CSCI-B 505 Applied Algorithms (3 cr.) № 5

Dr. H. Kurban
Computer Science
School of Informatics, Computing, and Engineering

Indiana University, Bloomington, IN, USA

November 4, 2021

Contents

Problem 1: Pattern-matching: The brute-force

Problem 1.1: The brute-force pattern-matching algorithm [10 pt.]

Describe a text $\mathcal{D}$ and a pattern $\mathcal{P}$ such that the brute-force pattern-matching algorithm runs in $\Omega(\mathrm{dp})$ time.The lengths of $\mathcal{D}$ and $\mathcal{P}$ are d and p, respectively.

Problem 1.2: Python's str class and pattern-matching [20 pt]

In this part, you are asked to modify three pattern matching programs given to you (See appendix). Run your modified programs for varying-length patterns and show your results.

The count method in Python's str class takes a text $\mathcal{D}$ and a pattern $\mathcal{P}$ and returns the maximum number of non-overlapping occurrences of a $\mathcal{P}$ within $\mathcal{D}$. As an example 'cdcdcdcdc'.count('cdc') returns 2.

1. Modify the brute-force pattern-matching to return non-overlapping occurrences of a $\mathcal{P}$ within $\mathcal{D}$.
2. Similar to the previous question (Problem 1.2.1), do the same on the Boyer-Moore program.
3. Similar to problem 1.2.1, modify the KMP program.

Problem 2: Experimental Analysis of Pattern-Matching Algorithms [20 pt.]

Perform an experimental analysis of pattern matching algorithms in terms of:

1. *Number of character comparison:* Perform an experimental analysis of the efficiency of the brute-force, the KMP and Boyer-Moore pattern matching algorithms for varying-length patterns.
2. *Relative speed comparison:* Perform an experimental comparison of the brute-force, KMP, and Boyer-Moore pattern-matching algorithms. Run each algorithm against large text documents using varying-length patterns and report the relative running times.

Problem 3: Matrix-chain Multiplication

The matrix-chain multiplication problem: Given a chain of $< \mathcal{D}_1, D_2, \ldots, D_n >$ of n matrices fully parenthesize the product $< \mathcal{D}_1 \cdot D_2 \cdots D_n >$ in a way so that the number of scalar multiplications is minimized. Each $\mathcal{D}_i$ has a $p_{i-1} \times p_i$ dimension and $i = 1, 2, \ldots, n$.

1. *The Brute-Force: [10 pt.]:* Implement a Python program to solve the matrix-chain multiplication problem by the brute force algorithm.
2. *Bottom-up Dynamic Programming [20 pt.]:* Implement a Python program to solve the matrix-chain multiplication problem using bottom-up dynamic programming approach.
3. *Dynamic Programming with Memoization [Extra Credit, 10 pt.]:* Implement a Python program to solve the matrix-chain multiplication problem using dynamic programming with memoization.

Problem 4: Longest Common Sub-sequence (LCS) Problem [20 pt.]

Implement a Python program to solve LCS problem using dynamic programming. Run your program to find the best sequence alignment between DNA strings. Show your results.

Longest Common Sub-sequence (LCS) problem: Given two character strings over some alphabet, find a longest string that is a sub-sequence of given two strings.

Data source: https://www.ncbi.nlm.nih.gov/genbank/

Directions

Please follow the syllabus guidelines in turning in your homework. While testing your programs, run them with a variety of inputs and discuss your findings. This homework is due Sunday, Nov 14, 2021 10:00pm. **OBSERVE THE TIME**. Absolutely no homework will be accepted after that time. All the work should be your own.

Appendix

Python program for the Brute-Force pattern-matching algorithm

```python
# Brute force
def find_brute(T, P):
    n, m = len(T), len(P)
    # every starting position
    for i in range(n-m+1):
        k = 0
        # conduct O(k) comparisons
        while k < m and T[i+k] == P[k]:
            k += 1
        if k == m:
            return i
    return -1
```

Python program for the Boyer-Moore pattern-matching algorithm

```python
# Boyer-Moore
def find_boyer_moore(T, P):
    n, m = len(T), len(P)
    if m == 0:
        return 0
    last = {}
    for k in range(m):
        last[P[k]] = k
    i = m-1
    k = m-1
    while i < n:
        # If match, decrease i,k
        if T[i] == P[k]:
            if k == 0:
                return i
            else:
                i -= 1
                k -= 1
        # Not match, reset the positions
        else:
            j = last.get(T[i], -1)
            i += m - min(k, j+1)
            k = m-1
    return -1
```

Python program for the Knuth-Morris-Pratt pattern-matching algorithm

```python
# KMP failure function
def compute_kmp_fail(P):
    m = len(P)
    fail = [0] * m
    j = 1
    k = 0
    while j < m:
        if P[j] == P[k]:
            fail[j] = k+1
            j += 1
            k += 1
        elif k > 0:
            k = fail[k-1]
        else:
            j += 1
    return fail
```

```python
# KMP
def find_kmp(T, P):
    n, m = len(T), len(P)
    if m == 0:
        return 0
    fail = compute_kmp_fail(P)
    # print(fail)
    j = 0
    k = 0
    while j < n:
        if T[j] == P[k]:
            if k == m-1:
                return j-m+1
            j += 1
            k += 1
        elif k > 0:
            k = fail[k-1]
        else:
            j += 1
    return -1
```