Time Complexity Analysis
There are always many ways to solve a problem.
  - Different programs (algorithms) produce the same results.
  - But they may have very different run time.

We are usually interested in how an algorithm performs when its **input is large**

Main question to ask:

How does computational time grow with the input size?

Usually the worst case analysis.

Counting Instructions

The first thing is to count how many fundamental instructions that will be executed.

For simplicity, we assume all the following instructions are executed with similar speed:

- Variable assignment ($a = 1$)
- Get a value in an array ($b[5]$)
- Comparing two values ($a \geq b$)
- Basic arithmetic operations: $+, -, \times, /$
The two-sum problem:
- Input: an array of \( n \) numbers \((a_1, \cdots, a_n)\), and a target \( T \)
- Output: return \( i, j \) such that \( a_i + a_j = T \)

A simple algorithm for two-sum:

```python
def two_sum(a, T):
    n = len(a)
    for i in range(n):
        for j in range(i+1, n):
            if a[i] + a[j] == T:
                return (i, j)
    return -1
```
A simple example: Two sum

- How many instructions do we need?
  
  ```python
  def two_sum(a, n, T):
      for i in range(n):  # n addition
          for j in range(i+1, n):  # n-i-1 addition
              if a[i]+a[j] == T:  # 1 addition, 2 loopups, 1 comparison
                  return (i, j)
      return -1
  ```

- In the worst case, line 4 executes $n(n - 1)/2$ times
- Line 3 executes $n - 1$ times
- Line 2 executes 1 time
- In the worse case, we have totally

$$2n(n-1) + \sum_{i=1}^{n-1} (n-i-1) + n - 1 = 2n^2 - 2n + n/2 + n - 1 = 2n^2 - 0.5n - 1$$
Asymptotic Behavior

- We only care about the speed when input \((n)\) is very large
  ⇒ Drop the lower-order terms
  ⇒ Run time \(\approx 2n^2\)

\[ n^2 \gg 1999n \text{ when } n \text{ is large} \]
Asymptotic Behavior

- Drop the constant (why?)
  - Constants are related to what language you use
  - It can make the calculation much easier

- Run time = $O(n^2)$
Big-O

- We say a function $f(n) = O(g(n))$ if:
  
  $f(n)$ grows at most in the order of $g(n)$

- Formal definition: $f(n) = O(g(n))$ if $f(n) \leq ag(n)$ when $a$ is large enough

- Examples:
  
  - $2n^2 - 0.5n - 1 = O(n^2), \quad n^3 + 2n^2 + n \log n = O(n^3)$
  - $n = O(n), \quad n = O(n^2)$
  - $n \neq O(\log n)$
A function \( f(n) = \Omega(g(n)) \):

- \( f(n) \) grows at least in the order of \( g(n) \)
- \( f(n) \geq ag(n) \) for all \( a \) if \( n \) is large enough

E.g., \( 2n^2 + n = \Omega(n) \), \( 2n^2 + n = \Omega(n^2) \)

\[ n^2 = \Omega(n^2), \quad n^2 = \Omega(n) \]
Big-O

- A function \( f(n) = \Omega(g(n)) \):
  - \( f(n) \) grows at least in the order of \( g(n) \)
  - \( f(n) \geq ag(n) \) for all \( a \) if \( n \) is large enough
  - E.g., \( 2n^2 + n = \Omega(n) \), \( 2n^2 + n = \Omega(n^2) \)
  - \( n^2 = \Omega(n^2) \), \( n^2 = \Omega(n) \)

- A function \( f(n) = \Theta(g(n)) \):
  - \( f(n) \) grows exactly in the order of \( g(n) \)
  - \( f(n) = \Omega(g(n)) \) and \( f(n) = O(g(n)) \)
Commonly used functions

- $\Theta(2^n)$: exponential growing
  - Very slow ($n$ should be smaller than 30)
  - Often for search algorithms, or when you list all the possible combinations

- $\Theta(n^3), \Theta(n^2), \Theta(n)$: commonly used in statistical computing, will see in the following lectures.

- $\Theta(\log n)$: sub-linear time, e.g., binary search.
Example: Two-sum with sorted list

- **Input/output:**
  - **Input:** an array of *n* sorted numbers \((a_1 \leq a_2 \leq \cdots \leq a_n)\), and a target \(T\)
  - **Output:** return \(i, j\) such that \(a_i + a_j = T\)

- Can we have faster algorithm if the input is sorted?
Example: Two-sum with sorted list

- **Input/output:**
  - **Input:** an array of \( n \) sorted numbers \((a_1 \leq a_2 \leq \cdots \leq a_n)\), and a target \( T \)
  - **Output:** return \( i, j \) such that \( a_i + a_j = T \)

- Can we have faster algorithm if the input is sorted?
- **Idea I:** solve two-sum in \( O(n \log n) \) time (using binary search)
- **Idea II:** a better linear-time algorithm to solve it in \( O(n) \) time
How to find an element in a sorted array?

- Naive approach: Linear search: check every elements in the array

```python
def bsearch(input_array, target):
    for i in range(len(input_array)):
        if input_array[i] == target:
            return i
    return -1
```

Takes $O(n)$ time. Doesn't use the information that input array is sorted.
How to find an element in a sorted array?

Naive approach: Linear search: check every elements in the array

```python
def bsearch(input_array, target):
    for i in range(len(input_array)):
        if input_array[i] == target:
            return i
    return -1
```

- Takes $O(n)$ time
- Doesn’t use the information that input array is sorted
Example: Two-sum with sorted list (Idea I)

- A better approach: Binary search
  - Main idea: if array\[i\] < target, then the target will be in \([i + 1, n - 1]\); if array\[i\] > target, the target will be in \([0, i - 1]\).
  - Check

![Binary search diagram](https://www.cs.usfca.edu/~galles/visualization/index.html)

- Target < 41
- Target > 41
Binary Search (Idea I)

- Maintain 2 pointers (begin, end): such that target is in [begin, end].
- Keep decreasing the interval by checking \( a[(\text{begin} + \text{end})/2] \)
- Need \( O(\log n) \) operations (since each step decreases (begin-end) by half)

```python
def bsearch(input_array, target):
    begin = 0
    end = len(input_array)-1
    while (begin <= end):
        mid = (begin+end)/2
        if input_array[mid] < target:
            begin = mid+1
        elif input_array[mid] > target:
            end = mid-1
        else:
            return mid
    return -1
```
For each element \( a_i \), try to find \( a_j = T - a_i \) using binary search

```python
def sorted_two_sum(a, T):
    for i in range(len(a)):
        j = bsearch(a, T-a[i])
        if j!=-1:
            return (i,j)
```

Overall time complexity: \( O(n \log n) \) time
For each element $a_i$, try to find $a_j = T - a_i$ using binary search

```python
def sorted_two_sum(a, T):
    for i in range(len(a)):
        j = bsearch(a, T-a[i])
        if j != -1:
            return (i, j)
```

Overall time complexity: $O(n \log n)$ time
Can we do better?
Can we do better?

Main observation: If $a[i] + a[j] > T$, then $a[i + 1] + a[k] > T$ for all $k \geq j$

For each $i = 1, 2, \cdots, n$, check whether there exists an $a[j] = T - a[i]$.

```python
def sorted_two_sum(a, T):
    end = len(a)-1
    for i in range(len(a)):
        while ((a[i] + a[end]) > T) && (end>i):
            end -= 1
        if end == i:
            break
```
Sorted Two-Sum (Idea II)

Target: 73

0 can be eliminated (impossible to form two-sum=target)

76, 88 can be eliminated (impossible to form two-sum=target)

0+88 > 73
0+76 > 73
0+68 < 73
Sorted Two-Sum (Idea II)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>9</th>
<th>26</th>
<th>31</th>
<th>41</th>
<th>42</th>
<th>53</th>
<th>68</th>
<th>76</th>
<th>88</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Target: 73

- 9 + 68 > 73
- 9 + 53 < 73
Sorted Two-Sum (Idea II)

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>26</th>
<th>31</th>
<th>41</th>
<th>42</th>
<th>53</th>
<th>68</th>
<th>76</th>
<th>88</th>
</tr>
</thead>
</table>

Target: 73

- **26+53 > 73**
- **26+42 < 73**
- **31+42 = 73**
Sorted Two-Sum (Idea II)

- At each step (one addition and one comparison), we will reduce \((\text{end} - i)\) by 1
- The algorithm runs in \(O(n)\) time!
Using sorted two-sum algorithms to solve two-sum.
First, sort the input array
  - We will discuss how to do this later
  - Requires $O(n \log n)$ time
Run either idea I or idea II for sorted two-sum problem
  - $O(n \log n)$ time or $O(n)$ time
Overall complexity: $O(n \log n)$ time
$\ll O(n^2)$ time using a naive approach
Sorting
Naive approaches:
- Insertion sort: $O(n^2)$
- Bubble sort: $O(n^2)$

Better approaches:
- Merge sort: $O(n \log n)$
- Sorting by Priority queue: $O(n \log n)$
- Quick sort: $O(n \log n)$ (in expectation)
Insertion Sort

- Main idea: Always maintain a sorted sublist
  Keep increasing the size of sorted sublist until it covers all the elements
- $O(n^2)$ time
Main idea:

Step 1: Divide the array into half  
Step 2: Sort each part  
Step 3: Merge two sorted arrays

```python
def merge_sort(a):
    n = len(a)
    if n > 1:
        mid = n/2+1
        merge_sort(a[:mid])
        merge_sort(a[mid:])
        merge(a, mid)  ## merge two sorted lists
        ## a[:mid] and a[mid:]
Merge two size $n$ sorted lists can be done in $O(n)$ time.

```python
def merge(a, mid):
    b = []
i = 0
j = mid
n = len(a)
while i<mid and j<n:
    if a[i] >= a[j]:
        b.append(a[j])
        j+=1
    else:
        b.append(a[i])
        i+=1
while i<mid:
    b.append(a[i])
    i+=1
while j<n:
    b.append(a[j])
    j+=1
return b
```
Merge Two Sorted Arrays

List 0

\[
\begin{array}{cccc}
0 & 4 & 5 & 9 \\
\end{array}
\]

\(i\)

List 1

\[
\begin{array}{cccc}
1 & 2 & 6 & 7 \\
\end{array}
\]

\(j\)

Combined List

\[
\begin{array}{c}
0 \\
\end{array}
\]

\(0 < 1\)
Merge Two Sorted Arrays

List 0

0 4 5 9

i

List 1

1 2 6 7

j

Combined List

0 1

4 > 1
Merge Two Sorted Arrays

List 0

0 4 5 9

i

4 > 2

List 1

1 2 6 7

j

Combined List

0 1 2
Merge Two Sorted Arrays

List 0

\[
\begin{array}{cccc}
0 & 4 & 5 & 9 \\
\end{array}
\]

i

List 1

\[
\begin{array}{cccc}
1 & 2 & 6 & 7 \\
\end{array}
\]

j

Combined List

\[
\begin{array}{cccc}
0 & 1 & 2 & 4 \\
\end{array}
\]
Merge Two Sorted Arrays

List 0

```
0 4 5 9
i
```

List 1

```
1 2 6 7
j
```

Combined List

```
0 1 2 4 5
```

5 < 6
| 99 |  6 | 86 | 15 | 58 | 35 | 86 |  4 |  0 |
Merge Sort

99  6  86  15  58  35  86  4  0

99  6  86  15  58  35  86  4  0
Merge Sort

99  6  86  15  58  35  86  4  0

99  6  86  15  58  35  86  4  0

99  6  86  15  58  35  86  4  0

99  6  86  15  58  35  86  4  0
Merge Sort

99 6 86 15 58 35 86 4 0

99 6 86 15 58 35 86 4 0

99 6 86 15 58 35 86 4 0

99 6 86 15 58 35 86 4 0

4 0
Merge Sort

99  6  86  15  58  35  86  0  4

Merge
Merge Sort
Merge Sort
Merge Sort

0 4 6 15 35 58 86 86 99

6 15 86 99
0 4 35 58 86
Merge Sort (Time Complexity)

- Each layer requires $O(n)$ time, totally $O(\log n)$ layers
  $\Rightarrow$ Overall time complexity: $O(n \log n)$

Total time for merging: $n \log n$
Quick Sort

- Similar divide-and-conquer idea:
  - Randomly choose a value \( v \) (from the list)
  - Divide into two parts: \( L = \{ i : a[i] \leq v \} \) and \( R = \{ i : a[i] > v \} \)
  - Sort \( L \) and \( R \) independently
  - Concatenate \( L \) and \( R \) to get the result

- In the worst case, \( v \) can be very bad (smallest or largest element), then the algorithm takes \( O(n^2) \)

- However, it can be proved that if \( v \) is chosen uniformly random from the list, then quick sort has expected time complexity \( O(n \log n) \)

- In practice, it is faster than merge sort.
Sort in python

- Sort a list:

```python
>>> a = [1, 4, 2, 5, 3]
>>> sorted(a)
```

- Get the index:

```python
>>> a = np.array([1, 4, 2, 5, 3])
>>> b = np.argsort(a)
>>> b
array([0, 2, 4, 1, 3])
>>> a[b]
array([1, 2, 3, 4, 5])
```
Coming up

- Data Structure

Questions?