CS1010 Tutorial 8
Group BC1A

22 October 2020

Topics for today

Objectives

- Recap on Topics (Searching, Sorting)
- Going through problem set 23, 24
- Feedback for Assignment 5, 6
- Summary

Corrections from previous tutorial

- ```
  //This loop will be of o(sqrt(n))
  for (long i = 0; i < sqrt(n); i += 1) {
      //Do something (o(1) time)
  }
  ```
- 22.2 (c)

n times. = 1, 2, = n.

```
long k = 1;
for (long j = 0; j < n; j +=1) { //Outer loop
    k *= 2;
    for (long i = 0; i < k; i += 1) { //Inner loop
        cs1010_println-long(i + j);
    }
}
```

n

$2, 4 \ldots 2^n$

×2 ×2 ×2.

Note that the inner loop always run 2^n times. This gives us the sequence $2, 4, 8, \ldots, 2^n$. To find the sum of all these value, use binary expansion or geometric series to help you. This will give you $\sum_{i=1}^{n} 2^n = 2^{(n+1)} - 2$. This makes the efficiency $o(2^{(n+1)} - 2) = o(2^{(n+1)}) = o(2(2^n)) = o(2^n)$.
```

2 3 4 5 | 5 5 6 7 8

2 3 4 5 5 | 5 6 7 8

Searching

- Linear search
  - Goes though the whole list of value and compare each value to the needed condition
  - o(n) efficiency
  - Works with any list of value
- Binary search
  - Uses divide and conquer method to search a list of sorted values efficiently
  - o(log2(n)) efficiency
  - Works only with a **SORTED** array

Problem 23.1 (b) Question

Instead of returning the position of the query `q`, modify an iterative version of the binary search such that it returns either:

- a position `k`, such as `a[k] <= q <= a[k+1]`.
- -1 if `q < a[0]`
- n-1 if `q > a[n-1]`

```
/**
 * Iterative binary search.
 * Look for q in list[i]..list[j].
 *
 * @pre list is sorted
 * @return -1 if not found, the position of q in list otherwise.
 */
long search(const long list[], long len, long q) {
  long i = 0;
  long j = len-1;
  while (i <= j) {
    long mid = (i+j)/2;
    if (list[mid] == q) {
      return mid;
    }
    if (list[mid] > q) {
          j = mid-1;
    } else {
      i = mid+1;
    }
  }
  return -1;
}
```

Problem 23.1 (b) Answer

```c
/**
 * Iterative binary search
 * Look for q in list[i]..list[j].
 *
 * @pre list is sorted
 * @return i-1 if not found, the position of q in list otherwise.
 */
long search(const long list[], long len, long q) {
  long i = 0;
  long j = len-1;
  while (i <= j) {
    long mid = (i+j)/2;
    if (list[mid] == q) {
      return mid;
    }
    if (list[mid] > q) {
      j = mid-1;
    } else {
      i = mid+1;
    }
  }
  return i-1; // change this line only
}
```

i=0

i=3

i+1

once found, terminate:

n

(n-1)+1 = n

n-1

Initial loop invariant

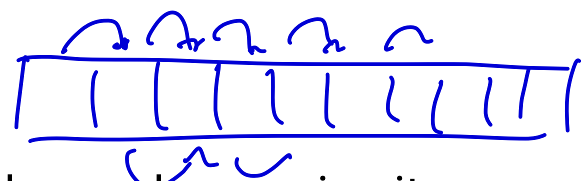
```
// { q is not in list[0]..list[i-1] and list[j+1]..list[n-1] }
```

Loop invariant after modification

```
// { (q > list[0]..list[i-1]) and q < (list[j+1]..list[n-1]) }
```

Sorting

- Counting sort
  - Sort by counting how many times a certain number exist within a list
  - Efficiency o(n + MAX)
- Selection sort
  - Sort by finding the largest/smallest value in list before swapping it to the correct position
  - Efficiency o(n^2)



- Bubble sort
  - Sort by comparing to the next number and swapping it
  - Efficiency o(n^2)
- Insertion sort
  - Sort by inserting unsorted values into their correct positions
  - Efficiency o(n^2) (Worse case)
- Visualising sorting algorithm
  - https://visualgo.net/en
  - https://panthema.net/2013/sound-of-sorting/

Counting sort

```
/**
 * Perform counting sort on the input in[] and store the sorted
 * numbers in out[].
 *
 * @param[in] in The array containing numbers to be sorted.
 * @param[out] out The array containing the sorted numbers.
 * @param[in] len The size of the input and output array.
 *
 * @pre in[i] is between 0 and MAX for all i.
 * @post out[] is sorted
 */
void counting_sort(const long in[], long out[], long len)
{
  long freq[MAX + 1] = { 0 };

  for (long i = 0; i < len; i += 1) { //A          o(n)
    freq[in[i]] += 1;
  }

  long outpos = 0;
  for (long i = 0; i <= MAX; i += 1) { //B                        max
    for (long j = outpos; j < outpos + freq[i]; j += 1) { //C
      out[j] = i;
    }
    outpos += freq[i];
  }
}
```

Efficiency analysis:

A is $o(n)$. B is $o(MAX)$. Take note that C will only reach $o(n)$ in **total** instead of $o(n)$ for each loop even if it is being loop though MAX times. Total efficiency = $o(n + n + MAX) = o(2n + MAX) = o(n + MAX)$

Selection sort

```
long max(long last, const long list[])
{
  long max_so_far = list[0];
  long max_index = 0;
  for (long i = 1; i <= last; i += 1) { //A
    if (list[i] > max_so_far) {
      max_so_far = list[i];
      max_index = i;
    }
  }
  return max_index;
}
```

max/min. o(n)

```
void selection_sort(long length, long list[])
{
  for (long i = 1; i < length; i += 1) { //B
    long max_pos = max(length - i, list);
    if (max_pos != length - i) {
      swap(&list[max_pos], &list[length - i]);
    }
  }
}
```

o(n)

Efficiency analysis:

A takes $o(n)$. B takes $o(n)$. Total efficiency = $o(n * n) = o(n^2)$.

Bubble sort

```
void bubble_pass(long last, long a[])
{
  for (long i = 0; i < last; i += 1) { //A
    if (a[i] > a[i+1]) {
      swap(a, i, i+1);
    }
  }
}

void bubble_sort(long n, long a[]) {
  for (long last = n - 1; last > 0; last -= 1) { //B
    bubble_pass(last, a);
  }
}
```

Efficiency analysis:

A takes $o(n)$. B takes $o(n)$. Total efficiency = $o(n * n) = o(n^2)$.

Insertion sort

```
void insert(long a[], long curr)
{
  long i = curr - 1;
  long temp = a[curr];
  while (i >= 0 && temp < a[i]) { //A
    a[i+1] = a[i];
    i -= 1;
  }
  a[i+1] = temp;
}

void insertion_sort(long n, long a[]) {
  for (long curr = 1; curr < n; curr += 1) { //B
    insert(a, curr);
  }
}
```

Efficiency analysis:

A takes $o(n)$ in worse case. B takes $o(n)$ in worse case. Total efficiency = $o(n * n) = o(n^2)$ in worse case.

Problem 24.1 Question

In the implementation of bubble sort above, we always make $n - 1$ passes through the array. It is, however, possible to stop the whole sorting procedure, when a pass through the array does not lead to any swapping. Modify the code above to achieve this optimization.

Problem 24.1 Answer

```
bool bubble_pass(long last, long a[])
{
  bool swapped = false;
  for (long i = 0; i < last; i += 1) {
    if (a[i] > a[i+1]) {
      swap(a, i, i+1);
      swapped = true;
    }
  }
  return swapped;
}

void bubble_sort(long n, long a[n]) {
  bool swapped = true;
  for (long last = len - 1; last > 0 && swapped; last -= 1) {
    swapped = bubble_pass(last, a);
  }
}
```

Problem 24.2 Question

(a) Suppose the input list to insertion sort is already sorted. What is the running time of insertion sort?

(b) Suppose the input list to insertion sort is inversely sorted. What is the running time of insertion sort?

Problem 24.2 Answer

a) If the input is already sorted, then we would never enter the loop

```
while (temp < a[i] && i >= 0) {
  :
}
```

So the function `insert` is $O(1)$ and insertion sort runs in $O(n)$ time.

b) If the input is inversely sorted, then we enter the loop every time. Not only that, `temp < a[i]` is true for every `i` we check until `i == 0`.

```
while (temp < a[i] && i >= 0) {
  :
}
```

So this is the worst case as for every element, we have to shift every elements to its left. It is still $O(n^2)$

Problem 24.3 Question

What is the loop invariant for the loop in the function `insert`?

```
void insert(long a[], long curr)
{
  long i = curr - 1;
  long temp = a[curr];
  while (i >= 0 && temp < a[i]) { //A
    a[i+1] = a[i];
    i -= 1;
  }
  a[i+1] = temp;
}
```

Problem 24.3 Answer

The invariant is: `temp` is smaller or equal to than `a[i+1]..a[curr]`.

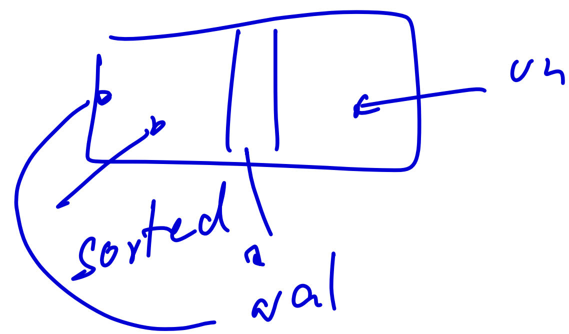
```
void insert(long a[], long curr)
{
  long i = curr - 1;
  long temp = a[curr];
  // { temp <= a[j], for all i+1 <= j <= curr }
  // This is true since i+1 is curr and temp is a[curr]
  while (temp < a[i] && i >= 0) {
    // { temp < a[i] }
    a[i+1] = a[i];
    i -= 1;
    // { temp < a[i+1]  }
    // The invariant { temp <= a[j], for all i+1 <= j <= curr }
    // remains true.
  }
  // { temp >= a[i] || i == -1 }
  // The invariant { temp <= a[j], for all i+1 <= j <= curr } was true at the
  // end of the loop.  So it remains true once we exit the loop.
  a[i+1] = temp;
}
```

Problem 24.4 Question

In certain scenarios, comparison is more expensive than assignment. For instance, comparing two strings is more expensive than assigning a string to a variable. In this case, we can reduce the number of comparisons during insertion sort by doing the following:

repeat

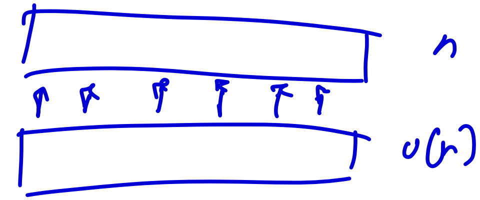
- take the first element X from unsorted partition
- use binary search to find the correct position to insert X
- insert X into the right place



until the unsorted partition is empty.

Implement the variation to insertion sort above. You may use your solution from Problem 23.1.

Problem 24.4 Answer



new place.
(Assignment)

Algorithm:

```
repeat
  take the first element X from the unsorted pile
  use binary search to find the correct position to insert X
  insert X into the right place
until the unsorted pile is empty
```

Understand that not all operations are equally expensive and choosing different sorting algorithms depending on whether comparison or assignment is expensive is important

```
void insert(long a[], long curr)
{
  long i = curr - 1;
  long temp = a[curr];
  long pos = search(a, curr-1, temp); // change this line
  // temp should go to a[pos+1]
  while (i > pos) { // no longer need to compare
    a[i+1] = a[i];
    i -= 1;
  }
  a[i+1] = temp;
}

void insertion_sort(long n, long a[n]) {
  for (long curr = 1; curr < n; curr += 1) {
    insert(a, curr);
  }
}
```

use binary here.

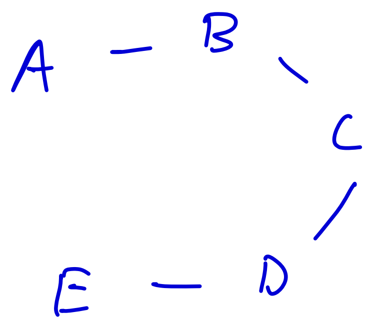
o(log n)

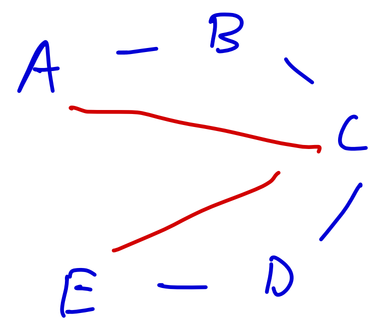
o(n)

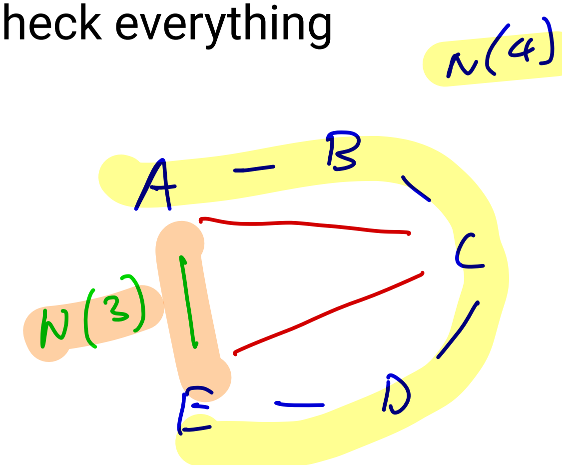
∴ o(n log n)

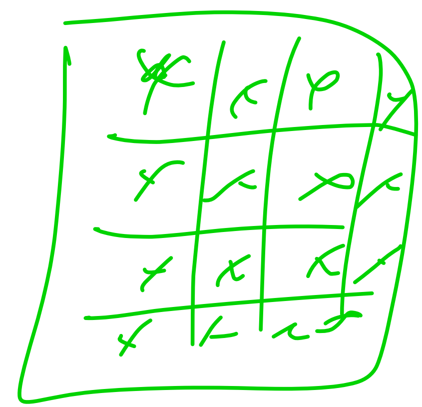
Assignment 5

- Remember to break the question down into smaller chunks
- Issue with using only one array to store and check everything
  - `(N(1) -> N(2) -> N(4) ...)`









Assignment 5

- For two people i, j and one intermediary m, consider only i to m (indirectly) and m to j (directly). Didn't consider i to m (directly) and m to j (indirectly).
    - Note that while i to m (indirectly) and m to j (directly) might not give you a full edge, i to m (directly) and m to j (indirectly) might and vice versa
    - This means we need to consider both cases before ruling out connection

A — B (Directly)

B ~ C (Indirectly)

A ~ C (Indirect)

||

B — C (direct)

A ~ B (Indirect)

A ~ C (Indirect)

Assignment 6

- Permutation
  - o(nk^2)
    - Typical nested loop and check every character
  - o(nk)
    - Use answer for frequency.c to help you check for the same frequency after finding out all possible substring of length k from s2
  - o(n + k)
    - Any ideas?

Question 3: Permutation (15 marks)

Write a program `permutation`, that, given two strings, consists of alphabets 'a' to 'z', S1 and S2, checks if S2 is a permutation of some substring of S1. A substring of length k is a consecutive sequence of k characters from a string.

For instance, `nus` is a permutation of a substring of `suntec`, since `suntec` contains `sun`. `ntu` is also a permutation of a substring of `suntec`, since `suntec` contains `unt`. `smu` is not a permutation of any substring of `suntec`.

Your program should read, from the standard input,

- a string S1, consists of k characters, chosen from `a` to `z`
- a string S2, consists of n characters, chosen from `a` to `z`

and print, to the standard output, `YES` if S2 is a permutation of some substring of length k from S1, and `NO` otherwise.

Permutation $0(nk^2)$

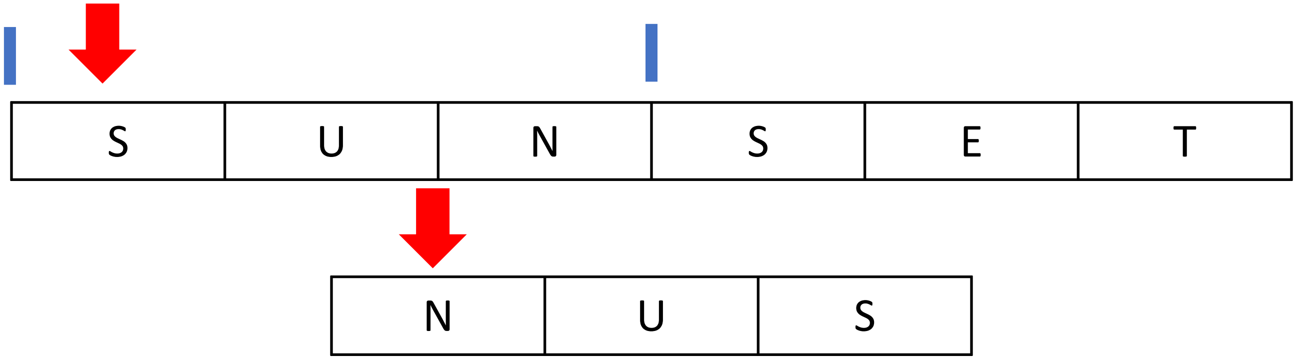
Find all possible substring of length k from s1, and then use a double for loop through s2 to check if each alphabet exists in the substring

S	U	N	S	E	T

N	U	S

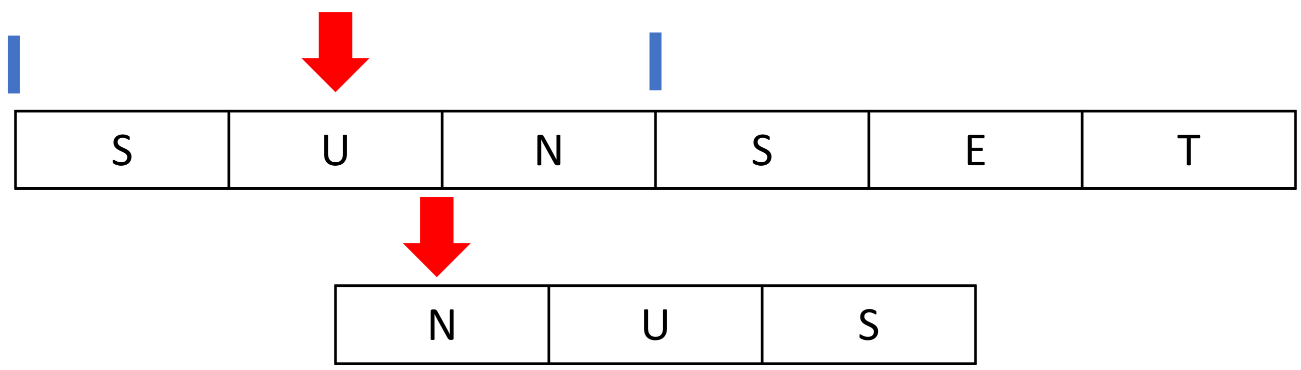
Permutation $O(nk^2)$

Find all possible substring of length k from s1, and then use a double for loop through s2 to check if each alphabet exists in the substring



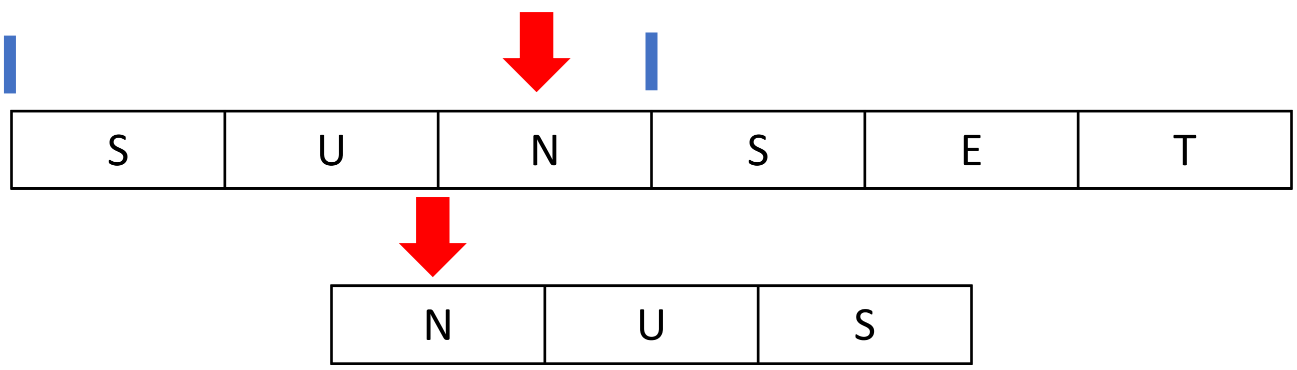
Permutation $O(nk^2)$

Find all possible substring of length k from s1, and then use a double for loop through s2 to check if each alphabet exists in the substring



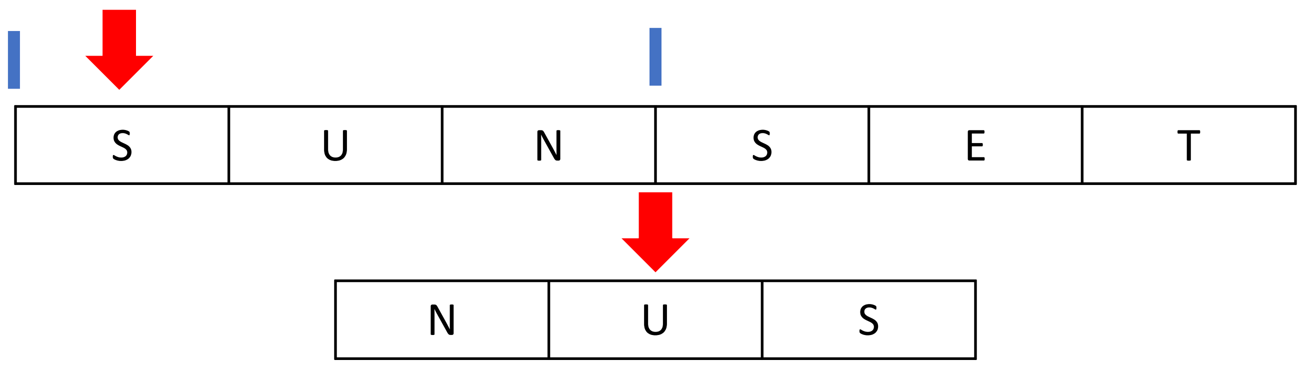
Permutation $O(nk^2)$

Find all possible substring of length k from s1, and then use a double for loop through s2 to check if each alphabet exists in the substring



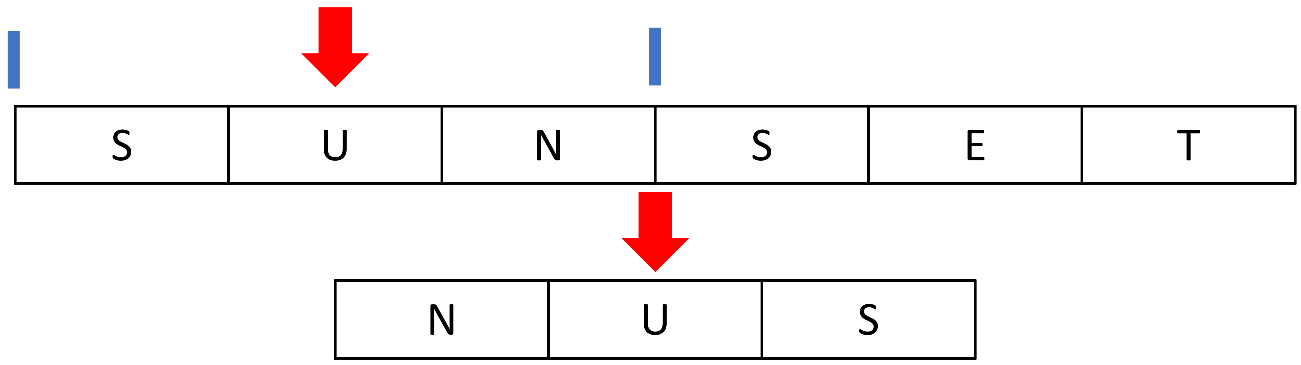
Permutation $O(nk^2)$

Find all possible substring of length k from s1, and then use a double for loop through s2 to check if each alphabet exists in the substring



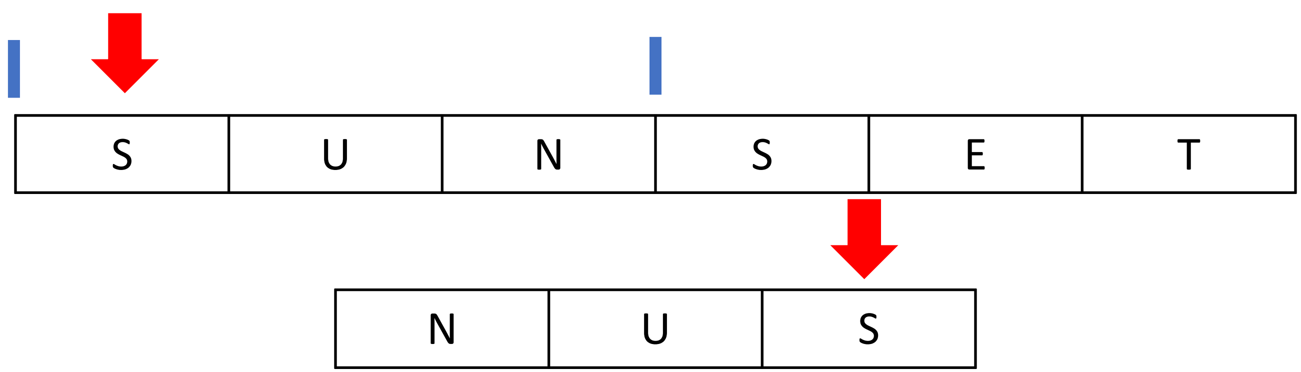
Permutation $O(nk^2)$

Find all possible substring of length k from s1, and then use a double for loop through s2 to check if each alphabet exists in the substring



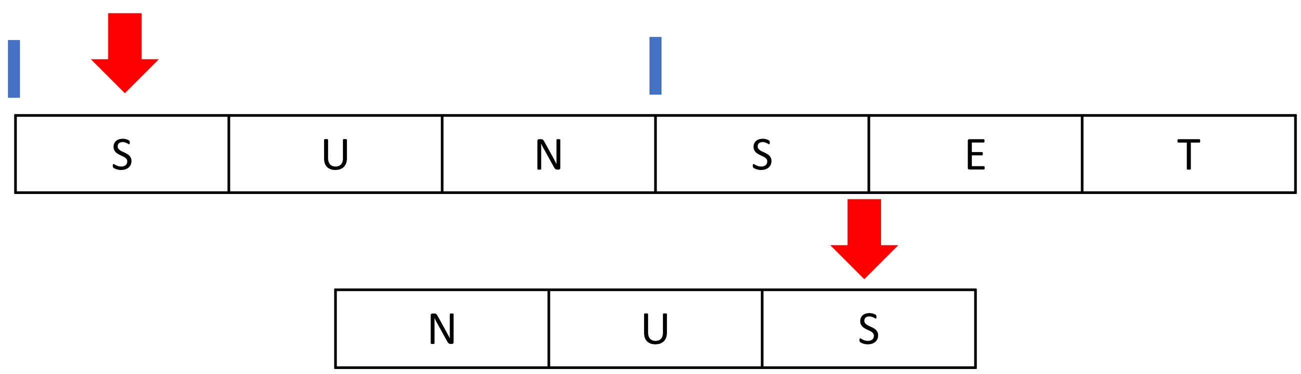
Permutation $O(nk^2)$

Find all possible substring of length k from s1, and then use a double for loop through s2 to check if each alphabet exists in the substring



Permutation $O(nk^2)$

Find all possible substring of length k from s1, and then use a double for loop through s2 to check if each alphabet exists in the substring



Permutation $O(nk^2)$

```
bool is_permutation(const char *s1, const char *s2, size_t start) {
  size_t k = strlen(s1);
  size_t n = strlen(s2);
  char *copy = malloc(n+1);
  strncpy(copy, s2, n+1);

  for (size_t i = 0; i < k; i += 1) {
    for (size_t j = start; j < k + start; j += 1) {
      if (copy[j] == s1[i]) {
        copy[j] = '*';
        j = n;
      }
    }
  }
  for (size_t j = start; j < k + start; j += 1) {
    if (copy[j] != '*') {
      free(copy);
      return false;
    }
  }
  // cs1010_println_string(copy);
  free(copy);
  return true;
}
```

Permutation $O(nk)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring

N	E	T	N	U	S

N	U	S

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1														1												
Str2														1					1		1					

Permutation $O(nk)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring

N	E	T	N	U	S

N	U	S

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1					1									1												
Str2														1					1		1					

Permutation $O(nk)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring

N	E	T	N	U	S

N	U	S

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1					1									1						1						
Str2														1					1		1					

Permutation $0(nk)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring

N	E	T	N	U	S

N	U	S

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1					1																					
Str2														1					1		1					

Permutation $0(nk)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring

N	E	T	N	U	S

N	U	S

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1					1															1						
Str2														1					1		1					

Permutation $O(nk)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring

N	E	T	N	U	S

N	U	S

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1					1									1						1						
Str2														1					1		1					

Permutation $O(nk)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring

N	E	T	N	U	S

N	U	S

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1														1					1		1					
Str2														1					1		1					

Permutation $O(nk)$

```c
bool find_permutation(char *s1, char *s2) {
  long k = strlen(s1);
  long n = strlen(s2);

  long freq1[26];
  long freq2[26];

  build_frequency_array(k, s1, freq1);
  build_frequency_array(k, s2, freq2);

  if (is_permutation(freq1, freq2)) {
    return true;
  }

  for (long start = 1; start <= n - k; start += 1) {
    build_frequency_array(k, s2 + start, freq2);
    if (is_permutation(freq1, freq2)) {
      return true;
    }
  }
  return false;
}
```

```c
void build_frequency_array(long len, const char s[len], long freq[26]) {
  for (long i = 0; i < 26; i += 1) {
    freq[i] = 0;
  }
  for (long i = 0; i < len; i += 1) {
    freq[s[i]-'a'] += 1;
  }
}
```

```c
bool is_permutation(const long freq1[26], const long freq2[26]) {
  for (long i = 0; i < 26; i += 1) {
    if (freq1[i] != freq2[i]) {
      return false;
    }
  }
  return true;
}
```

Permutation $O(n + k)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring – but smarter

N	E	T	N	U	S

N	U	S

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1														1						1	1					
Str2														1					1		1					

Permutation $0(n + k)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring – but smarter

N	E	T	N	U	S

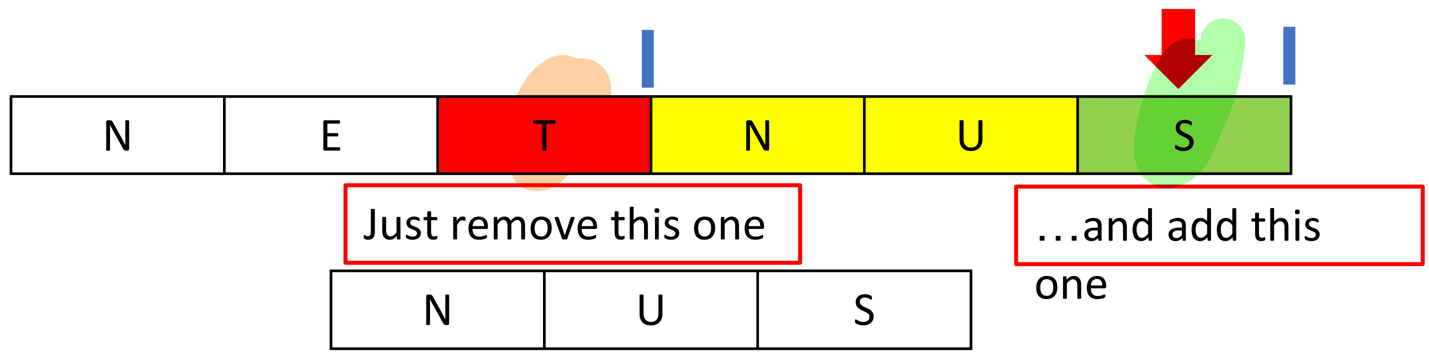
N	U	S

The frequency array for the second letter onwards will remain the same!

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1														1						1	1					
Str2														1					1		1					

Permutation $O(n+k)$

Find all possible substring of length k from s1, and then construct a frequency array from that substring – but smarter



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Str1														1					1	0	1					
Str2														1					1		1					

Permutation $O(n+k)$

```
bool find_permutation(char *s1, char *s2) {
  long k = strlen(s1);
  long n = strlen(s2);

  long freq1[26];
  long freq2[26];

  build_frequency_array(k, s1, freq1);
  build_frequency_array(k, s2, freq2);

  if (is_permutation(freq1, freq2)) {
    return true;
  }

  for (long start = 1; start <= n - k; start += 1) {
    update_frequency(freq2, s2, start, k);
    if (is_permutation(freq1, freq2)) {
      return true;
    }
  }
  return false;
}
```

```
void update_frequency(long freq2[26], const char *s2, int start, int k) {
  freq2[s2[start-1]-'a'] -= 1;
  freq2[s2[start+k-1]-'a'] += 1;
}
```