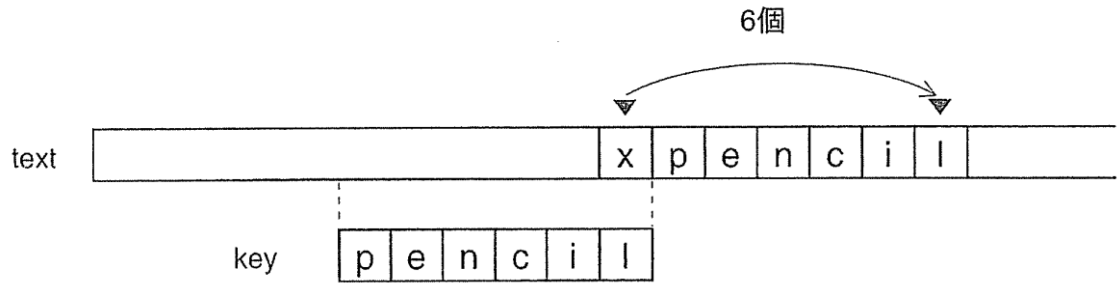
Advanced Class Lab06 04/28

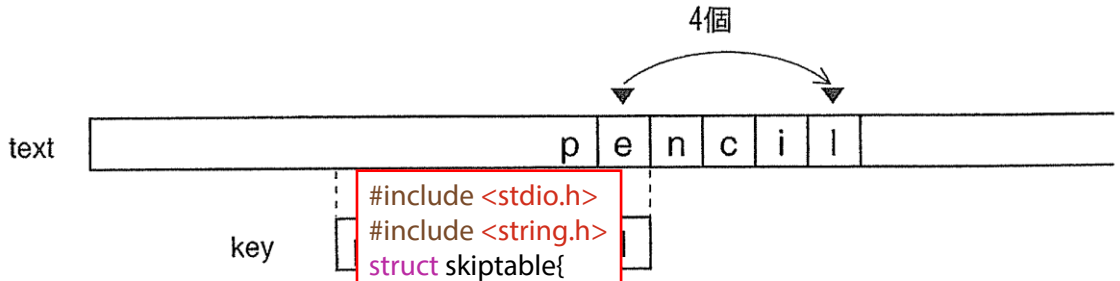
1. Boyer-Moore Algorithm

It is inefficient to match the correct pattern character by character. Boyer-Moore algorithm compares the most right character of target key with that of candidate string in text. When these two characters are not different, it is not to shift one character, but to shift according to the most right character of candidate string in text.

Take following figure as example; the target key is "pencil", and the most right character of candidate string in text is "x". Because it is impossible for "pencil" to exist in the range of next 5 characters, the next compared position can be shifted to next 6 characters, i.e., to skip 5 characters.



Take following figure as another example; the most right character of candidate string in text is "e". The most possible condition is as below figure, so the next compared position can be shifted to next 4 characters, i.e., to skip 3 characters.



In summary, we can cor able" as follows according to the characters of key.

```
#include <stdio.h>
#include <string.h>
struct skiptable{
   char character;
   int skipValue;
};
char *search(char
```

character	p	e	n	c	i	l	Others
skip value	5	4	3	2	1	6	6

If there are the same characters in the key, for example, "array", as follows.

character	a	r	r	a	y	Others
skip value	4	3	2	1	5	5

The skip value of the same character is considered as smaller one.

character	r	a	y	Others
skip value	2	1	5	5

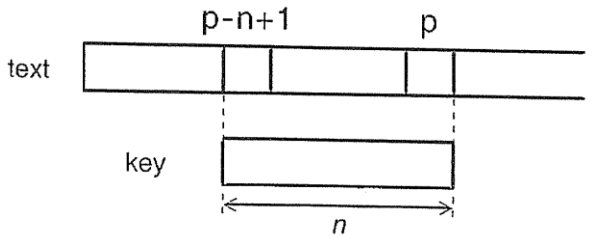
The summary of Boyer-Moore algorithm is as follows.

a) Construct the "skip table".

b) Repeat steps c) and d) before completing matching all characters in candidate text.

c) If the character at position *p* in candidate text is the same as the most right character of target key (represented as *key[n-1]*), compare the string (n characters and start from *p-n+1*) with target key. If they are well matched, the answer is the position *p-n+1*.

d) Shift the compared position *p* according to the skip value in "skip table".



Please implement Boyer-Moore algorithm as a function "char * search(char *text, char * key)" and verify it.

2. Hash

In general, we can manage a lot of amount of student data by simply setting student IDs as record numbers, and store it into array or files. However, many data cannot be managed by numbers, and we always set non-number data such as name as key values, and then we can use these key values to search what we want in the future. These data cannot be referenced and this an irregular manage method always take much time to search the data. We can use Hash method to set names as key values and construct the speed references to improve this problem.

ANN	3211—1234	EMY	3331—4567	CANDY	3333—1222	···

In computing, a hash table (also hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the correct value can be found.

Please consider following hash function. The name is composed of upper-cases, and we set names with upper-case letters as keys. In a name, there are n characters, A_0, A_1, A_2, …, A_{n-1}, which can constitute 26^n names. This is a big range, so we just use three letters, first letter, A_0, middle letter, $A_{n/2}$, and the last letter,

A_{n-1}, to implement hash function.

$$hash(A_0\ A_1\ A_2 \ldots A_{n-1}) = (A_0 + A_{n/2} * 26 + A_{n-1} * 26^2) \bmod 1000$$

For example, if key is "SUZUKI", then

$$hash("SUZUKI") = (('S'-'A')+('U'-'A')*26+('I'-'A')*26^2)\%1000$$
$$=(18+520+5408)\%1000$$
$$=946$$

We can consider number 946 as index of array or record number of file, and easily find the data of SUZUKI.

The following is a seudo code, please complete it, implement hash function int hash(char *s), and test your program.

```
#include <stdio.h>
#include <conio.h>
#define STACK_SIZE 10

struct int_stack
{
   int stack[STACK_SIZE];
   int sp;
};

int push(int_stack &s, int data)
{

}

int pop(int_stack &s, int &data)
{

}

int main()
{
   int x;
   int_stack s;
   s.sp = -1;
```

```
  while(scanf("%d",&x))
    if(!push(s,x))
          printf("Stack is full.\n");

  printf("Pop the data:\n");
  while(pop(s,x))
    printf("%d\n",x);

  getch();
}
```

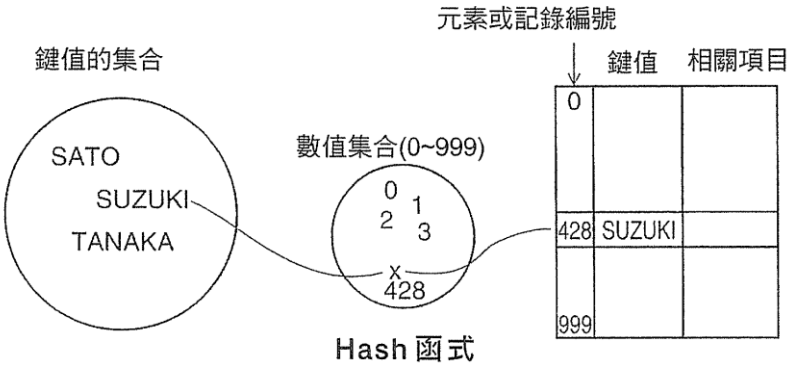
3. Hash with Collision

Ideally, the hash function should assign each possible key to a unique bucket, but this ideal situation is rarely achievable in practice (unless the hash keys are fixed; i.e. new entries are never added to the table after it is created). Instead, most hash table designs assume that "hash collisions"—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way. For example, if there are two keys, "SUZUKI" and "SAMURI", the numbers generated by hash function are both 946. This condition is called "collision". Problem 2 you implemented before cannot deal with this situation. Hash function needs to make occurrences of collisions as lower as possible. If collision happens, the data have to be stored in other position.

There are many methods to avoid collisions, we only introduce simple algorithm here. We add a status flag "empty" into data structure to show memory usage situation. "empty=1" represents in use, and "empty!=0" represents not in use. We can check "empty" flag to know if collision happens. And if it does, find next storage memory sequentially, and if it is empty, store the data in that position. Consider above example, "SAMURI" will be stored into 947.

When searching data, program will find the position with number generated by hash function. If the data is not matched, it will find next position circularly until it meet the empty slot. We assume there is at least one empty slot in the database.

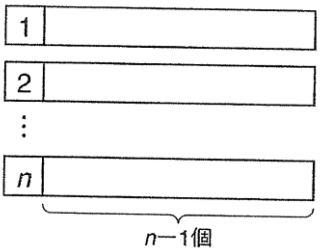
(Note that this is not an efficient method. If space of hash table is full, it is not easy to find other space to store collision data, and this data will be stored in a position far away from the original position.)



Please refer to above figure to modify program 2 and complete this program.

4. Permutation Combination

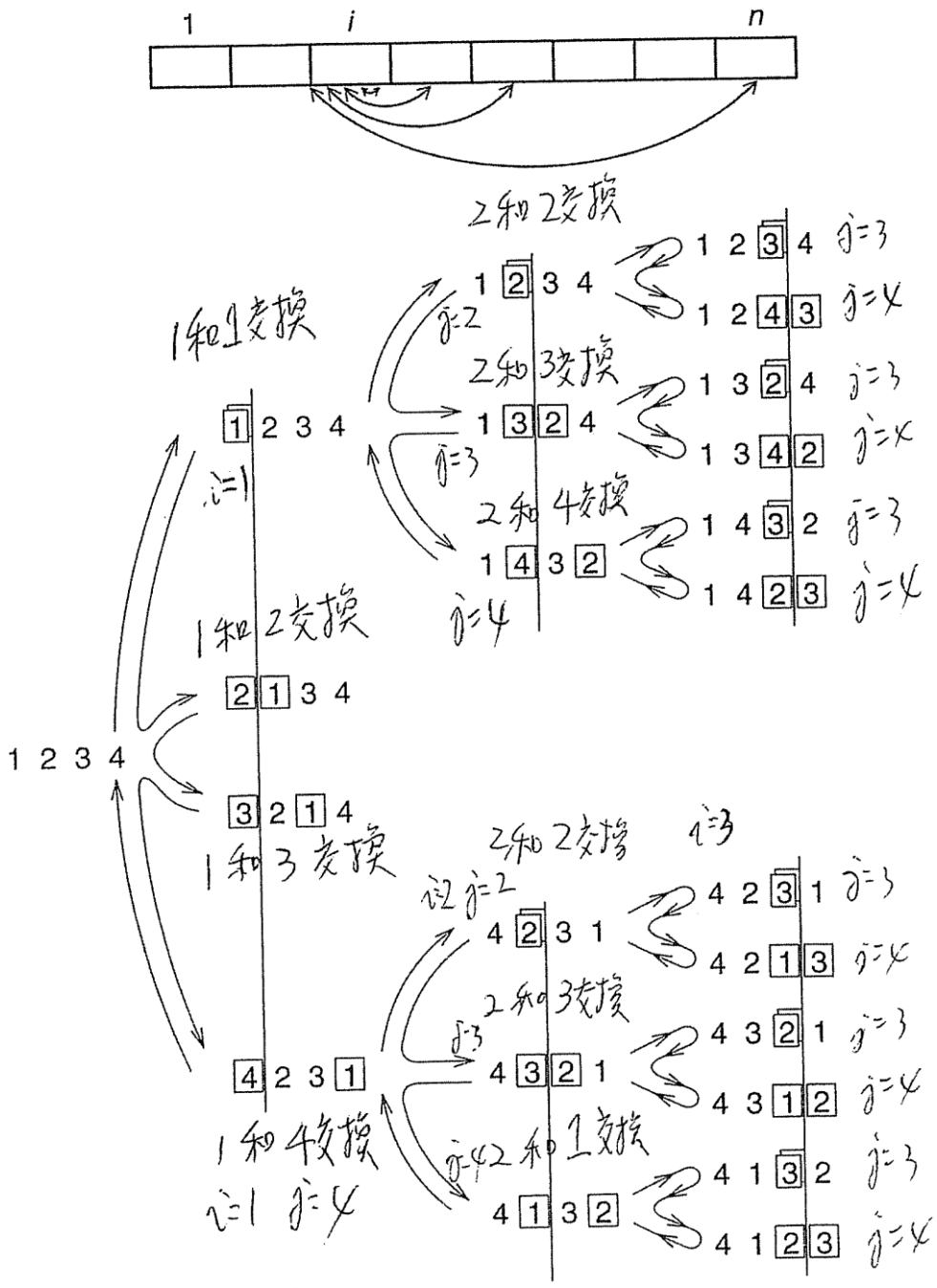
The permutation combination of 1, 2 , and 3 are 6 numbers, 123, 132, 213, 231, 312, 321. The permutation amount of n different numbers are n!. The permutation problem with 1, 2, 3, …, n shown as follows can be considered as another n-1 permutation problems with beginning from 1~n. And the n-1 permutation problems can also be considered as the same conditions.



If we want to make 1~n as begin of a number series, we need to exchange each item of 1~n sequentially . Take 1, 2, 3, and 4 as example.

a) **Exchange 1 and 1, fix 1 as begin (1,2,3,4), and be as permutation problem with 2, 3, and 4.**
b) **Exchange 1 and 2, fix 2 as begin (2,1,3,4), and be as permutation problem with 1, 3, and 4.**
c) **Exchange 1 and 3, fix 3 as begin (3,2,1,4), and be as permutation problem with 2, 1, and 4.**
d) **Exchange 1 and 4, fix 2 as begin (4,2,3,1), and be as permutation problem with 2, 3, and 1.**

You can implement it by recursion. Before calling recursive function, firstly do exchange of 1 and 1~n. After calling recursive function, recover all exchanged numbers. Refer to following figure, the exchange index will move from 1 to n sequentially in recursion.



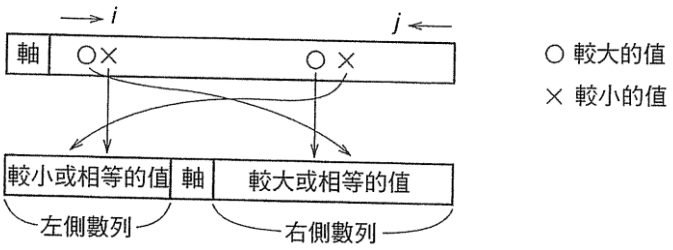
Please list all results of permutation combination of n numbers.
The result is as follows.

```
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 3 2
1 4 2 3
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 3 1
2 4 1 3
3 2 1 4
3 2 4 1
3 1 2 4
3 1 4 2
3 4 1 2
3 4 2 1
4 2 3 1
4 2 1 3
4 3 2 1
4 3 1 2
4 1 3 2
4 1 2 3
```

5. Quicksort

Quicksort, or partition-exchange sort, is a sorting algorithm developed by Tony Hoare that, on average, makes O(n log n) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons, though this behavior is rare. Quicksort is often faster in practice than other O(n log n) algorithms. Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log n) additional space used by the stack during the recursion.

Quicksort is a divide and conquer algorithm. It considers a suitable value as pivot, and first divides a large list into two smaller sub-lists according to pivot: the low elements and the high elements. Quicksort can then repeat above steps and recursively sort the sub-lists.

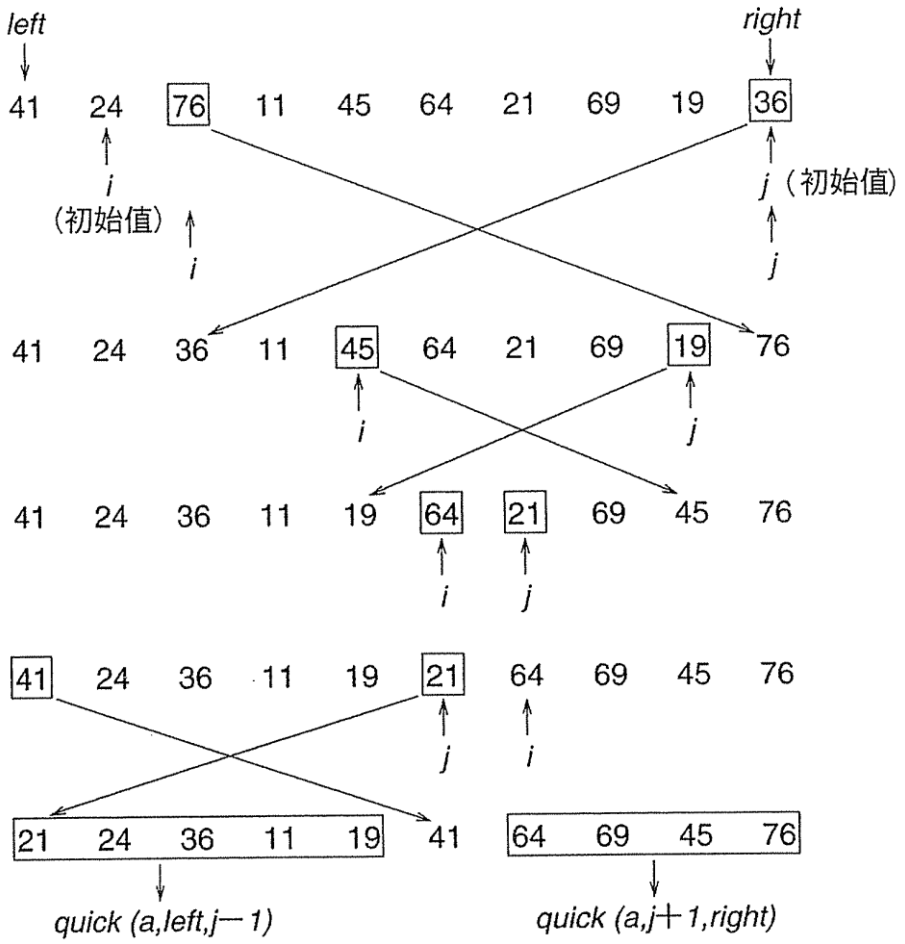


For convenience, we set pivot as the most left number of series. Set i as index from left of series, and j as index from right of series. The steps are:

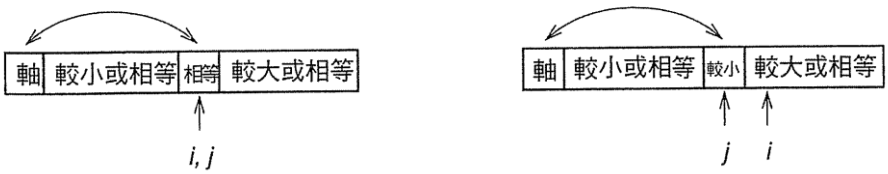
a) Because index 0 is pivot, start left sort from index 1.

b) **Move i from left to right, and find the item larger than pivot. Move j from right to left, and find the item smaller than pivot. And then exchange item i and item j. Repeat this step until i>=j, and then exchange pivot and item j.**

c) **Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values.**

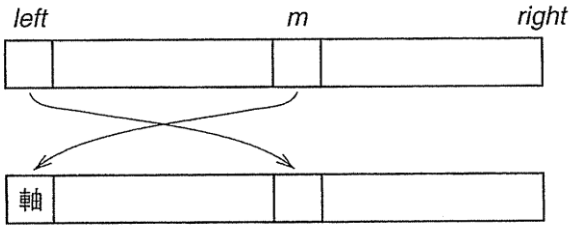


After exchanging i and j, the condition are shown as follows.



Therefore, next exchange of low series starts from left to j-1, and high series starts from j+1 to right. Pivot is at the correct position, so it will not be exchanged in later work.

P.S. It is inefficient to set the most left item or the most right item as pivot. Setting middle item of series as pivot has better performance, so you just exchange middle item with the most left item before starting one-round sort.



Please sort data in ascending order by quick sort algorithm.