6 Algorithm I

Hui Xu, xuh@fudan.edu.cn

Learning Objective:

- Learn how to use Python to simulate probability problems.
- Understand the basic concepts of sorting and common sorting algorithms.
- Explore the principles of string matching and common string-matching algorithms.

6.1 Probability Simulation

When a problem is difficult to solve analytically, simulation can be used to estimate its outcome. This section illustrates this approach using two probability problems.

6.2 Probability Simulation

When a problem is difficult to solve analytically, simulation can be used to estimate its outcome. This section illustrates this approach using two probability problems.

MF 20006

6.2.1 Monty Hall Problem

The Monty Hall problem is a famous probability puzzle based on a game show scenario. It demonstrates how simulation can help us understand counterintuitive results in probability. The rules of the Monty Hall problem are as follows:

- There are three doors. Behind one door is a prize, while behind the other two are goats.
- The contestant chooses one of the three doors.
- The host, who knows what is behind each door, opens one of the two unchosen doors, always revealing a goat.
- The contestant decides whether to stay with the original choice or switch to the other unopened door.
- The contestant wins the prize if the final choice is the door hiding the prize.

Which strategy gives a higher probability of winning: staying or switching? At first, it may seem that both options have an equal chance of winning. However, a Python simulation can help us estimate the actual probabilities. After many trials, we expect the probability of winning by staying to be approximately 1/3, while the probability of winning by switching is approximately 2/3.

```
import random

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)

    # If choice != prize, one possible door remains;
    # otherwise, two possible doors remain.
    possible_doors = doors - {choice, prize}
    opened = random.choice(list(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

print(f"Winning probability by staying: {stay_wins / trials:.3f}")
print(f"Winning probability by switching: {switch_wins / trials:.3f}")
```

Code 6.1: Monty Hall simulation

6.2.2 Kelly Criterion

Imagine a betting game in which each round has a known probability of winning. If you win, your capital increases according to the odds; if you lose, you lose the amount you bet. The question is: *What fraction of your capital should you bet in each round to maximize long-term growth?*

Simulation Experiment Suppose the probability of winning is $p = 0.6$, and the net odds are 2. We start with a capital of 100 and compare three strategies: betting 30%, 40% and 50% of the current capital in each round. Code 6.2 plots how capital evolves under the three betting strategies with 200 rounds. Although both strategies may experience fluctuations, their long-term growth rates differ. Repeated experiments can help us investigate which betting fraction tends to produce faster long-term growth.

The Kelly Criterion gives a theoretical answer to this question. The optimal fraction of capital to bet is $f = p - \frac{1-p}{\text{odds}}$. For $p = 0.6$ and odds $= 2$, the formula gives $f = 0.6 - \frac{1-0.6}{2} = 0.4$. Therefore, the Kelly Criterion recommends betting 40% of the current capital in each round under these assumptions. This theoretical result helps explain why the 40% strategy is expected to achieve a higher long-term growth rate.

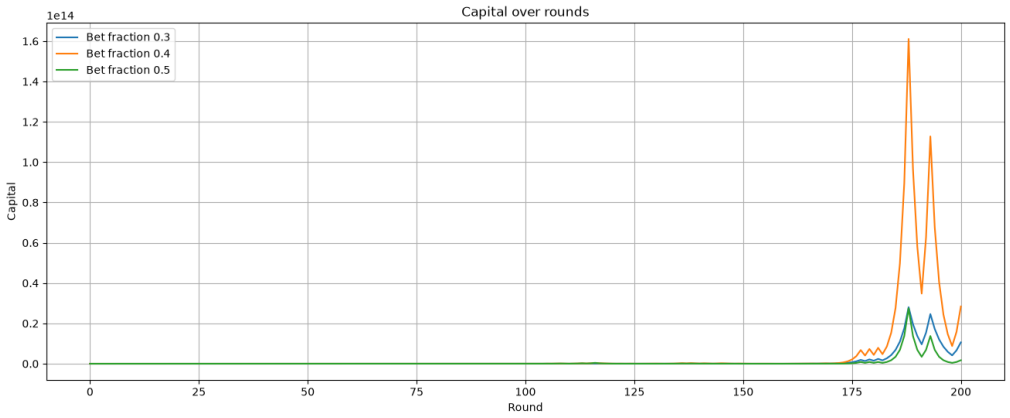


Figure 6.1: Visualization of the performance of different betting strategies.

```python
import random
import matplotlib.pyplot as plt

prob_win = 0.6
odds = 2

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

rounds = 200
game = [random.random() < prob_win for _ in range(rounds)]

history1 = simulate(game, 100, 0.3)
history2 = simulate(game, 100, 0.4)
history3 = simulate(game, 100, 0.5)

plt.figure(figsize=(20, 6))
plt.plot(history1, label='Bet fraction 0.3')
plt.plot(history2, label='Bet fraction 0.4')
plt.plot(history3, label='Bet fraction 0.5')
plt.xlabel('Round')
plt.ylabel('Capital')
plt.title('Capital over rounds')
plt.legend()
plt.grid(True)
plt.show()
```

Code 6.2: Simulating experiment with different betting strategies

6.3 Sorting Algorithm

Given a list of elements, the sorting problem is to arrange the elements in a specific order, typically in ascending or descending order. Figure 6.2 demonstrates an example.

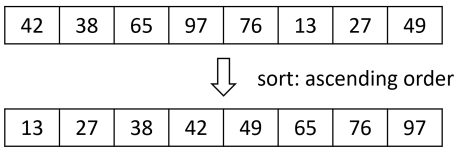


Figure 6.2: Demonstration of a sorting problem.

Sorting is a fundamental problem in computer science, and many algorithms have been developed to solve it efficiently. This section introduces several commonly used algorithms, including selection sort, bubble sort, quick sort, and radix sort.

6.3.1 Selection Sort

Selection sort is a simple comparison-based sorting algorithm. The idea is to repeatedly find the largest (or smallest) element from the unsorted portion of the list and move it to its correct position at the end (or beginning) of the sorted portion. This process is repeated until the entire list is sorted.

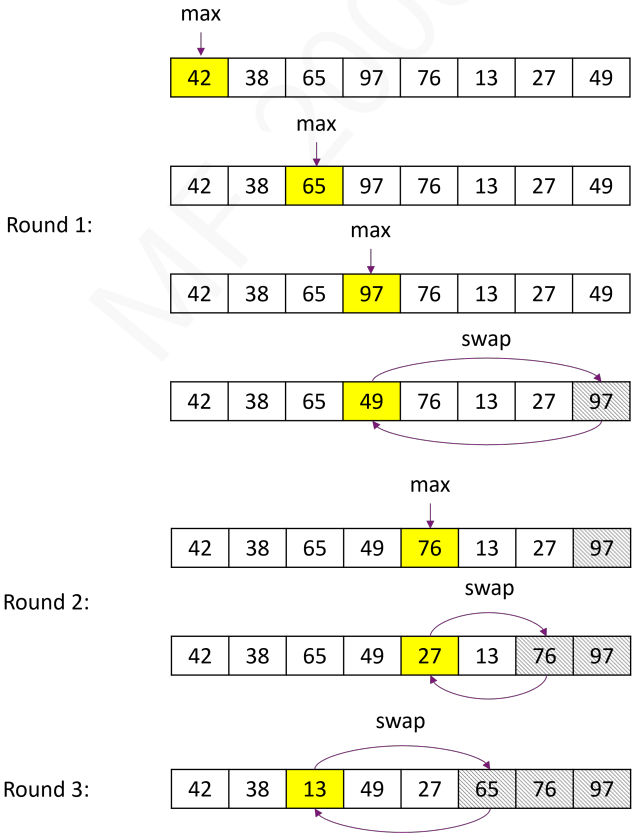


Figure 6.3: Demonstration of selection sort.

Figure 6.3 illustrates the process of selection sort step by step. In the first round, the algorithm finds the largest element and moves it to the end of the list. To do this, it maintains an index that records the position of the current maximum element and compares each remaining element against it. In the second

round, the algorithm repeats the process on the unsorted portion of the list, again selecting the largest element and moving it to its correct position. This procedure continues until the entire list is sorted. Code 6.3 demonstrates the algorithm with Python.

```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]
```

Code 6.3: Selection sort

6.3.2 Bubble Sort

Bubble sort is a simple comparison-based sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. Figure 6.4 illustrates the first pass, in which the largest element "bubbles up" to the end of the list.

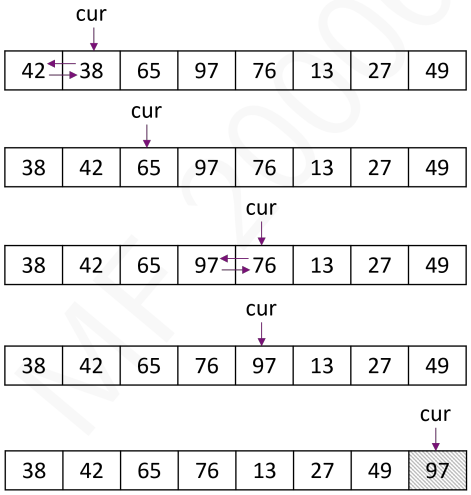


Figure 6.4: Demonstration of bubble sort.

Listing 6.4 shows the implementation. After each pass, the sorted suffix grows by one element. The swapped flag allows the algorithm to terminate early if no swaps occur.

```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
```

Code 6.4: Bubble sort

6.3.3 Quick Sort

Quick sort is a divide-and-conquer sorting algorithm. In the divide-and-conquer strategy, a complex problem is divided into smaller, more manageable subproblems that can be solved independently and then combined to form the final solution. The main idea is to select a pivot element, and partition the list into elements less than, equal to, and greater than the pivot. It then recursively sorts the resulting sublists until each sublist contains at most one element, producing a fully sorted list.

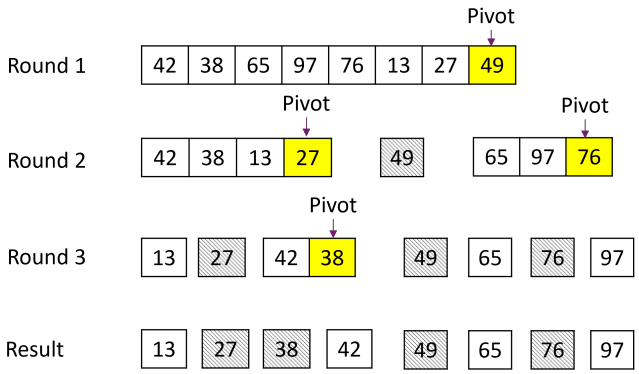


Figure 6.5: Demonstration of quick sort.

Figure 6.5 illustrates the partitioning and recursive process of quick sort based on the implementation of Code 6.5. In this implementation, the list is split into three parts relative to the pivot. The algorithm then recursively sorts the left and right sublists and combines them with the middle elements. This recursive approach ensures that all elements are eventually placed in the correct order.

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

Code 6.5: Quick sort

Quick sort is much faster than both selection sort and bubble sort for large datasets. It typically runs in $O(n \log n)$ time by dividing the list around a pivot and sorting the parts recursively, though in the worst case it can take $O(n^2)$. In contrast, both selection sort and bubble sort have average and worst-case time complexities of $O(n^2)$, making them inefficient for large inputs.

6.3.4 Radix Sort

Radix sort is a non-comparative sorting algorithm that sorts numbers by processing their digits. Unlike comparison-based algorithms, it organizes numbers digit by digit, typically starting from the most significant digit or the least significant digit. In each pass, numbers are placed into buckets according to the value of the current digit. The algorithm then processes the next digit recursively or iteratively until the entire list is sorted.

Figure 6.7 illustrates the step-by-step process of radix sort. Since the largest element is 97, the algorithm first distributes the numbers into buckets based on the ten's digit. After that, it processes the

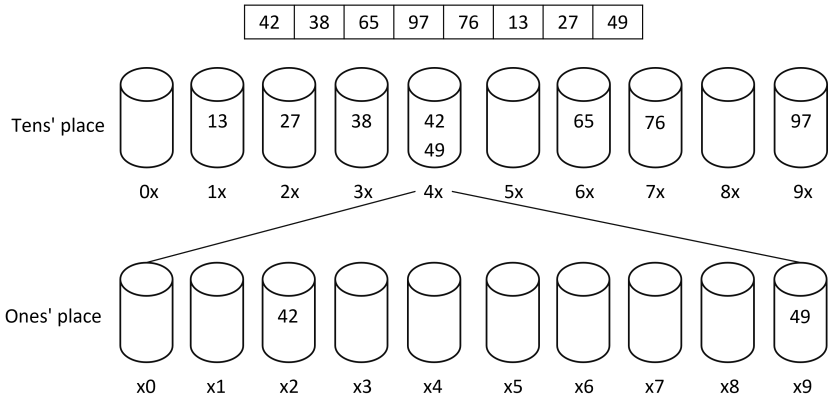


Figure 6.6: Demonstration of bubble sort.

unit digit to further organize the numbers. This two-step process results in a fully sorted list. Code 6.6 demonstrates the algorithm implementation in Python. The function `radix_sort` first determines the highest place value in the list. The helper function `sort_helper` then distributes elements into buckets based on the current digit and recursively sorts each bucket by the next lower digit. By processing all digits in this way, the algorithm produces a fully sorted list.

```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)

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
```

Code 6.6: Radix sort

Radix sort has a time complexity of $O(d \times n)$, where n is the number of elements, d is the number of digits in the largest number. It runs in linear time for fixed digit lengths, making it efficient for integers or strings of limited size. However, it requires additional space for buckets.

6.4 String Matching

String matching is the problem of finding all occurrences of a pattern string within a larger text string. It is also a fundamental problem in computer science, with applications in text processing, search engines, and bioinformatics. This section introduces two approaches: the naive brute-force method and the more efficient Knuth-Morris-Pratt (KMP) algorithm.

6.4.1 Naive Approach

The naive approach checks for the pattern at every possible position in the text. Although simple to implement, it can be inefficient for long texts and patterns.

```python
def str_match_brute(text: str, pattern: str) -> List[int]:
    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
```

Code 6.7: Naive string matching

6.4.2 Knuth-Morris-Pratt (KMP) Algorithm

The KMP algorithm improves efficiency by avoiding redundant comparisons. It preprocesses the pattern to compute the *longest prefix-suffix (LPS)* array, which is used to skip characters in the text that have already been matched.

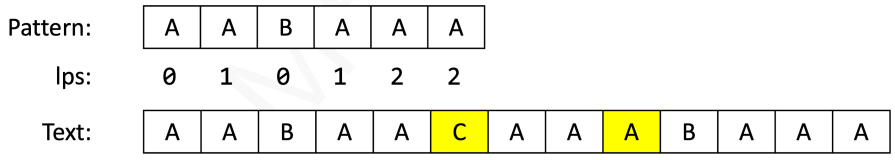


Figure 6.7: Demonstration of string matching via the longest prefix-suffix.

LPS is defined as the length of the longest proper prefix which is also a suffix of a string. For example, the LPS of a target pattern string "AABAAA" can be calculated as follows:

- LSP(0) = 0, because the "A" does not have a prefix or suffix.
- LSP(1) = 1, because the "AA" has a matched prefix and suffix "A".
- LSP(2) = 0, because the "AAB" does not have a matched prefix and suffix.
- 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".

If the LPS of the target pattern is known, we can use it to avoid unnecessary comparisons in the text. When a mismatch occurs between the text and the pattern, instead of restarting the match from the next

character in the text, we can use the LPS array to determine the next position in the pattern to compare. This allows the algorithm to skip characters that are guaranteed to match, improving efficiency.

```
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
```

Code 6.8: KMP string matching

Code 6.8 demonstrates the KMP string matching algorithm using the LPS array. The main execution steps of the algorithm for the given text and pattern are summarized in Table 6.1. This table shows the values of the text index i, the pattern index j, and the corresponding actions taken during each step.

Table 6.1: KMP Matching Steps

i	j	Action / Notes
0	0	pattern[0] = text[0], i ← 1, j ← 1
1	1	pattern[1] = text[1], i ← 2, j ← 2
⋮	⋮	⋮
5	5	pattern[5] ≠ text[5], j ← lps[4] = 2
5	2	pattern[2] ≠ text[5], j ← lps[1] = 1
5	1	pattern[1] ≠ text[5], j ← lps[0] = 0
5	0	pattern[0] ≠ text[5], i ← 6
6	0	pattern[0] = text[6], i ← 7, j ← 1
⋮	⋮	⋮
8	2	pattern[2] ≠ text[8], j ← lps[1] = 1
8	1	pattern[1] = text[8], i ← 9, j ← 2
9	2	pattern[2] = text[9], i ← 10, j ← 3
10	3	pattern[3] = text[10], i ← 11, j ← 4
11	4	pattern[4] = text[11], i ← 12, j ← 5
12	5	pattern[5] = text[12], i ← 13, j ← 6, result.append(7), j = lps(5) = 2
13	2	i == n, terminate

```python
def compute_lps(pattern: str) -> List[int]:
    m = len(pattern)
    lps = [0] * m
    l = 0
    i = 1
    while i < m:
        if pattern[l] == pattern[i]:
            l += 1
            lps[i] = l
            i += 1
        else:
            if l != 0:
                l = lps[l-1]
            else:
                lps[i] = 0
                i += 1
    return lps
```

Code 6.9: Longest prefix-suffix computation

The LPS array for a given pattern can be computed using the algorithm shown in Code 6.9. Table 6.2 presents the main steps of the LPS computation for the pattern "`AABAAA`". Each row shows the current value of `l` (the length of the current longest prefix-suffix), the index i, and the action taken.

Table 6.2: LPS Computation Steps

l	i	Action / Notes
0	1	pattern[0] = pattern[1], l ← 1, lps[1] ← 1, i ← 2
1	2	pattern[1] ≠ pattern[2], l ← lps[0] = 0
0	2	pattern[0] ≠ pattern[2], lps[2] ← 0, i ← 3
0	3	pattern[0] = pattern[3], l ← 1, lps[3] ← 1, i ← 4
1	4	pattern[1] = pattern[4], l ← 2, lps[4] ← 2, i ← 5
2	5	pattern[2] ≠ pattern[5], l ← lps[1] = 1
1	5	pattern[1] = pattern[5], l ← 2, lps[5] ← 2, i ← 6
2	6	i == m, terminate

KMP is especially efficient when the pattern contains repeated sub-patterns. In such cases, naive string matching would repeatedly re-examine characters that are known to match, leading to unnecessary comparisons. By using the LPS array, KMP avoids this redundancy by skipping portions of the text that have already been matched, resulting in a worst-case time complexity of $O(n + m)$, where n is the length of the text and m is the length of the pattern.

Excercise

1) Given the list of numbers

$$[29, 10, 14, 37, 13, 7, 21, 18],$$

demonstrate the process of applying the quick sort algorithm step by step. For each step, indicate:

- the pivot chosen,
- the partitioning of the list into elements less than, equal to, and greater than the pivot, and
- the recursive sorting of each sublist until the entire list is sorted.

2) Given a DNA pattern string, *e.g.,* "`AGATAG`", calculate its LPS array step by step using the algorithm in Code 6.9. Then, using the computed LPS array, demonstrate the process of applying the KMP algorithm to search for this pattern in a longer DNA sequence, *e.g.,* "`CAGATAGATAGT`". For each step, show the values of the sequence index i, pattern index j, and the actions taken.

MF 20006