🔟 Miscellaneous Topics

Naive Pattern Searching

- Naive pattern searching is a simple approach to finding a specific pattern (word or phrase) in a larger body of text.
- **Components:**
  - Text to scan
  - Pattern to search for
- The pattern is slid along the text one character at a time, checking for matches.

Process:

1. Slide the pattern along the text character by character.
2. Count the number of following characters that match the pattern.
3. If the count equals the pattern length, a match is found.

Performance:

- Worst-case scenario: O(nk) comparisons (n: length of text, k: length of pattern).
- Main cause of slow performance: Constant backtracking to the next character of the input text.
- Knuth–Morris–Pratt (KMP) algorithm improves performance by:
  - Tracking pattern prefixes.
  - Intelligently skipping through the text.
  - Preventing excessive backtracking.
  - Achieving runtime of O(n+k).

Advantages of KMP:

- More optimized than naive approach.
- Avoids redundant character comparisons.
- Efficiently integrates pattern and text iterations.
- Prevents unnecessary backtracking.

```
def pattern_search(text, pattern):
  print("Input Text:", text, "Input Pattern:", pattern)
  for index in range(len(text)):
    print("Text Index:", index)
    match_count = 0
    for char in range(len(pattern)):
      print("Pattern Index:", char)
      if pattern[char] == text[index + char]:
        match_count += 1
      else:
        break
    if match_count == len(pattern):
      print(pattern, "found at index", index)


text = "HAYHAYNEEDLEHAYHAYHAYNEEDLEHAYHAYHAYHAYNEEDLE"
pattern = "NEEDLE"
pattern_search(text, pattern)

# New inputs to test
text2 = "SOMEMORERANDOMWORDSTOpatternSEARCHTHROUGH"
pattern2 = "pattern"
text3 = "This   still      works with    spaces"
pattern3 = "works"
text4 = "72261545782461270420268217999255207204739 6"
pattern4 = "42"
pattern_search(text2, pattern2)
pattern_search(text3, pattern3)
pattern_search(text4, pattern4)
```

Brute Force Algorithms

A *brute force* algorithm is a straightforward method that solves a problem by going through every possible choice *one by one* until a solution is found. Instead of utilizing clever techniques, brute force algorithms rely on sheer computing power to solve problems.

Time Complexity:

- Typically denoted by Big O notation.
- Brute force algorithm's time complexity: **O(N)** (N: size of input data).
- Example: Flipping through N pages in a textbook.

Space Complexity:

- Varies from problem to problem.
- No general rule, determined case by case.

Disadvantages:

- Slow and inefficient for large or unorganized data.
- Cost increases rapidly with problem size.

Advantages:

- Easier to implement for programmers.
- Simplicity reduces chances of inconsistencies or bugs.
- Some brute force algorithms use less memory compared to optimized counterparts.
  Example: Bubble sort (slower but less memory) vs. Merge sort.

> 💡 **Remember that while brute force algorithms have their advantages, they are generally not suitable for real-world problems with large and complex datasets.**