[ICDS Spring 2025]

Algorithms: Part II

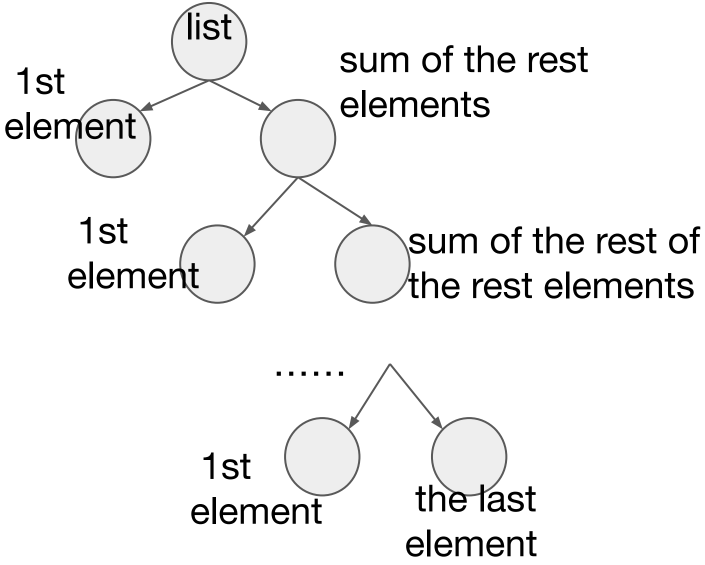
Complexity and Search

Recap: Algorithm Basics

Definition of algorithm in CS:

An algorithm is an **ordered** set of **executable** steps to <u>accomplish a task</u>.

- **Order of execution**
- Steps are **executable**
- Steps are **unambiguous**
- Execution must **end**



Recap: Algorithm Basics

Which is the most preferred way to represent an algorithm?

A. Programming language Python

B. Natural language Chinese

C. Natural language English

D. Pseudo-code

Scan to answer!

Algorithm is an abstraction - like a story;

How to represent an algorithm is like telling a story - in different versions

Recap: Algorithm Basics

Which sorting algorithm does this animation illustrate?

6 5 3 1 8 7 2 4

A. Insertion sort
B. Selection sort
C. Bubble sort
D. Merge sort

Scan to answer!

Rationale: any pair of neighbors must be sorted in a fully sorted list

Agenda

- Computational complexity
    - How to evaluate the efficiency of an algorithm?
    - Big-O notation
    - Complexity of real code
- Search
    - Linear search
    - Binary search
- Appendix: Hash table (read after-class)

Computational Complexity

Motivation

- One task often has multiple solutions.
  - e.g., there are about 20 algorithms for sorting

- Which algorithm should we use?

We compare algorithms on their **runtime** and **space usage**

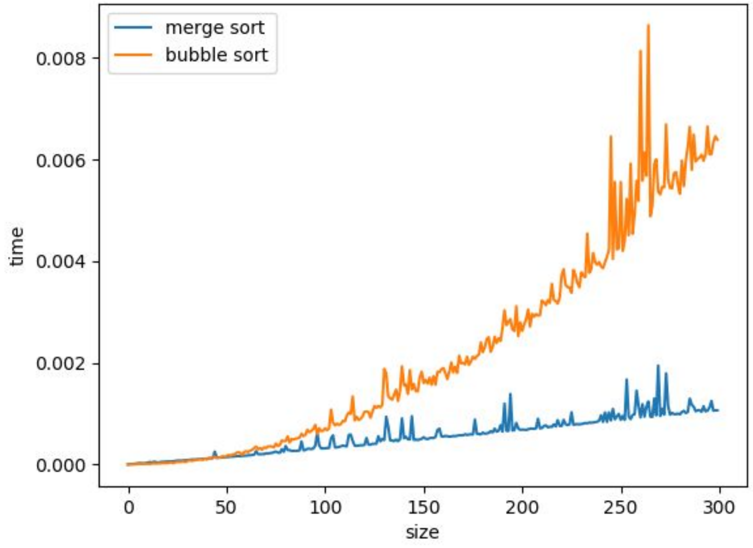
Coding Exercise: Timing the running time

- Try it out on the sorting code, compare bubble sort and merge sort

```python
# Function runtime timer
def check_func_time(func, *args):
    start_time = time.time()
    func(*args)
    end_time = time.time()
    return end_time - start_time
```

In Python, a function can be an argument to another function.

- The `check_func_time()` returns the runtime that `func` takes.
- `*args`: all non-keywords arguments
- `**kargs`: all keywords arguments



How to evaluate an algorithm fairly?

- Runtime:
  - less runtime is better
  - always gets high attention
- Memory usage (space):
  - less memory usage is better.
  - **not** that important since large RAM gets cheaper nowadays

- Comparing the runtime is complex: it is influenced by many factors
  - CPU clock
  - Bandwidth of the bus
  - Programming languages (Python is slower than C)
  - ……

To make it a fair play

- Instead of comparing the runtime → we compare the **total number of steps** that are executed by the algorithm
- a step ⇒ an elementary operation (what an elementary operation is depends on the resolution you choose in practice)

- We assume that each elementary operation takes fixed amount of time to perform by computers.
- **Time complexity** of an algorithm is represented by the total amount of elementary operations that required for executing the algorithm.

The Resolution of Elementary Operations

The most elementary operation in a computer: turning on/off a logic gate
⇒ but it is not necessary to count steps using the on/off of a logic gate; since the circuits for many higher level operations (e.g., adding, subtracting, etc.) in different machines are similar.

To simplify the step counting, **in our course**, we treat the following operations in Python as primitive operations,

- Assign a value to a variable, e.g., a = 1.6
- Compare two variables, e.g., a < b, a == b
- Logic operators, e.g., a OR b, a AND b
- Arithmetic operations, e.g., a + b, a x b
- Indexing an elements, e.g., a[0], b[“key1”] (in fact, it depends on the data structure that stores the data)

Number of operations of the bubble sort

```python
def swap(a, b):
    x = a
    a = b
    b = x
    return a, b

def bubble_sort_basic(my_list):
    N = len(my_list)
    for i in range(0, N):
        for j in range(0, N-1):
            if my_list[j] > my_list[j+1]:
                swap(my_list[j], my_list[j+1])
    return my_list #Note: not necessary; swapping is in-place
```

Loading values takes 2 operations, Compare two values takes 1 operation
Swap takes 3 operations.

Inner loop: repeating the swap and comparison for n-1 times (the worst case)

Outer loop: repeating the inner loop for n times

$$\text{The total operations} = 6 \times (n-1) \times n = 6n^2 - 6n$$

Given an n, we will know how many steps are needed.

The worst case assumption

The steps needed to completed a task also rely on the condition of the dataset:

- Sorting [1, 2, 3, 4] and [4, 2, 3, 1] will take different numbers of operations though they both have 4 elements.

Here, we assume the input is the worst case when evaluating an algorithm.

- e.g., when counting the operations to sort a list ascendingly, without specification, we will assume that the given list is in decreasing order.

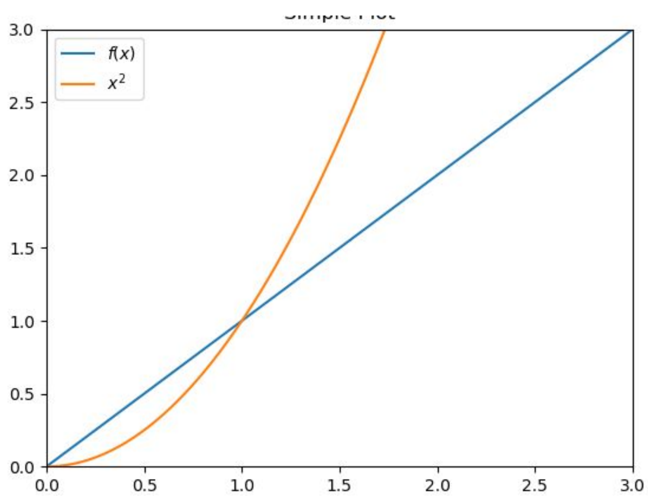
Infinite size assumption (so, we use Big-O)

It relates to the scale of the dataset, so the comparison is between functions, not exact values. For example,

- $f(n) = n$
- $g(n) = n^2$

when $n < 1$, $f(n) > g(n)$.

when $n > 1$, $f(n) < g(n)$.



To simplify the comparison, we often represent the complexity of an algorithm by setting the n to be infinite (i.e., a very large dataset).

Big-O notation (Asymptotic notation)

- When we compare the efficiency of algorithms, we compare the upper bound of their step functions. **The big-O notation is an upper bound of f(x),** which is defined as the following,

Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants C and k such that

$$|f(x)| \leq C|g(x)|,$$

whenever $x \geq k$.

Use of Big-O notation

- The Big-O notation does **not** care about the constant factors, it only describe the long-term growth rate of functions.

e.g., under the Big-O notation, $f_1(x) = x^5 + x^2 + 10$ and $f_2(x) = 100x^5$ are equivalent, becaues

In the view of Big-O, $f_1(x)$ and $f_2(x)$ are equal.

$$\lim_{x \to +\infty} \frac{f_1(x)}{x^5} = \frac{x^5 + x^2 + 10}{x^5} = 1$$

$$\lim_{x \to +\infty} \frac{f_2(x)}{x^5} = \frac{100x^5}{x^5} = 100$$

and both of $f_1(x)$ and $f_2(x)$ are $O(x^5)$.

Note: It is true that the growth of $f_1(x)$ is slower than $f_2(x)$, but as long as n increases, such difference turns to be insignificant.

Important complexity classes and their code structures

Since algorithms are combinations of selection and repetition, their complexity functions often fall into one of the following function families, (sometimes, a combination of them)

- $O(1)$: functions of constant complexity
- $O(\log n)$: functions of logarithmic complexity
- $O(n)$: functions of linear complexity
- $O(n \log n)$: functions of log-linear complexity
- $O(n^k)$: functions of polynomial complexity
- $O(C^n)$: functions of exponential complexity

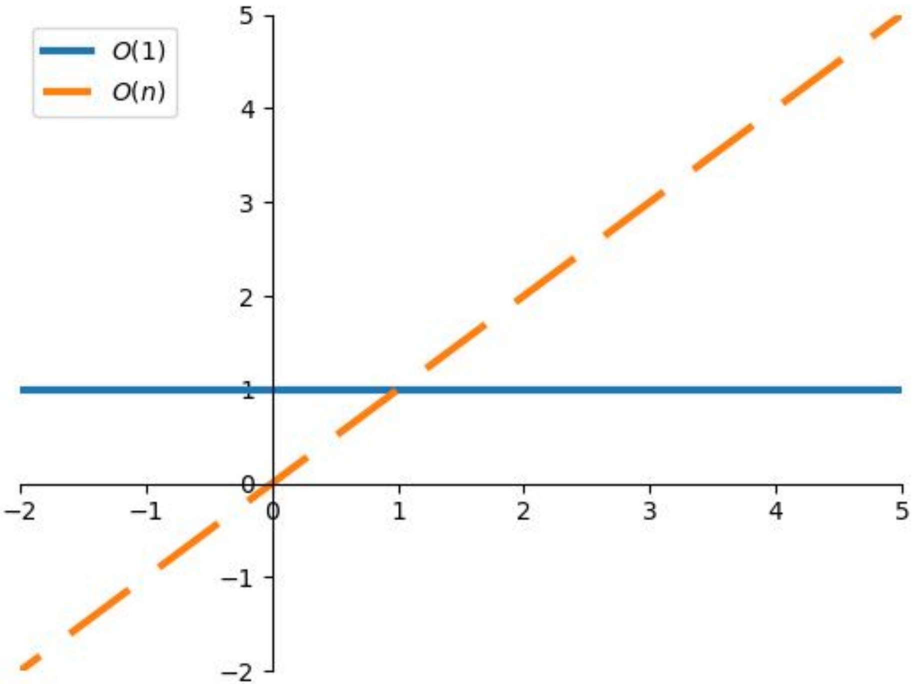
Constant complexity and its curve

O(1):

the complexity does not change with the increase of n

```python
def func(a_list):
    a = 1
    b = 2
    return a+b
```

Whatever the length of "a_list", the number of operations in func() are fixed.

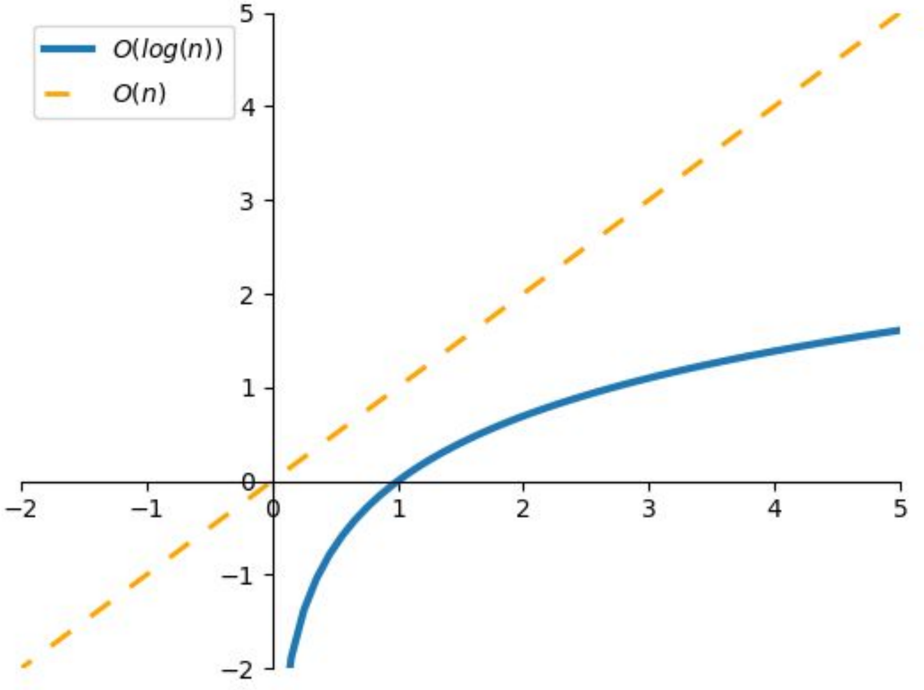


Logarithmic complexity and its curve

O(log(n)): the base of log does not matter.

```
def func(n):
    while n > 1:
        n /= 10
    return
```

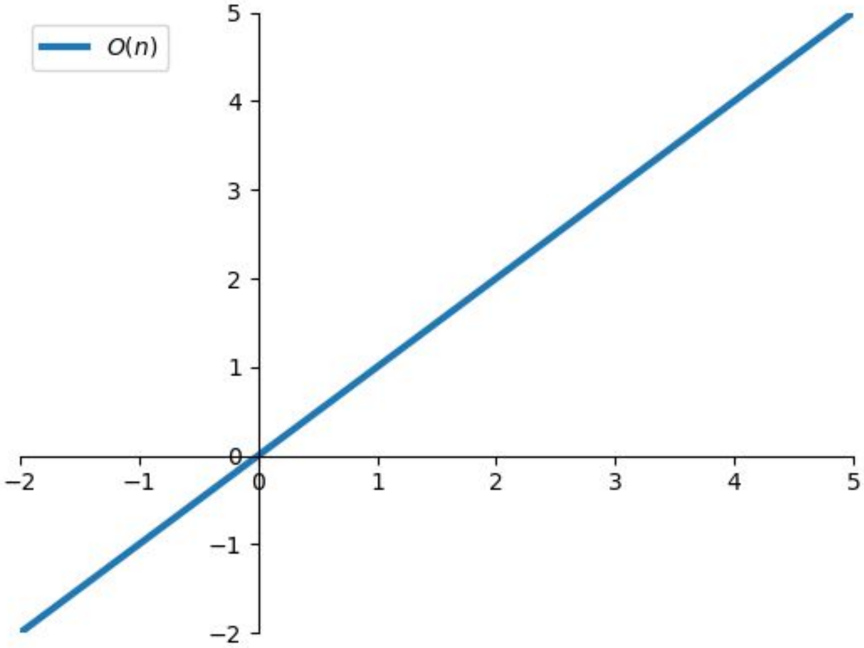
If n = 1000, how many iteration will the above while loop has？



Linear complexity

O(n)

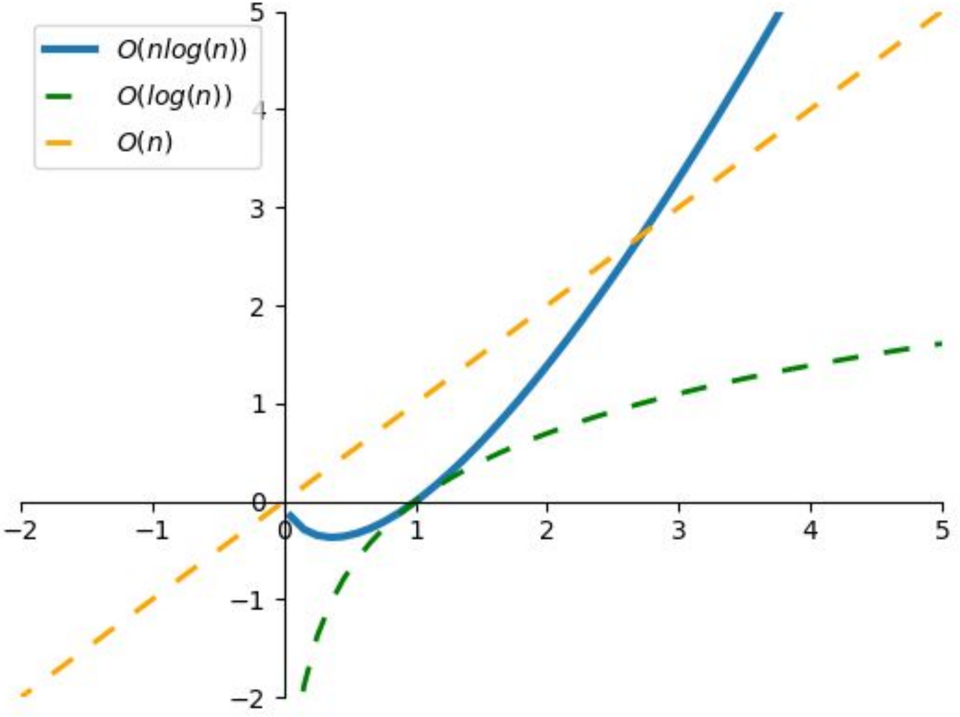
```
def func(n):
    j = 0
    for i in range(n):
        j += 1
    return
```



Log-linear complexity

```python
def fun(n):
    while n > 1:
        n = n//2
        for i in range(n):
            a = 1
    return
```

Another example is the merge sort.



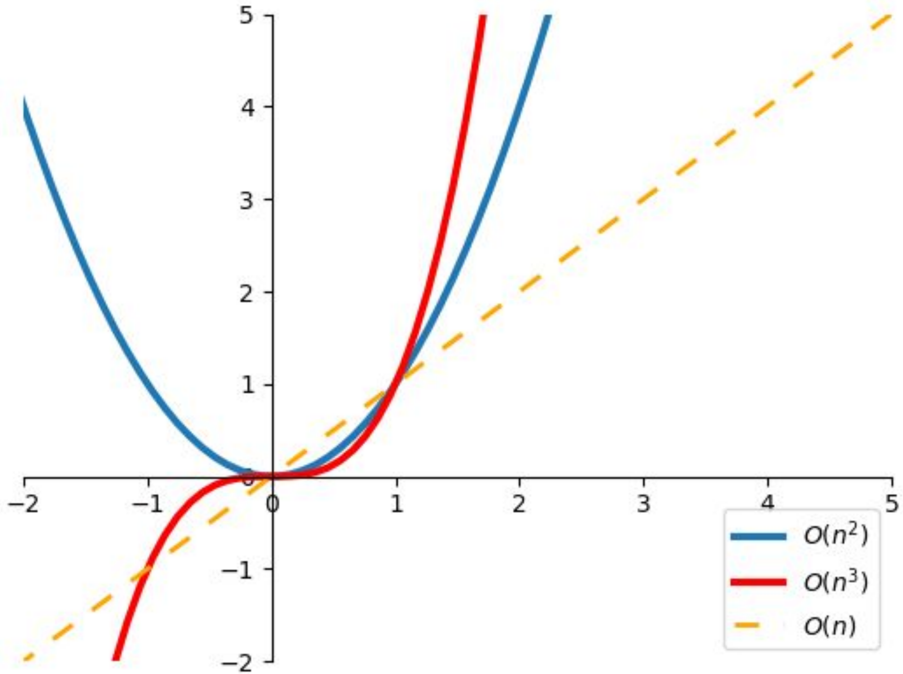
Polynomial complexity

$O(n^2)$

```
def func(n):
    for i in range(1, n):
        for j in range(1, n):
            print(i*j)
    return
```

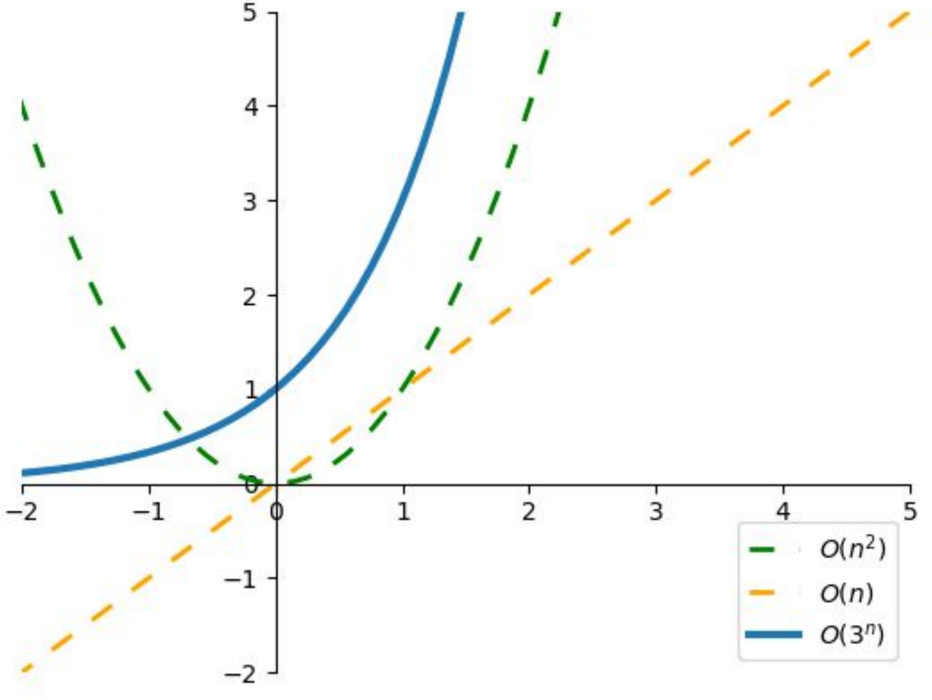
$O(n^3)$

```
def func(n):
    for i in range(1, n):
        for j in range(1, n):
            for k in range(1, n):
                print(i+j+k)
    return
```



Exponential complexity

Rarely a company will pay for a program that requires exponential time to run.



How does it look like:

- Please plot the curves of x**2 and 2**x in [0, 10].
- [Exponential growth and epidemics]()

Comparison of complexity classes

$O(C^n)$

$O(n^k)$

$O(n \log n)$

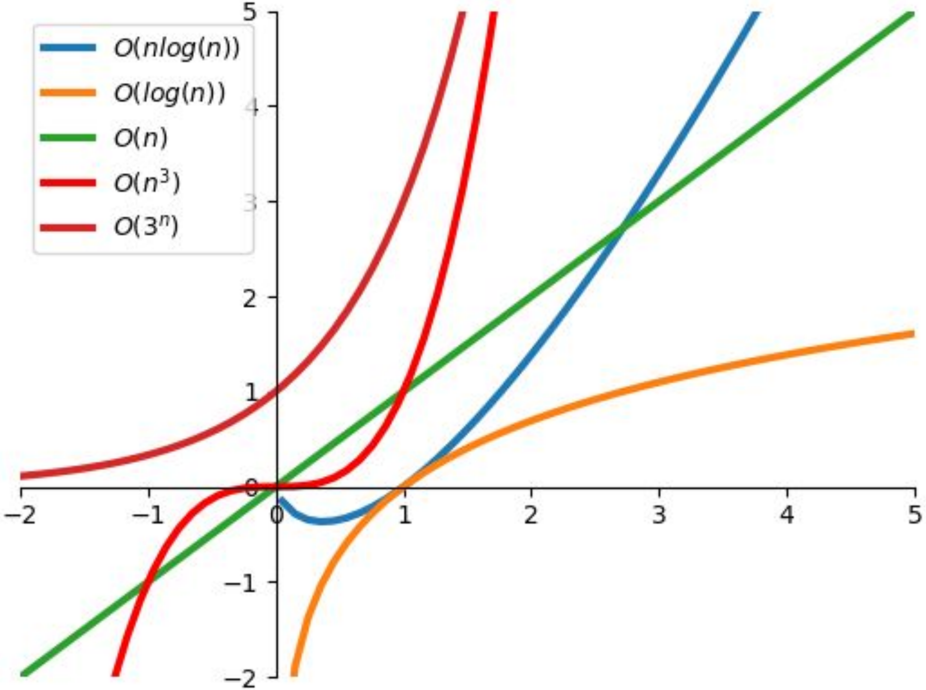
$O(n)$

$O(\log n)$

$O(1)$

High complexity

Low complexity



Polynomial and Non-polynomial problems

Polynomial Problem (**P**): the problem whose complexity of solution is **bounded** by a polynomial. So, we say a problem is **P,** means the problem can be solved within reasonable time.

If a problem's complexity is $O(2^n)$, is it a P?

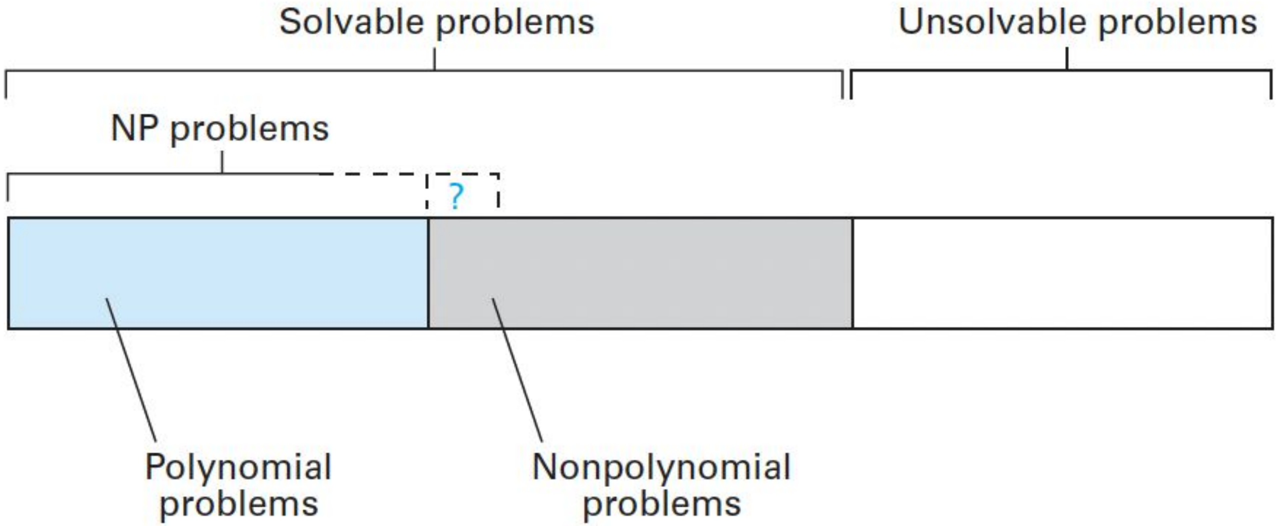
Nondeterministic Polynomial Problems

Nondeterministic Polynomial Problem (**NP**): A problem can be solved in polynomial time **by** a **nondeterministic algorithm** is called a NP problem.

<u>Nondeterministic algorithm</u>: the algorithm that contains some uncertainty. It can solve the problem in practice, but the answer varies each time.

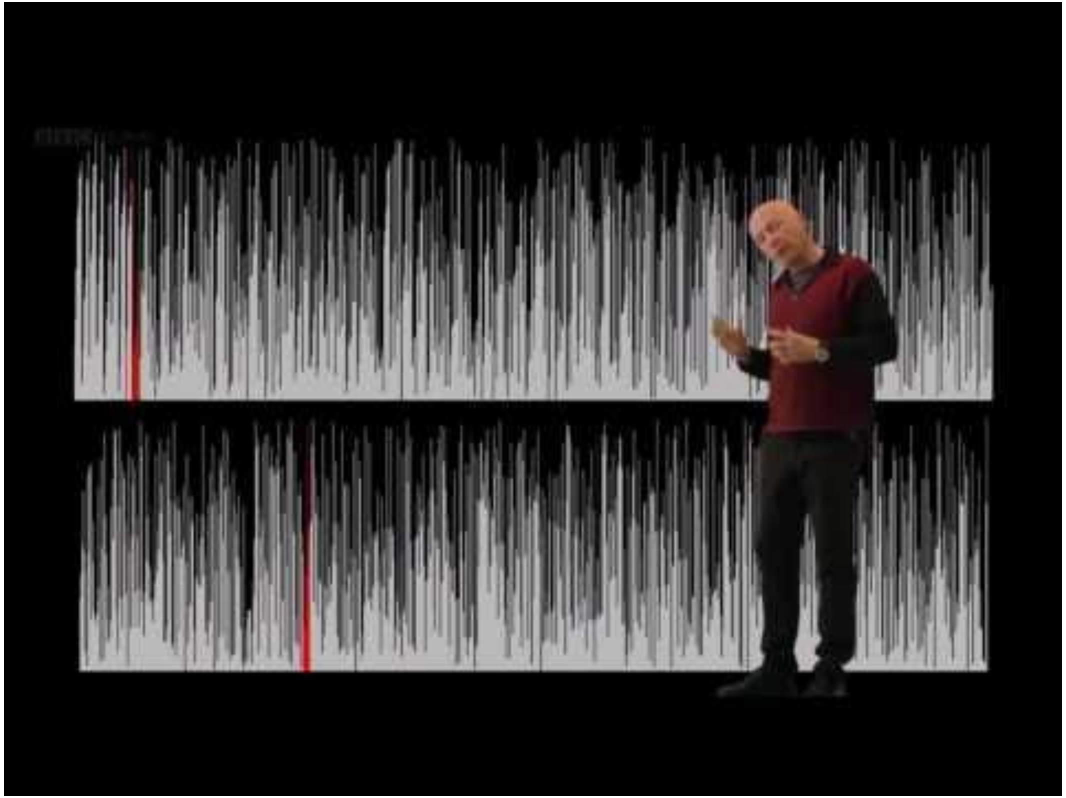
- You can cook a steak by a “nondeterministic algorithm”: “frying the steak for two minutes, then, put a few salt on it.”. But each time, the steak you produced is different due to the ambiguous “a few salt”. ⇒ by the way, if we want to convert this recipe into a runnable program, we can use a random number generator to simulate the “a few”.

Taxonomy of the Real-World Problems



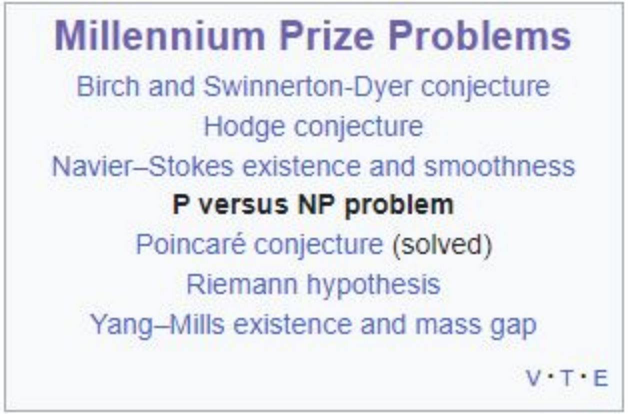
- P is in NP
- Travelling Salesman Problem - TSP (NP-hard)

Supplementary: Travelling Salesman Problem



Supplementary: Millennium Prize Problems

- **P vs. NP problem**: major unsolved problem in theoretical computer science
- Informally, it asks whether every problem whose solution can be quickly verified can also be quickly solved.
- Solve it to win $1 million prize!



Complexity of real code

This is O(n)

```
def fact(n):
    """Assumes n is a natural number
       Returns n!"""
    answer = 1
    while n > 1:
       answer *=  n
       n -= 1
    return answer
```

Complexity of real code

It takes constant operations.

It takes x operations.

It takes x^2 operations.

So, this code is $O(n^2)$.

```
def f(x):
    """Assume x is an int > 0"""
    ans = 0
    #Loop that takes constant time
    for i in range(1000):
        ans += 1
    print 'Number of additions so far', ans
    #Loop that takes time x
    for i in range(x):
        ans += 1
    print 'Number of additions so far', ans
    #Nested loops take time x**2
    for i in range(x):
        for j in range(x):
            ans += 1
            ans += 1
    print 'Number of additions so far', ans
    return ans
```

Bubble sort

```python
# Optimized bubble sort function
def bubble_sort(my_list):
    N = len(my_list)
    for i in range(1,N):
        swapped = False
        for j in range(0,N-i):
            if my_list[j] > my_list[j+1]:
                my_list[j], my_list[j+1] = my_list[j+1], my_list[j]
                swapped = True
        if swapped == False:
            break
    return my_list
```

Time complexity: $O(n^2)$
Space complexity: O(1) (when swap the two elements)

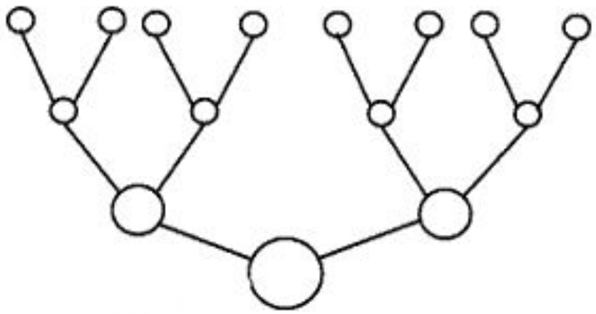
Merge sort

```python
def merge_sort(m):

    if len(m) <= 1:
        return m

    middle = len(m) // 2
    left = m[:middle]
    right = m[middle:]

    left = merge_sort(left)
    right = merge_sort(right)
    return merge(left, right)
```



$O(\log(n))$ “layers”

Recall “recursion is a kind of loop”

Merge sort

$O(\log(n))$ passes

```python
def merge_sort(m):

    if len(m) <= 1:
        return m

    middle = len(m) // 2
    left = m[:middle]
    right = m[middle:]

    left = merge_sort(left)
    right = merge_sort(right)
    return merge(left, right)
```

Time: O(n log(n))
Space: O(n) (why?)

$O(n)$ each pass

```python
# Merge function definition
def merge(left, right):
    result = []
    left_idx, right_idx = 0, 0

    while left_idx < len(left) and right_idx < len(right):
        # change the direction of this comparison
        # to change the direction of the sort
        if left[left_idx] <= right[right_idx]:
            result.append(left[left_idx])
            left_idx += 1
        else:
            result.append(right[right_idx])
            right_idx += 1

    if left:
        result.extend(left[left_idx:])
    if right:
        result.extend(right[right_idx:])

    return result
```

Search

Game time!

- **Rules:**
  - All of you as a team vs. me
  - I have a number in my mind in range [1, 20], try fewest #guess to bingo!
    - 1st guess:
      - 2nd guess:
        - 3rd guess:

Scan to record!

Linear search: the brute-force way

- What's the complexity of exhaustive search?

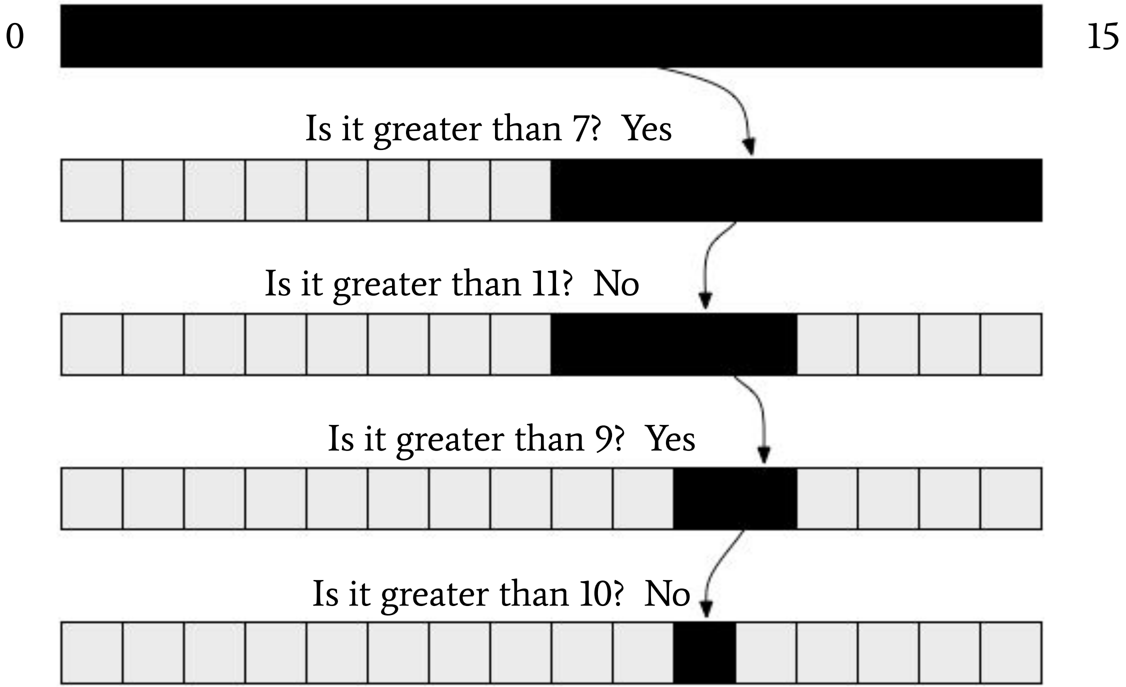
```
def search(L, e):
    for i in range(len(L)):
        if L[i] == e:
            return True
    return False
```

The complexity is linear, i.e., O(n). (assuming the operations inside the loop can be done in constant time.)

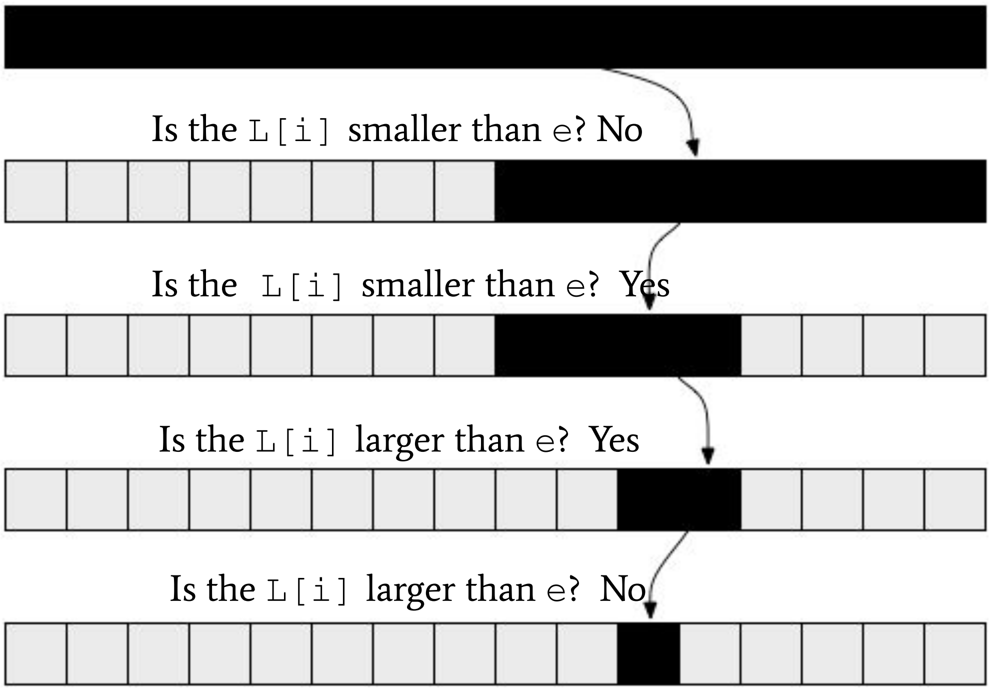
This looks good … but when there are millions of items, you have to wait for minutes or even hours.

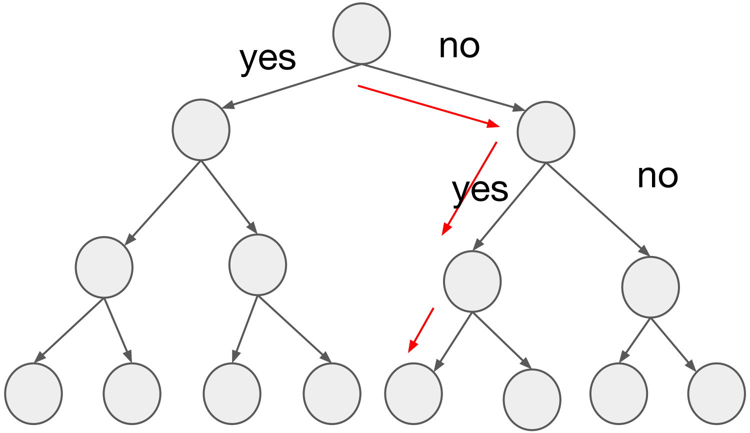
If the list is sorted, can you do better?

Guess a number: search a sorted list



Binary search





The search can be represented by a tree diagram. Each node is a list of numbers. For each node, we do the following,

- a parent node splits to two children nodes by an index `i`
- the value at `i` > `e` ? → go left if yes; go right if no; stop if equal

Binary search: when the list is sorted

```
def bSearch(L, e):
    """Assume L is a list, the e
    Returns True if e is in L and False otherwise

    if len(L) == 1:
        if L[0] == e:
            return True
        return False

    mid = len(L)//2

    if L[mid] == e:
        return True
    elif L[mid] > e:
        return bSearch(L[:mid], e)
    else:
        return bSearch(L[mid:], e)
```

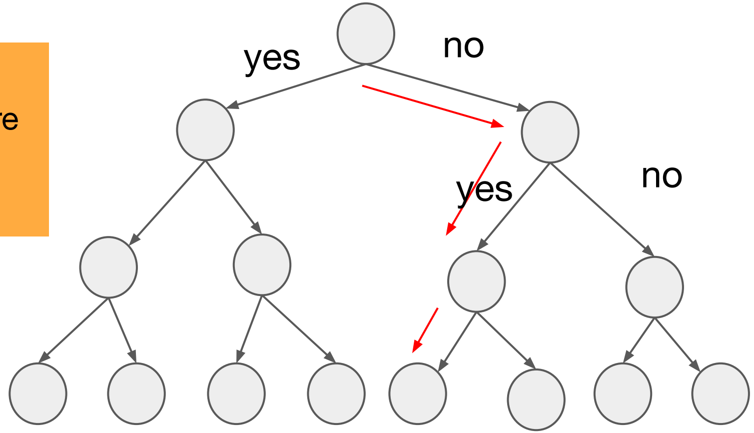
When there is only one element left, compare it with e and return true if they are equal; otherwise, false. (this is called the stop condition)

We set the i to be the middle of the list.

stop (return) if L[i] = e

Go left if L[i] > e

Go right if L[i] > e



The search can be represented by a tree diagram. Each node is a list of numbers. For each node, we do the following,

- a parent node splits to two children nodes by an index `i`
- the value at `i` > `e` ? → go left if yes; go right if no; stop if equal

What is the complexity of the binary search?

Binary search code (another version)

The algorithm stops when the length of the interval is 1.

Each time the interval shrinks by a half.

The time complexity is O(log(n)).

```
def search(L, e):
    """Assumes L is a list, the elements of which are in
          ascending order.
       Returns True if e is in L and False otherwise"""

    def bSearch(L, e, low, high):
        #Decrements high - low
        if high == low:
            return L[low] == e
        mid = (low + high)//2
        if L[mid] == e:
            return True
        elif L[mid] > e:
            if low == mid: #nothing left to search
                return False
            else:
                return bSearch(L, e, low, mid - 1)
        else:
            return bSearch(L, e, mid + 1, high)

    if len(L) == 0:
        return False
    else:
        return bSearch(L, e, 0, len(L) - 1)
```

The framework of search

- Search is the way to solve many real-world problems
    - Finding the answer, not creating the answer
    - Many problems can be described as problems of search
        - Minimize the cost of … → Find the minimized cost of …
        - Sort the list → find the order of the list …
- How do we conduct search?
    - Knowing the accept/reject conditions
    - Splitting the possible solutions into two parts (e.g., one + others)
    - Checking one part of possible solution with the conditions
    - Repeat the above two steps until find the solution satisfied.

This is done by recursion.

Implement recursion in programs

- In programming, recursion is a **trick** by which a function calls itself. In algorithm design, it is a **top-down** approach.
    - We pretend that the function has been defined already and call it when we define it.
- What's in the code of a recursion?
    - a stop condition
    - the call of the function itself
    - statements for solving the task/subtask
- When to use it?
    - In solving almost all problems

Converting the linear search into recursion

```python
def linearSearch(l, x):
    for e in l:
        if e == x:
            return True
    return False



##Test
if __name__ == "__main__":
    l = [3, 4, 5, 8, 9, 10]
    print(linearSearch(l, 5))
    print(linearSearch(l, 2))
```

```python
def linearSearchRecursive(l, x):
    pass

##Test
if __name__ == "__main__":
    l = [3, 4, 5, 8, 9, 10]
    print(linearSearch(l, 5))
    print(linearSearch(l, 2))
    print(linearSearchRecursive(l, 5))
    print(linearSearchRecursive(l, 2))
```

When to use binary search?

- Given an **unsorted** list of length *n*
- You will search *m* times
- What's the complexity using naive search, and binary search?

  - Brutal search: $O(m * n)$
  - Binary search: $O(n\log(n) + m\log(n))$ (here, $O(n\log(n))$ is for sorting the list)

Typically, you do search many, many times, so, m is often very large in practices

Appendix: Hash table

Can we do faster than binary search?

Imagine there is a 100 x 1 table. Each cell has an index starts from 0 to 99. The table is used as a data warehouse to store integers in [1, 100]. Given any integer i, where 1 ≤ i ≤ 100, it always stores i into the cell whose index is i-1.

0	empty
1	2
2	empty
...	...
99	100

Now, you want to find whether 81 is in the table or not, what will you do?

Do we have to start from the first cell and check the content one by one until we find 81? No

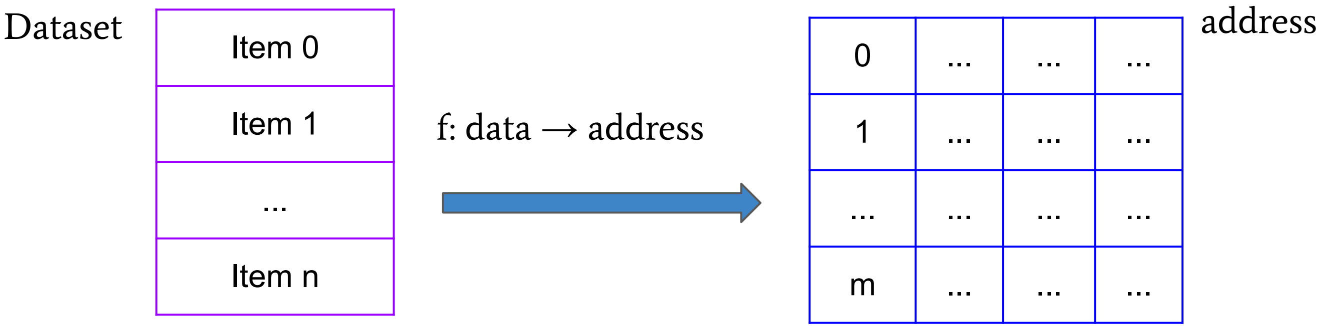
How about using the binary search? No

Because we've know how the table stores its data. We know 81 should be stored at the 80th cell. So, we just need to check the 80th cell, if 81 is there, return True. Otherwise, return False.

Hashing

Computer scientists proposed a technique namely hashing

- Instead of traversing the list and checking whether the item is in the list, we can define a structure that uses a function to map the item to where it stores.
- We “calculate” the position of an item rather than “search” it.



Hashing

- It can find an item in O(1)
  - Because calculating an index by the hash function usually takes fixed number of steps.
- It is handy.
  - Almost any data attribute can be a key since everything in computer memory is a binary string.
    - For example, our names are binary numbers when stored in a computer.

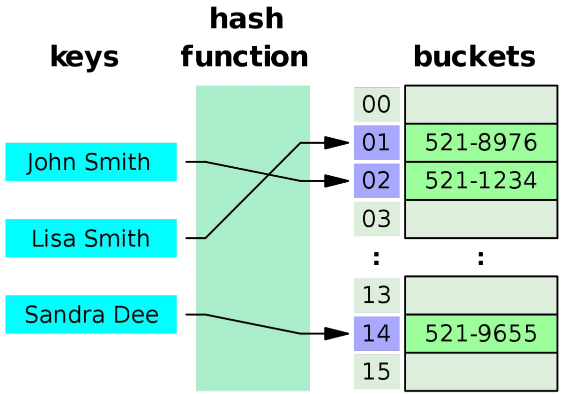
```
In [6]: ##covert a character to the ASCII number in Python
   ...: ord("a")
Out[6]: 97
```

Several concepts

Hash function: maps the inputs to integer indices (also called hash code).

Hash table: a data structure that contains **keys**, **buckets**, and a **hash function** which maps every key to a bucket where the desired value can be found.

- The hash function determines which bucket to store the item.
- Ideally, each bucket should only contain one data item.



A small phone book as a hash table (from Wikipedia)

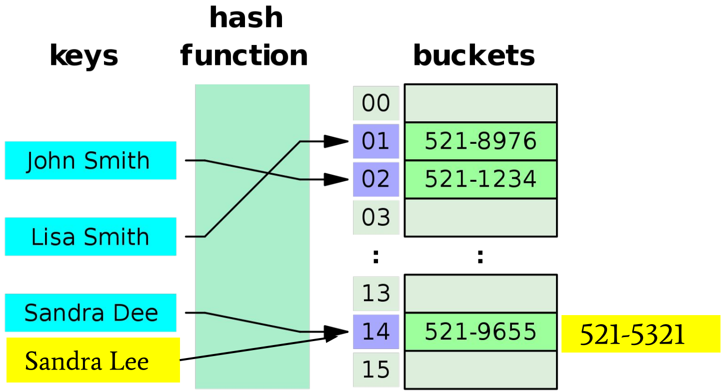
Several concepts:

Collision: the hash function generates the same index for two or more different keys. (Note: a good hash function has a very low probability of collision.)

Some collision resolutions:

- Separate chaining: storing elements with same hash value in a linked list
  - In the worst case, its complexity is O(n).

- Trade space for time: making the hash table large enough.

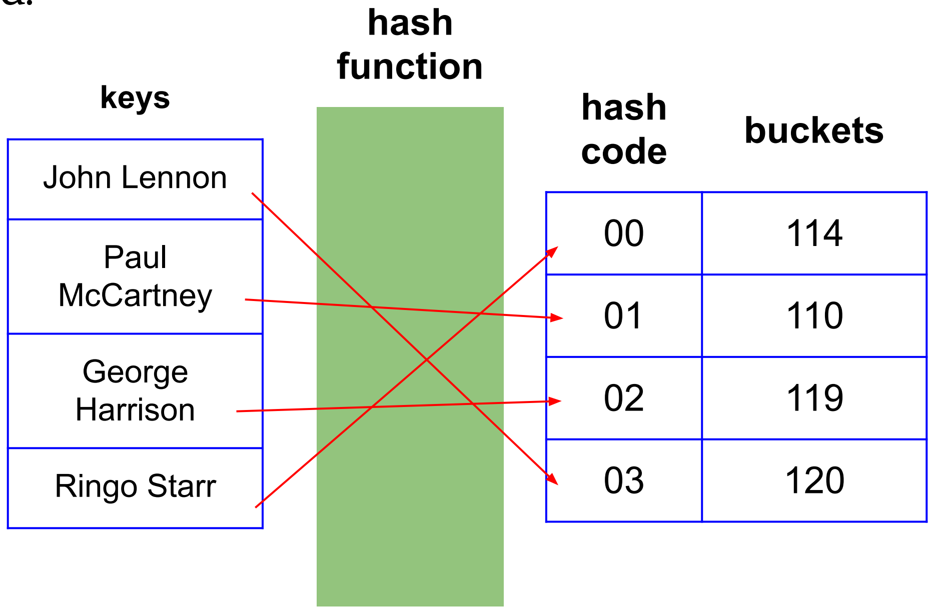


Hash table: a summary

- A hash table uses a hash function to compute an index into an array of buckets where the desired value can be found.

Dataset

Name	Phone number
John Lennon	120
Paul McCartney	110
George Harrison	119
Ringo Starr	114



A hash table

Readings

Chapter 9, 10, 11