# Leet Code Training Day 54 String KMP

In general, the string pattern search algorithm is a O(N^2) time complexity. This is because we must map the pattern in every possible position on the original text. Such algorithms is not an issue to search a very short pattern, but will be the problem if the target pattern is very long.

To avoid duplicated work, there is an algorithm called KMP (Knuth Morris Pratt) Pattern Searching. This algorithm helps the scenario where the search substring has repeated pattern. Please look at the following example:

1) txt[] = “AAAAAAAAAAAAAAAAAB”, pat[] = “AAAAB”

2) txt[] = “ABABABCABABABCABABABC”, pat[] = “ABABAC” (not a worst case, but a bad case for Naive)

The KMP matching algorithm uses degenerating property (pattern having the same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst-case complexity to O(n+m).

The basic idea behind KMP’s algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will match anyway.

Matching Overview

txt = “AAAAABAAABA”

pat = “AAAA”

We compare first window of txt with pat

txt = “AAAAABAAABA”

pat = “AAAA” [Initial position]

We find a match. This is same as Naive String Matching.

In the next step, we compare next window of txt with pat.

txt = “AAAAABAAABA”

pat = “AAAA” [Pattern shifted one position]

This is where KMP does optimization over Naive. In this second window, we only compare the fourth A of pattern with fourth character of current window of text to decide whether current window matches or not. Since we know the first three characters will anyway match, we skipped matching first three characters.

**Preprocessing Overview:**

KMP algorithm preprocesses pat[] and constructs an auxiliary lps[] of size m (same as the size of the pattern) which is used to skip characters while matching.

Name lps indicates the longest proper prefix which is also a suffix. A proper prefix is a prefix with a whole string not allowed. For example, prefixes of “ABC” are “”, “A”, “AB” and “ABC”. Proper prefixes are “”, “A” and “AB”. Suffixes of the string are “”, “C”, “BC”, and “ABC”.

We search for lps in subpatterns. More clearly we focus on sub-strings of patterns that are both prefix and suffix.

For each sub-pattern pat[0..i] where i = 0 to m-1, lps[i] stores the length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i].

lps[i] = the longest proper prefix of pat[0..i] which is also a suffix of pat[0..i].

Note: lps[i] could also be defined as the longest prefix which is also a proper suffix. We need to use it properly in one place to make sure that the whole substring is not considered.

Examples of lps[] construction:

For the pattern “AAAA”, lps[] is [0, 1, 2, 3]

For the pattern “ABCDE”, lps[] is [0, 0, 0, 0, 0]

For the pattern “AABAACAABAA”, lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

For the pattern “AAACAAAAAC”, lps[] is [0, 1, 2, 0, 1, 2, 3, 3, 3, 4]

For the pattern “AAABAAA”, lps[] is [0, 1, 2, 0, 1, 2, 3]

**Preprocessing Algorithm:**

In the preprocessing part, We calculate values in lps[]. To do that, we keep track of the length of the longest prefix suffix value (we use len variable for this purpose) for the previous index.

1. We initialize lps[0] and len as 0.
2. If pat[len] and pat[i] match, we increment len by 1 and assign the incremented value to lps[i].
3. If pat[i] and pat[len] do not match and len is not 0, we update len to lps[len-1]

See computeLPSArray() in the below code for details

Illustration of preprocessing (or construction of lps[]):

1. pat[] = “AAACAAAA” => len = 0, i = 0: lps[0] is always 0, we move to i = 1 => len = 0, i = 1:
2. Since pat[len] and pat[i] match, do len++, store it in lps[i] and do i++. Set len = 1, lps[1] = 1, i = 2 => len = 1, i = 2:
3. Since pat[len] and pat[i] match, do len++, store it in lps[i] and do i++. Set len = 2, lps[2] = 2, i = 3 => len = 2, i = 3:
4. Since pat[len] and pat[i] do not match, and len > 0, Set len = lps[len-1] = lps[1] = 1 => len = 1, i = 3:
5. Since pat[len] and pat[i] do not match and len > 0, len = lps[len-1] = lps[0] = 0 => len = 0, i = 3:
6. Since pat[len] and pat[i] do not match and len = 0, Set lps[3] = 0 and i = 4 => len = 0, i = 4:
7. Since pat[len] and pat[i] match, do len++, Store it in lps[i] and do i++. Set len = 1, lps[4] = 1, i = 5 => len = 1, i = 5:
8. Since pat[len] and pat[i] match, do len++, Store it in lps[i] and do i++. Set len = 2, lps[5] = 2, i = 6 => len = 2, i = 6:
9. Since pat[len] and pat[i] match, do len++, Store it in lps[i] and do i++. len = 3, lps[6] = 3, i = 7 => len = 3, i = 7:
10. Since pat[len] and pat[i] do not match and len > 0, Set len = lps[len-1] = lps[2] = 2 => len = 2, i = 7:
11. Since pat[len] and pat[i] match, do len++, Store it in lps[i] and do i++. len = 3, lps[7] = 3, i = 8
12. We stop here as we have constructed the whole lps[].

The code for KMP in C++ is as follows:

/// <summary>

/// generate kmp array

/// </summary>

vector<int> LeetCodeString::kmp(string& s)

{

vector<int> dp(s.size());

for (size\_t i = 1; i < s.size(); i++)

{

int pos = dp[i - 1];

while (pos != 0 && s[i] != s[pos])

{

pos = dp[pos - 1];

}

dp[i] = pos + (s[i] == s[pos] ? 1 : 0);

}

return dp;

}

## [459. Repeated Substring Pattern](https://leetcode.com/problems/repeated-substring-pattern/)

Easy

Given a string s, check if it can be constructed by taking a substring of it and appending multiple copies of the substring together.

**Example 1:**

**Input:** s = "abab"

**Output:** true

**Explanation:** It is the substring "ab" twice.

**Example 2:**

**Input:** s = "aba"

**Output:** false

**Example 3:**

**Input:** s = "abcabcabcabc"

**Output:** true

**Explanation:** It is the substring "abc" four times or the substring "abcabc" twice.

**Constraints:**

* 1 <= s.length <= 104
* s consists of lowercase English letters.

### Analysis:

We can use KMP to find the longest match. If this is divisible by the length of the whole string, then we have a repeatable pattern.

/// <summary>

/// Leet Code 459. Repeated Substring Pattern

///

/// Easy

///

/// Given a string s, check if it can be constructed by taking a substring

/// of it and appending multiple copies of the substring together.

///

/// Example 1:

/// Input: s = "abab"

/// Output: true

/// Explanation: It is the substring "ab" twice.

///

/// Example 2:

/// Input: s = "aba"

/// Output: false

///

/// Example 3:

/// Input: s = "abcabcabcabc"

/// Output: true

/// Explanation: It is the substring "abc" four times or the substring

/// "abcabc" twice.

///

/// Constraints:

/// 1. 1 <= s.length <= 10^4

/// 2. s consists of lowercase English letters.

/// </summary>

bool LeetCodeString::repeatedSubstringPattern(string s)

{

vector<int> dp = kmp(s);

if ((dp.back() != 0) && (s.size() % (s.size() - dp.back()) == 0))

{

return true;

}

else

{

return false;

}

}

## [686. Repeated String Match](https://leetcode.com/problems/repeated-string-match/)

Medium

Given two strings a and b, return the minimum number of times you should repeat string a so that string b is a substring of it. If it is impossible for b​​​​​​ to be a substring of a after repeating it, return -1.

**Notice:** string "abc" repeated 0 times is "", repeated 1 time is "abc" and repeated 2 times is "abcabc".

**Example 1:**

**Input:** a = "abcd", b = "cdabcdab"

**Output:** 3

**Explanation:** We return 3 because by repeating a three times "ab**cdabcdab**cd", b is a substring of it.

**Example 2:**

**Input:** a = "a", b = "aa"

**Output:** 2

**Constraints:**

* 1 <= a.length, b.length <= 104
* a and b consist of lowercase English letters.

### Analysis:

The idea is to find match with KMP on string B and return -1 if we must reset to zero after size of A.

/// <summary>

/// Leet Code 686. Repeated String Match

///

/// Medium

///

/// Given two strings a and b, return the minimum number of times you

/// should repeat string a so that string b is a substring of it. If

/// it is impossible for b to be a substring of a after repeating it,

/// return -1.

///

/// Notice: string "abc" repeated 0 times is "", repeated 1 time is

/// "abc" and repeated 2 times is "abcabc".

///

/// Example 1:

/// Input: a = "abcd", b = "cdabcdab"

/// Output: 3

/// Explanation: We return 3 because by repeating a three times

/// "abcdabcdabcd", b is a substring of it.

///

/// Example 2:

/// Input: a = "a", b = "aa"

/// Output: 2

/// Constraints:

/// 1 . 1 <= a.length, b.length <= 10^4

/// 2. a and b consist of lowercase English letters.

/// </summary>

int LeetCodeString::repeatedStringMatch(string a, string b)

{

vector<int> dp = kmp(b);

size\_t i = 0, j = 0;

while (j < b.size())

{

if (a[i % a.size()] == b[j])

{

i++;

j++;

}

else

{

if (i - j >= a.size()) return -1;

while (j != 0 && a[i % a.size()] != b[j])

{

j = dp[j - 1];

}

if (a[i % a.size()] != b[j])

{

i++;

}

}

}

return (i + a.size() - 1) / a.size();

}

## [214. Shortest Palindrome](https://leetcode.com/problems/shortest-palindrome/)

Hard

You are given a string s. You can convert s to a

palindrome

 by adding characters in front of it.

Return the shortest palindrome you can find by performing this transformation.

**Example 1:**

**Input:** s = "aacecaaa"

**Output:** "aaacecaaa"

**Example 2:**

**Input:** s = "abcd"

**Output:** "dcbabcd"

**Constraints:**

* 0 <= s.length <= 5 \* 104
* s consists of lowercase English letters only.

### Analysis:

The idea is to patch the string, a delimiter and a reverse of string and using KMP to match the longest part.

/// <summary>

/// Leet Code 214. Shortest Palindrome

///

/// Hard

///

/// You are given a string s. You can convert s to a palindrome by adding

/// characters in front of it.

///

/// Return the shortest palindrome you can find by performing this

/// transformation.

///

/// Example 1:

/// Input: s = "aacecaaa"

/// Output: "aaacecaaa"

///

/// Example 2:

/// Input: s = "abcd"

/// Output: "dcbabcd"

///

/// Constraints:

/// 1. 0 <= s.length <= 5 \* 10^4

/// 2. s consists of lowercase English letters only.

/// </summary>

string LeetCodeString::shortestPalindrome(string s)

{

string reverse\_s = s;

std::reverse(reverse\_s.begin(), reverse\_s.end());

string str\_s = s + "#" + reverse\_s;

vector<int> dp = kmp(str\_s);

string result;

for (size\_t i = 0; i < s.size() - dp.back(); i++)

{

result.push\_back(s[s.size() - 1 - i]);

}

result.append(s);

return result;

}

## [1316. Distinct Echo Substrings](https://leetcode.com/problems/distinct-echo-substrings/)

Hard

Return the number of **distinct** non-empty substrings of text that can be written as the concatenation of some string with itself (i.e. it can be written as a + a where a is some string).

**Example 1:**

**Input:** text = "abcabcabc"

**Output:** 3

**Explanation:** The 3 substrings are "abcabc", "bcabca" and "cabcab".

**Example 2:**

**Input:** text = "leetcodeleetcode"

**Output:** 2

**Explanation:** The 2 substrings are "ee" and "leetcodeleetcode".

**Constraints:**

* 1 <= text.length <= 2000
* text has only lowercase English letters.

### Analysis:

If we use KMP, then from every position, extend KMP array. But an easier way is to check any length, maximum s.size() / 2, see if we can find a repeated substring within the original string.

/// <summary>

/// Leet code #1316. Distinct Echo Substrings

///

/// Hard

///

/// Return the number of distinct non-empty substrings of text that can be

/// written as the concatenation of some string with itself.

///

/// Example 1:

/// Input: text = "abcabcabc"

/// Output: 3

/// Explanation: The 3 substrings are "abcabc", "bcabca" and "cabcab".

/// Example 2:

/// Input: text = "leetcodeleetcode"

/// Output: 2

/// Explanation: The 2 substrings are "ee" and "leetcodeleetcode".

///

/// Constraints:

/// 1. 1 <= text.length <= 2000

/// 2. text has only lowercase English letters.

/// </summary>

int LeetCodeString::distinctEchoSubstrings(string text)

{

unordered\_set<string> result\_set;

for (size\_t len = 1; len <= text.size() / 2; len++)

{

int count = 0;

for (int i = 0; i + len < text.size(); i++)

{

if (text[i] == text[i + len])

{

count++;

}

else

{

count = 0;

}

if (count == len)

{

result\_set.insert(text.substr(i, len));

count--;

}

}

}

return result\_set.size();

}

There is another solution which use KMP

/// <summary>

/// Leet code #1316. Distinct Echo Substrings

///

/// Hard

///

/// Return the number of distinct non-empty substrings of text that can be

/// written as the concatenation of some string with itself.

///

/// Example 1:

/// Input: text = "abcabcabc"

/// Output: 3

/// Explanation: The 3 substrings are "abcabc", "bcabca" and "cabcab".

/// Example 2:

/// Input: text = "leetcodeleetcode"

/// Output: 2

/// Explanation: The 2 substrings are "ee" and "leetcodeleetcode".

///

/// Constraints:

/// 1. 1 <= text.length <= 2000

/// 2. text has only lowercase English letters.

/// </summary>

int LeetCodeString::distinctEchoSubstringsII(string text)

{

int n = text.size();

unordered\_set<string> result\_set;

for (int s = 0; s < (int)text.size(); s++)

{

vector<int> kmp(n);

int i = 1;

int j = 0;

while (s + i < n)

{

if (text[s + i] == text[s + j])

{

j++;

kmp[i] = j;

i++;

// must be even length

if (i % 2 == 1) continue;

// if duplicate as half half, the remaining prefix length

// must be divisible to total length

if ((i / 2) % (i - j) == 0)

{

result\_set.insert(text.substr(s, i));

}

}

else if (j == 0)

{

i++;

}

else

{

j = kmp[j - 1];

}

}

}

return result\_set.size();

}

## [1392. Longest Happy Prefix](https://leetcode.com/problems/longest-happy-prefix/)

Hard

A string is called a **happy prefix** if is a **non-empty** prefix which is also a suffix (excluding itself).

Given a string s, return the ***longest happy prefix*** of s. Return an empty string "" if no such prefix exists.

**Example 1:**

**Input:** s = "level"

**Output:** "l"

**Explanation:** s contains 4 prefix excluding itself ("l", "le", "lev", "leve"), and suffix ("l", "el", "vel", "evel"). The largest prefix which is also suffix is given by "l".

**Example 2:**

**Input:** s = "ababab"

**Output:** "abab"

**Explanation:** "abab" is the largest prefix which is also suffix. They can overlap in the original string.

**Constraints:**

* 1 <= s.length <= 105
* s contains only lowercase English letters.

### Analysis:

This is the typical KMP case.

/// <summary>

/// Leet Code 1392. Longest Happy Prefix

///

/// Hard

///

/// A string is called a happy prefix if is a non-empty prefix which is also

/// a suffix (excluding itself).

///

/// Given a string s, return the longest happy prefix of s. Return an empty

/// string "" if no such prefix exists.

///

/// Example 1:

/// Input: s = "level"

/// Output: "l"

/// Explanation: s contains 4 prefix excluding itself ("l", "le", "lev",

/// "leve"), and suffix ("l", "el", "vel", "evel"). The largest prefix which

/// is also suffix is given by "l".

///

/// Example 2:

/// Input: s = "ababab"

/// Output: "abab"

/// Explanation: "abab" is the largest prefix which is also suffix. They can

/// overlap in the original string.

///

/// Constraints:

/// 1. 1 <= s.length <= 10^5

/// 2. s contains only lowercase English letters.

/// </summary>

string LeetCodeString::longestPrefix(string s)

{

vector<int> kmp\_table(s.size());

int i = 1;

int j = 0;

while (i < (int)s.size())

{

if (s[i] == s[j])

{

j++;

kmp\_table[i] = j;

i++;

}

else if (j == 0)

{

kmp\_table[i] = 0;

i++;

}

else

{

j = kmp\_table[j - 1];

}

}

return s.substr(0, kmp\_table[s.size() - 1]);

}

## [2223. Sum of Scores of Built Strings](https://leetcode.com/problems/sum-of-scores-of-built-strings/)

Hard

You are **building** a string s of length n **one** character at a time, **prepending** each new character to the **front** of the string. The strings are labeled from 1 to n, where the string with length i is labeled si.

* For example, for s = "abaca", s1 == "a", s2 == "ca", s3 == "aca", etc.

The **score** of si is the length of the **longest common prefix** between si and sn (Note that s == sn).

Given the final string s, return the ***sum*** of the ***score*** of every si.

**Example 1:**

**Input:** s = "babab"

**Output:** 9

**Explanation:**

For s1 == "b", the longest common prefix is "b" which has a score of 1.

For s2 == "ab", there is no common prefix so the score is 0.

For s3 == "bab", the longest common prefix is "bab" which has a score of 3.

For s4 == "abab", there is no common prefix so the score is 0.

For s5 == "babab", the longest common prefix is "babab" which has a score of 5.

The sum of the scores is 1 + 0 + 3 + 0 + 5 = 9, so we return 9.

**Example 2:**

**Input:** s = "azbazbzaz"

**Output:** 14

**Explanation:**

For s2 == "az", the longest common prefix is "az" which has a score of 2.

For s6 == "azbzaz", the longest common prefix is "azb" which has a score of 3.

For s9 == "azbazbzaz", the longest common prefix is "azbazbzaz" which has a score of 9.

For all other si, the score is 0.

The sum of the scores is 2 + 3 + 9 = 14, so we return 14.

**Constraints:**

* 1 <= s.length <= 105
* s consists of lowercase English letters.

### Analysis:

The KMP can tell you from the current position, the longest prefix it can match. Since we also want to know all the possible matches, including shorter ones, we should count the iterated matches.

If we move from left to right, theoretically, we have all already calculated the accumulated count already.

/// <summary>

/// Leet Code 2223. Sum of Scores of Built Strings

///

/// Hard

///

/// You are building a string s of length n one character at a time,

/// prepending each new character to the front of the string. The

/// strings are labeled from 1 to n, where the string with length i

/// is labeled si.

///

/// For example, for s = "abaca", s1 == "a", s2 == "ca", s3 == "aca", etc.

/// The score of si is the length of the longest common prefix between si

/// and sn (Note that s == sn).

///

/// Given the final string s, return the sum of the score of every si.

///

/// Example 1:

/// Input: s = "babab"

/// Output: 9

/// Explanation:

/// For s1 == "b", the longest common prefix is "b" which has a score of 1.

/// For s2 == "ab", there is no common prefix so the score is 0.

/// For s3 == "bab", the longest common prefix is "bab" which has a score

/// of 3.

/// For s4 == "abab", there is no common prefix so the score is 0.

/// For s5 == "babab", the longest common prefix is "babab" which has a

/// score of 5.

/// The sum of the scores is 1 + 0 + 3 + 0 + 5 = 9, so we return 9.

///

/// Example 2:

/// Input: s = "azbazbzaz"

/// Output: 14

/// Explanation:

/// For s2 == "az", the longest common prefix is "az" which has a

/// score of 2.

/// For s6 == "azbzaz", the longest common prefix is "azb" which has a

/// score of 3.

/// For s9 == "azbazbzaz", the longest common prefix is "azbazbzaz" which

/// has a score of 9.

/// For all other si, the score is 0.

/// The sum of the scores is 2 + 3 + 9 = 14, so we return 14.

///

/// Constraints:

/// 1. 1 <= s.length <= 10^5

/// 2. s consists of lowercase English letters.

/// </summary>

long long LeetCodeString::sumScores(string s)

{

vector<int> arr = kmp(s);

vector<int> dp(s.size());

long long result = 0;

for (size\_t i = 0; i < arr.size(); i++)

{

if (arr[i] == 0) dp[i] = 0;

else

{

dp[i] = dp[arr[i] - 1] + 1;

}

result += (long long)dp[i];

}

result += (long long)s.size();

return result;

}

## [3008. Find Beautiful Indices in the Given Array II](https://leetcode.com/problems/find-beautiful-indices-in-the-given-array-ii/)

Hard

You are given a **0-indexed** string s, a string a, a string b, and an integer k.

An index i is **beautiful** if:

* 0 <= i <= s.length - a.length
* s[i..(i + a.length - 1)] == a
* There exists an index j such that:
  + 0 <= j <= s.length - b.length
  + s[j..(j + b.length - 1)] == b
  + |j - i| <= k

Return the array that contains beautiful indices in ***sorted order from smallest to largest***.

**Example 1:**

**Input:** s = "isawsquirrelnearmysquirrelhouseohmy", a = "my", b = "squirrel", k = 15

**Output:** [16,33]

**Explanation:** There are 2 beautiful indices: [16,33].

- The index 16 is beautiful as s[16..17] == "my" and there exists an index 4 with s[4..11] == "squirrel" and |16 - 4| <= 15.

- The index 33 is beautiful as s[33..34] == "my" and there exists an index 18 with s[18..25] == "squirrel" and |33 - 18| <= 15.

Thus we return [16,33] as the result.

**Example 2:**

**Input:** s = "abcd", a = "a", b = "a", k = 4

**Output:** [0]

**Explanation:** There is 1 beautiful index: [0].

- The index 0 is beautiful as s[0..0] == "a" and there exists an index 0 with s[0..0] == "a" and |0 - 0| <= 4.

Thus we return [0] as the result.

**Constraints:**

* 1 <= k <= s.length <= 5 \* 105
* 1 <= a.length, b.length <= 5 \* 105
* s, a, and b contain only lowercase English letters.

### Analysis:

The search target is very long, we can use KMP to optimize it. Also, we need to find all the occurrences. We put the search target as the prefix of the original string and delimit it by a character of '#'.

Finally, we iterate it by using two queue for all the occurrences indexes.

/// <summary>

/// Leet Code 3008. Find Beautiful Indices in the Given Array II

///

/// Hard

///

/// You are given a 0-indexed string s, a string a, a string b, and an

/// integer k.

///

/// An index i is beautiful if:

/// 0 <= i <= s.length - a.length

/// s[i..(i + a.length - 1)] == a

/// There exists an index j such that:

/// 0 <= j <= s.length - b.length

/// s[j..(j + b.length - 1)] == b

/// |j - i| <= k

/// Return the array that contains beautiful indices in sorted order from

/// smallest to largest.

///

/// Example 1:

/// Input: s = "isawsquirrelnearmysquirrelhouseohmy", a = "my",

/// b = "squirrel", k = 15

/// Output: [16,33]

/// Explanation: There are 2 beautiful indices: [16,33].

/// - The index 16 is beautiful as s[16..17] == "my" and there exists an

/// index 4 with s[4..11] == "squirrel" and |16 - 4| <= 15.

/// - The index 33 is beautiful as s[33..34] == "my" and there exists an

/// index 18 with s[18..25] == "squirrel" and |33 - 18| <= 15.

/// Thus we return [16,33] as the result.

///

/// Example 2:

/// Input: s = "abcd", a = "a", b = "a", k = 4

/// Output: [0]

/// Explanation: There is 1 beautiful index: [0].

/// - The index 0 is beautiful as s[0..0] == "a" and there exists an index 0

/// with s[0..0] == "a" and |0 - 0| <= 4.

/// Thus we return [0] as the result.

///

/// Constraints:

/// 1. 1 <= k <= s.length <= 5 \* 10^5

/// 2. 1 <= a.length, b.length <= 5 \* 10^5

/// 3. s, a, and b contain only lowercase English letters.

/// </summary>

vector<int> LeetCodeString::beautifulIndicesII(string s, string a, string b, int k)

{

string str\_a = a + "#" + s;

string str\_b = b + "#" + s;

vector<int> kmp\_a = kmp(str\_a);

vector<int> kmp\_b = kmp(str\_b);

queue<int> queue\_a, queue\_b;

for (size\_t i = 0; i < kmp\_a.size(); i++)

{

if (kmp\_a[i] >= (int)a.size())

queue\_a.push(i - 2 \* a.size());

}

for (size\_t i = 0; i < kmp\_b.size(); i++)

{

if (kmp\_b[i] >= (int)b.size())

queue\_b.push(i - 2 \* b.size());

}

vector<int> result;

while (!queue\_a.empty() && !queue\_b.empty())

{

if (abs(queue\_a.front() - queue\_b.front()) <= k)

{

result.push\_back(queue\_a.front());

queue\_a.pop();

}

else if (queue\_b.front() < queue\_a.front())

{

queue\_b.pop();

}

else

{

queue\_a.pop();

}

}

return result;

}