String

字串處理

Radix Sort

基數排序

Counting Sort

- 輸入 n 個範圍 $0 \sim k-1$ 的數字，將其排序後輸出

0	1	2	3	4	5
7	1	2	2	7	1

Counting Sort

- 輸入 n 個範圍 $0 \sim k-1$ 的數字，將其排序後輸出

0	1	2	3	4	5
7	1	2	2	7	1

Bucket

0	1	2	3	4	5	6	7
0	2	2	0	0	0	0	2

Counting Sort

- 輸入 n 個範圍 $0 \sim k-1$ 的數字，將其排序後輸出

0	1	2	3	4	5
7	1	2	2	7	1

Bucket

0	1	2	3	4	5	6	7
0	2	2	0	0	0	0	2

0	1	2	3	4	5
1	1	2	2	7	7

Counting Sort

```
vector<unsigned> counting_sort(const vector<unsigned> &Arr, unsigned k) {
  vector<unsigned> Bucket(k, 0);
  for (auto x : Arr)
    ++Bucket[x];
  vector<unsigned> Ans;
  for (unsigned x = 0; x < k; ++x)
    while (Bucket[x]--)
      Ans.emplace_back(x);
  return Ans;
}
```

不喜歡這樣的寫法，基本上只能用在數字上

標準的 Counting Sort

```
vector<unsigned> counting_sort(const vector<unsigned> &Arr, unsigned k) {
  vector<unsigned> Bucket(k, 0);
  for (auto x : Arr)
    ++Bucket[x];
  partial_sum(Bucket.begin(), Bucket.end(), Bucket.begin());
  vector<unsigned> Ans(Arr.size());
  for (auto Iter = Arr.rbegin(); Iter != Arr.rend(); ++Iter)
    Ans[--Bucket[*Iter]] = *Iter;
  return Ans;
}
```

標準的 Counting Sort

0	1	2	3	4	5
7	1	2	2	7	1

Bucket

0	1	2	3	4	5	6	7
0	2	2	0	0	0	0	2

標準的 Counting Sort

0	1	2	3	4	5
7	1	2	2	7	1

std::partial_sum

Bucket

0	1	2	3	4	5	6	7
0	2	4	4	4	4	4	6

Ans

0	1	2	3	4	5

標準的 Counting Sort

0	1	2	3	4	5
7	1	2	2	7	1

std::partial_sum

Bucket

0	1	2	3	4	5	6	7
0	1	4	4	4	4	4	6

Ans

0	1	2	3	4	5
	1				

標準的 Counting Sort

0	1	2	3	4	5
7	1	2	2	7	1

std::partial_sum

Bucket

0	1	2	3	4	5	6	7
0	1	4	4	4	4	4	5

Ans

0	1	2	3	4	5
	1				7

標準的 Counting Sort

0	1	2	3	4	5
7	1	2	2	7	1

std::partial_sum

Bucket

0	1	2	3	4	5	6	7
0	1	3	4	4	4	4	5

Ans

0	1	2	3	4	5
	1		2		7

標準的 Counting Sort

0	1	2	3	4	5
7	1	2	2	7	1

std::partial_sum

Bucket

0	1	2	3	4	5	6	7
0	1	2	4	4	4	4	5

Ans

0	1	2	3	4	5
	1	2	2		7

標準的 Counting Sort

0	1	2	3	4	5
7	1	2	2	7	1

std::partial_sum

Bucket

0	1	2	3	4	5	6	7
0	0	2	4	4	4	4	5

Ans

0	1	2	3	4	5
1	1	2	2		7

標準的 Counting Sort

0	1	2	3	4	5
7	1	2	2	7	1

std::partial_sum

Bucket

0	1	2	3	4	5	6	7
0	0	2	4	4	4	4	4

Ans

0	1	2	3	4	5
1	1	2	2	7	7

標準的 Counting Sort – 性質

- 可以排序非整數物件 (例如某個 struct 只需要對某個 member 排序)
- 它是個**穩定排序 (stable sort)**

Radix Sort

- 輸入 n 個範圍 $0\sim99$ 的數字，將其排序後輸出

0	1	2	3	4	5
76	15	23	27	71	14

Radix Sort

- 輸入 n 個範圍 0~99 的數字，將其排序後輸出

- 根據 counting sort 的 stable 性質
  可以先對個位數排序，再對十位數排序

0	1	2	3	4	5
76	15	23	27	71	14

→

0	1	2	3	4	5
71	23	14	15	76	27

Radix Sort

- 輸入 n 個範圍 0~99 的數字，將其排序後輸出

- 根據 counting sort 的 stable 性質
  可以先對個位數排序，再對十位數排序

0	1	2	3	4	5
76	15	23	27	71	14

0	1	2	3	4	5
71	23	14	15	76	27

0	1	2	3	4	5
14	15	23	27	71	76

只需要大小是 10 的額外空間！

2 位數 - Radix Sort

```
void bucket_sort(vector<unsigned> &Arr) {
  auto counting_sort = [&](auto getKey) {
    vector<unsigned> Bucket(10, 0);
    for (auto x : Arr)
      ++Bucket[getKey(x)];
    partial_sum(Bucket.begin(), Bucket.end(), Bucket.begin());
    vector<unsigned> Ans(Arr.size());
    for (auto Iter = Arr.rbegin(); Iter != Arr.rend(); ++Iter)
      Ans[--Bucket[getKey(*Iter)]] = move(*Iter);
    return Ans;
  };

  Arr = counting_sort([&](unsigned x) { return x % 10; });
  Arr = counting_sort([&](unsigned x) { return x / 10; });
}
```

將 counting-sort 的部分拆開來

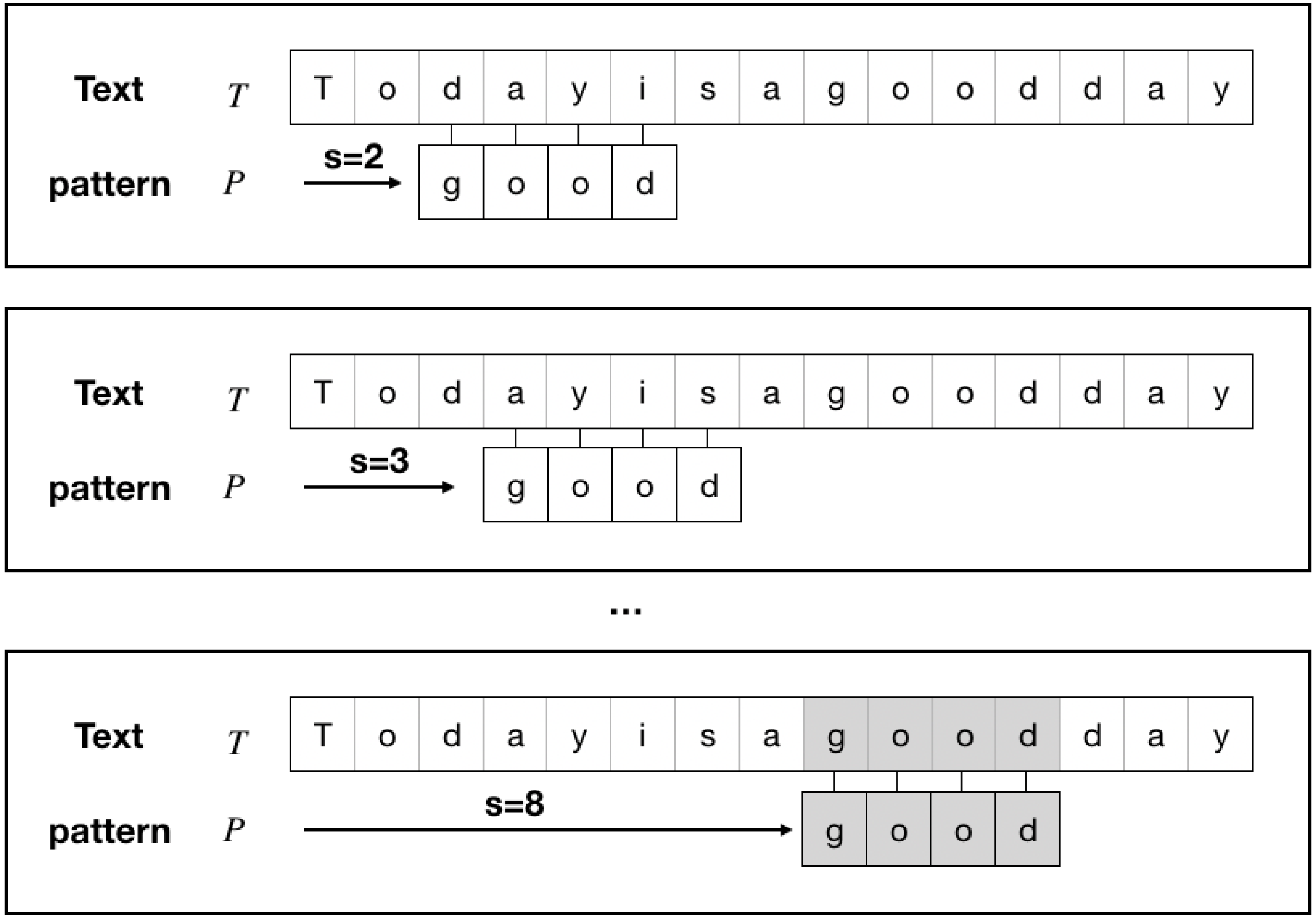
```cpp
template <class Ty, class BucketTy, class FuncTy>
void counting_sort(const Ty &Input, Ty &Output, BucketTy &Bucket,
                   FuncTy getKey) {
  fill(Bucket.begin(), Bucket.end(), 0);
  for (auto x : Input)
    ++Bucket[getKey(x)];
  partial_sum(Bucket.begin(), Bucket.end(), Bucket.begin());
  for (auto Iter = Input.rbegin(); Iter != Input.rend(); ++Iter)
    Output[--Bucket[getKey(*Iter)]] = move(*Iter);
}
```

unsigned - Radix Sort

```
void bucket_sort(vector<unsigned> &Arr) {
  vector<unsigned> Bucket(10, 0), Tmp(Arr.size());
  unsigned base = 1;
  for (unsigned T = 1; T <= 10; ++T) {
    counting_sort(Arr, Tmp, Bucket,
                  [base](unsigned x) { return (x / base) % 10; });
    Arr.swap(Tmp);
    base *= 10;
  }
}
```

字串匹配問題

給你兩個字串 $Text, Pattern$
問 $Pattern$ 是不是 $Text$ 的子字串



最簡單的暴力法

```
size_t matching(const string &text, const string &pattern) {
  for (size_t i = 0; i < text.size(); ++i) {
    bool match = true;
    for (size_t j = 0; j < pattern.size(); ++j)
      if (i + j >= text.size() || text[i + j] != pattern[j]) {
        match = false;
        break;
      }
    if (match) return i;
  }
  return std::string::npos;
}
```

最簡單的暴力法 – C++ string 內建 $O(n^2)$

```
size_t matching(const string &text, const string &pattern) {
  return text.find(pattern);
}
```

暴力法會變成 $O(n^2)$ 的例子

- $Text = aaaa \ldots aaaaaa$
- $Pattern = aaa \ldots aaab$
- $len(Pattern) = \left\lfloor \frac{len(Text)}{2} \right\rfloor$

小祕密 – GCC 實作的 strstr 是 $O(n)$ 的

```cpp
#include <cstring>
#include <iostream>
using namespace std;

int main() {
  char Tc[] = "This is C-style string with an Egg.";
  char Sc[] = "Egg";

  if (auto res = strstr(Tc, Sc); res != nullptr) {
    cout << "found " << Sc << " at " << res - Tc << '\n';
  }
  return 0;
}
```

Suffix Array

後綴數組

苜蓿芽 (alfalfa)

後綴 (Suffix)

0	1	2	3	4	5	6	7
a	a	b	a	a	a	a	b

Suffix

0	a	a	b	a	a	a	a	b
1	a	b	a	a	a	a	b	
2	b	a	a	a	a	b		
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
6	a	b						
7	b							

後綴數組 (Suffix Array, SA)

0	1	2	3	4	5	6	7
a	a	b	a	a	a	a	b

Suffix Sorted In Lexicographic Order

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Rank

0	1	2	3	4	5	6	7
3	5	7	0	1	2	4	6

後綴數組上二分搜

0	1	2	3	4	5	6	7
a	a	b	a	a	a	a	b

Suffix Sorted In Lexicographic Order

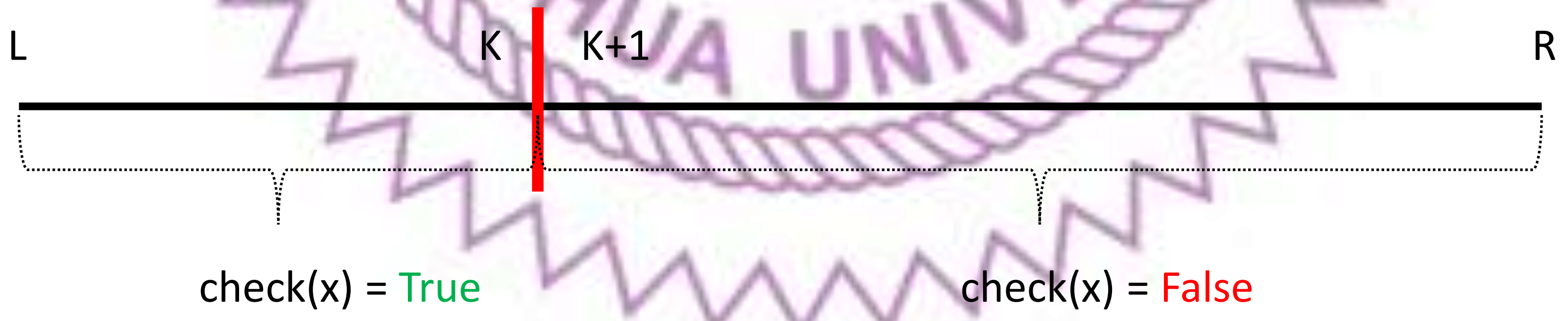
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

Pattern

a	b	a

複習：模板化的二分搜

```cpp
template <class Ty, class FuncTy>
pair<Ty, Ty> binarySearch(Ty L, Ty R, FuncTy check) {
  if (check(R) == true) return {R, R + 1};
  if (check(L) == false) return {L - 1, L};
  while (L + 1 < R) {
    Ty Mid = L + (R - L) / 2;
    if (check(Mid)) L = Mid;
    else R = Mid;
  }
  return {L, R};
}
```



後綴數組上二分搜 $O(|pattern| \log|text|)$

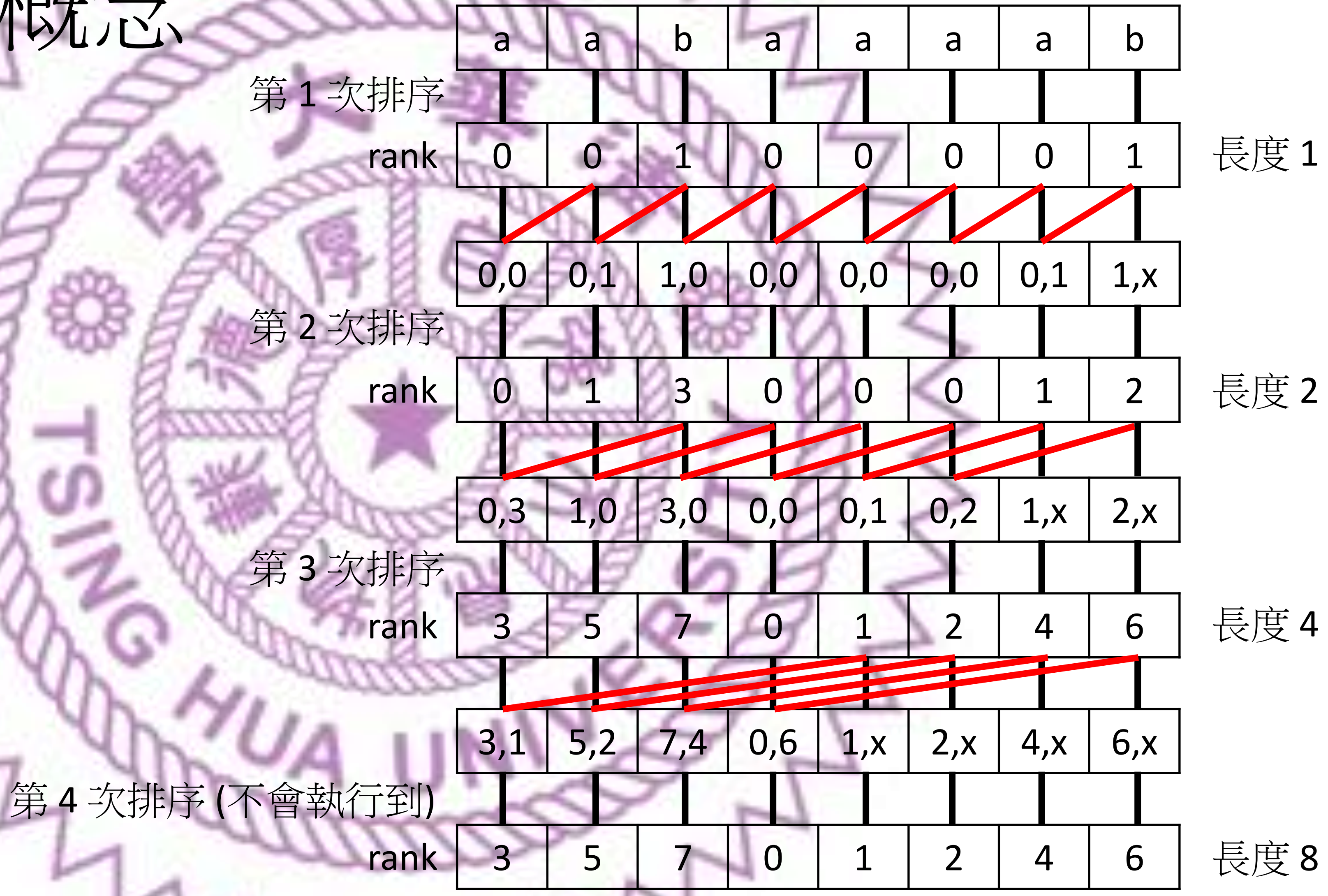
```
pair<vector<int>, vector<int>> buildSuffixArray(const string &text) {
  // TODO
}

size_t matching(const string &text, const string &pattern) {
  auto [SA, Rank] = buildSuffixArray(text);
  auto [L, R] = binarySearch(0, int(SA.size()) - 1, [&](int Idx) {
    return text.substr(SA[Idx]) < pattern;
  });
  if (R < SA.size() && pattern == text.substr(SA[R], pattern.size()))
    return SA[R];
  return std::string::npos;
}
```

構造後綴數組常見方法

- 倍增法：$O(n\log n)$
- DC3： $O(n)$
- SA_IS： $O(n)$

自己學放模板

倍增法基本概念



a a b a a a a b
第 1 次排序
rank 0 0 1 0 0 0 0 1
長度 1
0,0 0,1 1,0 0,0 0,0 0,0 0,1 1,x
第 2 次排序
rank 0 1 3 0 0 0 1 2
長度 2
0,3 1,0 3,0 0,0 0,1 0,2 1,x 2,x
第 3 次排序
rank 3 5 7 0 1 2 4 6
長度 4
3,1 5,2 7,4 0,6 1,x 2,x 4,x 6,x
第 4 次排序 (不會執行到)
rank 3 5 7 0 1 2 4 6
長度 8

倍增法實作

$k = 0$

SA

0	1	2	3	4	5	6	7

Index

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

Rank

0	1	2	3	4	5	6	7
‘a’	‘a’	‘b’	‘a’	‘a’	‘a’	‘a’	‘b’

0	a	a	b	a	a	a	a	b
1	a	b	a	a	a	a	b	
2	b	a	a	a	a	b		
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
6	a	b						
7	b							

倍增法實作

$k = 0$

SA

0	1	2	3	4	5	6	7
0	1	3	4	5	6	2	7

Sort by rank

Index

0	1	2	3	4	5	6	7
0	1	2	3	4	5	6	7

Rank

0	1	2	3	4	5	6	7
‘a’	‘a’	‘b’	‘a’	‘a’	‘a’	‘a’	‘b’

0	a	a	b	a	a	a	a	b
1	a	b	a	a	a	a	b	
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
6	a	b						
2	b	a	a	a	a	b		
7	b							

倍增法實作

$k = 1$

SA

0	1	2	3	4	5	6	7
0	1	3	4	5	6	2	7

update Index

Index

0	1	2	3	4	5	6	7
7	0	2	3	4	5	1	6

Rank

0	1	2	3	4	5	6	7
‘a’	‘a’	‘b’	‘a’	‘a’	‘a’	‘a’	‘b’

	0	1	2	3	4	5	6	7
	a	a	b	a	a	a	a	b
rank	0	0	1	0	0	0	0	1
	0,0	0,1	1,0	0,0	0,0	0,0	0,1	1,x

0	a	a	b	a	a	a	a	b
1	a	b	a	a	a	a	b	
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
6	a	b						
2	b	a	a	a	a	b		
7	b							

倍增法實作

$k = 1$

SA

0	1	2	3	4	5	6	7
0	1	3	4	5	6	2	7

update Index

Index

0	1	2	3	4	5	6	7
7	0	2	3	4	5	1	6

Rank

0	1	2	3	4	5	6	7
‘a’	‘a’	‘b’	‘a’	‘a’	‘a’	‘a’	‘b’

	0	1	2	3	4	5	6	7
	a	a	b	a	a	a	a	b
rank	0	0	1	0	0	0	0	1
	0,0	0,1	1,0	0,0	0,0	0,0	0,1	1,x

7	b							
0	a	a	b	a	a	a	a	b
2	b	a	a	a	a	b		
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
1	a	b	a	a	a	a	b	
6	a	b						

Update Index

```
template <class Ty>
void updateIndex(Ty &Index, const Ty &SA, int k) {
  int n = SA.size();
  Index.clear();
  for (int i = n - k; i < n; ++i)
    Index.emplace_back(i);
  for(auto x: SA)
    if (x >= k)
      Index.emplace_back(x - k);
}
```

倍增法實作

$k = 1$

Sort by rank

SA

0	1	2	3	4	5	6	7
0	3	4	5	1	6	7	2

Index

0	1	2	3	4	5	6	7
7	0	2	3	4	5	1	6

Rank

0	1	2	3	4	5	6	7
‘a’	‘a’	‘b’	‘a’	‘a’	‘a’	‘a’	‘b’

0	a	a	b	a	a	a	a	b
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

倍增法實作

$k = 1$

SA

0	1	2	3	4	5	6	7
0	3	4	5	1	6	7	2

Index

0	1	2	3	4	5	6	7
7	0	2	3	4	5	1	6

Rank

0	1	2	3	4	5	6	7
0	1	3	0	0	0	1	2

update Rank

0	a	a	b	a	a	a	a	b
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

Update Rank

```cpp
template <class Ty>
int updateRank(Ty &NewRank, Ty &Rank, const Ty &SA, int k) {
  int n = SA.size();
  int Idx = 0;
  NewRank[SA[0]] = 0;
  Rank.resize(n * 2, -1);
  auto Compare = [&](int a, int b) {
    return Rank[a] != Rank[b] || Rank[a + k] != Rank[b + k];
  };
  for (int i = 1; i < n; ++i)
    NewRank[SA[i]] = (Idx += Compare(SA[i - 1], SA[i]));
  return Idx + 1; // size of Bucket
}
```

Update Rank

```
template <class Ty>
int updateRank(Ty &NewRank, const Ty &Rank, const Ty &SA, int k) {
  int n = SA.size();
  int Idx = 0;
  NewRank[SA[0]] = 0;
  auto Compare = [&](int a, int b) {
    return Rank[a] != Rank[b] || a + k >= n || Rank[a + k] != Rank[b + k];
  };
  for (int i = 1; i < n; ++i)
    NewRank[SA[i]] = (Idx += Compare(SA[i - 1], SA[i]));
  return Idx + 1; // size of Bucket
}
```

倍增法實作

$k = 2$

update Index

SA	0	1	2	3	4	5	6	7
	0	3	4	5	1	6	7	2

Index	0	1	2	3	4	5	6	7
	6	7	1	2	3	4	5	0

Rank	0	1	2	3	4	5	6	7
	0	1	3	0	0	0	1	2

	0	1	2	3	4	5	6	7
	a	a	b	a	a	a	a	b
rank	0	1	3	0	0	0	1	2
	0,3	1,0	3,0	0,0	0,1	0,2	1,x	2,x

0	a	a	b	a	a	a	a	b
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

倍增法實作

$k = 2$

SA

0	1	2	3	4	5	6	7
0	3	4	5	1	6	7	2

update Index

Index

0	1	2	3	4	5	6	7
6	7	1	2	3	4	5	0

Rank

0	1	2	3	4	5	6	7
0	1	3	0	0	0	1	2

	0	1	2	3	4	5	6	7
	a	a	b	a	a	a	a	b
rank	0	1	3	0	0	0	1	2
	0,3	1,0	3,0	0,0	0,1	0,2	1,x	2,x

6	a	b						
7	b							
1	a	b	a	a	a	a	b	
2	b	a	a	a	a	b		
3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b

倍增法實作

$k = 2$

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Sort by rank

Index

0	1	2	3	4	5	6	7
6	7	1	2	3	4	5	0

Rank

0	1	2	3	4	5	6	7
0	1	3	0	0	0	1	2

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

倍增法實作

$k = 2$

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Index

0	1	2	3	4	5	6	7
6	7	1	2	3	4	5	0

Rank

0	1	2	3	4	5	6	7
3	5	7	0	1	2	4	6

update Rank

每個後綴的 rank 都不一樣就可以結束了

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

倍增法程式碼

```
pair<vector<int>, vector<int>> buildSuffixArray(const string &text) {
  int n = text.size(), BucketSize = 1 << (sizeof(char) * 8);
  vector<int> SA(n), Rank(n), Index(n), Bucket(max(n + 1, BucketSize));
  for (int i = 0; i < n; ++i)
    Rank[Index[i] = i] = text[i];
  counting_sort(Index, SA, Bucket, [&](int x) { return Rank[x]; });
  for (int k = 1;; k *= 2) {
    updateIndex(Index, SA, k);
    counting_sort(Index, SA, Bucket, [&](int x) { return Rank[x]; });
    Rank.swap(Index);
    BucketSize = updateRank(Rank, Index, SA, k);
    if (BucketSize >= n) break;
    Bucket.resize(BucketSize);
  }
  return {SA, Rank};
}
```

最常共同前綴 (Longest Common Prefix, LCP)

- S = "abcdefgh"
- T = "abcefgh"
- $lcp(S, T)$ = "abc"

高度數組 (Height)

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Height

0	1	2	3	4	5	6	7
0							

定義：$Height[i] = lcp(SA[i], SA[i-1])$
$Height[0] = 0$

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

高度數組 (Height)

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Height

0	1	2	3	4	5	6	7
0	3						

定義：
$$Height[i] = lcp(SA[i], SA[i-1])$$
$$Height[0] = 0$$

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

高度數組 (Height)

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Height

0	1	2	3	4	5	6	7
0	3	2					

定義：$Height[i] = lcp(SA[i], SA[i-1])$
$Height[0] = 0$

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

高度數組 (Height)

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Height

0	1	2	3	4	5	6	7
0	3	2	3				

定義：
$$Height[i] = lcp(SA[i], SA[i-1])$$
$$Height[0] = 0$$

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

高度數組 (Height)

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Height

0	1	2	3	4	5	6	7
0	3	2	3	1			

定義：$Height[i] = lcp(SA[i], SA[i-1])$
$Height[0] = 0$

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

高度數組 (Height)

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Height

0	1	2	3	4	5	6	7
0	3	2	3	1	2		

定義：$Height[i] = lcp(SA[i], SA[i-1])$
$Height[0] = 0$

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

高度數組 (Height)

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Height

0	1	2	3	4	5	6	7
0	3	2	3	1	2	0	

定義：$Height[i] = lcp(SA[i], SA[i-1])$
$Height[0] = 0$

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

高度數組 (Height)

SA

0	1	2	3	4	5	6	7
3	4	5	0	6	1	7	2

Height

0	1	2	3	4	5	6	7
0	3	2	3	1	2	0	1

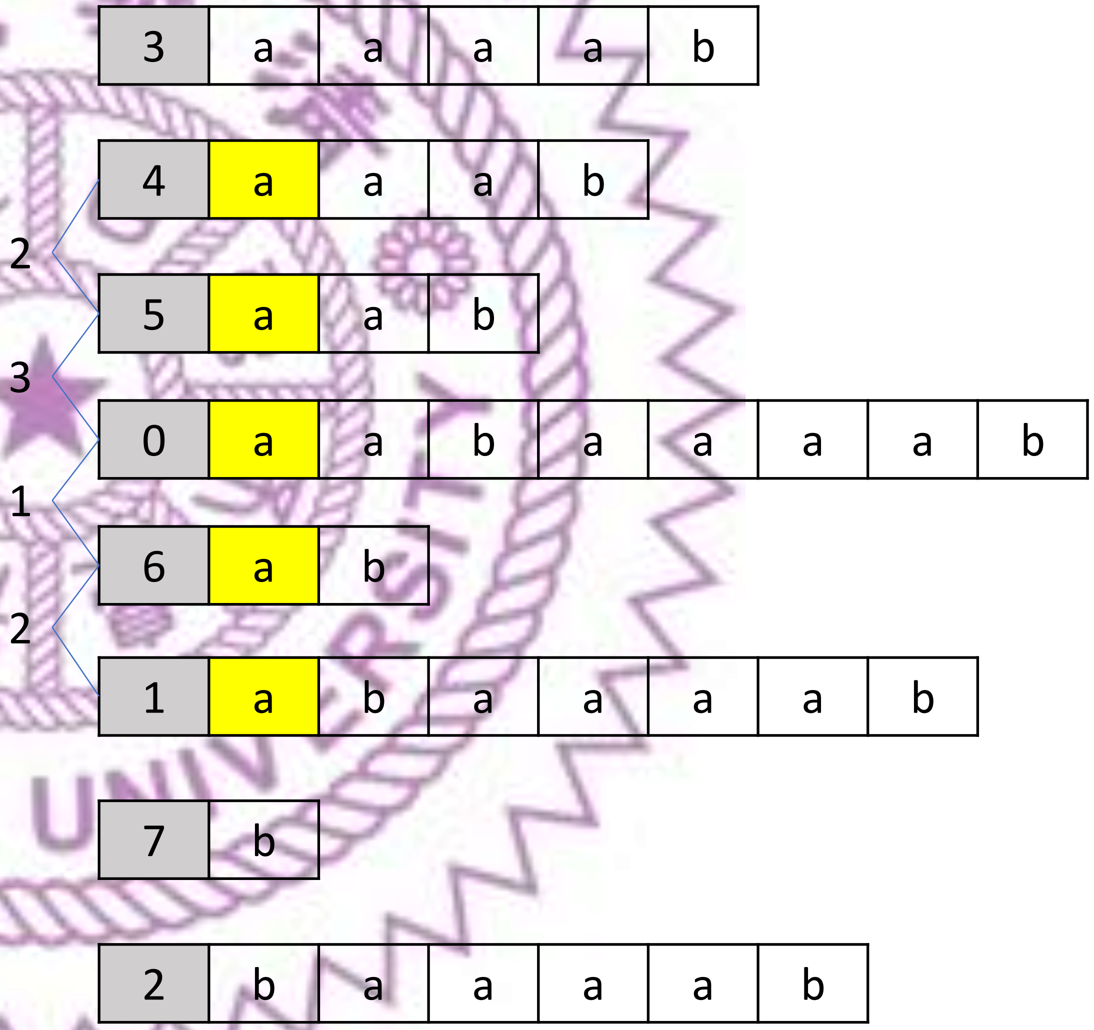
定義：$Height[i] = lcp(SA[i], SA[i-1])$
$Height[0] = 0$

3	a	a	a	a	b			
4	a	a	a	b				
5	a	a	b					
0	a	a	b	a	a	a	a	b
6	a	b						
1	a	b	a	a	a	a	b	
7	b							
2	b	a	a	a	a	b		

高度數組 (Height) – 性質

SA	0	1	2	3	4	5	6	7
	3	4	5	0	6	1	7	2

Height	0	1	2	3	4	5	6	7
	0	3	2	3	1	2	0	1



$$lcp(SA[i], SA[j]) = \min_{i \le k \le j}\{Height[k]\}$$

RMQ 有 $O(1)$ 作法

定理

$$Height\left[Rank[i]\right] \geq Height\left[Rank[i-1]\right] - 1$$

- 證明很複雜 https://oi-wiki.org/string/sa/#on-%E6%B1%82-height-%E6%95%B0%E7%BB%84%E9%9C%80%E8%A6%81%E7%9A%84%E4%B8%80%E4%B8%AA%E5%BC%95%E7%90%86

構造高度數組 $O(n)$

k 不會超過 n，最多被減 n 次

```
template <class Ty>
vector<int> buildHeight(const string &text, const Ty &SA, const Ty &Rank) {
  int n = SA.size(), k = 0;
  vector<int> Height(n, 0);
  for (int i = 0; i < n; ++i) {
    if (Rank[i] == 0) continue;
    if (k) --k;
    while (text[i + k] == text[SA[Rank[i] - 1] + k]) ++k;
    Height[Rank[i]] = k;
  }
  return Height;
}
```

高度數組好處

- 原本的字串匹配複雜度 $O(|pattern| \times \log|text|)$
- 透過高度數組 $O(|pattern| + \log|text|)$
- Udi Manber*. Gene Myers# (1990).
  Suffix arrays: a new method for on-line string searches.

乘法變加法
自己試試看實作

最長共同子字串(Longest Common Substring)

- A = aaaba, B = abaa
- A, B 的最長共同子字串是 aba

利用後綴數組

	5	$	a	b	a	a					
	9	a									
1											
	4	a	$	a	b	a	a				
1											
	8	a	a								
2											
	0	a	a	a	b	a	$	a	b	a	a
	1	a	a	b	a	$	a	b	a	a	
	2	a	b	a	$	a	b	a	a		
3											
	6	a	b	a	a						
0											
	3	b	a	$	a	b	a	a			
2											
	7	b	a	a							

a	a	a	b	a

a	b	a	a

0	1	2	3	4	5	6	7	8	9
a	a	a	b	a	$	a	b	a	a

某個沒出現過的符號

LCS 會相鄰，直接在 Height 上面找就行

Longest Common Substring

```
string getLongestCommonSubstring(const string &A, const string &B) {
  string S = A + "$" + B;
  auto [SA, Rank] = buildSuffixArray(S);
  auto Height = buildHeight(S, SA, Rank);
  int nA = A.size(), Ans = 0, Idx = -1;
  for (size_t i = 1; i < Height.size(); ++i)
    if (Height[i] > Ans)
      if ((nA < SA[i - 1] && SA[i] < nA) || (nA > SA[i - 1] && SA[i] > nA))
        Ans = Height[Idx = i];
  if (Idx != -1)
    return S.substr(SA[Idx], Ans);
  return "";
}
```

最小表示法 (Minimal String Rotation)

- $S = \mathrm{bacd}$

- Rotation of S:
  - bacd
  - $\boxed{\mathrm{acdb}}$
  - cdba
  - dbac

Minimal String Rotation

利用後綴數組

- 設 $S' = SS$
- 求出 S' 的後綴數組 SA
- 找出最小的 $SA[k]$ 使得 $0 \le SA[k] < |S|$
- $S'[SA[k], SA[k] + |S| - 1]$ 就是答案

	0	1	2	3				
S'	b	a	c	d	b	a	c	d

SA	5	1	4	0	6	2	7	3

最長迴文子字串 (Longest Palindromic Substring)

- S = aabaaaab
- S 的最長迴文子字串是 baaaab

```cpp
string getLongestPalindromicSubstring(const string &S) {
  auto S2 = S;
  std::reverse(S2.begin(), S2.end());
  return getLongestCommonSubstring(S, S2);
}
```

專門的 $O(n)$ 演算法

- 最小表示法：
  - String Booth's Algorithm

- 最長迴文子字串
  - Manacher Algorithm

KMP 演算法

Knuth-Morris-Pratt

Mississippi

基本概念：共同前後綴

在 T 中尋找 P ，兩字串已知共同前綴部分為 R (橘色部分)

T:	a	b	a	a	b	a	.	.	.	.	.	.
P:	a	b	a	a	b	c						

若 R 中存在一個不等於 R 的前綴等於後綴 Q (綠色部分)

a	b	a	a	b

T:	a	b	a	a	b	a	.	.	.	.	.	.
P:				a	b	a	a	b	c			

把 P 往後移動 $|R| - |Q|$ 的距離也有機會產生匹配

次長共同前後綴

- 共同前後綴有非常多個，我們要找盡量長的
- 字串 S 的最長共同前後綴等同於 S，所以沒有討論意義
- 所以要找次長的那個做為移動依據

a	b	a	a	b	a

a	b	a	a	b	a

a	b	a	a	b	a

前缀函数

- 給定字串 S，定義前缀函数 $\pi(i)$

$$\begin{cases} \pi(i) = S[0:i] \text{ 的次長共同前後綴終點位置} \\ \pi(0) = -1 \end{cases}$$

	0	1	2	3	4	5	6	7
	a	b	c	a	a	b	c	d
π	-1	-1	-1	0	0	1	2	-1

前缀函数

- 給定字串 S，定義前缀函数 $\pi(i)$

$$\begin{cases} \pi(i) = S[0:i] \text{ 的次長共同前後綴終點位置} \\ \pi(0) = -1 \end{cases}$$

	0	1	2	3	4	5	6	7
	a	b	c	a	a	b	c	d
π	-1	-1	-1	0	0	1	2	-1

前缀函数

- 給定字串 S，定義前缀函数 $\pi(i)$

$$\begin{cases} \pi(i) = S[0{:}\,i] \text{ 的次長共同前後綴終點位置} \\ \pi(0) = -1 \end{cases}$$

	0	1	2	3	4	5	6	7
	a	b	c	a	a	b	c	d
π	-1	-1	-1	0	0	1	2	-1

前缀函数 – 動態規劃

- Case 1:

$$\pi(i-1) = -1$$

- 直接比較 $S[0], S[i]$

由於 $\pi(8) = -1$
$\pi(9)$ 要用 $S[9]$ 跟 $S[0]$ 做比較

	0	1	2	3	4	5	6	7	8	9
	A	B	C	X	X	X	A	B	Z	A
π	-1	-1	-1	-1	-1	-1	0	1	-1	

前缀函数 – 動態規劃

- Case 1:

$$\pi(i-1) = -1$$

- 直接比較 $S[0], S[i]$

由於 $\pi(8) = -1$
$\pi(9)$ 要用 $S[9]$ 跟 $S[0]$ 做比較

	0	1	2	3	4	5	6	7	8	9
	A	B	C	X	X	X	A	B	Z	A
π	-1	-1	-1	-1	-1	-1	0	1	-1	0

前缀函数 – 動態規劃

- Case 2:

$$S[i] = S[\pi(i-1)+1]$$

- $\pi(i) = \pi(i-1) + 1$

	0	1	2	3	4	5	6	7	8	9
	A	B	C	D	X	X	A	B	C	D
π	-1	-1	-1	-1	-1	-1	0	1	2	

前缀函数 – 動態規劃

- Case 2:

$$S[i] = S[\pi(i-1)+1]$$

- $\pi(i) = \pi(i-1)+1$

	0	1	2	3	4	5	6	7	8	9
	A	B	C	D	X	X	A	B	C	D
π	-1	-1	-1	-1	-1	-1	0	1	2	3

前缀函数 – 動態規劃

- Case 3:

$$S[i] \neq S[\pi(i-1)+1]$$

- 透過 $S[0\sim\pi(i-1)]$ 的次長共同前後綴繼續檢查直到產生 case 1 或 case 2 為止

j

	0	1	2	3	4	5	6	7	8	9	10	11
	A	B	C	A	B	D	A	B	C	A	B	A
π	-1	-1	-1	0	1	-1	0	1	2	3	4	

前缀函数 – 動態規劃

- Case 3:

$$S[i] \neq S[\pi(i-1)+1]$$

- 透過 $S[0 \sim \pi(i-1)]$ 的次長共同前後綴繼續檢查直到產生 case 1 或 case 2 為止

j

	0	1	2	3	4	5	6	7	8	9	10	11
	A	B	C	A	B	D	A	B	C	A	B	A
π	-1	-1	-1	0	1	-1	0	1	2	3	4	

前缀函数 – 動態規劃

- Case 3:

$$S[i] \neq S[\pi(i-1)+1]$$

- 透過 $S[0\sim\pi(i-1)]$ 的次長共同前後綴繼續檢查直到產生 case 1 或 case 2 為止

j

	0	1	2	3	4	5	6	7	8	9	10	11
	A	B	C	A	B	D	A	B	C	A	B	A
π	-1	-1	-1	0	1	-1	0	1	2	3	4	

前缀函数 – 動態規劃

- Case 3:

$$S[i] \neq S[\pi(i-1)+1]$$

- 透過 $S[0 \sim \pi(i-1)]$ 的次長共同前後綴繼續檢查直到產生 case 1 或 case 2 為止

j

	0	1	2	3	4	5	6	7	8	9	10	11
	A	B	C	A	B	D	A	B	C	A	B	A
π	-1	-1	-1	0	1	-1	0	1	2	3	4	

前缀函数 – 動態規劃

- Case 3:

$$S[i] \neq S[\pi(i-1)+1]$$

- 透過 $S[0 \sim \pi(i-1)]$ 的次長共同前後綴繼續檢查直到產生 case 1 或 case 2 為止

j

	0	1	2	3	4	5	6	7	8	9	10	11
	A	B	C	A	B	D	A	B	C	A	B	A
π	-1	-1	-1	0	1	-1	0	1	2	3	4	0

前綴函數 – 程式碼

```
vector<int> buildPi(const string &S) {
  vector<int> Pi(S.size());
  Pi[0] = -1;
  for (size_t i = 1; i < S.size(); ++i) {
    int j = Pi[i - 1];
    while (j != -1 && S[i] != S[j + 1]) // case 3
      j = Pi[j];
    if (S[i] == S[j + 1]) Pi[i] = j + 1; // case 2
    else Pi[i] = -1; // case 1
  }
  return Pi;
}
```

前綴函數 – $O(n)$

另一種把 j 放外面的寫法

```
vector<int> buildPi(const string &S) {
  vector<int> Pi(S.size());
  int j = -1;
  Pi[0] = j;
  for (size_t i = 1; i < S.size(); ++i) {
    while (j != -1 && S[i] != S[j + 1]) // case 3
      j = Pi[j];
    if (S[i] == S[j + 1]) Pi[i] = j += 1; // case 2
    else Pi[i] = j = -1; // case 1
  }
  return Pi;
}
```

- while 最多執行 j 次
- while 每次執行 j 都會變小
- j 只會在 case 2 那行增加
- j 只會被操作 $O(n)$ 次

找出所有匹配的位置 $O(n)$

pattern

p	p	a	p

text

p	p	p	a	p	a

某個沒出現過的符號

	0	1	2	3	4	5	6	7	8	9	10
	p	p	a	p	$	p	p	p	a	p	a
π	-1	0	-1	0	-1	0	1	1	2	3	-1

找出所有匹配的位置 $O(n)$

pattern

p	p	a	p

text

p	p	p	a	p	a

某個沒出現過的符號

	0	1	2	3	4	5	6	7	8	9	10
	p	p	a	p	$	p	p	p	a	p	a
π	-1	0	-1	0	-1	0	1	1	2	3	-1

找出所有匹配的位置 $O(n)$

```
vector<size_t> matching_all(const string &text, const string &pattern) {
  string S = pattern + '$' + text;
  size_t nP = pattern.size();
  auto Pi = buildPi(S);
  vector<size_t> ans;
  for (size_t i = nP + 1; i < S.size(); i++) {
    if (Pi[i] + 1 == nP)
      ans.push_back(i - 2 * nP);
  }
  return ans;
}
```

空間更少的做法

```
vector<size_t> matching_all(const string &text, const string &pattern) {
  auto Pi = buildPi(pattern);
  vector<size_t> ans;
  int j = -1;
  for (size_t i = 0; i < text.size(); ++i) {
    while (j != -1 && pattern[j + 1] != text[i])
      j = Pi[j];
    if (pattern[j + 1] == text[i]) ++j;
    if (j + 1 == pattern.size()) {
      ans.emplace_back(i + 1 - pattern.size());
      j = Pi[j];
    }
  }
  return ans;
}
```

UVa 11475

- 給你一個字串 S
- 問這要在 S “結尾”最少增加幾個字，才能使 S 變成迴文
- Ex: $S = \text{cabaa}$
  加入 bac ➔ cabaabac 變成迴文

想法

c	a	b	a	a

reverse normal

	0	1	2	3	4	5	6	7	8	9	10
	a	a	b	a	c	$	c	a	b	a	a
π	-1	0	-1	0	-1	-1	-1	0	-1	0	1

字串最小週期

- 若存在某個字串 R 使得字串 $S = RRR \ldots RR$ 則稱 R 為 S 的週期

- 輸入 S ，請找出 S 長度最小的週期

想法

	0	1	2	3	4	5	6	7	8	9	10	11
	a	b	a	a	b	a	a	b	a	a	b	a
π	-1	-1	0	0	1	2	3	4	5	6	7	8

```
string getMinPeriod(const string &S) {
  auto Pi = buildPi(S);
  size_t PeriodLen = S.size() - Pi.back() - 1;
  if (S.size() % PeriodLen != 0)
    return S;
  return S.substr(0, PeriodLen);
}
```

Z Algorithm

Gusfield’s Algorithm

banana

拓展 KMP

- 與 KMP 的 π 定義很像，但不一樣

	0	1	2	3	4	5	6
	A	B	A	B	A	A	B
Z	0	0	3	0	1	2	0

- 給定字串 S
- 定義

$$\begin{cases} Z[i] = lcp(S[i:|S|-1], S) \\ Z[0] = 0 \end{cases}$$

$Z[2] = 3$，因為 $S[2\sim6]$ = ABAAB
與整個字串的開頭 3 個字 ABABAAB 一樣

Z Algorithm

- 為了快速計算 Z ，我們維護一個範圍 $[L, R]$
- 該範圍滿足 $S[L \sim R] = S[0: R - L]$

			L		R		
	0	1	2	3	4	5	6
	A	B	A	B	A	A	B
Z	0	0	3	0	1	2	0

Z Algorithm

- 為了快速計算 Z ，我們維護一個範圍 $[L, R]$
- 該範圍滿足 $S[L{\sim}R] = S[0{:}R-L]$

			L		R		
	0	1	2	3	4	5	6
	A	B	A	B	A	A	B
Z	0	0	3	0	1	2	0

Case 1: i 不再 $[L, R]$ 範圍內

- 當前的 L, R 完全沒有用，直接暴力求答案
- 並把 L, R 更新成找到的範圍

```cpp
vector<int> buildZ(const string &S) {
  int L = 0, R = 0, n = S.size();
  vector<int> Z(n);
  for (int i = 1; i < n; ++i) {
    if (R < i) { // Case 1
      while (S[Z[i]] == S[i + Z[i]])
        ++Z[i];
      L = i, R = i + Z[i] - 1;
    } else {
      // TODO
    }
  }
  return Z;
}
```

Case 2: $L \le i \le R$ 且 $i + Z[i - L] - 1 < R$

$$Z[i] = Z[i - L]$$

根據定義，橘色區域會一樣

Case 2: $L \leq i \leq R$ 且 $i + Z[i-L] - 1 < R$

$$Z[i] = Z[i-L]$$

根據定義，橘色區域會一樣

根據 $Z[i-L]$ 可知綠色部分也相等

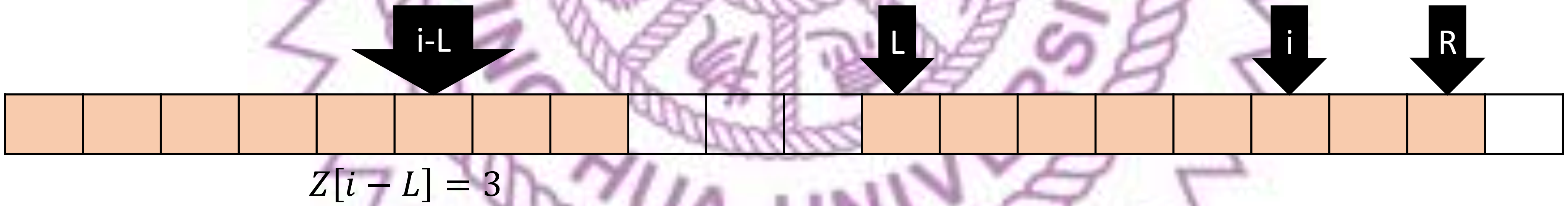
$Z[i-L] = 2$

Case 2: $L \leq i \leq R$ 且 $i + Z[i - L] - 1 < R$

```cpp
vector<int> buildZ(const string &S) {
  int L = 0, R = 0, n = S.size();
  vector<int> Z(n);
  for (int i = 1; i < n; ++i) {
    if (R < i) { // Case 1
      while (S[Z[i]] == S[i + Z[i]])
        ++Z[i];
      L = i, R = i + Z[i] - 1;
    } else if (i + Z[i - L] - 1 < R) { // Case 2
      Z[i] = Z[i - L];
    } else {
      // TODO
    }
  }
  return Z;
}
```

Case 3: $L \leq i \leq R$ 且 $i + Z[i-L] - 1 \geq R$

根據定義，橘色區域會一樣



Case 3: $L \leq i \leq R$ 且 $i + Z[i-L] - 1 \geq R$

- 和 Case 1 一樣的操作，但是綠色部分不用檢查
- 因為綠色部分一樣，所以 $Z[i] \geq R - i + 1$

根據定義，橘色區域會一樣

根據 $Z[i-L]$ 可知綠色部分也相等

$Z[i-L] = 3$

Z Algorithm

重複了

```
vector<int> buildZ(const string &S) {
  int L = 0, R = 0, n = S.size();
  vector<int> Z(n);
  for (int i = 1; i < n; ++i) {
    if (R < i) { // Case 1
      Z[i] = 0;
      while (S[Z[i]] == S[i + Z[i]])
        ++Z[i];
      L = i, R = i + Z[i] - 1;
    } else if (i + Z[i - L] - 1 < R) { // Case 2
      Z[i] = Z[i - L];
    } else { // Case 3
      Z[i] = R - i + 1;
      while (S[Z[i]] == S[i + Z[i]])
        ++Z[i];
      L = i, R = i + Z[i] - 1;
    }
  }
  return Z;
}
```

Z Algorithm

```
vector<int> buildZ(const string &S) {
  int L = 0, R = 0, n = S.size();
  vector<int> Z(n);
  for (int i = 1; i < n; ++i) {
    if (i <= R && i + Z[i - L] - 1 < R) { // Case 2
      Z[i] = Z[i - L];
    } else { // Case 1 & 3
      Z[i] = max(0, R - i + 1);
      while (S[Z[i]] == S[i + Z[i]])
        ++Z[i];
      L = i, R = i + Z[i] - 1;
    }
  }
  return Z;
}
```

要小心會有負數
不能用 unsigned

Z Algorithm – 時間複雜度

```cpp
vector<int> buildZ(const string &S) {
  int L = 0, R = 0, n = S.size();
  vector<int> Z(n);
  for (int i = 1; i < n; ++i) {
    if (i <= R && i + Z[i - L] - 1 < R) { // Case 2
      Z[i] = Z[i - L];
    } else { // Case 1 & 3
      Z[i] = max(0, R - i + 1);
      while (S[Z[i]] == S[i + Z[i]])
        ++Z[i];
      L = i, R = i + Z[i] - 1;
    }
  }
  return Z;
}
```

- 每次執行 while R 都會增加
- R 最多只會是 n
- while 只會執行 n 次
- $O(n)$

找出所有匹配的位置 $O(n)$

pattern

p	p	a	p

text

p	p	p	a	p	a

某個沒出現過的符號

	0	1	2	3	4	5	6	7	8	9	10
	p	p	a	p	$	p	p	p	a	p	a
Z	0	1	0	1	0	2	4	1	0	1	0

CSES - Finding Periods

- 若存在某個字串 R 使得字串 $S = RRR \dots RR$
  則稱 R 為 S 的週期，注意此題最一次重複可以是 R 的前綴
- 目標是找出所有的 R

- Ex: $S = \mathrm{abcabca}$ ，合法的 R 有：
- abc
- abcabc
- $\mathrm{abcabca}$

觀察

	0	1	2	3	4	5	6
	a	b	c	a	b	c	a
Z	0	0	0	4	0	0	1

	0	1	2	3	4	5	6
	a	b	c	a	b	c	a
Z	0	0	0	4	0	0	1

CSES - Finding Periods

```
vector<int> findingPeriods(const string &S) {
  auto Z = buildZ(S);
  vector<int> Ans;
  for (size_t i = 0; i < Z.size(); ++i) {
    if (Z[i] == Z.size() - i)
      Ans.emplace_back(i);
  }
  Ans.emplace_back(Z.size());
  return Ans;
}
```

Rabin-Karp Rolling Hash

滾動雜湊

Roll a Hash Joint

滾動雜湊

- 設 $S = s_0 s_1 \ldots s_{n-1}$，給定兩個質數 P, M
- 定義

$$H(S,P,M) = (s_0 P^{n-1} + s_1 P^{n-2} + \cdots + s_{n-1}) \% M$$

滾動雜湊

- 設 $S = s_0 s_1 \dots s_{n-1}$ 給定兩個質數 P, M
- 定義

$$H(S, P, M) = (s_0 P^{n-1} + s_1 P^{n-2} + \cdots + s_{n-1}) \% M$$

- 設 $S' = s_0 s_1 \dots s_{n-1} s_n$

$$\begin{aligned} H(S', P, M) &= (s_0 P^{n} + s_1 P^{n-1} + \cdots + s_{n-1} P + s_n) \% M \\ &= (H(S, P, M) \times P + s_n) \% M \end{aligned}$$

程式碼

- 建議在 code book 裡面紀錄一些 int, long long 範圍質數的表
- 很多演算法需要質數的幫忙

```
vector<long long> rollingHash(const string &S, long long prime,
                              long long prime_mod) {
  vector<long long> H(S.size() + 1);
  for (size_t i = 1; i <= S.size(); ++i) {
    H[i] = (H[i - 1] * prime + S[i - 1]) % prime_mod;
  }
  return H;
}
```

子字串的 hash

- 設 $S' = s_L s_{L+1} s_{L+2} \dots s_R$
- 設 $S^R = s_0 s_1 s_2 \dots s_R$
- 設 $S^{L-1} = s_0 s_1 s_2 \dots s_{L-1}$

$$H(S', P, M) = (s_L P^{R-L} + s_{L+1} P^{R-L-1} + \cdots + s_R)\%M$$
$$H(S^R, P, M) = (s_0 P^R + s_1 P^{R-1} + \cdots + s_R)\%M$$
$$H(S^{L-1}, P, M) = (s_0 P^{L-1} + s_1 P^{L-2} + \cdots + s_{L-1})\%M$$

子字串的 hash

- 設 $S' = s_L s_{L+1} s_{L+2} \dots s_R$
- 設 $S^R = s_0 s_1 s_2 \dots s_R$
- 設 $S^{L-1} = s_0 s_1 s_2 \dots s_{L-1}$

$$H(S',P,M) = (s_L P^{R-L} + s_{L+1} P^{R-L-1} + \cdots + s_R)\%M$$

$$H(S^R,P,M) = (s_0 P^R + s_1 P^{R-1} + \cdots + s_R)\%M$$

$$P^{R-L+1} \times H(S^{L-1},P,M) = (s_0 P^R + s_1 P^{R-1} + \cdots + s_{L-1} P^{R-L+1})\%M$$

$$\boxed{H(S',P,M) = \left(H(S^R,P,M) - P^{R-L+1} \times H(S^{L-1},P,M)\right)\%M}$$

取得子字串的 hash $O(1)$

```
vector<long long> buildBase(int n, long long prime, long long prime_mod) {
  vector<long long> Base(n + 1);
  Base[0] = 1;
  for (size_t i = 1; i < Base.size(); ++i)
    Base[i] = Base[i - 1] * prime % prime_mod;
  return Base;
}
```

```
long long getSubStrHash(const vector<long long> &H,
                        const vector<long long> &Base, int L, int R,
                        long long prime, long long prime_mod) {
  return (H[R + 1] - (H[L] * Base[R - L + 1]) % prime_mod + prime_mod) %
         prime_mod;
}
```

唬爛法

- 若兩字串 A, B
- $H(A, P, M) = H(B, P, M)$
- 則我們認定 $A = B$

碰撞問題

- 碰撞定義：

$$H(A,P,M) = H(B,P,M), A \neq B$$
$$H(A,P,M) \neq H(B,P,M), A = B$$

- 若不考慮 modulo，H 函數相當於是將字串編碼成 P 進位的整數

- 因此碰撞只跟取餘數 M 有關，單次比較碰撞機率為：

$$\frac{1}{M}$$

碰撞問題

- 如果進行 k 次比較，希望有 90 % 的正確率， M 要多大?

$$\left(1-\frac{1}{M}\right)^k \geq 0.9$$

$$M \geq \frac{1}{1-0.9^{\frac{1}{k}}}$$

- 有人推了近似公式： $M \geq 5k^2$
- 如果還是不夠用，就用多個 $M_1, M_2, \ldots$ 做 Hash
  比較用 tuple 包起來一起比較

特殊質數

- 不要用特殊質數，例如 2^k-1 的質數

- 很容易可以設計出碰撞的測資

Trie

字典樹

儲存、查找字串

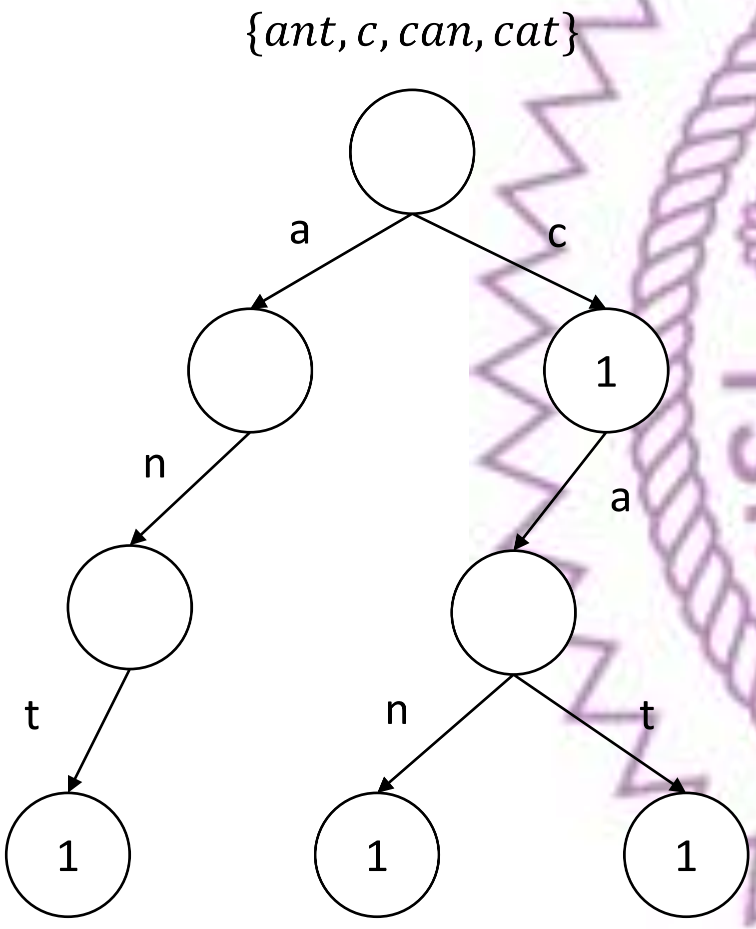
```
int main() {
  set<string> ST;
  ST.emplace("ant");
  ST.emplace("can");
  ST.emplace("cat");
  cout << ST.count("can") << endl;
  return 0;
}
```

- 設 n 表示 ST.size()

- 若插入/刪除/查詢一個字串 S
- 花費的時間為 $O(|S| \log n)$

有沒有辦法去除這個 log ?

Trie (讀做 Try)



- 字典樹的邊表示一個字母
- 如果到該節點已經有單字組成了則在節點做標記

```
struct node {
  int hit = 0;
  node *next[26] = {}; // a-z
};
```

加入資料很簡單 $O(|S|)$

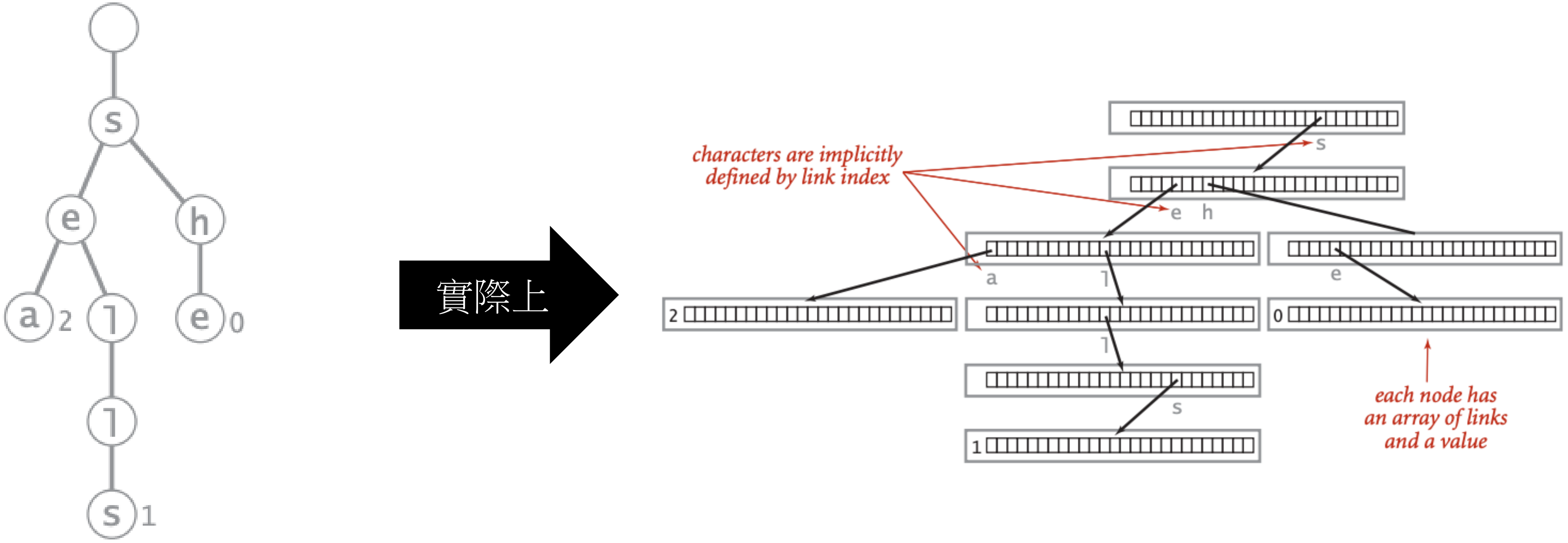
```
void insert(const char *S, node *&root) {
  if (!root)
    root = new node;
  if (*S == '\0') {
    ++root->hit;
  } else {
    insert(S + 1, root->next[*S - 'a']);
  }
}
```

Trie 的好處

- 假設有一些字串 $S_1, S_2, \ldots, S_k$ ，將其加入 Trie 只要 $O(\sum_i |S_i|)$

- 對一個字串 T 查詢：
  - 有幾個 $S_i = T$
  - 有幾個 S_i 是 T 的前綴
- 複雜度都是 $O(|T|)$

- 大多數**字串處理自動機**都基於 Trie 的結構



實際上
characters are implicitly defined by link index
each node has an array of links and a value

注意 Trie 的空間

特殊題型 0-1 Trie

- 把**數字**當作是 由 0 或 1 組成的 二進位字串，用 Tire 維護

- HDU4825:
  給一個數字集合
  詢問數字 s 與集合內的哪一個數字 xor 起來最大

- 洛谷P4551:
  問一個有權重的樹，所有路徑中，權重 xor 最大值為何