COMP36111: Advanced Algorithms I
Lecture 3: String Matching

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2017–18

Outline

The string matching problem

The Rabin-Karp algorithm

The Knuth-Morris-Pratt algorithm

- Suppose we are given some English text

  > *A blazing sun upon a fierce August day was no greater rarity in southern France then, than at any other time, before or since. Everything in Marseilles, and about Marseilles, had stared at the fervid sky, and been stared at in return, until a staring habit had become universal there.*

  and a search string, say "Marseilles".
- We would like to find all instances (or just the first instance) of the search string in the text.
- What's the best way?

- Suppose we are given some English text

  > *A blazing sun upon a fierce August day was no greater rarity in southern France then, than at any other time, before or since. Everything in Marseilles, and about Marseilles, had stared at the fervid sky, and been stared at in return, until a staring habit had become universal there.*

  and a search string, say "Marseilles".
- We would like to find all instances (or just the first instance) of the search string in the text.
- What's the best way?

- As usual, we start by modelling the data:
  - let Σ be a finite non-empty set (the alphabet);
  - let $T = T[0], \ldots, T[n-1]$ be a string length n over a fixed alphabet Σ;
  - let $P = P[0], \ldots, P[m-1]$ be a string length m over Σ;
- We formalize the notion of an occurrence of one string in another:
  - string P occurs with shift i in string T if $P[j] = T[i+j]$ for all j $(0 \leq i < |P|)$.
- The we have the following problem

MATCHING

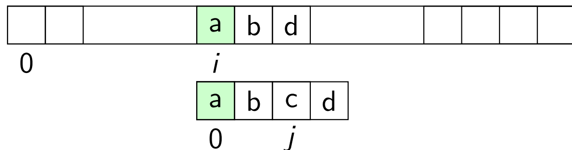
Given: strings T and P over some fixed alphabet Σ.

Return: the set of integers i such that P occurs in T with shift i.

- Here is a really stupid algorithm

```
begin naiveMatch(T,P)
    I ← ∅
    for i = 0 to |T| − |P|
        j ← 0
        until j = |P| or T[i + j] ≠ P[j]
            j++
        if j = |P|
            I = I ← {i}
    return I
end
```

- Graphically

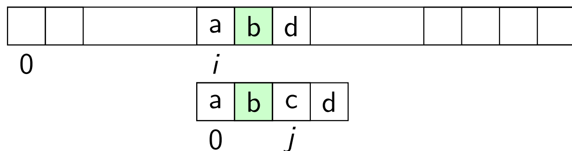


- Running time is $O(|T| \cdot |P|)$.

- Here is a really stupid algorithm

```
begin naiveMatch(T,P)
    I ← ∅
    for i = 0 to |T| − |P|
        j ← 0
        until j = |P| or T[i + j] ≠ P[j]
            j++
        if j = |P|
            I = I ← {i}
    return I
end
```

- Graphically

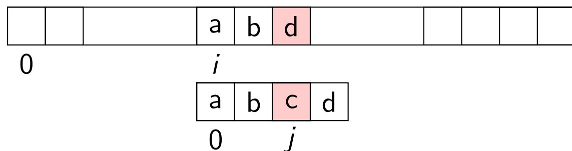


- Running time is $O(|T| \cdot |P|)$.

- Here is a really stupid algorithm

```
begin naiveMatch(T,P)
    I ← ∅
    for i = 0 to |T| − |P|
        j ← 0
        until j = |P| or T[i + j] ≠ P[j]
            j++
        if j = |P|
            I = I ← {i}
    return I
end
```

- Graphically



- Running time is $O(|T| \cdot |P|)$.

Outline

The string matching problem

The Rabin-Karp algorithm

The Knuth-Morris-Pratt algorithm

- Let $n = |T|$ and $m = |P|$.
- Think of the elements of Σ as digits in a base-b numeral, where $b = |\Sigma|$.
- Then P is the number $P[0] \cdot b^{m-1} + \cdots + P[m-1] \cdot b^0$.
- Similarly, $T[i, \cdots, i+m-1]$ is $T[i] \cdot b^{m-1} + \cdots + T[i+m-1] \cdot b^0$.
- To calculate $T[i+1, \cdots, i+m]$ from $T[i, \cdots, i+m-1]$, write:

$$T[i+1, \cdots, i+m] = (T[i, \cdots, i+m-1] - T[i] \cdot b^{m-1}) \cdot b + T[i+m].$$

- These numbers can get a bit large.
- However, we can work modulo q, for some constant q (usually a prime) such that bq is about the size of a computer word.
- Of course, we have

$$T[i+1,\cdots,i+m] = (T[i,\cdots,i+m-1] - T[i]\cdot b^{m-1})\cdot b + T[i+m] \pmod{q}.$$

- If $T[i,\cdots,i+m-1] \neq P \bmod q$, then we know we do not have a match at shift i.
- If $T[i,\cdots,i+m-1] = P \bmod q$, then we simply check explicitly that $T[i,\cdots,i+m-1] = P$.

- The worst-case running time of this algorithm is also $O(|T| \cdot |P|)$.
- On average, however, it works must better:
  - A rough estimate of the probability of a spurious match is $1/q$, since this is the probability that a random number will take a given value modulo q. (Well, that's actually nonsense, but never mind.)
  - A reasonable estimate of the number of matches is $O(1)$, since patterns are basically rare.
- This leads to an expected performance of about $O(n + m + m(n/q))$
- Thus, expected running time will be about $O(n + m)$, since presumably $q > m$.

- Here is the algorithm

```
begin Rabin-Karp(T,P,q,b)
    I ← ∅
    m ← |P|
    t ← T[0] · b^(m−1) + ··· + T[m − 1] · b^0  mod q
    p ← P[0] · b^(m−1) + ··· + P[m − 1] · b^0  mod q
    i → 0
    while i ≤ |T| − m
        if p = t
            j ← 0
            while P[j] = T[i + j] and j < |P|
                j++
            if j = |P|
                I ← I ∪ {i}
        t ← (t − T[i] · b^(m−1)) · b + T[i + m]  mod q
        i++
    return I
end
```

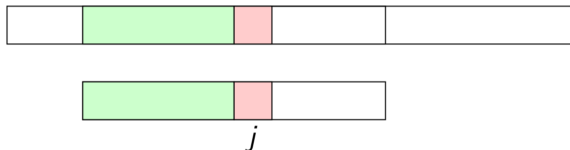
Outline

The string matching problem

The Rabin-Karp algorithm

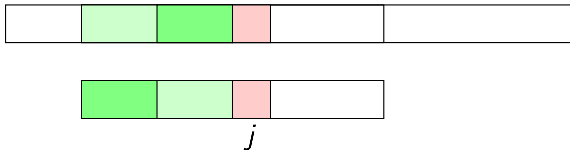
The Knuth-Morris-Pratt algorithm

- The basic idea of this algorithm is as follows. Suppose we have a mismatch at some pattern position j.



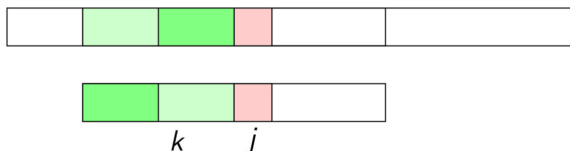
- Call $P[0, \ldots, j-1]$ the good prefix.
- Now let us look at the longest prefix of the good prefix which is also a proper suffix of the good prefix.

- The basic idea of this algorithm is as follows. Suppose we have a mismatch at some pattern position j.



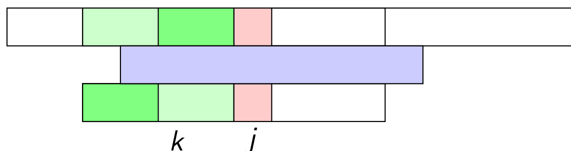
- Call $P[0, \ldots, j-1]$ the good prefix.
- Now let us look at the longest prefix of the good prefix which is also a proper suffix of the good prefix.

- Suppose that the length of the longest such proper prefix is k.
- It should be clear that there cannot be any matches involving shifts of less than k (for then, k would not be maximal).



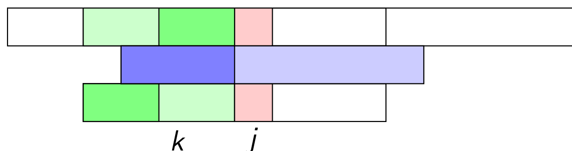
- So we can shift the pattern up by k.
- Denote the length k of the longest prefix of $P[0, \ldots, j-1]$ that is also a proper suffix of $P[0, \ldots, j-1]$ by $\pi(j)$, for all j $(1 \leq j \leq |P|)$, and set $\pi(0) = 0$.

- Suppose that the length of the longest such proper prefix is k.
- It should be clear that there cannot be any matches involving shifts of less than k (for then, k would not be maximal).



- So we can shift the pattern up by k.
- Denote the length k of the longest prefix of $P[0,\ldots,j-1]$ that is also a proper suffix of $P[0,\ldots,j-1]$ by $\pi(j)$, for all j $(1 \le j \le |P|)$, and set $\pi(0) = 0$.

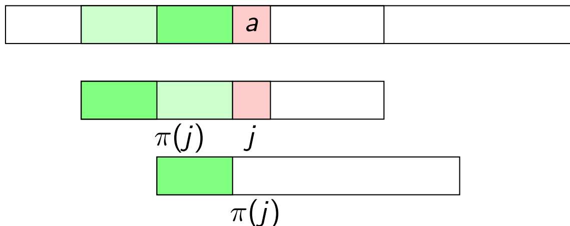
- Suppose that the length of the longest such proper prefix is k.
- It should be clear that there cannot be any matches involving shifts of less than k (for then, k would not be maximal).



- So we can shift the pattern up by k.
- Denote the length k of the longest prefix of $P[0, \ldots, j-1]$ that is also a proper suffix of $P[0, \ldots, j-1]$ by $\pi(j)$, for all j $(1 \leq j \leq |P|)$, and set $\pi(0) = 0$.

- Don't do it! Don't shift the pattern up by k!
- Instead, think what would happen to j if you did. The length of the good prefix would change:

$$j \leftarrow \pi(j).$$



- Now repeat the process: either we get a match and make progress (incrementing j), or we get a mismatch and execute $j \leftarrow \pi(j)$.

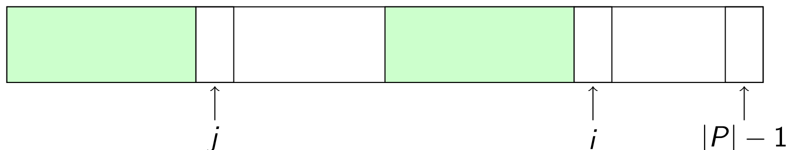
- Here is the algorithm.

```
begin Knuth-Morris-Pratt(T,P)
    I ← ∅
    compute the function π
    i ← 0, j ← 0
    while i < |T|
        if P[j] = T[i]
            if j = |P| − 1
                I ← I ∪ {i − |P| + 1}
                i++, j ← π[|P|]
            i++, j++
        else if j > 0
            j ← π[j]
        else
            i++
    return I
end
```

- The following algorithm computes π.

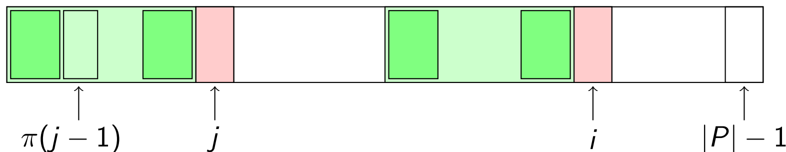
```
begin compute-π(P)
    i ← 1
    j ← 0
    f(0) ← 0
    while i < |P|
        if P[i] = P[j]
            f(i) ← j + 1
            i++, j++
        else if j > 0
            j ← f(j − 1)
        else
            f(i) ← 0
            i++
    return f
end
```

- Here is the general idea behind compute-$\pi(P)$



- j points to the square after the longest confirmed good-prefix for $P[0] \ldots P[i-1]$.
- if we get a match, we can simply increase both i and j, and update $\pi(i)$.

- Here is the general idea behind `compute-`π`(`P`)`



- j points to the square after the longest confirmed good-prefix for $P[0]\ldots P[i-1]$.
- if we get a match, we can simply increase both i and j, and update $\pi(i)$.
- if we get a mis-match, we do not have to start at the beginning of the pattern; we can use $\pi(j-1)$ to jump over characters we know will match.

- The running time of `Knuth-Morris-Pratt(`T`,`P`)` (ignoring the construction of π is $O(|T|)$.
  - Letting $k = i - j$, each iteration of the loop either increments i or increases k by at least 1, and neither quantity reduces.
  - Hence, the while loop can execute at most $2|T|$ times.
- Note that π is one-off: it depends only P and not on T, so its computation is not critical.
- In fact, however, the running time of `compute-`π`(`P`)` is $O(|P|)$.
- Hence, overall running time is $O(|P| + |T|)$.