

STRING MATCHING

Chapter 23-23.4 of GT

DRAFT

DISCLAIMER: *Report an error / typo to the author so that he resolves it.
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis*

Strings

References

1. D.E. Knuth, J.H. Morris Jr., V. R. Pratt. "Fast pattern matching in strings", SIAM J Computing, Vol 6 (2), June 1977.
2. R. S. Boyer, J. S. Moore. "A fast string searching algorithm". Dec 29, 1975. SRI, Menlo Park, CA. Xerox PARC, Palo Alto, CA.
3. R. M. Karp, M. O. Rabin, Michael O. "Efficient randomized pattern-matching algorithms". IBM Journal of Research and Development. 31 (2) 1987.
4. Algorithm Design: Foundations, analysis, and internet examples. M. T. Goodrich and R. Tamassia. Wiley, 2001, ISBN 0-471-38365-1.

DRAFT

Strings

Definitions

Definition 1 (Strings). A string is a sequence of zero or more characters. Each character belongs to a set of characters known as the alphabet and denoted by Σ .

Definition 2 (Length of a string). The *length of a string* S is denoted by $|S|$, and it is the number of its characters.

Definition 3 (Empty or null string). A string of length 0 is called the *null string* and denoted by ϵ .

Definition 4 (Representation of a string). We represent string S by an array of length equal to the length of the string.

We then talk about the character at offset i to be $S[i]$. We try to avoid expressions the i -th character of $S[i]$ since depending on indexing this might be $S[i]$ (Pascal indexing) or $S[i - 1]$ (C/C++ or Java indexing).

Strings

More definitions

Definition 5 (Substrings). A *substring* of a string S is a string $S[i..j]$, where $j \geq i$ that includes all characters of S starting at offset i and ending (inclusive of) offset j . A substring $S[i..j]$, with $i > j$ is considered to be the null string ϵ .

Definition 6 (Concatenation of strings). For two strings x and y the concatenation of x and y is string xy , where y follows x . The length of string xy is $|x| + |y|$,

Definition 7 (String prefix). A string w is a prefix of x if $x = wy$.

Definition 8 (String suffix). A string w is a suffix of x if $x = yw$.

String x is a prefix of itself and also a suffix of itself. By default the null string ϵ is a prefix and suffix of every string.

Definition 9 (Equality of P to a T starting at i). Later on for two strings P and T of length $|P| = m$ and $|T| = n$ with $n > m$ we are going to determine whether $P = T(i)$. $T(i)$ or T_i would represent the substring $T[i..i + m - 1]$ i.e. the substring T starting at offset i of length the length of P . If the length is less than m then $T(i, k)$ would be used to indicate a length k match i.e. $P[0..k - 1]$ matches $T[i..i + k - 1]$.

String Matching

Naive Algorithm

The **string matching** problem is defined as follows.

Problem 1 (String Matching).

Input. A string T of length n and a pattern P of length m ; the characters of T and P are drawn from a finite alphabet Σ .

Output. The first index i such that P matches the m characters of text T starting at position i that is, $T[i..i+m-1]$ is equal to $P[0..m-1]$. (We also say that P matches T starting at i .)

Theorem 1. *Algorithm NaiveMatching solves the string matching problem in time $T(n, m) = O(nm)$.*

Proof.

```
NaiveMatching(P[0..m-1], T[0..n-1])
1. for (i=0; i<=n-m; i++)           // Check T[i...] against P[...]
2.   if P[0..m-1] == T[i..i+m-1]
3.     return(i);
4. return(-1);
```

Since this is the first 'algorithm', let us give a C or C++ interpretation of it.

```
NaiveMatching(P[0..m-1], T[0..n-1])
1. for (i=0; i<=n-m; i++) {          // Check T[i...] against P[...]
2.   j=0;                             // From start j=0 of P
3.   while ((j<m) && (T[i+j] == P[j])) { // T[i+j] : P[j]
4.     j++;
5.   }                                // to the end j=m-1
6.   if (j==m)                         // or j==m equivalently
7.     return(i);                     //T[i..i+m-1] matches P[0..m-1]
8. }
9. return(-1);                        //-1 means no match.
```

Naive Matching Properties

Namely, in the NaiveMatching algorithm (interpretation) the pattern is overlayed with the text for each starting position of the text (indexed by i). For each such overlay we compare the pattern to the text starting at position i . This is performed through lines 2-4. If a match is found $j == m$ is true and the algorithm returns with the index of the match (starting at i). Otherwise i is incremented by 1 and the algorithm continues. When the for..loop exits no match exists and no match (-1) is returned.

The running time of the algorithm is $T(n, m) = O(nm)$ as the loop of line 1 is executed $O(n)$ times and the while loop of lines 3-4 $O(m)$ times. ■

DRAFT

String Matching

Rabin-Karp fingerprinting algorithm

Method 1 (Rabin Karp fingerprinting). The Rabin-Karp algorithm is another approach to the matching problem. It tries to establish first a non-match of a fingerprint of P to a finger print of $T(i)$ based on the premise that fingerprint matches are faster than string matches. It performs quite well in practice and can also be generalized for two dimensional (i.e. image) matching. Although its (worst case) running time is $O(nm)$, its *average case running time* is better (linear to n).

Note 1 (Fast vs Quick). Do not forget that in computer science some practical algorithms include those that exhibit good average or expected running time performance, even if they have suboptimal (worst-case) running time. Hashing and quick-sort are some interesting examples: they are Quick (in practice) but not asymptotically optimal (i.e. not Fast)!

Definition 10 (Comparing fingerprints). The Rabin-Karp algorithm uses the notion of finger-printing. Instead of comparing $T[i..i + m - 1]$ to $P[0..m - 1]$ i.e. T_i to P , it compares a fingerprint of $T[i..i + m - 1]$ to a fingerprint of $P[0..m - 1]$.

Note 2 (A fingerprint is a hash). Fingerprint is a better name for the misnomer “hash”.

Rabin-Karp also employs elementary number-theoretic notions and is referred to as a finger-printing algorithm. In order to generate a fingerprint of T or P , we employ some “hashing” mechanism. So it is also referred to as a “hashing” algorithm.

Rabin-Karp fingerprinting algorithm

Fingerprints

Definition 11 (Fingerprint matching). In Rabin-Karp matching (sub)string $T_i = T[i..i + m - 1]$ to $P = P[0..m - 1]$ is realized by comparing the fingerprints t_i and p of T_i and P respectively.

The characters of the pattern P and text T belong to some alphabet Σ . We view each character of Σ as a digit in radix- d notation where $d = |\Sigma|$. For example a character from $\Sigma = \{a, b, \dots, z\}$ can be viewed as a digit of radix-26. Therefore the text T and pattern P can be viewed as n -digit and m -digit numbers respectively radix of the appropriate base (26 or otherwise).

Definition 12 (Fingerprint p of order m of string P). Let alphabet Σ be such that $d = |\Sigma|$. For a string P of alphabet Σ of length m we denote by p its fingerprint's (denary) value. Then p is given by the following expression.

$$p = P[0] \cdot d^{m-1} + \dots + P[m-2] \cdot d + P[m-1]. \quad (1)$$

The fingerprint p is the evaluation of a polynomial $P(x)$ of degree $m-1$ (or degree bound m) at $x = d$, where the coefficient of degree i is $P[m-1-i]$.

Definition 13 (Fingerprint t_i of order m of string $T(i) = T_i$). For a text $T[..]$ we denote by t_i the fingerprint of the length m substring starting at i , i.e. of the string $T(i) = T_i = T[i..i + m - 1]$.

$$t_i = T[i] \cdot d^{m-1} + \dots + T[i+m-2] \cdot d + T[i+m-1].$$

Rabin-Karp fingerprinting algorithm

Fingerprint matching

Definition 14 (String matching becomes fingerprint matching). String matching between P of length m and T of length n , where $n \geq m$ is reduced to deciding whether there exists an index i such that $p = t_i$, where p is the fingerprint of order m of P and t_i is the fingerprint of order m of $T(i) = T_i$.

Lemma 1. *Computing t_i and p is a polynomial evaluation.*

Corollary 1. In order to compute p we need to evaluate a polynomial of degree $m - 1$ and degree bound m . Horner's rule achieves this in $O(m)$ additions and multiplications.

Rabin-Karp Algorithm

Approximation 1

Problem 2 (Rabin-Karp Infinite precision).

Input. A text $T[.]$ of length n and a pattern $P[.]$ of length m ; the characters of T and P are drawn from a finite alphabet Σ . The fingerprints of T are the fingerprints T_i of order m i.e. $t_0, \dots, t_{n-m-1}$ and the fingerprint of P of order m is p . (In other words t_i is the fingerprint of T_i i.e. $T[i..i+m-1]$.)

Output. The first index i such that t_i is equal to p .

Corollary 2. All t_i , $i = 0, \dots, n - m - 1$ can be computed in time $O(m + n)$.

Proof. Computing one t_i indeed takes time $\Theta(m)$ since it involves a polynomial evaluation.

This is indeed the case for t_0 that is computed from $T[0..m-1]$ in $\Theta(m)$ time.

We can incrementally compute t_{i+1} from t_i though in only constant time $\Theta(1)$ through the recurrence

$$t_{i+1} = d(t_i - d^{m-1}T[i]) + T[i+m] \quad (2)$$

Note that t_{i+1} is the fingerprint of $T[i+1..i+m]$ and t_i that of $T[i..i+m-1]$. In the formula above, the subtractive term $d^{m-1}T[i]$ removes the contribution of $T[i]$ in t_i from t_{i+1} , since T_{i+1} has no contribution related to $T[i]$, the multiplication by d “shifts left” the $T[i+1..i+m-1]$ contribution and the last term adds to that the contribution of the character $T[i+m]$. The computational cost of computing t_{i+1} from t_i is constant. d^{m-1} is precomputed during the calculation of t_0 that takes $\Theta(m)$ time and its cost $O(\lg m)$ is included in $\Theta(m)$. All other calculations include one addition, one subtraction and two multiplications that is a constant number of operations. Note that t_i can be arbitrarily large depending on m and the alphabet size d . On finite precision architectures overflow problems might occur. ■

Rabin-Karp Algorithm

Approximation 1 continued

Example 1. Let $m = 3$ and $n = 6$ for $d = 10$. The alphabet is the set of digits 0 through 9.

```
0 1 2 3 4 5      m=3,n=6,d=10
3 1 2 4 8 9      t0 = T[0..2] = 312
                  t1 = T[1..3] = 10 * (t0 - 10**2 * T[0]) + T[3] = 124
                  t2 = T[2..4] = 10 * (t1 - 10**2 * T[1]) + T[4] = 248
                  t3 = T[3..5] = 10 * (t2 - 10**2 * T[2]) + T[5] = 489
```

Theorem 2. *RabinKarp1 has running time $O(n + m)$. It works assuming one word can accommodate a p or a t_i .*

```
RabinKarp1(P[0..m-1], T[0..n-1])
0. Compute fingerprint p from P;
1. for (i=0; i<=n-m; i++)
2.   compute fingerprint ti from ti-1;
3.   if p == ti
4.     if P[0..m-1] == T[i..i+m-1]
5.       return(i);
6. return(-1);
```

Proof. Algorithm RabinKarp1 in the worst case is still $O(nm)$. More often than not though, the check of P and T_i of line 4 would return a match rather than a spurious match! So on the average it would take $O(n + m)$ time. ■

Rabin-Karp algorithm

Approximation 1 issues

There is a variety of issues related to an implementation of Version 1 of Rabin-Karp. We raise them through a sequence of remarks.

Remark 1 (Numeric overflows). Equation 2 can cause overflow problems for large m and d and finite precision integer arithmetic.

One can eliminate this problem if we perform all computations modulo R , where R is a very large prime number. In practice one may choose R large enough such that dR or $dR + R$ can be represented within the built-in integer data type allowed by a programming language's execution model.

But then spurious matches might appear.

Definition 15 (Spurious match). An operation might indicate a match but there is no actual match.

This refers to cases where $P == T_i$ is sufficient for $p = t_i$ but $p == t_i$ is necessary for $P == T_i$. Thus $p == t_i$ might indicate a match or not (and it is then a spurious match) and then a further comparison is needed to conclude whether $P == T_i$ or not.

Example 2. Let the hash of a date of birth is the date of a month. If two persons were born July 4, 1776 and July 4, 1976, the hashes for each one of their dates of birth would be a 4. A spurious match does not obviously translate to them having the same date of birth.

Rabin-Karp algorithm

Introduction

We rewrite equation 2 for the recursive formulation of t_{i+1} indicated belows as equation 3; the new fingerprint generated would be denoted by u_i . Fingerprint u_i incorporates modulo R calculations rather than infinite precision arithmetic calculations involved to derive t_i , the (arbitrary precision) fingerprint of $T_i = T(i) = T[i..i + m - 1]$. Note that a rewrite of equation 1 for p is also needed along the same lines and shown below as equation 5.

$$t_{i+1} = d(t_i - d^{m-1}T[i]) + T[i + m] \quad (3)$$

Method 2 (Finite $(\text{mod } R)$ fingerprint u_i of T_i). We compute a finite precision fingerprint modulo- R u_{i+1} for T_{i+1} from u_i of T_i as follows.

$$u_{i+1} = (d(u_i - (d^{m-1} \text{ mod } R)T[i]) + T[i + m]) \text{ mod } R. \quad (4)$$

Moreover we call $\mathcal{P}$ the modulo- R derivation of the fingerprint p of P as indicated below.

$$\mathcal{P} = (P[0] \cdot (d^{m-1} \text{ mod } R) + \dots + P[m-2] \cdot d + P[m-1]) \text{ mod } R. \quad (5)$$

Rabin-Karp algorithm

Spurious matches

The problem of using modulo arithmetic is that finding a match between u_i and $\mathcal{P}$ does not necessarily translate into a match between t_i and p . A match for $\mathcal{P}$ vs u_i might be a spurious match and p and t_i might be different resulting in P not matching T_i . Therefore a match between $\mathcal{P}$ and u_i must continue with a comparison between P and $T(i)$ to establish either a match or a spurious match!

Definition 16 (Spurious match in RabinKarp). For a pattern P of length m and text T of length n , $n > m$ and alphabet Σ of length (cardinality) d , let p, t_i be fingerprints of P and $T(i) = T_i$ respectively and for some prime number R let $\mathcal{P}, u_i$ be the modulo- R fingerprints of P and $T(i) = T_i$ as established by Equation 3 and Equation 5.

(a) Then we have a spurious match (spurious hit, or false positive) at i if

$$u_i = \mathcal{P} \wedge t_i \neq p \quad (\text{implying } T_i \neq P)$$

(b) a real match, and

$$u_i = \mathcal{P} \wedge t_i = p \quad (\text{implying } T_i = P)$$

(c) a miss

$$u_i \neq \mathcal{P} \quad (\text{implying } T_i \neq P)$$

Given that we do not calculate t_i and p but only u_i and $\mathcal{P}$ a spurious match is resolved by comparing $T[i..i + m - 1] = P[0..m - 1]$ to check whether P matches T_i and thus $t_i = p$ is indeed the case or it is a spurious match.

Rabin-Karp algorithm

Spurious match example

Example 3. We use $d = 1$ even though $\Sigma = \{0, 1, \dots, 9\}$ and thus $|\Sigma| = 10!$ Redo this example for $d = 10!$

```
P: 3141 , d=1 !      Thus p=3+1+4+1=9. All ops mod R=11.  
   012345678901234567890 <-----these are indexes (indices)  
T: 115231423141313042210 <-----this is the text  
-----  
t0=9 ; spurious match (1152 vs 3141)  
t1=0;  
t2=0  
-----  
t8=9 and a HIT (3141 vs 3141)
```

DRAFT

Rabin-Karp algorithm

The algorithm

Theorem 3. *RabinKarp has average case running time $O(n + m)$ and running time $T(n, m) = O(nm)$. It works assuming one word can accommodate a p or a t_i .*

```
RabinKarp (P[0..m-1], T[0..n-1], d, R)
1. h = d**(m-1) mod R;
2. calP = 0;
3. u = 0; u[0] = 0;
4. for(j=0; j<m; j++) {
5.   calP=(d*calP+P[j]) mod R;
6.   u=(d*u + T[j]) mod R;
7. }
8. u[0]=u;
9. for(i=0; i<=n-m; i++){
10.  if (calP == u[i]) {
11.    k=0; // runs through the pattern
12.    while ((k<m) && (T[i+k] == P[k])) {
13.      k++;
14.    }
15.    if (k==m) // a match has been found
16.      return(i); //T[i..i+m-1] is P[0..m-1]
17.  }
18.  if (i < n-m)
19.    u[i+1] = (d(u[i]-R* T[i])+T[i+m]) mod R;
```

Proof. Though the running time performance of RabinKarp is dependent on its worst case performance and the former can be $O(nm)$, its average case performance is significantly lower $O(n + m)$ as long as $R > m$. ■

Theorem 4. For a binary alphabet $\Sigma = \{0, 1\}$ and thus $p < 2^m$ and $t_i < 2^m$, and thus a $u_{i+1} < R$ a i_{i+1} update is given by

$$u_{i+1} = (2(u_i - (2^{m-1} \bmod R)T[i]) + T[i + m]) \bmod R, \quad (6)$$

and for a choice of $R = A \cdot m \lg(A \cdot m)$ for some $A > 1$, we have the following.

$$Pr(\text{spurious match at } i) = Pr(\tau_i = \mathcal{P}) = Pr(t_i \bmod R = p \bmod R) \leq 2/A.$$

Proof. What is the probability that $p \bmod R = t_i \bmod R$? This is equivalent to $(p - t_i) \equiv 0 \bmod R$ or equivalently that R divides $p - t_i$.

Integer R is a prime number. Both p and t_i , by the way they are defined and given that $|\Sigma| = 2$ are $p < 2^m$ and $t_i < 2^m$ which both imply that $|p - t_i| \leq 2^m$.

The number of distinct prime numbers that divide $p - t_i$, with $|p - t_i| \leq 2^m$, is at most m . This is because if $z_1, \dots, z_s$ are these distinct prime divisors of the difference $z_i \geq 2$ and thus $2^s \leq z_1 \cdot \dots \cdot z_s \leq 2^m$, which implies that $s \leq m$.

The number of prime numbers between 1 and R however, by the Prime Number Theorem is about $R/\ln R$ which is

$$(Am \ln Am)/(\ln Am + \ln \ln Am) \leq Am/2$$

The probability that R is one of these s numbers $z_1, \dots, z_s$ is at most $s/(Am/2) \leq 2/A$.

We have thus established.

$$Pr(\text{spurious match at } i) = Pr(p \bmod R = t_i \bmod R) \leq 2/A.$$

■

Rabin-Karp Algorithm

Probability of spurious matches

Theorem 5. *For the conditions of theorem 4*

$$Pr(\text{spurious match in text}) = Pr(\cup_{i=0}^{n-1} \text{spurious match at } i) \leq 2n/A$$

Proof. Then the probability that there is a spurious match somewhere in the text is

$$\begin{aligned} Pr(\text{spurious match in text}) &= Pr(\cup_{i=0}^{n-1} \text{spurious match at } i) \\ &= \sum_{i=0}^{n-1} Pr(\text{spurious match at } i) \\ &\leq n \times Pr(\text{spurious match at } i) \\ &\leq n \times 2/A = 2n/A \end{aligned}$$

■

Corollary 3. For the conditions of Corollary 4, if furthermore $A = 2n^2$ we have

$$Pr(\text{spurious match in text}) \leq 1/n.$$

Choosing larger values for A further decreases this probability.

Rabin-Karp Algorithm

Expected running time

Lemma 2 (RabinKarp expected running time). *The expected running time of RabinKarp is $T(n, m) = O(n + m)$.*

Proof. If there is a spurious match the running time may go as high as $O(nm)$. In all other cases it remains linear $O(n + m)$. The probability of a spurious match anywhere in the text is $1/n$ by way of Corollary 3, a rare event. The expected running time of Rabin-Karp thus is $T(n, m) = O(nm)(1/n) + O(n + m)(1 - 1/n) = O(n + m)$. ■

DRAFT

Rabin-Karp Algorithm

Example

```
R=911 ==> 10**3 mod 911 = 89, 10**2 mod 911 = 100, 10**1 mod 911 =10, etc
P:      8  2  8  4   fing(P) = 8 *89 + 2 *100 + 8 *10 +4 *1 % R = 85
        0  1  2  3  4  5  6  7  8  9 10 11 12
T:      2  7  1  8  2  8  1  8  2  8  4  5  9
ALL OPS BELOW % R i.e. % 911
i=0      fing(2718)= 2*89+7*100+1*10+8= 896
i=1      fing(7182)= 10*(896-89*2)+2 = 805
i=2      fing(1828) = 10 * ( 805 - 89*7)+8 = 6
i=3      fing(8281)= 10*(6 - 89*1)+1 = 82
i=4      fing(2818)= 10*(82-89*8)+8 = 85  spurious hit
          2818 does not match P even if fing(2818)==fing(P)
i=5      fing(8182) = 10*(85-89*2)+2= 894
i=6      fing(1828) = 10*(894-89*8)+8 = 917
i=7      fing(8284)= 10*(917-89*1)+4= 85 hit!
          8284 matches P
```

DRAFT

The Boyer-Moore algorithm

Introduction

If the pattern P is relatively long (say more than 4-5 characters) and the alphabet is reasonably large (i.e. strings are non-binary strings), then an algorithm due to R.S. Boyer and J.S. Moore is perhaps the most efficient (and practical) string matching algorithm.

The Boyer-Moore algorithm differs from the brute-force naive algorithm in that it checks the pattern against the text not left-to-right but right-to-left.

If a mismatch is found the action performed on the pattern differs from the approach of the naive algorithm that shifts the pattern one position to the right. In the Boyer-Moore algorithm, if the mismatched character $c = T[i]$ of the text that was compared to $P[j]$ does not appear in the pattern P , then the pattern is shifted right j positions past $T[i]$, and thus in the following iteration, a comparison of $P[.]$ against $T[i + 1..i + m]$ will be performed.

This is because c does not appear in P and therefore no character in the portion of the pattern $P[..j - 1]$ will ever match $T[i]$ and thus the portion of the text $T[..i]$ can be skipped.

If however, c appears in $P[..j - 1]$, we shift P to the right by an appropriate number of positions. Let t be the maximum index in $..j - 1$ that a c appears. Then we shift P to the right $j - t$ positions, so that $c = P[t]$ matches $T[i] = c$. If c does not appear in $P[..j - 1]$ but it does appear in $P[j + 1..]$, then and only then do we shift right one position only and Boyer-Moore behaves like the naive algorithm).

The actions performed due to the mismatched c describe a method that is known as the bad-character heuristic or the character-jump. How do we compute the last occurrence of a character c in P ? We use function `BadCharacter`.

The Boyer-Moore algorithm

BadCharacter function

Proposition 3. For a pattern P of length m whose characters are drawn from some alphabet Σ , let A be the set of nominal values into which the characters of Σ are encoded. Therefore characters are represented with numbers $0, \dots, |\Sigma| - 1$. The *BadCharacter* function computes array *last* of length m . Element *last*[i] is the last (rightmost) position in P occupied by a character whose nominal value is i .

Proof. Array *last* computes the rightmost occurrence of a character of the alphabet Σ denoted as A below.

```
BadCharacter(P,m,A)          // A is the alphabet
1. for(i=0;i<|A|;i++)
2.   last[i]= Left-1;         //Left=0, Right=m for   P[0..m-1] (C/C++)and
3.   for(j=Left;j<Right;j++){ //Left=1, Right=m+1 for P[1..m]   (Pascal)
4.     last[P[j]]=j;
5.   return(last[]);
```

The running time of the *BadCharacter* algorithm is $O(m + |\Sigma|)$ and thus can be performed quite efficiently. ■

The Boyer-Moore algorithm

Theorem 6 (Boyer-Moore). *The Boyer-Moore algorithm has running time $T(n, m) = O(nm)$ but its average case running time is a lower $T(n, m) = O(n + m)$.*

Proof. The Boyer-Moore algorithm is described below.

```
BoyerMoore(P[0..m-1], T[0..n-1], Alphabet)
0. last[] = BadCharacter(P, m, Alphabet);
1. i=m-1; // runs through the text
2. j=m-1; // runs through the pattern
3. while (i <= n-1) {
4.   if (P[j]==T[i]) {
5.     if (j==0)
6.       return("Match at i");
7.     else {
8.       i--;
9.       j--;
10.    }
11.  }
12. else {      //mismatch P[j] differs from c=T[i]
13.   i=i+m- MIN(j, 1+last[T[i]]);
14.   j=m-1;    //restart from right end of P
15. }
16.}
17.return(-1); // No match found
```

Line 12 is encountered when $c = T[i] \neq P[j]$. There are three cases to be considered: (a) c is nowhere in P , (b) c appears in P but rightmost occurrence is in $P[0..j-1]$, and (c) c appears in P but rightmost occurrence is in $P[j+1..m-1]$. Line 12 leads to lines 13 and 14. Line 14 resets j to $m-1$ (rightmost character of P); line 13 resets i to $i = i + m - \min(j, 1 + \text{last}[c])$.

The Boyer-Moore algorithm

Proof of correctness

We described the three cases in more detail

Case (a). In this case $P[0..m-1]$ is to determine a match with $T[i-j..i-j+m-1]$. When $c = T[i]$ is encountered and mismatches $P[j]$ i.e. $c = T[i] \neq P[j]$ we have established a match between $T[i+1..i-j+m-1]$ with $P[j+1..m-1]$. Since c appears nowhere in P we plan to reset the matching so that next time $P[0..m-1]$ would be tested against $T[i+1..i+m]$. Thus j needs to be reset to $m-1$, and i needs to be reset to $i+m$. Line 14 correctly resets j to $m-1$. Line 13 resets i to $i = i+m - \min(j, 1 + \text{last}[c])$; since c is not in P , then $\text{last}[c] = -1$, therefore $\min(j, 1 + \text{last}[c])$ evaluates to $\min(j, 1 + (-1)) = 0$ and thus Line 13 resets i to $i+m$ indeed. Case resolved.

Case (b) In this case $P[0..m-1]$ is to determine a match with $T[i-j..i-j+m-1]$. When $c = T[i]$ is encountered and mismatches $P[j]$ i.e. $c = T[i] \neq P[j]$ we have established a match between $T[i+1..i-j+m-1]$ with $P[j+1..m-1]$ as before. Now c appears in $P[0..j-1]$ and thus $t = \text{Last}[c] < j$ i.e. $1 + \text{Last}[c] = 1 + t \leq j$ and thus $i = i+m - \min(j, 1 + \text{last}[c]) = i+m-1-t$.

We plan to reset the matching by shifting P to the right and aligning $c = T[i]$ with $P[t] = c$. In other words after the realignment we will match $P[0..m-1]$ to $T[i-t..i+m-1-t]$. Thus j needs to be reset to $m-1$, and i needs to be reset to $i+m-1-t$. Line 14 correctly resets j to $m-1$. Line 13 resets i to $i = i+m - \min(j, 1 + \text{last}[c]) = i+m-1-t$ as observed earlier. Case resolved.

Case (c). In this case $P[0..m-1]$ is to determine a match with $T[i-j..i-j+m-1]$. When $c = T[i]$ is encountered and mismatches $P[j]$ i.e. $c = T[i] \neq P[j]$ we have established a match between $T[i+1..i-j+m-1]$ with $P[j+1..m-1]$ as before. Now c appears in $P[j+1..m-1]$ and thus $t = \text{Last}[c] \geq j$ i.e. $1 + \text{Last}[c] = 1 + t \geq j+1 > j$ and thus $i = i+m - \min(j, 1 + \text{last}[c]) = i+m-j$.

We plan to reset the matching by shifting P to the right one position and aligning $c = T[i]$ with $P[j-1]$. In other words after the realignment we will match $P[0..j-1..m-1]$ to $T[i-j+1..i+m-j]$. Thus j needs to be reset to $m-1$, and i needs to be reset to $i+m-j$. Line 14 correctly resets j to $m-1$. Line 13 resets i to $i = i+m - \min(j, 1 + \text{last}[c]) = i+m-j$ as observed earlier. Case resolved. ■

Line 13 and 14 are in text graphics on the following page.

The Boyer-Moore algorithm

Pictorial case analysis

Case 1: Initial State <----- Right to Left---|

T[i-j]	c = T[i]	T[i+1]	T[i-j+m-1]
	!=	=	=
P[0]	P[j]	P[j+1]	P[m-1]

Observation: $c=T[i]$ is nowhere in $P[0..m-1]$ $\Rightarrow t=-1$

ACTION: Shift P to the right j positions

NEW ALIGNMENT RESET $i = i+m-i+m-\text{MIN}(j,t+1)$, $j = m-1$

T[i+1]	...	T[i+m]
?	?	?
P[0]	...	P[m-1]

Case 2: Initial State <----- Right to Left---|

T[i-j]	c = T[i]	T[i+1]	T[i-j+m-1]
	!=	=	=
P[0]	P[j]	P[j+1]	P[m-1]

Observation: $c=T[i]$ is in $P[0..j-1]$ and $t=\text{last}[c] \leq j-1 \Rightarrow 1+t \leq j$

ACTION: Shift P to the right j-t positions

NEW ALIGNMENT RESET $i = i+m-1-t = i+m-\text{MIN}(j,1+t)$, $j = m-1$

T[i-t]	...	T[i]	...	T[i+m-1-t]
?	?	=	?	?
P[0]	...	P[t]	...	P[m-1]

Case 3: Initial State <----- Right to Left---|

T[i-j]	c = T[i]	T[i+1]	T[i-j+m-1]
	!=	=	=
P[0]	P[j]	P[j+1]	P[m-1]

Observation: $c=T[i]$ is in $P[j+1..m-1]$ and $t=\text{last}[c] \geq j+1 \Rightarrow 1+t > j$

ACTION: Shift P to the right ONE position

NEW ALIGNMENT RESET $i = i+m-j = i+m-\text{MIN}(j,1+t)$, $j = m-1$

T[i-j+1]	...	T[i]	...	T[i+m-j]
?	?	=	?	?
P[0]	...	P[j-1]	...	P[m-1]

The Boyer-Moore algorithm

Examples

Example 4. An example that shows the benefits of using Boyer-Moore: $n = 43$, $m = 11$.

```

T= ABRACADABRZZABRACDABRAZZZZZZZZZABRACADABRA
      1
ABRACADABRA      765432
      ABRACADABRA
            8
      ABRACADABRA
            9
      ABRACADABRA
            43210
      ABRACADABRA
            54321098765
      ABRACADABRA
            01234567890
P= ABRACADABRA      c = A B C D R Z
      last(c)=10 8 4 6 9 -1

```

Example 5. An example that slows down BoyerMoore: $n = 13$, $p = 5$.

```
T=  AAAAAAAAABAAAA          n=
54321
BAAAA
09876
BAAAA
54321
BAAAA
09876
BAAAA
1
BAAAA
65432
BAAAA

P=  BAAAA                      c = A  B
                                last(c)= 4  0
```

The Boyer-Moore algorithm

Good and Bad

Note 3 (Best case). The best case for Boyer-Moore is an all "a" text and an all "b" pattern. Boyer-Moore will only compare n/m characters, before failure is declared.

Question 1 (Worst case). What is a worst case text and pattern scenario in Boyer-Moore? In the worst-case Boyer-Moore is an $O(nm)$ time algorithm.

One of the example of the previous page might provide you ideas about the worst case.

Question 2. Is there a better algorithm? Can we improve the performance of Boyer-Moore?

Answer. The answer starts on the next page. ■

The Knuth-Morris-Pratt algorithm

Introduction

The Knuth-Morris-Pratt (KMP) algorithm is a linear time algorithm in the worst-case. It asymptotically outperforms Boyer-Moore (BM). Boyer-Moore (BM) exhibits the worst-case performance of the naive algorithm on certain inputs (e.g. binary strings of short length) that KMP avoids. KMP also requires some precomputation, similar to the one of BM. KMP works left-to-right as opposed to BM that works right to left.

The KMP precomputation involves the calculation of a failure function f , that is utilized when the comparison of $P[j]$ to $c = T[i]$ fails i.e. $P[j] \neq c = T[i]$, and the pattern needs to be shifted to the right and matching needs to restart.

Method 3 (Action for KMP). For some i, j we have

$$P[0..j-1] = T[i-j..i-1], \quad P[j] \neq T[i].$$

We need to determine how many characters is P to be shifted right. (For the naive algorithm the default is one shift right.)

Let us call P_k the prefix of P consisting of the first k characters of $P[0..k-1]$ and P_{-k} the suffix of P consisting of the last k characters of P .

The question that arises then is by using the lessons learned from the BM algorithm how to shift P to the right by more than one position. If we shift right P by say $d > 1$ positions we have the following cases illustrated by the examples of the next page.

The Knuth-Morris-Pratt algorithm

Overview examples

Consider the following case.

Example 6.

	01234567890123456
	i
T:	ABCDABCDABCDYBCDY
	0 j m-1
P:	ABCDABCDYBCDY

Discussion of example 6. A mismatch occurs at $i = 8$ and $j = 8$ between the text and the pattern. $P[0..7]$ matches $T[0..7]$ but $A = c = T[8] \neq Y = P[8]$. BM would advise to shift the pattern P 4 positions to the right. This is because the rightmost $c = A$ is at position $t = 4$ of P and the mismatch occurs at $i = 8$ and $j = 8$ of the text. So BM proposes to move P to the right by $j - t = 8 - 4 = 4$ positions, and this is the right thing to do! ■

Consider now the following example, a variation of the previous one.

Example 7.

	01234567890123456
	i
T:	ABCAABCDABCDYBCDY
	0 j m-1
P:	ABCAABCDYBCDY

Discussion of example 6. A mismatch occurs at $i = 8$ and $j = 8$ between the text and the pattern. $P[0..7]$ matches $T[0..7]$ but $A = c = T[8] \neq Y = P[8]$. BM would advise to shift the pattern P 4 positions to the right. This is because the rightmost $c = A$ is at position $t = 4$ of P and the mismatch occurs at $i = 8$ and $j = 8$ of the text. So BM proposes to move P to the right by $j - t = 8 - 4 = 4$ positions.

The Knuth-Morris-Pratt algorithm

Overview example continued

In that case BM (be reminded that it does a right to left matching starting at $j = m - 1$ and comparing $P[m - 1]$ to $T[16]$ first and pursues a red-herring sequence of matches until it realizes a mismatch at $j = 3$ and $i = 7$.

```

                                01234567890123456
                                i=16
T:                               ABCAABCDABCDYBCDY
                                0           j=m-1
P:                               ABCAABCDYBCDY

```

So in the former case BM makes the correct decision, in the latter case it wastes 10 character comparisons. Why ? because we know that at $j = 3$ of the pattern a D is a better match for the D of $i = 7$ of the text. An A at position $j = 3$ of P would never match the D of position $i = 7$ of T . ■

Proposition 4. *On a mismatch between $P[j]$ and $c = T[i]$ KMP would shift right the pattern P a number of positions $j - f[j - 1]$ where $f[m]$ is the length of the longest prefix of P that is a suffix of $P[1 \dots m]$.*

Proof. With reference to example 6, KMP observes a mismatch at $j = 8$ and $i = 8$ for $P[8] = Y \neq A = c = T[8]$. Then KMP tries to determine the longest prefix $P[0..t]$ that matches for some $t \geq 0$, the pattern itself $P[...j-1]$ to the last matched character of it which was at position $j - 1 = 7$. It then realizes that $P[0..t] = P[0..3] = ABCD$ matches $P[..7] = P[..j-1] = ABCD$. It then shifts the pattern to the right $j - t$ positions. But t in this case is the length of the longest prefix of P that is a suffix of the prefix of P ending at position $j - 1$. In other words the $j - f[j - 1]$ of this proposition 4. So the actions of BM and KMP are identical for different reasons.

With reference to the other example 7 though, KMP shifts the pattern $j - f[j - 1] = 8 - 0 = 8$ positions to the right. There is no prefix of P that is a suffix of $P[..7]$ that warrants a lesser shift to the right. KMP makes the right decision. ■

The Knuth-Morris-Pratt algorithm

Introduction concluded

Example 8. We provide an example used in the KMP paper (1977).

```

0 1 2 3 4 5 6 7 8 9
P= a b c a b c a c a b
f=[0 0 0 1 2 3 4 0 1 2 ]
P: abcabcacab
01234567890123456789012345
T: babcbabcbabcaabcbabcbacabcb
abcbacab
x shift 1
abcbacab
***x shift 3
abcbacab
x shift 1
abcbacab
*****x shift 3
abcbacab
----x shift 3
abcbacab
-x shift 1
abcbacab
*****x shift 3
abcbacab
----***** match at i-m+1=24-10+1 = 15

```

The Knuth-Morris-Pratt algorithm

The algorithm

Theorem 7. *Function $KMP(P,T)$ requires time $O(n + m)$. The calculation involved in line f for the computation of the failure function f also takes $O(m)$ time.*

Proof. The code is from [4] (Tamassia and Goodrich).

```
KMP(P,T)
1.  f[] = LongestPrefix(P);
2.  i = 0;
3.  j = 0;
4.  while (i <= n-1) {
5.    if P[j] == T[i] {
6.      if (j == m-1) {
7.        return("match at i-m+1");
8.      }
9.      i++; // P[j] == T[i] advance i
10.     j++; //      and j
11.   }
12.   else { // mismatch at j but P[0..j-1] matches T[..i-1]
13.     if (j > 0) // try to advance pattern a little
14.       j = f[j-1];
15.     else // no luck; advance text one character to the right.
16.       i++;
17.   }
18. }
19. return("no match");
```

How do we compute array f ? Element $f[j]$ is the length of the longest prefix $P[0..j]$ of P that is a suffix of $P[1..j]$. For the boundary case $f[0] = 0$.

For example let $P = ababababca$. Then $f = [0, 0, 1, 2, 3, 4, 5, 6, 0, 1]$.

Knuth-Morris-Pratt algorithm

Time analysis

The analysis of the Knuth-Morris-Pratt algorithm consists of examining the three possible cases of matches/mismatches and also examine how the difference $i - j$ behaves. Let us call $D = i - j$, i.e. the difference in shift between the text and the pattern. Naturally, $D \leq n$.

The three cases that we have are as follows.

- Case 1. If $T[i] == P[j]$ then, i increases by one and j increases by one and D remains the same $D = (i + 1) - (j + 1) = i - j$.
- Case 2. If $T[i] \neq P[j]$ and $j > 0$, then i remains the same, j however changes from j into $f[j - 1]$ a decrease by $j - f[j - 1]$. Then D changes from $i - j$ into $i - f[j - 1]$. Since $f[j - 1] < j$, this means that D is increased by $j - f[j - 1]$.
- Case 3. If $T[i] \neq P[j]$ and $j == 0$, then i increases by 1 and j remains the same. This leads to $D = (i + 1) - j$, i.e. an increase by 1 in the value of D .

As a conclusion,

- i increases by at least one in cases 1 and 3.
- D increases by at least one in cases 2 and 3.

Therefore at every iteration of the loop, either i or D increases by at least one. The maximum value for either i or D is n . Therefore the loop of KMP is executed at most $2n$ times.

In order to complete the analysis we need to examine the running time of line 1. By a similar argument it is $O(2m) = O(m)$. Therefore KMP has running time $O(n + m)$. ■

We find $f[]$ by using KMP but we match the pattern against a text which is the pattern P ! The array f can thus be computed as follows.

Knuth-Morris-Pratt Failure Function Computation

Theorem 8. *The calculation involved in line 1 of KMP for the computation of the failure function f also takes $O(m)$ time. This calculation is realized by function LONGESTPREFIX.*

Proof.

```
LongestPrefix (P);
1. f[0]=0; // boundary case
2. i=1; // i-j >=1
3. j=0;
4. while (i <= m-1) {
5.   if P[j] == P[i] { //P[0..j] matches P[i-j..i]
6.     f[i]=j+1;      // j+1 matched characters
7.     i++; //P[j]==T[i] advance i and j
8.     j++; // maintains i-j >=1
9.   }
10.  else { // mismatch at j but P[0..j-1] matches P[i-j..i-1]
11.    if (j>0)
12.      j=f[j-1];
13.    else { // no luck; no match
14.      f[i]=0;
15.      i++;
15a.  }
16.  }
17. }
18. return(f[]);
```

Trivial. Follows from the previous page. ■

Notice Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 1.

Let the pattern P be and the text T be respectively

ABRACADABRA

ABRACAVABRACALABRAABRACADABRA

- (a) Illustrate the working of the Boyer-Moore algorithm. What is the total number of comparisons (text vs pattern) performed?
- (b) Illustrate the working of the Knuth-Morris-Pratt algorithm. What is the $f[]$ array. What is the total number of comparisons (text vs pattern) performed?
- (c) Which of the two algorithms is faster? Explain.

Proof. Practice! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 2.

In Rabin-Karp, let the pattern be $p = 35$ and all arithmetic operations be performed modulo q where $q = 10$. Are there any spurious (false) matches if we search for the pattern p in text t shown below? What is the number of character comparisons (not fingerprint!) performed to find all occurrences of the pattern in the text?

271828182845904523536

Explain.

Proof. Practice! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 3.

Give an example of a pattern and a text for which Boyer-Moore exhibits worst case $O(nm)$ performance (n is text length, and m is pattern length). Explain.

Proof. Practice! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem Set

Problem 4.

- (a) Give an input T, P for which NaiveMatching exhibits worst-case performance over a binary alphabet of letters a, b .
- (b) Give an input T, P for which Boyer-Moore exhibits worst-case performance over a binary alphabet of letters a, b .
- (c) How many comparisons does BM perform if $P = a^m$ and $T = (a^{m-1}b)^{n/m}$. Assume m/n and a^m means m consecutive occurrences of a .

Proof. Practice! ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page