MF20006: Introduction to Computer Science

Lecture 6: Algorithm I

Hui Xu

xuh@fudan.edu.cn

Outline

1. Probability Simulation

2. Sorting Algorithm

3. String Matching

1. Probability Simulation

The Monty Hall Problem

- ❑A famous probability puzzle named after the host of the TV show Let's Make a Deal.
  - ➢There are 3 doors.
  - ➢Behind 1 door is a car (the prize).
  - ➢Behind the other 2 doors are goats.

The Monty Hall Problem

1) The player chooses one door.
2) The host opens one of the other doors, always revealing a goat.
3) The player is given a choice: stay with the original door, or switch to the other unopened door?

1. Chosen by the player

2. Opened by the host

Solve the Problem with Python

```
stay_wins = 0
switch_wins = 0
trials = 100000

for _ in range(trials):
    doors = {0, 1, 2}
    prize = random.randint(0, 2)
    choice = random.randint(0, 2)

    possible_doors = doors - {choice, prize}
    opened = random.choice(possible_doors)

    # If the player stays
    if choice == prize:
        stay_wins += 1

    # If the player switches
    switch_choice = (doors - {choice, opened}).pop()
    if switch_choice == prize:
        switch_wins += 1
```

Result

```python
print(f"Trials: {trials}")
print(f"Win rate if stay   : {stay_wins / trials:.3f}")
print(f"Win rate if switch : {switch_wins / trials:.3f}")
```

```
Trials: 100000
Win rate if stay   : 0.333
Win rate if switch : 0.667
```

Kelly Criterion

- **There is a bet (or investment) with known odds and probability of winning, e.g.,**
  - Net odds of 2:1 mean you win 2 units (excluding your stake) for every 1 unit bet.
  - The probability of winning is 60%.
- **What fraction of my capital should I bet to maximize my long-term growth rate (*e.g.,* after 100 times of bet) of wealth?**

Simulation Experiment

```
prob_win = 0.6
odds = 2

def simulate(game, capital, bet_fraction):
    for win in game:
        bet = capital * bet_fraction
        if win:
            capital += bet * odds
        else:
            capital -= bet
    return capital

rounds = 10000
# The range of random.random() is [0.0, 1.0)
game = [random.random() < prob_win for _ in range(rounds)]

result1 = simulate(game, 100, 0.3)
result2 = simulate(game, 100, 0.4)
print(f"Result with bet fraction 0.3: {result1:.2f}")
print(f"Result with bet fraction 0.4: {result2:.2f}")
```

Kelly Criterion

$$f = p - \frac{1-p}{odds}$$

When odds =2, p=0.6,

$$f = 0.6 - \frac{1-0.6}{2} = 0.4$$

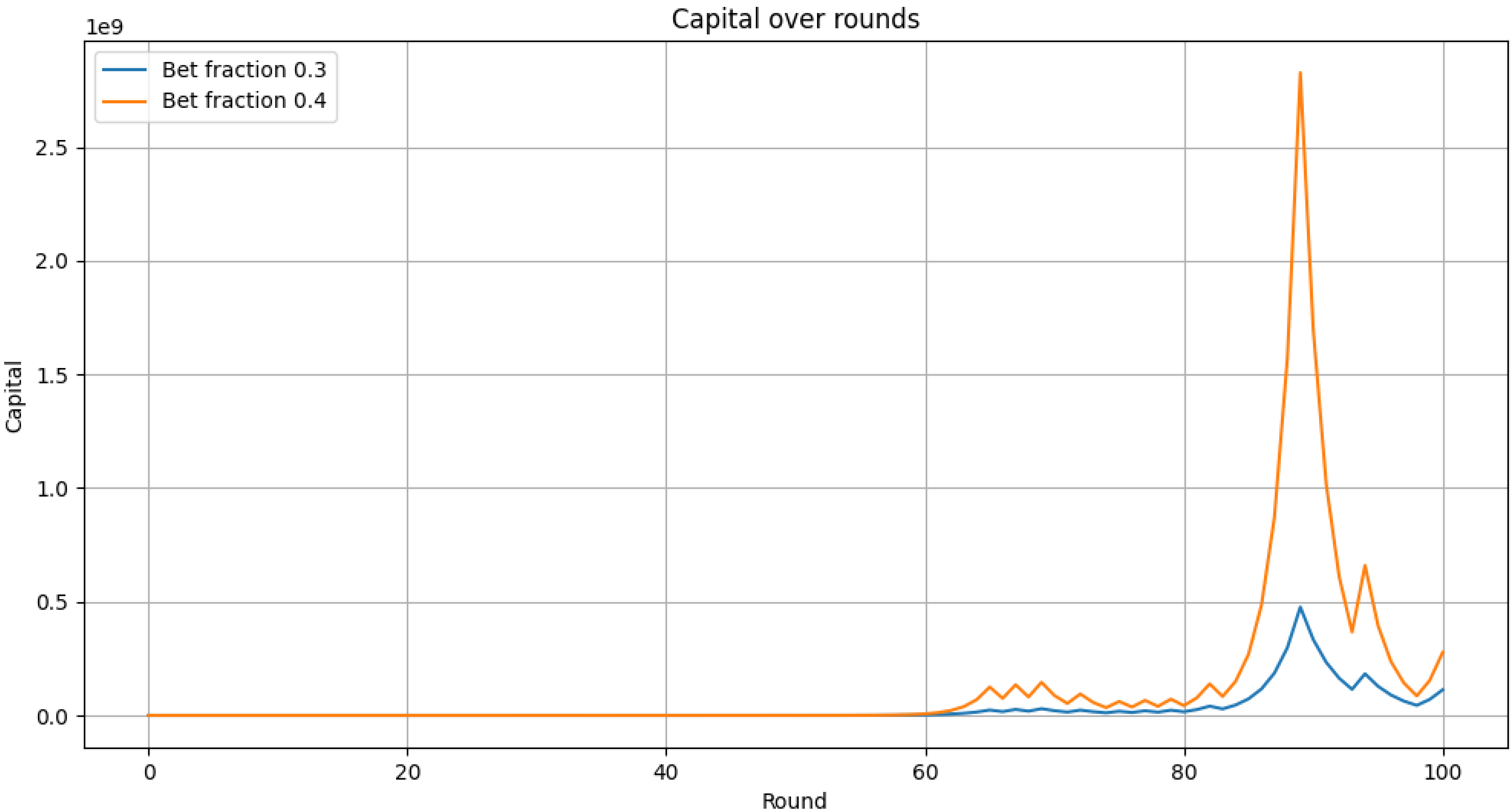
Simulation Experiment: Logging the Capital History

```
def simulate(game, capital, bet_fraction):
    history[capital]
    for win in game:
        bet = capital * bet_fraction
        if win:
            capital += bet * odds
        else:
            capital -= bet
    return history

rounds = 10000
game = [random.random() < prob_win for _ in range(rounds)]

history1 = simulate(game, 100, 0.3)
history2 = simulate(game, 100, 0.4)
```

Result Analysis via Visualization



Result Analysis via Visulization

```python
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.plot(history1, label='Bet fraction 0.3')
plt.plot(history2, label='Bet fraction 0.4')
plt.xlabel('Round')
plt.ylabel('Capital')
plt.title('Capital over rounds')
plt.legend()
plt.grid(True)
# plt.savefig("capital_simulation.png", dpi=300)
plt.show()
```

2. Sorting Algorithm

Scenario

- We want to display stocks in ascending or descending order by name, price, volume, or other criteria.

Seria	Symbol	Name	Price	Chg	% Chg	Volume	Turnover	Market
1	800000	Hang Seng Index	22736.87	+623.36	+2.82%	0	261.5B	0
2	00700	TENCENT	477.600	+11.400	+2.45%	24.66M	11.69B	4.432T
3	SPY	SPDR S&P 500 ETF	572.980	+5.160	+0.91%	43.01M	24.56B	589.9B
4	TSLA	Tesla	250.080	+9.420	+3.91%	86.73M	21.52B	798.92B
5	AAPL	Apple	226.800	+1.130	+0.50%	37.35M	8.436B	3.448T
6	FUTU	Futu Holdings Ltd	127.980	+5.190	+4.23%	14.55M	1.815B	17.651B
7	NVDA	NVIDIA	124.920	+2.070	+1.68%	244.5M	30.31B	3.064T

The Sorting Problem

- **Given an array of elements, output a new array sorted in either ascending or descending order.**
- **Classic solutions:**
  - Selection sort
  - Bubble sort
  - Quick sort
  - Radix sort

49	38	65	97	76	13	27	49

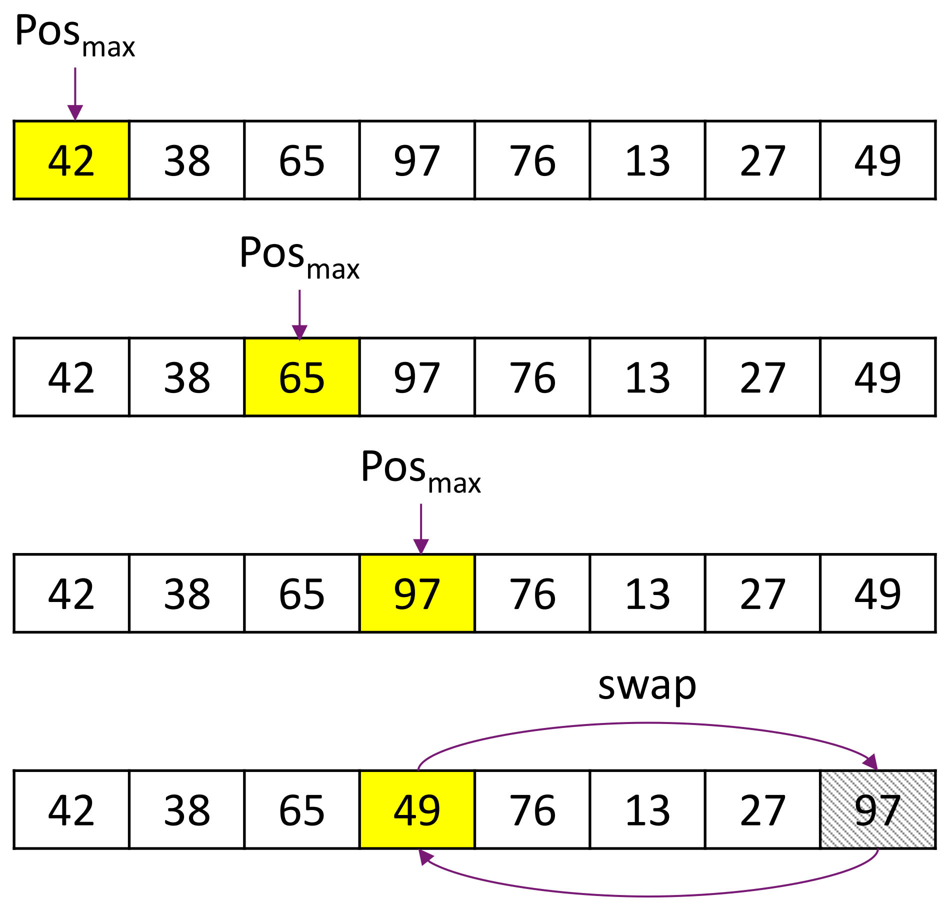
sort (ascending)

13	27	38	49	49	65	76	97

Selection Sort

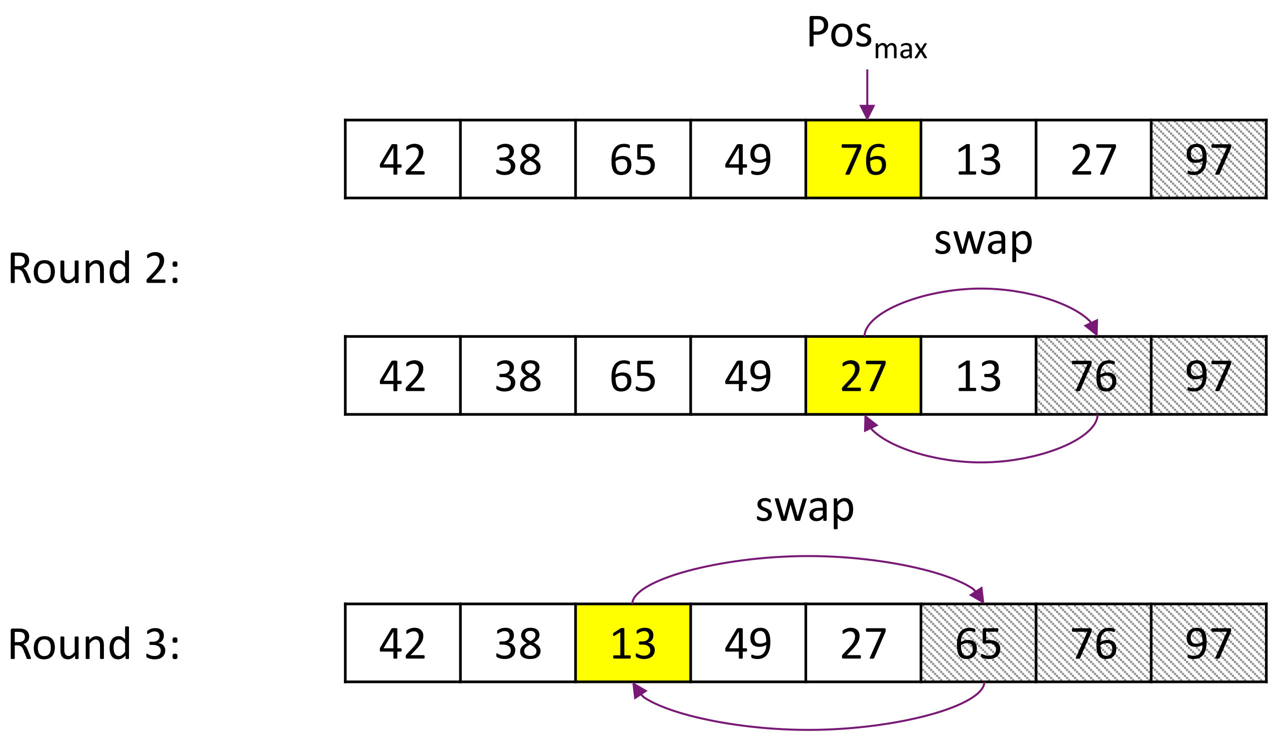
- In each round, find the largest element.
- Swap it with the last unsorted element.

Round 1:



Selection Sort

- **Repeat the selection and swap operations iteratively.**



…

Selection Sort Algorithm

```python
def selection_sort(l):
    n = len(l)
    for i in range(n):
        max_idx = 0
        for j in range(1, n - i):
            if l[j] > l[max_idx]:
                max_idx = j
        l[max_idx], l[n-1-i] = l[n-1-i], l[max_idx]
```

Complexity and Big O Notation

❑Complexity analysis:

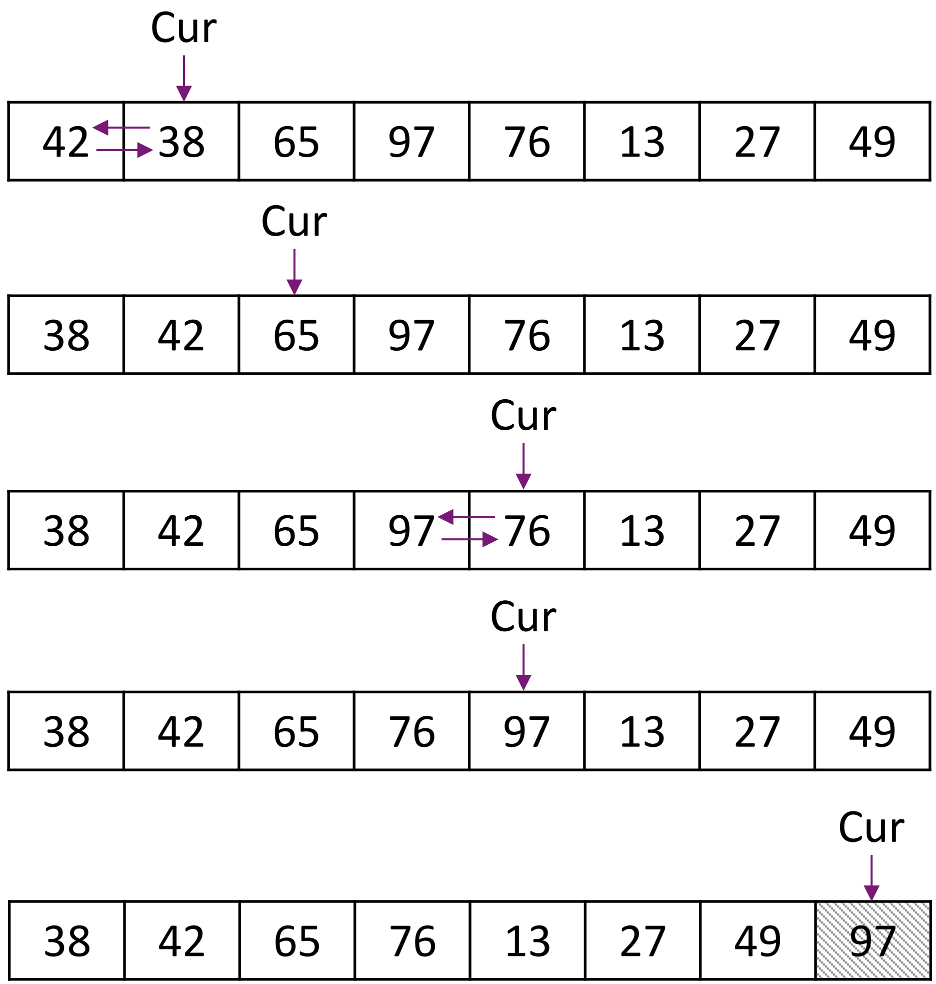
- How many rounds do we need to perform?
- How many comparisons are needed in each round?
- How many comparisons are needed in total?

❑Order of approximation: $O(n^2)$

- $(n-1) \times \frac{(n-1)+1}{2} = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$

Bubble Sort

- ❑Swap two adjacent elements if they are not in ascending order.
- ❑Similar to selection sort, but it performs multiple swaps in each pass.



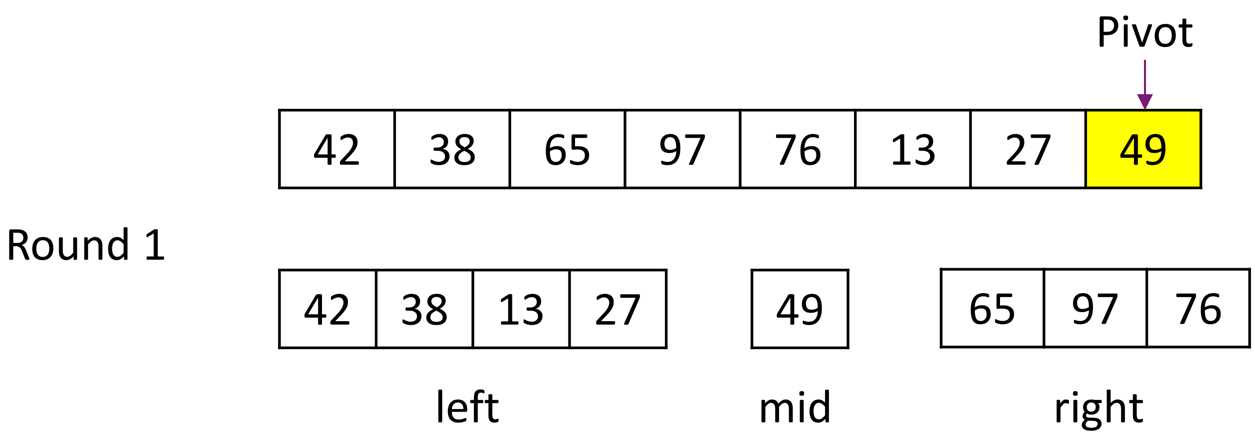
Bubble Sort Algorithm

```python
def bubble_sort(l):
    n = len(l)
    for i in range(n):
        swapped = False
        for j in range(n - i - 1):
            if l[j] > l[j + 1]:
                l[j], l[j + 1] = l[j + 1], l[j]
                swapped = True
        if not swapped:
            break
```

Bubble sort performs better if the array is already sorted.

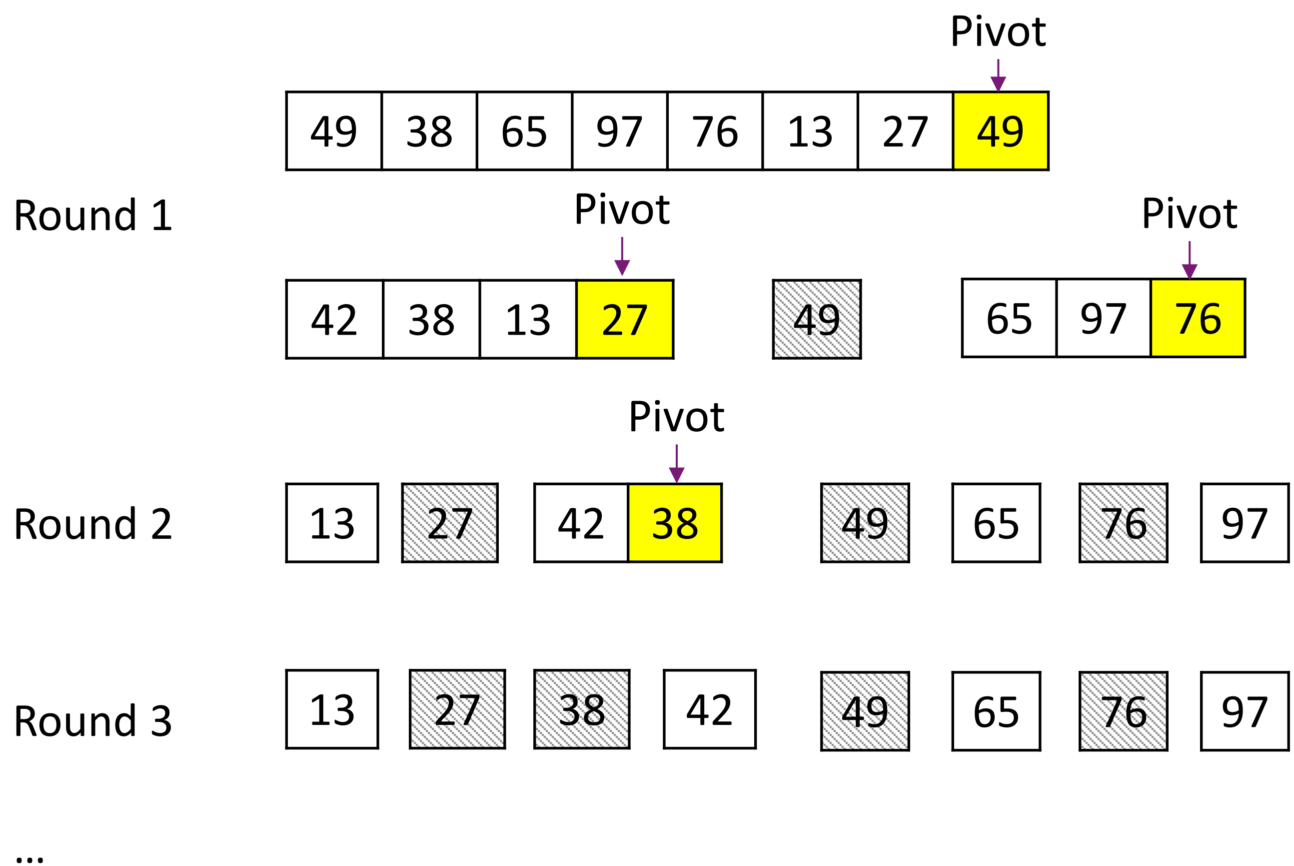
Quick Sort: Divide-and-Conquer Approach

- ❑Select a pivot element from the array and partition the other elements into two sub-arrays in each round.
- ❑All elements in the left array are less than the pivot.
- ❑All elements in the right array are grater than or equal to the pivot.
- ❑Recursively sorted the sub-arrays.



Quick Sort: Time Complexity

- ❑**Average complexity:** $O(nlogn)$
- ❑**Worst-case complexity:** $O(n^2)$



Quick Sort Algorithm

```
def quick_sort(l):
    if len(l) <= 1:
        return
    pivot_index = partition(l)
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

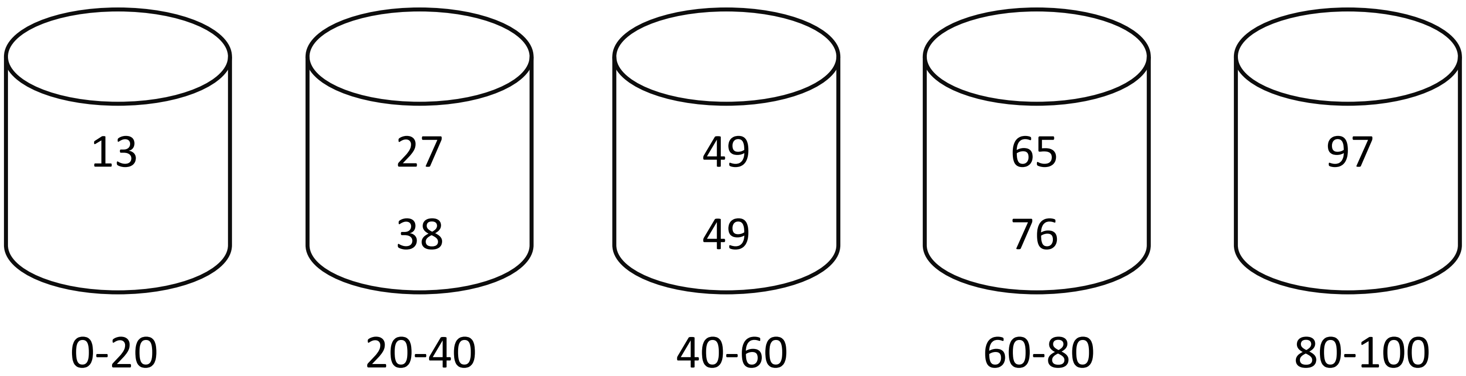
Space Complexity Analysis

- ❑Each recursive call builds new lists.
- ❑These lists together hold all elements of the array.
- ❑Therefore, each level of recursion allocates $O(n)$ additional space.
- ❑Average complexity: $O(nlogn)$

Quicker: Bucket Sort

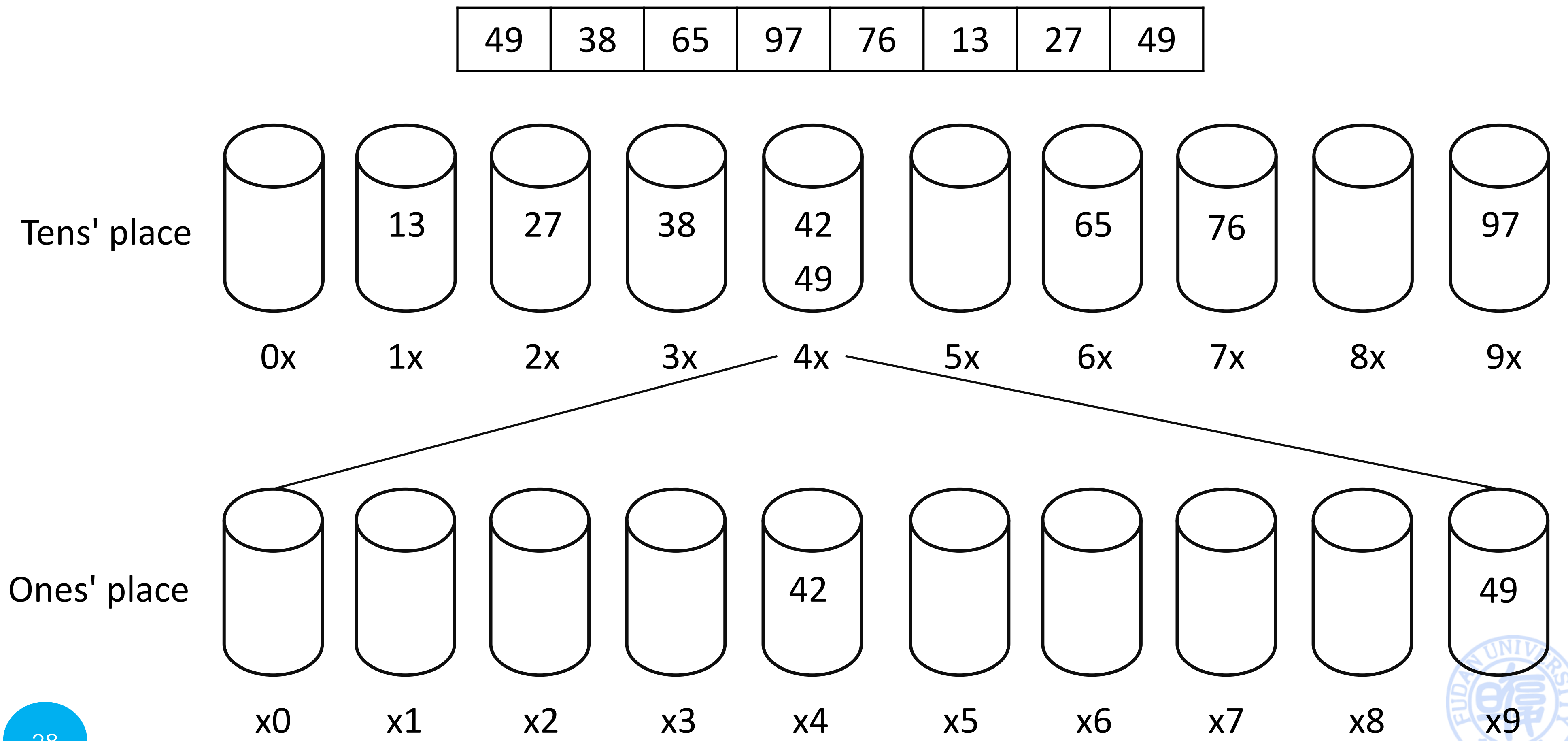
- Instead of dividing the elements into two subset, we distribute them into multiple subsets or buckets.

49	38	65	97	76	13	27	49



Radix Sort

- ❑Distribute the elements based on the digits at each position. Then, select the elements in ascending order.



Radix Sort Algorithm

```python
def radix_sort(l):
    # Find the maximum number to know how many digits we need
    max_num = max(l)
    # Determine the highest place (e.g., 1000s, 100s, 10s, 1s)
    exp = 1
    while max_num // exp >= 10:
        exp *= 10

    return sort_helper(l, exp)
```

Radix Sort Algorithm

```python
def sort_helper(arr, exp):
    if len(l) <= 1 or exp == 0:
        return l  # Base case: one element or no more digits

    # Create 10 buckets for digits 0-9
    buckets = [[] for _ in range(10)]
    for ele in l:
        digit = (ele // exp) % 10
        buckets[digit].append(ele)

    # Recursively sort each bucket by the next lower digit
    result = []
    for b in buckets:
        if b:
            result.extend(sort_helper(b, exp // 10))
    return result
```

Complexity of Radix Sort

- Suppose the list length is n, and each element has at most w digits.
- We need to distribute the elements $w \times n$ times.
- Cost: Additional space is required to keep track of the distributions

Exercise

❑Design experiments to study the performance of sorting algorithms.

```
size = 10000
data = [random.randint(0, 10000) for _ in range(size)]
print(f"\nArray size: {size}")
print(f"Selection Sort: {benchmark(selection_sort, data):.6f} s")
print(f"Bubble Sort:    {benchmark(bubble_sort, data):.6f} s")
print(f"Quick Sort:     {benchmark(quick_sort, data):.6f} s")
print(f"Radix Sort:     {benchmark(radix_sort, data):.6f} s")
```

```
Array size: 10000
Selection Sort: 3.014266 s
Bubble Sort:    6.400471 s
Quick Sort:     0.019425 s
Merge Sort:     0.030293 s
Radix Sort:     0.018060 s
```

3. String Matching

The String-Matching Problem

- How to find the place that a string pattern appears in a text?

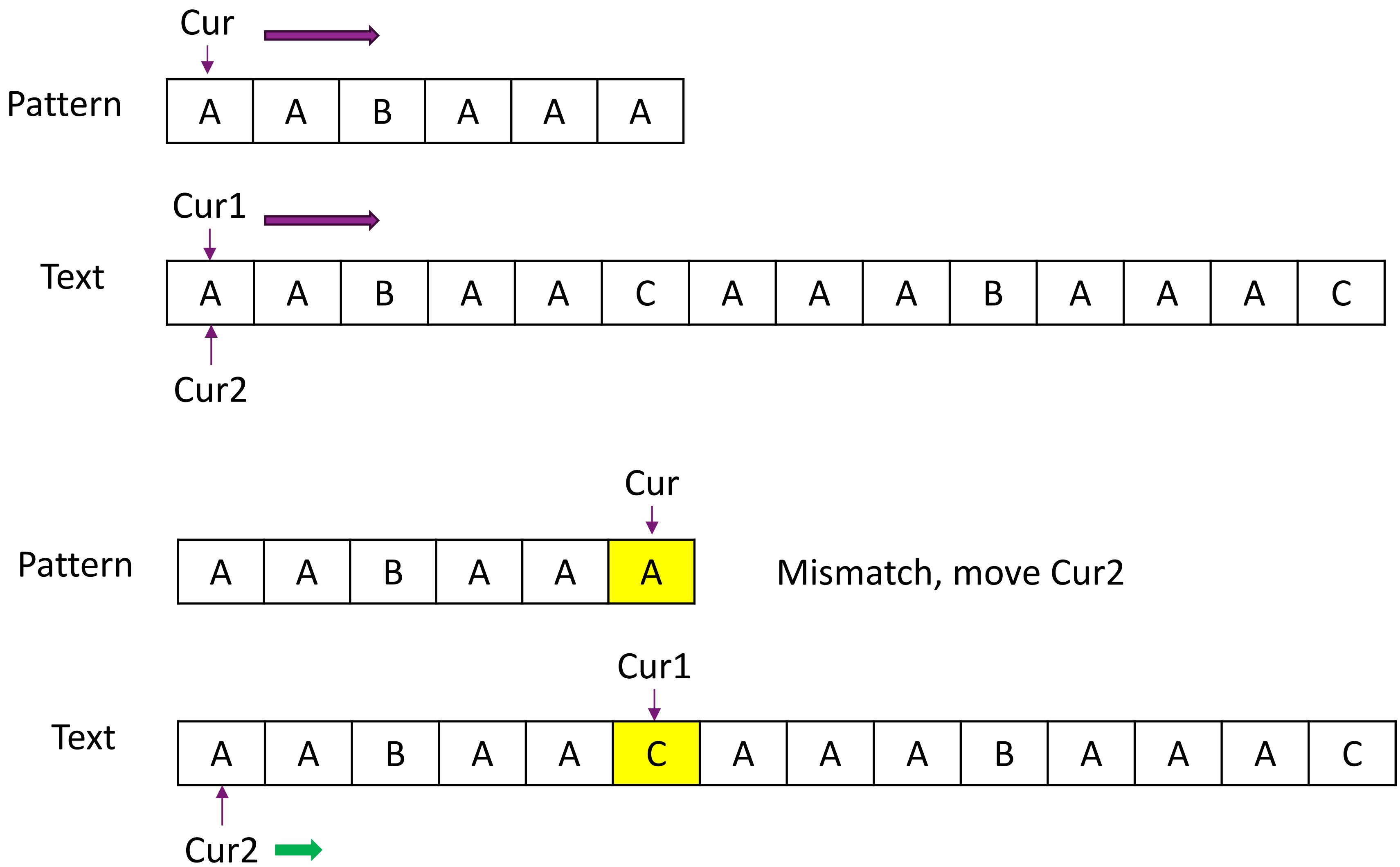
Pattern

A	A	B	A	A	A

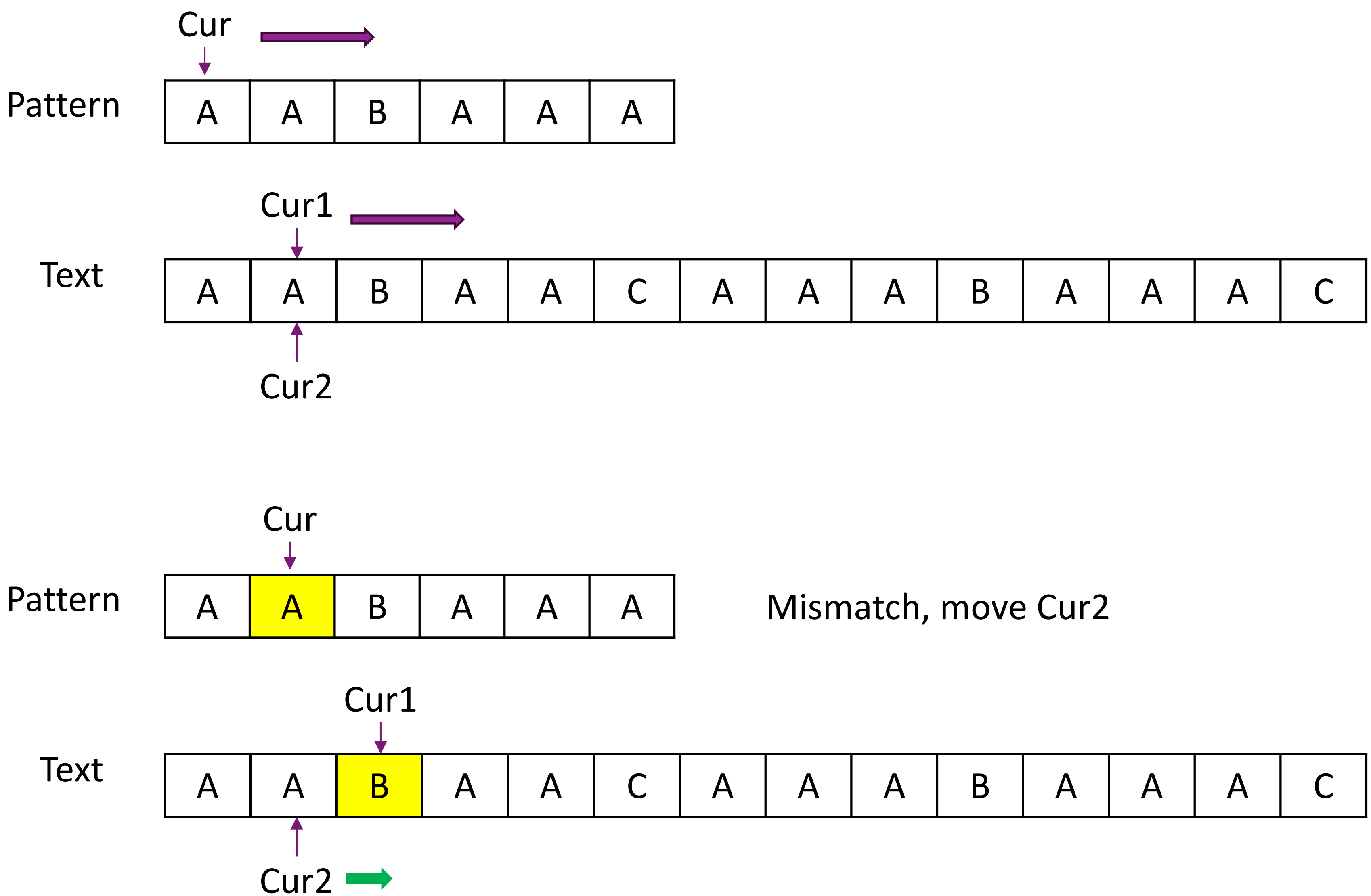
Text

A	A	B	A	A	C	A	A	A	B	A	A	A	

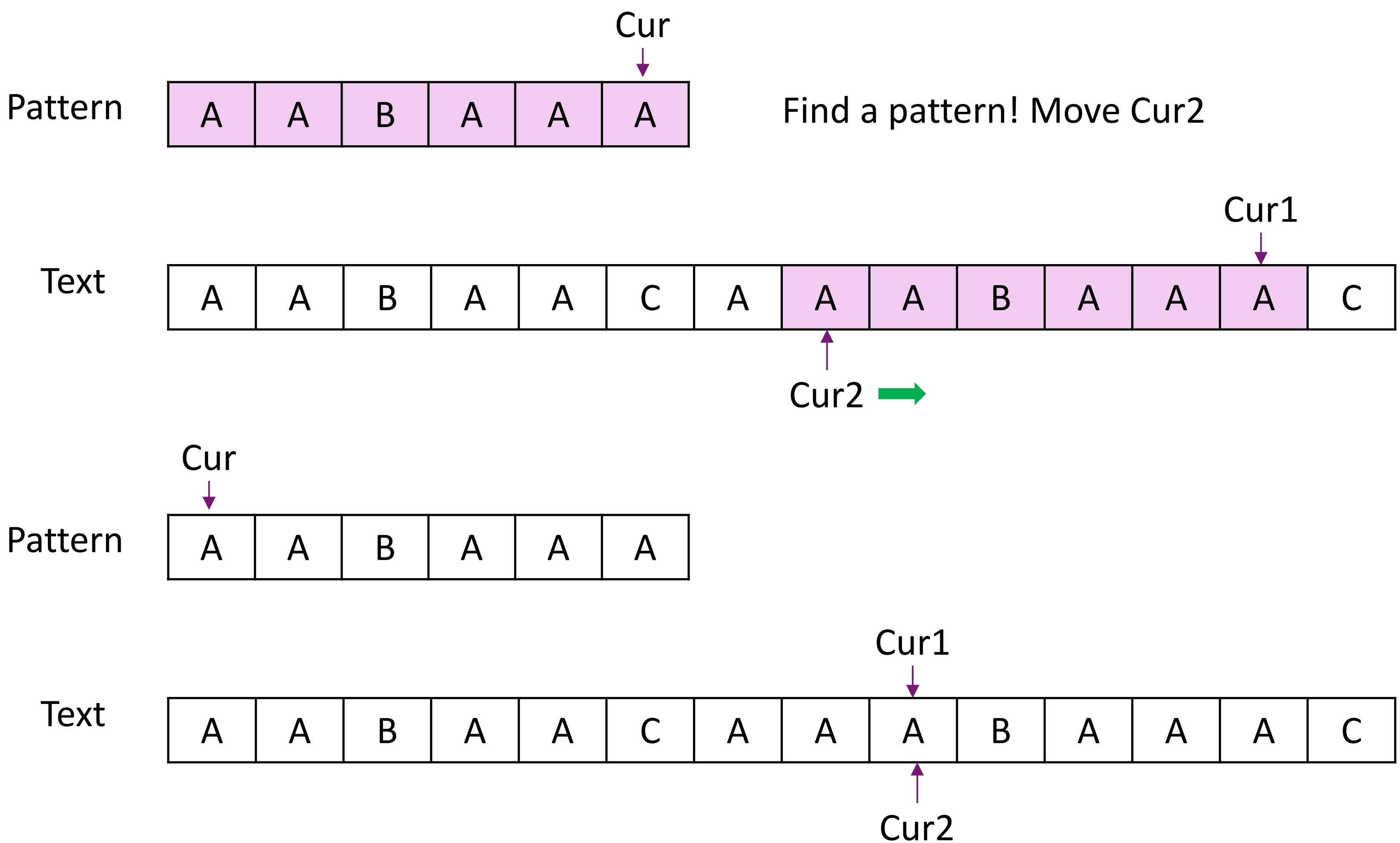
Naïve Approach



Naïve Approach



Result



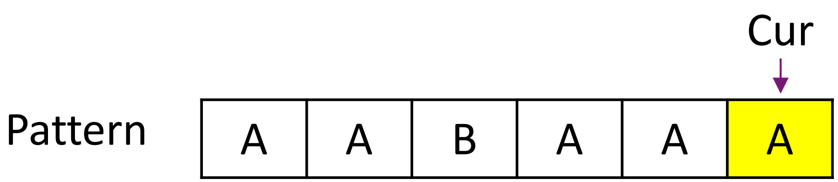
Complexity: $O(l_1 * l_2)$

Implementation in Python

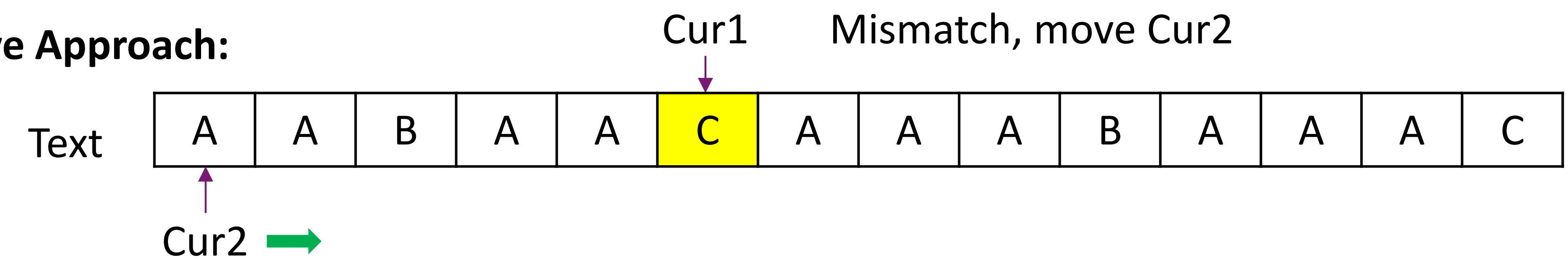
```python
def naive_search(text, pattern):
    n, m = len(text), len(pattern)
    result = []
    for i in range(n - m + 1):
        if text[i:i+m] == pattern:
            result.append(i)
    return result
```

KMP Algorithm

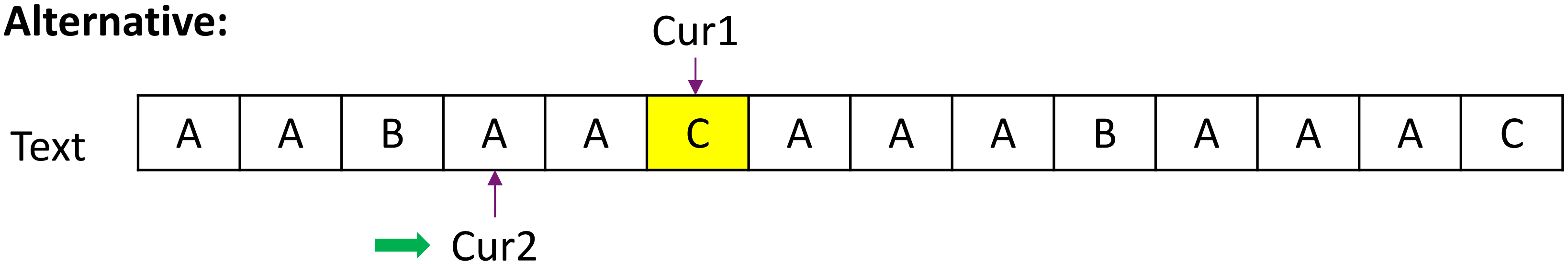
- ❑Keep records of the prefix appeared in the already matched substring.
- ❑Continue from the next position if such prefix does not exist.



Naive Approach:



Alternative:



Analyze the Pattern: Longest Prefix Suffix (LPS)

- The LPS of a pattern at position i indicates the length of the longest proper prefix of the pattern (up to i), which is also a suffix.
- Move several steps to the left based on the LPS value.

A	A	B	A	A	A
0	1	0	1	2	2

- LSP(0) = 0 because the "A" does not have a prefix nor suffix.
- LSP(1) = 1 because the "AA" has a matched prefix and suffix "A".
- LSP(2) = 0 because the "AAB" has a matched prefix and suffix "F".
- LSP(3) = 1 because the "AABA" has a matched prefix and suffix "A".
- LSP(4) = 2 because the "AABAA" has a matched prefix and suffix "AA".
- LSP(5) = 2 because the "AABAAA" has a matched prefix and suffix "AA".

Search with LPS

```
def kmp_search(text, pattern):
    n, m = len(text), len(pattern)
    lps = compute_lps(pattern)
    result = []
    i = j = 0
    while i < n:
        if text[i] == pattern[j]:
            i += 1
            j += 1
        if j == m:
            result.append(i - j)
            j = lps[j-1]
        elif i < n and text[i] != pattern[j]:
            if j != 0:
                j = lps[j-1]
            else:
                i += 1
    return result
```

Demo: Using LPS

Pattern:	A	A	B	A	A	A
lps:	0	1	0	1	2	2

Text:	A	A	B	A	A	C	A	A	A	B	A	A	A

```
i = 0; j = 0; pattern[0] = text[0]; i += 1; j += 1;
i = 1; j = 1; pattern[1] = text[1]; i += 1; j += 1;
...
i = 5; j = 5; pattern[5] != text[5]; j = lps[j-1] = 2
i = 5; j = 2; pattern[2] != text[5]; j = lps[j-1] = 1
i = 5; j = 1; pattern[1] != text[5]; j = lps[1-1] = 0
i = 6; j = 0; pattern[0] != text[0]; i = 7; j = 1;
...
i = 8; j = 2; pattern[2] != text[8]; j = lps[2-1] = 1
i = 8; j = 1; pattern[1] = text[8]; i += 1; j += 1;
i = 9; j = 2; pattern[2] = text[9]; i += 1; j += 1;
...
```

Compute LPS

```
def compute_lps(pattern):
    m = len(pattern)
    lps = [0] * m
    len = 0
    i = 1
    while i < m:
        if pattern[i] == pattern[length]:
            len += 1
            lps[i] = len
            i += 1
        else:
            if len != 0:
                len = lps[len-1]
            else:
                lps[i] = 0
                i += 1
    return lps
```

Demo: Steps of LPS Computating

Pattern:	A	A	B	A	A	A
lps:	0	1	0	1	2	2

```
len = 0; i = 1; pattern[0] = pattern[1]; len += 1; lps[1] = 1; i += 1
len = 1; i = 2; pattern[1] != pattern[2]; len = lps[0] = 0;
len = 0; i = 2; pattern[0] != pattern[2]; lps[2] = 0; i += 1
len = 0; i = 3; pattern[0] = pattern[3]; len += 1; lps[3] = 1; i += 1
len = 1; i = 4; pattern[1] = pattern[4]; len += 1; lps[4] = 2; i += 1
len = 2; i = 5; pattern[2] != pattern[5]; len = lps[1] = 1
len = 1; i = 5; pattern[1] = pattern[5]; len += 1; lps[5] = 2; i += 1
```

Question

- Design a specific string matching case such that the KMP algorithm outperforms the naïve approach.