Lecture 8: Recursion & Searching

김수경

이화여자대학교 인공지능융합전공 소속

저작권 안내

(주)업스테이지가 제공하는 모든 교육 콘텐츠의 지식재산권은
운영 주체인 (주)업스테이지 또는 해당 저작물의 적법한 관리자에게 귀속되어 있습니다.

콘텐츠 일부 또는 전부를 복사, 복제, 판매, 재판매 공개, 공유 등을 할 수 없습니다.
유출될 경우 지식재산권 침해에 대한 책임을 부담할 수 있습니다.

유출에 해당하여 금지되는 행위의 예시는 다음과 같습니다.

- 콘텐츠를 재가공하여 온/오프라인으로 공개하는 행위
- 콘텐츠의 일부 또는 전부를 이용하여 인쇄물을 만드는 행위
- 콘텐츠의 전부 또는 일부를 녹취 또는 녹화하거나 녹취록을 작성하는 행위
- 콘텐츠의 전부 또는 일부를 스크린 캡쳐하거나 카메라로 촬영하는 행위
- 지인을 포함한 제3자에게 콘텐츠의 일부 또는 전부를 공유하는 행위
- 다른 정보와 결합하여 Upstage Education의 콘텐츠임을 알아볼 수 있는 저작물을 작성, 공개하는 행위
- 제공된 데이터의 일부 혹은 전부를 Upstage Education 프로젝트/실습 수행 이외의 목적으로 사용하는 행위

01

Searching

Searching Problem

Given a list of objects and a single target, locate it if it exists.

Example:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	-7	15	2	6	-1	5	4	10	-4	21

Where is 4? → [6].

Where is 17? → It does not exist.

How to solve it?

What is the most efficient way?

Linear Search

- Let's try the simplest algorithm: checking one by one!
    - Check [0]. If it contains the target value, return 0. Otherwise, move to the next.
    - Check [1]. If it contains the target value, return 1. Otherwise, move to the next.
    - …
    - Check [N-1]. If it contains the target value, return N-1. Otherwise, return -1.

```
def linear_search(list, value):
  for i in range(len(list)):
    if list[i] == value:
      return i
  return -1
```

Time complexity? O(N)

Can we do better?

Searching Problem

Now, let's consider an additional condition:

Given a sorted list of objects and a single target, locate it if it exists.

Example:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	-7	-4	-1	2	4	5	6	10	15	21

Where is 4? → [4].

Where is 17? → It does not exist.

Can we use the linear search method to solve this version? **Yes!**

Can we solve it more efficiently? 🤔

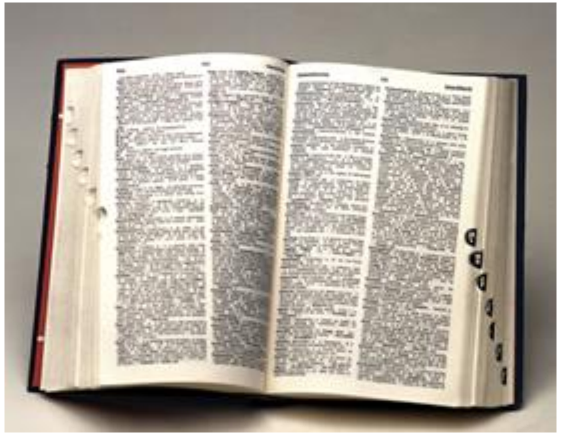
Analogy to Dictionary Search

- Suppose you are searching for a word in English dictionary.
- If you use the “linear search” method, how many words do you expect to see before you find the word?

N/2, where N is the number of words in the dictionary!

- Do you really do this? Any better way?

[1]



[1] https://www.britannica.com/summary/dictionary

Binary Search

- Linear search does work for a sorted list, but does NOT take advantage of the fact that it is sorted.
- Main idea: Check what value is stored in the middle. If it is smaller than the target, we can ignore all values before this. Otherwise, we can ignore all values after this.
- For instance, target = **15**:

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Value	-7	-4	-1	2	4	5	6	10	15	21

15 should appear on the right side of this!
→ We can safely ignore all values on the left.

Binary Search

- In the next step, we repeat the same procedure with the reduced list:

Index	5	6	7	8	9
Value	5	6	10	15	21

Again, 15 should appear on the right side of this!
→ We can safely ignore all values on the left.

- Repeat with the reduced list:

Index	8	9
Value	15	21

Okay, we found it!

Binary Search

- What if we are searching a non-existent value? For instance, 14?

Index	8	9
Value	15	21

14 should be on the left of this.
→ We can safely ignore all values on the right.

- Removing them gives now an empty list. We confirm that there is no 14.

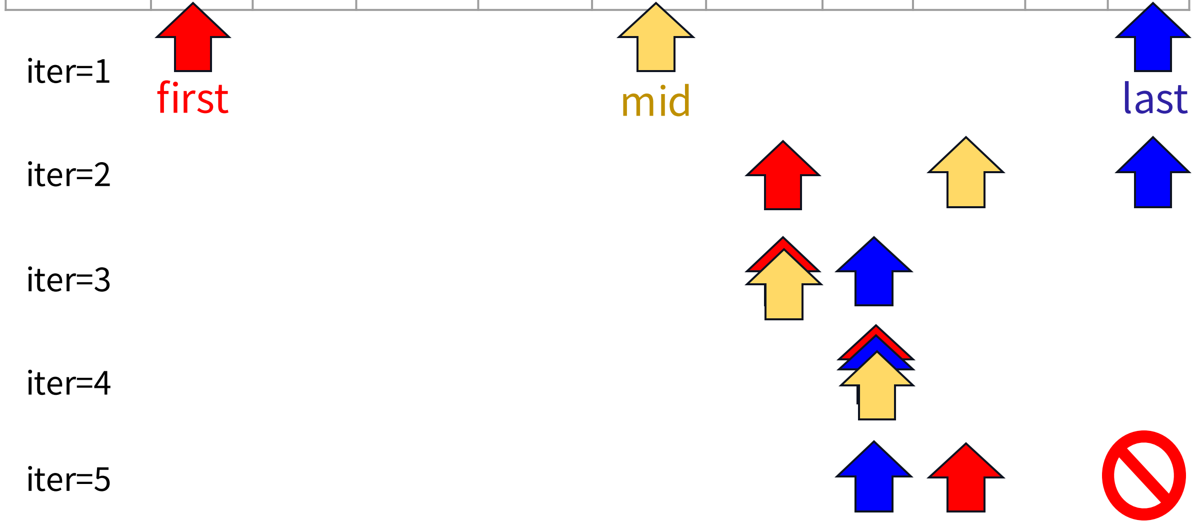
Index
Value

Binary Search

Example: value = 7?

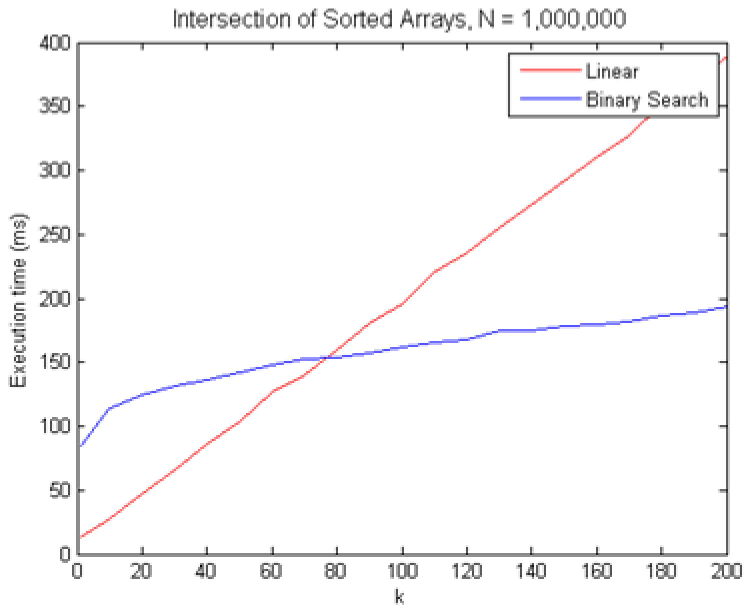
```
def binary_search(list, value):
  first = 0
  last = len(list) - 1

  while first <= last:
    mid = (first + last) // 2
    if list[mid] == value:
      return mid
    elif list[mid] < value:
      first = mid + 1
    else:
      last = mid - 1

  return -1
```

Index	0	1	2	3	4	5	6	7	8	9
Value	-7	-4	-1	2	4	5	6	10	15	21



Time Complexity

- Linear search
  - Applicable to **any** list
  - Time complexity: O(*N*)
- Binary search
  - Applicable to a **sorted** list
  - Time complexity: **O(log *N*)**



Additional Problems

- Does the binary search work when the list contains duplicates?
  - To make it return any of those duplicates?
  - To make it return the leftmost one?
  - To make it return the rightmost one?

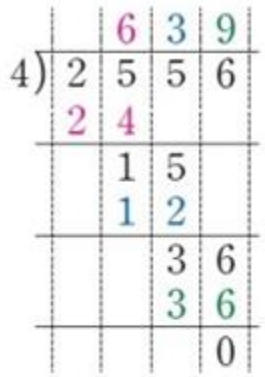
- Find the first element greater than or equal to the query.

02

Recursion

Recursive Algorithm

- An algorithm that **calls itself for subproblems**.
- The subproblems are of **exactly the same type** as the original problem, with **smaller scale**.
- “Complex problems can be seen much simpler.”
- Examples:
  - Division in elementary school
  - Binary search
  - Sorting (selection sort, merge sort, ⋯)
  - Tree traversal



[1]

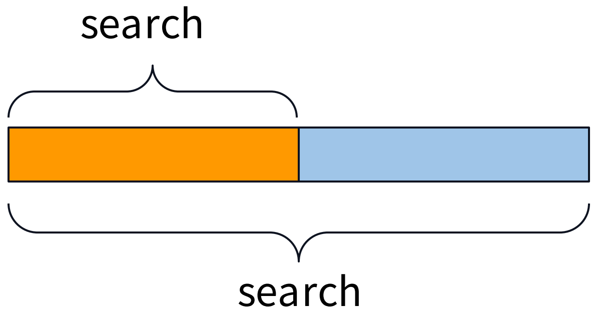
[1]
https://www.google.com/url?sa=i&url=https%3A%2F%2Fschool.jbedu.kr%2F_cmm%2FfileDownload%2Fclass%2Fjhwasan%2F2020%2FG02020602%2F273057%2Fca81112587983642fbb5586a42845cb0&psig=AOvVaw3LT6c5Y1nA850MDX2_clu0&ust=1763800053288000&source=images&cd=vfe&opi=89978449&ved=0CBEQjRxqFwoTCLCR87DpgpEDFQAAAAAdAAAAABAE

Key Components in Recursive Algorithm

- **Base case**: the simplest case where the algorithm can trivially conclude without additional recursive call.
  - *E.g.*, empty list in searching, single element in sorting, ⋯
  - Usually, this case is simply solved by O(1).
- **General case**: the algorithm may call one or more recursive call, to solve exactly the same problem in smaller scale.
  - Recursive call must be **strictly smaller**; otherwise it may not finish infinitely.
  - Important! Make sure if the same thing is not repeatedly called.
- Keep in mind: your algorithm must **finish** with **correct answer** for all **possible inputs**. Then, make it as **efficient** as possible.

Binary Search: Recursive Version

```
def binary_search(list, value, first, last):
  if first > last:
    return -1
  else:
    mid = (first + last) // 2
    if list[mid] == value:
      return mid
    elif list[mid] < value:
      return binary_search(list, value, mid+1, last)
    else:
      return binary_search(list, value, first, mid-1)

binary_search(list, value, 0, len(list) - 1)
```



"Same problem of different size"

Time complexity? **O(log *N*)**

Up to O(log *N*) recursive calls.
Each call takes O(1): computing mid, comparison with value, return output.

Reversing a String

- Print a string in reverse order.
    - *E.g.*, ACTGCC → CCGTCA

```
return reverse(str[1:]) + str[0]
```

Non-recursive version:

```
def reverse(str):
  output = ""
  for i in range(len(str)):
    output += str[-i-1]
  return output
```

Recursive version:

```
def reverse(str):
  if not str:
    return str
  else:
    return str[-1] + reverse(str[:-1])
```

Combination

- Combination: a selection of *r* items from a set that has *n* distinct members, such that the order of selection does not matter.

$$_nC_r = \frac{n!}{r!(n-r)!}$$

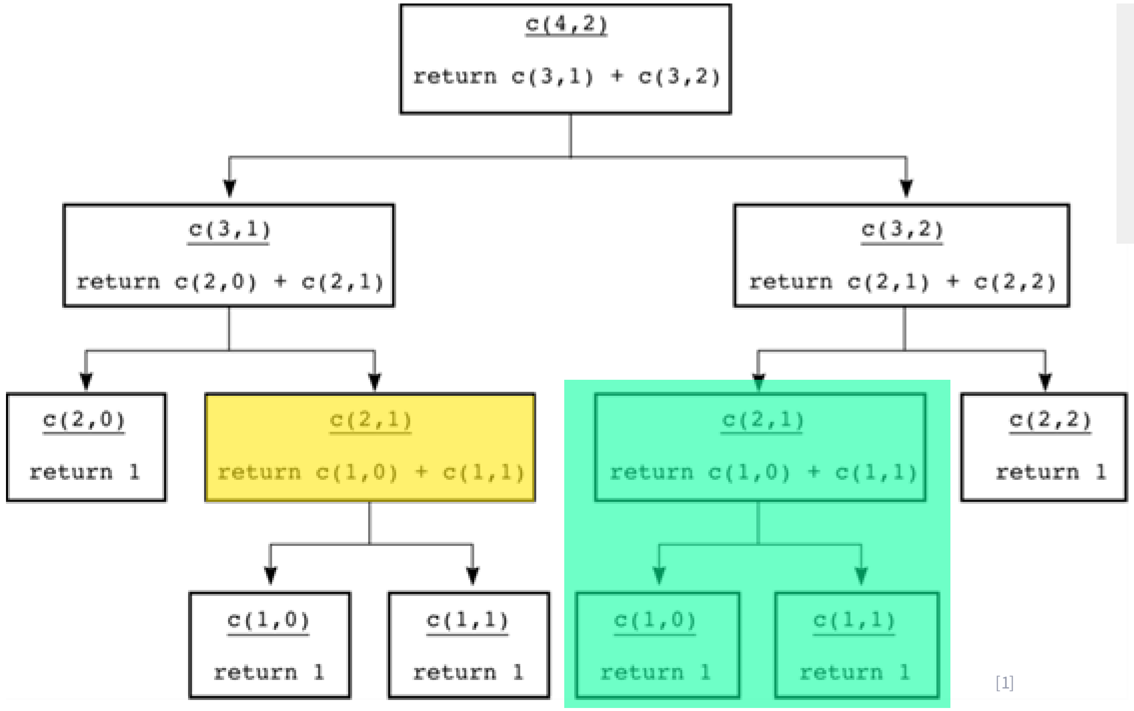
- Recursive definition:
  - C(*n*, *r*) = 1 if *r* = 0 or *n*
  - C(*n*, *r*) = 0 if *r* > *n*
  - C(*n*, *r*) = C(*n* - 1, *r* - 1) + C(*n* - 1, *r*)

$$_nC_r = {}_{n-1}C_{r-1} + {}_{n-1}C_r$$

```
def comb(n, r):
  if r == 0 or r == n:
    return 1
  elif r > n:
    return 0
  else:
    return comb(n-1, r-1) + comb(n-1, r)
```

This implementation is not efficient.
Why?

Combination



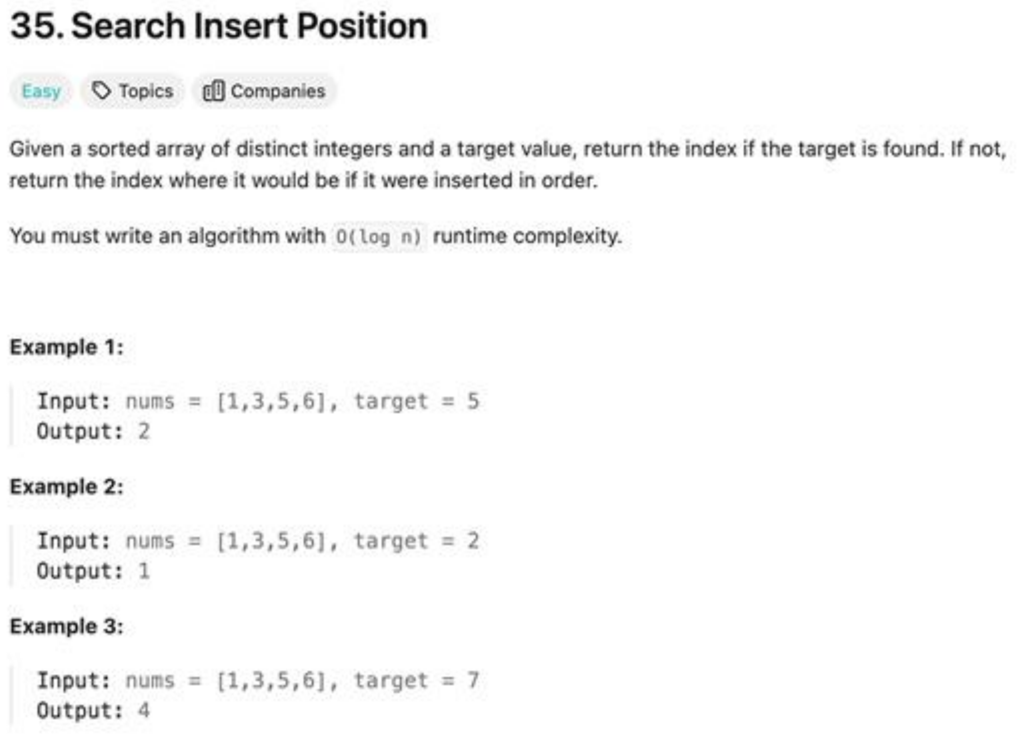
```
def comb(n, r):
    if r == 0 or r == n:
        return 1
    elif r > n:
        return 0
    else:
        return comb(n-1, r-1) + comb(n-1, r)
```

[1]. https://storm.cis.fordham.edu/~yli/documents/CISC2200Fall15/Recursive_7.pdf

Recursion Summary

- It provides a simpler way to interpret the problem.
- It becomes extremely **inefficient** if there's **duplicated** computation.
- Make sure that your algorithm always meets the **base case** to terminate.

Problem 1 – Binary Search



35. Search Insert Position

Easy Topics Companies

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with `O(log n)` runtime complexity.

Example 1:

```
Input: nums = [1,3,5,6], target = 5
Output: 2
```

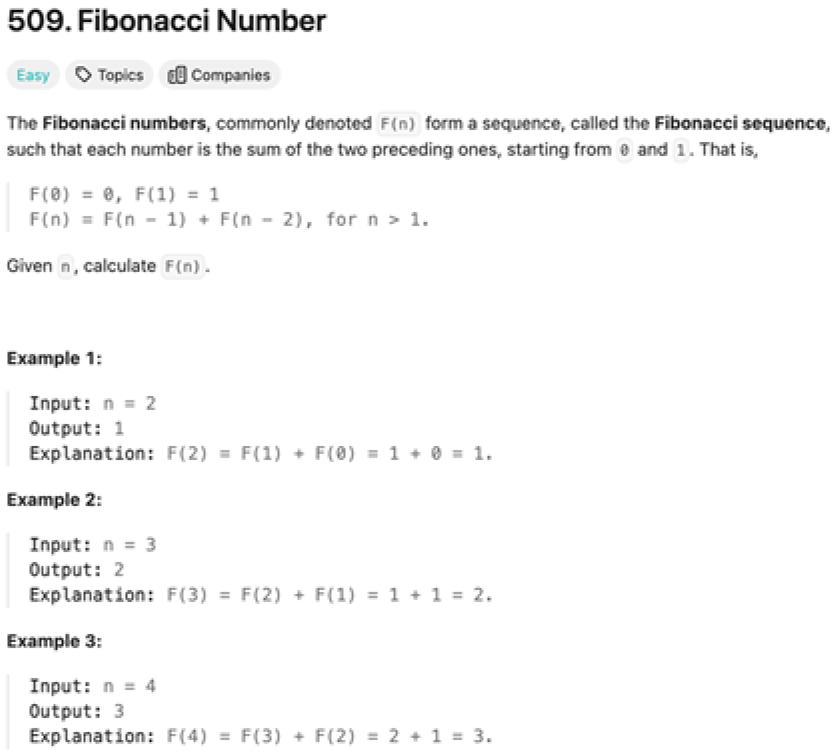
Example 2:

```
Input: nums = [1,3,5,6], target = 2
Output: 1
```

Example 3:

```
Input: nums = [1,3,5,6], target = 7
Output: 4
```

Problem 2 - Recursion



509. Fibonacci Number

Easy · Topics · Companies

The **Fibonacci numbers**, commonly denoted `F(n)` form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from `0` and `1`. That is,

```
F(0) = 0, F(1) = 1
F(n) = F(n - 1) + F(n - 2), for n > 1.
```

Given `n`, calculate `F(n)`.

Example 1:

```
Input: n = 2
Output: 1
Explanation: F(2) = F(1) + F(0) = 1 + 0 = 1.
```

Example 2:

```
Input: n = 3
Output: 2
Explanation: F(3) = F(2) + F(1) = 1 + 1 = 2.
```

Example 3:

```
Input: n = 4
Output: 3
Explanation: F(4) = F(3) + F(2) = 2 + 1 = 3.
```

upstage

Building intelligence for the future of work