisons:
• Assume len(list) = n
• The max < list[i] comparison is made n times.
• Each time i is incremented, a test is made to see if i < len(list).
• One last comparison determines that i ≥ len(list).
• Exactly $2n + 1$ comparisons are made.
• Consider the dominant term (as well as ignoring the coefficient)
• Hence, the time complexity of the max algorithm is $O(n)$. 
def max2(list):
    max = list[0]
    i=1
    while i< len(list):
        if max < list[i]: max = list[i]
        i+=1
    return max

Exact counting:
Count the number of comparisons:
- The max < list[i] comparison is made \( n-1 \) times.
- Each time \( i \) is incremented, a test is made to see if \( i < \text{len}(\text{list}) \).
- One last comparison determines that \( i \geq \text{len}(\text{list}) \).
- Exactly \( 2(n-1) + 1 = 2n - 1 \) comparisons are made.
- Consider the dominant term (as well as ignoring the coefficient)
- Hence, the time complexity of the max2 algorithm is \( O(n) \).
def silly(n):
    n = 17 * n**(1/2)
    n = n + 3
    print("n is: {}.").format(n)
    if n > 1997:
        print('very big!')
    elif n > 97:
        print('big!')
    else:
        print('not so big!')

Exact counting of the number of comparisons:
- Assume there is not any comparisons inside functions print or format
- Exactly 2 comparisons are made.
- Hence, the time complexity of the silly algorithm is $O(1)$.
- The number of comparisons in print/format is NOT depending on $n$
Estimating big-O

- Instead of calculating the exact number of operations, and then use the dominant term,

- Let’s just focus on the dominant parts of the algorithm in the first place.

- Dominant parts of algorithms are loops and function calls.

- Hence, two things to watch:
  1. We need to carefully estimate the number of iterations in the loops in terms of algorithm’s input size, i.e. $n$.
  2. If a called function depends on $n$ (i.e. it has loops that are in terms of $n$), we should take them into consideration.
watch loops and functions
Time complexity: Example 1 (revisited)

1. ```python
def max(list):
    max = list[0]
    for i in range(len(list)):
        if max < list[i]: max = list[i]
    return max
```

Calculating big-O:
Focus on the dominant part of the code
(normally loops, also be careful about function calls)
- Assume len(list) = \( n \)
- The dominant part is the `for` loop starting at line 3
  - Line 2 is minor, so is line 1, line 4, and line 5
  - None of these lines have a loop or a function call
- The `for` loop in line 3 iterates roughly \( n \) times
- Hence, the time complexity of the max algorithm is \( O(n) \).
Time complexity: Example 2 (revisited)

```python
1. def max2(list):
2.     max = list[0]
3.     i=1
4.     while i < len(list):
5.         if max < list[i]: max = list[i]
6.         i+=1
7.     return max
```

Calculating big-O:
Focus on the dominant part of the code
- Assume len(list) = n
- The dominant part is the **while** loop starting at line 4
- This **while** loop iterates roughly n times
- Hence, the time complexity of the max2 algorithm is $O(n)$. 
def silly(n):
    n = 17 * n**(1/2)
    n = n + 3
    print("n is: {}.").format(n))
    if n > 1997:
        print(‘very big!’)
    elif n > 97:
        print(‘big!’)
    else:
        print(‘not so big!’)

Calculating big_O:
Focus on the dominant parts (loops and function calls) of the code
- There is no loop; but there are some function calls
- The number of operations in print/format is NOT depending on \( n \)
- In other words, these function calls require constant amount of time
- Hence, the overall time complexity of the silly algorithm is \( O(1) \).
def silly2(n):
    n = 17 * n**2
    n = n + 3
    print("n is: {}.").format(n)

if n > 1997:
    for i in range(n):
        print('so big!')
elif n > 97:
    print('big!')
else:
    print('not so big!')

Calculate big_O:
Time complexity: Example 5

```python
def silly2(n):
    n = 17 * n**2
    n = n + 3
    print("n is: {}.".format(n))
    if n > 1997:
        print('so big!')
    elif n > 97:
        for i in range(n):
            print('big!')
    else:
        print('not so big!')
```

Calculate big_O:
What is the time complexity for this code fragment?

```python
sum = 0
for i in range(n//2):
    sum += i * j
```

The loop (roughly) iterates $\frac{1}{2} n$ times. Hence, it is $O(n)$
What is the time complexity for this code fragment?

```python
sum = 0
for i in range(n//2):
    for j in range(n**2):
        sum += i * j
```

The outer loop iterates \( \frac{1}{2} n \times n^2 \) times. Hence, it is \( O(n^3) \).
What is the time complexity for this code fragment?

```python
sum = 0
for i in range(n//2):
    sum += i
i = 1
for j in range(n**2):
    sum += i * j
```

The loops iterate $\frac{1}{2} n + n^2$ times. Hence, it is $O(n^2)$.
What is the time complexity for this code fragment?

```python
result = []
if len(lst)>0:
    result.append(lst[0])
    for i in range(len(lst)-1):
        if lst[i] != lst[i+1]:
            result.append(lst[i+1])
return result
```

It is $O(n)$
What is the time complexity for this code fragment?

```python
res = []
for i in lst:
    if i not in res:
        res.append(i)
```
What is the time complexity for this code fragment?

```python
sum = 0
if n % 2 == 0:
    for i in range(n*n):
        sum += 1
else:
    for i in range(5, n+3):
        sum += i
```

The loops iterate either $n^2$ or $n+3-5$ times. Hence, it is $\mathcal{O}(n^2)$.
What is the time complexity for this code fragment?

```python
i, sum = 0, 0
while i < n * n:
    sum += i
    i += 1
```

The loop iterate $n^2$ times. Hence, it is $O(n^2)$
What is the time complexity for this code fragment?

```python
i, sum = 0, 0
while i**2 < n:
    j = 0
    while j**2 < n:
        sum += i * j
        j += 1
    i += 1
```

The outer loop iterates $n^{\frac{1}{2}} * n^{\frac{1}{2}}$ times:. Hence, it is $O(n)$
What is the time complexity for this code fragment?

```python
p, q, sum = 0, 0, 0
while p**2 < n:
    while q**2 < n:
        sum += p * q
        q += 2
    p += 2
```

It’s $O(n^{1/2})$
What is the time complexity for this code fragment?

```python
def twoness(n):
    count = 0
    while n > 1:
        n = n // 2
        count = count + 1
    return count
```

The loop iterate $\log n$ times. Hence, it is $\mathcal{O}(\log n)$
Official Course Evaluation

Please don’t forget to do it.

Thanks 😊