INTRODUCTION

This week we cover the following subjects

11.1 STRING SEARCHING

- Like sorting, searching for strings within another string is an interesting exercise in algorithms because searching is an important activity that is done very often and because many innovative algorithms have been developed for it.

- We start off with a *base string* and a *search string* whose location we want to find within the base string. If we cannot find the search string then we return a -1 location.

11.1.1 Brute Force Approach

- The simplest way of string searching is the brute force approach. In this method, we go through the base string one character at a time and see if the characters from that point are the same as the one in the search string.

base string

t	a	d	h	e	x	o	n	u	…/ /…	l	z	e	y	n

search string

g	e	c	y	x

t	a	d	h	e	x	o	n	u	…/ /…	l	z	e	y	n
	g	e	c	y	x									

t	a	d	h	e	x	o	n	u	…/ /…	l	z	e	y	n

g	e	c	y	x

- The upshot of this is that for each character in the base string we may have to test every character in the search string to see if a match is found or not. If the length of the base string is *M* and the length of the base search string is *N* then the order of searching is *O*(*MN*). This can lead to very slow search times under certain circumstances.

```
sally% cat testbrute.cpp

/************************************************\
   Test program for demonstrating string
   searching algorithms
\************************************************/

#include <sys/time.h>
#include <time.h>
#include <stdlib.h>

#include <iostream>
#include <string>

using namespace std;

double difUtime(struct timeval *first,
                struct timeval *second);
int randNum();
int search(const string &baseString,
           const string &searchString);
```

```
int search(const string &baseString,
           const string &searchString)
{
   // look for searchString in baseString.
   // Return the starting location in
   // baseString where the first instance of
   // searchString is found
   // If not found return -1

   int blen = baseString.size();
   int slen = searchString.size();
   int i, j;

   for (i=0; i<blen-slen+1; i++) {
      for (j=0; j<slen; j++) {
         if (baseString[i+j] != searchString[j])
             break;
      }
      if (j == slen) return i;
   }
   return -1;
}

int main()
{
   const int BASESIZE = 1000000;
   const int SEARCHSIZE = 1000;

   string baseString = "", searchString = "";
   int i;

   // data for calculating timing
   struct timeval first, second;
   double usecs;

   try {

      /******************************************\
         Fill baseString with random letters in
         range 'a' to 'z'
         Fill searchString with start of
         baseString
      \******************************************/
```

```
for (i=0; i<BASESIZE; i++)
   baseString += ('a' + randNum() % 26);
for (i=0; i<SEARCHSIZE; i++)
   searchString += baseString[i];

cout << "Size of base string = " <<
   baseString.size() << "\n";
cout << "Size of search string = " <<
   searchString.size() << "\n";

/****************************************\
   find the location of searchString in
   basetring
\****************************************/

cout << "\nBase string filled with random
    letters\n";
cout << "Search string first " <<
    SEARCHSIZE << " letters in base\n";
gettimeofday(&first, NULL);
i = search(baseString, searchString);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "Location of found string = " <<
   i << "\n";
cout << "Time to search = " << usecs <<
   " seconds\n";

/****************************************\
   fill baseString with mostly a's but with
   a few b's
   refill searchString with last SEARCHSIZE
   letters in baseString
\****************************************/

for (i=0; i<BASESIZE; i++) {
   if (randNum() % 100 == 0)
      baseString[i] = 'b';
   else
      baseString[i] = 'a';
}
for (i=0; i<SEARCHSIZE; i++)
   searchString[i] =
      baseString[BASESIZE-SEARCHSIZE+i];
```

```
/***************************************\
   find the location of searchString in
   baseString
\***************************************/

cout << "\nBase string filled mostly with
   a's but some b's\n";
cout << "Search string last " << SEARCHSIZE
   << " letters in base\n";
gettimeofday(&first, NULL);
i = search(baseString, searchString);
gettimeofday(&second, NULL);
usecs = difUtime(&first, &second);
cout << "Location of found string = " <<
   i << "\n";
cout << "Time to search = " << usecs <<
   " seconds\n";

/***************************************\
   fill baseString with all a's except
   last with a b
   refill searchString
\***************************************/

for (i=0; i<BASESIZE-1; i++)
   baseString[i] = 'a';
baseString[BASESIZE-1] = 'b';
for (i=0; i<SEARCHSIZE; i++)
   searchString[i] =
      baseString[BASESIZE-SEARCHSIZE+i];

/***************************************\
   find the location of searchString in
   baseString
\***************************************/

cout << "\nBase string filled with a's
   except last is b\n";
cout << "Search string last " << SEARCHSIZE
   << " letters in base\n";
gettimeofday(&first, NULL);
i = search(baseString, searchString);
gettimeofday(&second, NULL);
```

```
      usecs = difUtime(&first, &second);
      cout << "Location of found string = " <<
         i << "\n";
      cout << "Time to search = " << usecs <<
         " seconds\n";   } catch(...) {
      cout <<
        "\nERROR - undefined Exception thrown\n";
      exit(1);
   }

   return 0;
}
```

```
sally% testsearch
Size of base string = 1000000
Size of search string = 1000

Base string filled with random letters
Search string first 1000 letters in base
Location of found string = 0
Time to search = 0.000407 seconds

Base string filled mostly with a's but some b's
Search string last 1000 letters in base
Location of found string = 999000
Time to search = 11.6288 seconds

Base string filled with a's except last is b
Search string last 1000 letters in base
Location of found string = 999000
Time to search = 245.63 seconds
```

Analysis:

- If the size of the search string is M and the size of the string being searched N then the search time of the brute-force approach is $O(NM)$.

11.1.2 Boyer-Moore Algorithm

Before we get into the main part of the algorithm, we create a lookup table showing the last position in the search string for all the characters it contains.

search string

0	1	2	3	4
t	r	e	a	t

ascii values

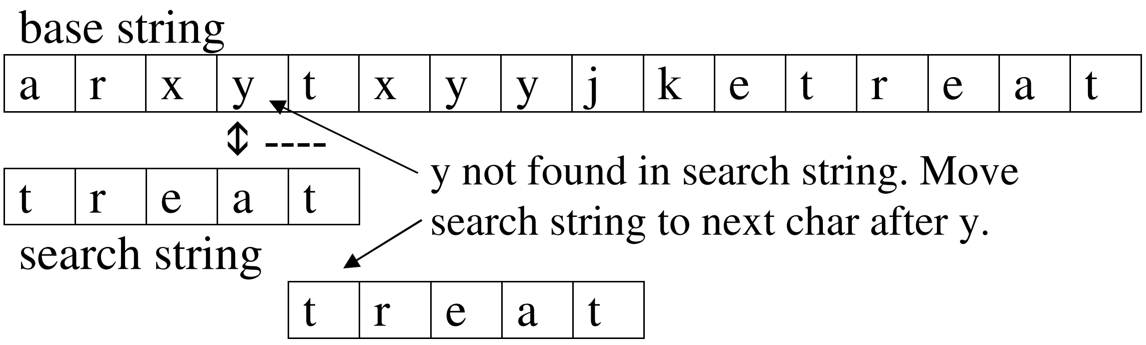
't' = 116 'r' = 114 'e' = 101 'a' = 97

last position lookup

0	…	…	…	97	…	101	…	…	114	115	116	…	…	…	127
-1	…	…	…	3	…	2	…	…	1	-1	4	…	…	…	-1

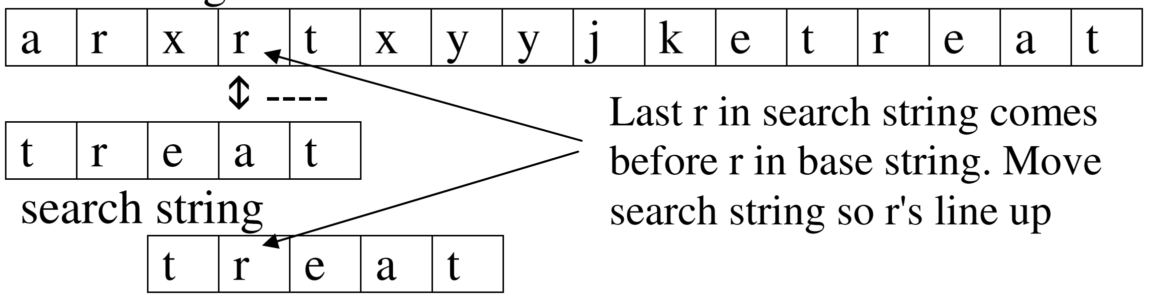
When we start the search, we compare the search string to the first characters in the base string being searched. To do the check, we start at the end of the search string, not the front. 3 possibilities can occur

Possibility 1:


base string
a r x y t x y y j k e t r e a t
⇕ ----
t r e a t
search string
y not found in search string. Move search string to next char after y.
t r e a t

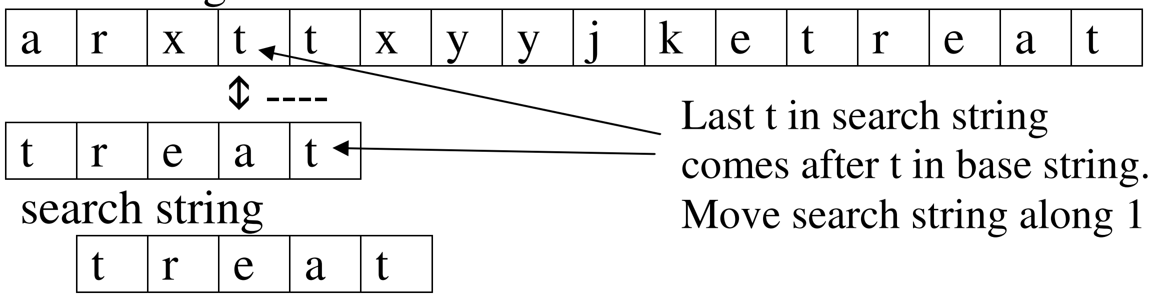
Possibility 2:

base string

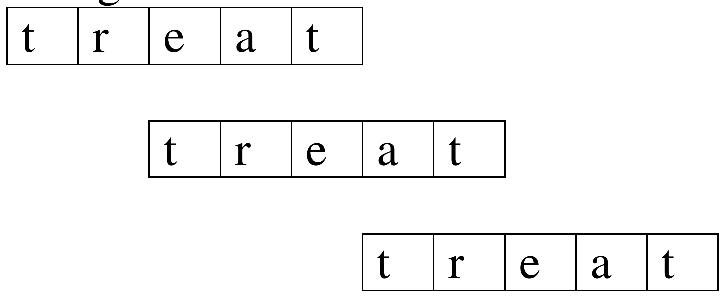

a r x r t x y y j k e t r e a t
⇕ ----
t r e a t
search string
t r e a t
Last r in search string comes before r in base string. Move search string so r's line up

Possibility 3:

base string


a r x t t x y y j k e t r e a t
⇕ ----
t r e a t
search string
t r e a t
Last t in search string comes after t in base string. Move search string along 1

Continuing the search we get



The following code implements the algorithm

```
#include <vector>

vector<int> buildLast(const string &searchString)
{
   int i;
   const int NUMASCII = 128;
   vector<int> last(NUMASCII);
   for (i=0; i<NUMASCII; i++) last[i] = -1;

   for (i=0; i<(int) searchString.size(); i++) {
      last[searchString[i]] = i;
   }
   return last;
}

int search(const string &baseString,
           const string &searchString)
{
   // look for searchString in baseString
   // using boyer-moore algorithm.
   // Return the starting location in baseString
   // where the first instance of searchString is
   // found.
   // If not found return -1

   int blen = baseString.size();
   int slen = searchString.size();

   // searchString bigger that baseString?
   if (slen > blen) return -1;

   vector<int> last = buildLast(searchString);
   int p, i = slen-1, j = slen-1;

   do {
      if (baseString[i] == searchString[j]) {
         if (j == 0) return i;
         else {
            i--;
            j--;
         }
      } else {
         p = baseString[i];
         i = i + slen - min(j, 1 + last[p]);
         j = slen - 1;
      }
   } while (i < blen);

   return -1;
}
```

```
sally% testsearch

Base string filled with random letters
Search string first 1000 letters in base
Location of found string = 0
Time to search = 0.001182 seconds

Base string filled mostly with a's but some b's
Search string last 1000 letters in base
Location of found string = 999000
Time to search = 5.6643 seconds

Base string filled with a's except last is b
Search string last 1000 letters in base
Location of found string = 999000
Time to search = 1.0577 seconds
```

Analysis:

- If the size of the search string is *M* and the size of the string being searched *N* then the search time of the Boyer-Moore algorithm is $O(N+M)$.

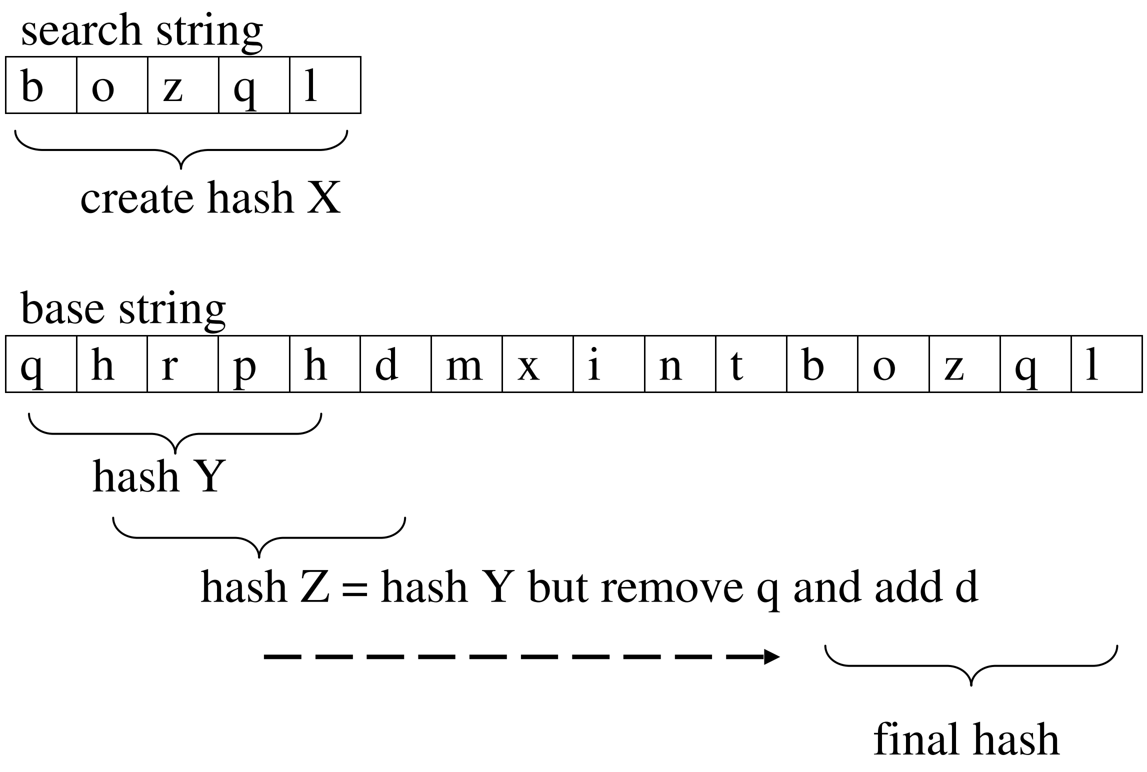
11.1.3 Rabin-Karp

- Instead of checking character by character using the brute force approach, we can use concepts from hashing.

- To do this we create a hash value of the search string. Assuming the search string has a length of *M*, we then check every *M* set of characters in the base string to find a matching hash value.

- If the two hash's match an explicit check must be made of the string to ensure it is the correct one. Even though the hash value will have a large range it is possible for two different strings to have the same hash value.

- The hash value of each substring in the base can be calculated very quickly if we use a hashing algorithm that allows us to remove the character we are discarding and add the new one.


search string
b o z q l
create hash X
base string
q h r p h d m x i n t b o z q l
hash Y
hash Z = hash Y but remove q and add d
final hash

The hash algorithm is of the form
$h_M = (s[0])d^{M-1} + s[1]d^{M-2} + \cdots + s[M-2]d^1 + s[M-1]d^0) \bmod Q$
where d is some constant, Q is a large prime number, M is the size of the search string and $s[x]$ is the integer contents of the search string at the location x.

For example, if d = 32 and Q = 33554393 then hashing the previous search string "bozql" would be
$h_M = ('b'\times 32^4 + 'o'\times 32^3 + 'z'\times 32^2 + 'q'\times 32^1 + 'l'\times 32^0) \bmod 33554393$

A large search string would create a value far too large to fit in an integer variable before we could do the modulus arithmetic. Fortunately, modulus arithmetic has some very nice properties that allow us to get around this. In particular.

$$h_M = \left(h_{M-1} \times d + s[M-1]\right) \bmod Q$$

To remove the first component we need to subtract that part. That is

$$h_M - d^{M-1}s[0]$$

Modulus arithmetic allows us to reduce this to

$$h_M - \left(d^{M-1)}\right) \bmod Q \times s[0]$$

The problem with this is that $d^{M-1} \bmod Q \times s[0]$ maybe much larger than h_M, thus creating a negative number. To get around this we add a multiple of Q which will allow the modulus to be unaffected. Thus

$$h_M + Q \times s[0] - \left(d^{M-1}\right) \bmod Q \times s[0]$$

This can be further simplified to

$$h_M + \left(Q - \left(d^{M-1}\right) \bmod Q\right) \times s[0]$$

Putting this into code

```
int search(const string &baseString,
           const string &searchString)
{
   // look for searchString in baseString
   // using rabin-karp algorithm.
   // Return the starting location in baseString
   // where the first instance of searchString is
   // found
   // If not found return -1

   const int q = 33554393;
   const int d = 32;
   unsigned int i, j, hashs=0, hashb=0, ds=1;
   unsigned int slen = searchString.size();
   unsigned int blen = baseString.size();

   // calculate q - ((d ^ (slen-1)) mod q)
   for (i=1; i<slen; i++) ds = (ds*d) % q;
   ds = q-ds;

   // create hash of searchString and
   // first slen chars in baseString
   for (i=0; i<slen; i++) {
      hashb = (hashb*d + baseString[i]) % q;
      hashs = (hashs*d + searchString[i]) % q;
   }

   // search for matching hash
   for (i=0; i <= blen - slen; i++) {
      if (hashs == hashb) {
         // confirm search string has been found
         for (j=0; j<slen; j++) {
            if (baseString[i+j] !=
                 searchString[j]) break;
         }
         if (j == slen) return i;
      }

      // remove left most char
      hashb = (hashb + baseString[i]*ds) % q;
      // add new right char
      hashb = (hashb*d + baseString[i+slen]) % q;
   }
   return -1;
}
```

```
sally% testsearch

Size of base string = 1000000
Size of search string = 1000

Base string filled with random letters
Search string first 1000 letters in base
Location of found string = 0
Time to search = 0.00122 seconds
```

```
Base string filled mostly with a's but some b's
Search string last 1000 letters in base
Location of found string = 999000
Time to search = 0.873396 seconds

Base string filled with a's except last is b
Search string last 1000 letters in base
Location of found string = 999000
Time to search = 1.16284 seconds
```

Analysis:

- If the size of the search string is M and the size of the string being searched N then the search time of the Rabin-Karp algorithm is $O((N-M)M)/Q$. Given that Q is very large, under most circumstances the search time reduces to $O(N-M)$.

11.1.4 Summary of Results

	Brute force	Boyer-Moore	Rabin-Karp
test 1	0.000407	0.001182	0.00122
test 2	11.6288	5.6643	0.873396
test 3	245.63	1.0577	1.16284

Size of base string = 1,000,000 letters

Test 1
Base string filled with random letters 'a' to 'z'
Search string first 1000 letters in base

Test 2
Base string filled mostly with a's but some b's
Search string last 1000 letters in base

Test 3
Base string filled with a's except last is b
Search string last 1000 letters in base