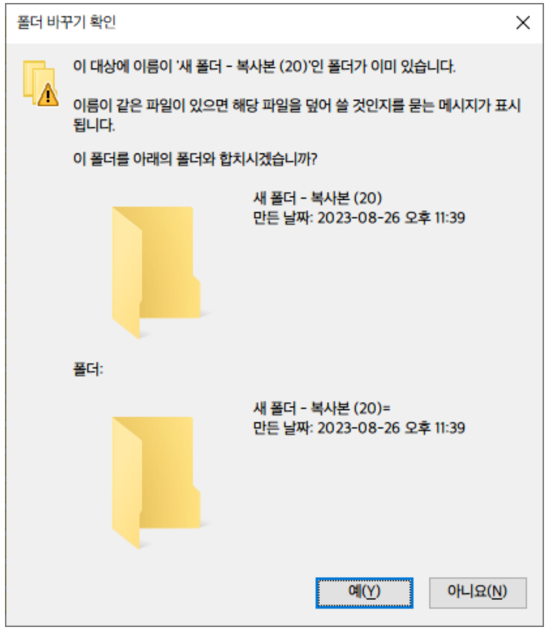
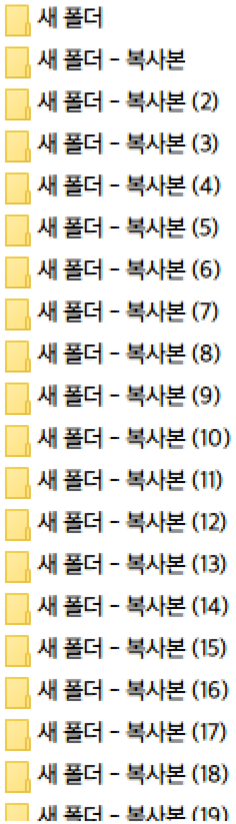
Hashing

들어가기 전에: 파일 시스템

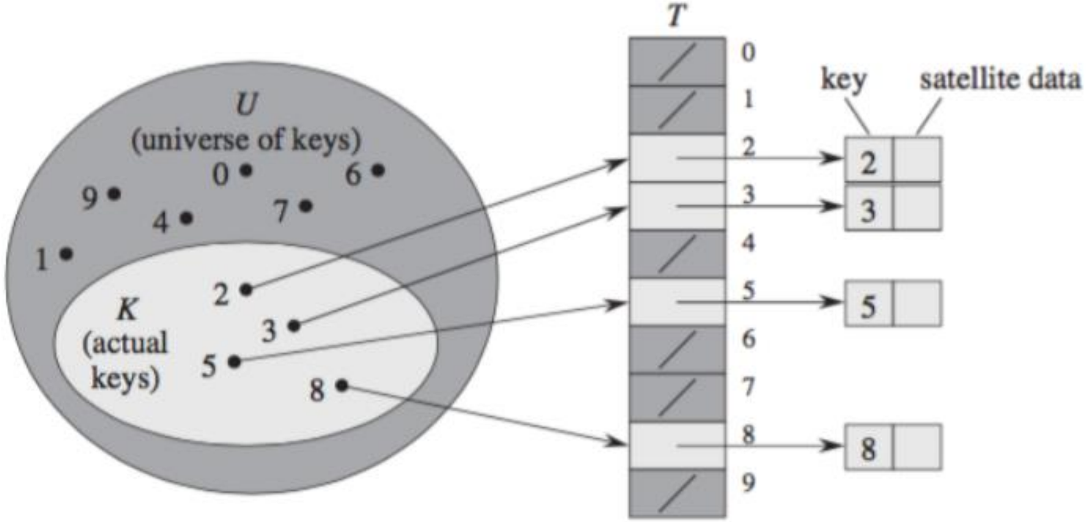
- 운영체제의 파일 시스템을 먼저 생각해 보자
- 하나의 디렉토리 안에 존재 가능한 파일의 수 제한은 없다
- 폴더 내 파일들을 모두 보여줘야 한다
- 같은 이름을 가진 파일이 이미 존재하는지 알아야 한다
- 어떻게 빠른 시간 안에 사용자에게 연산해서 보여줄까?





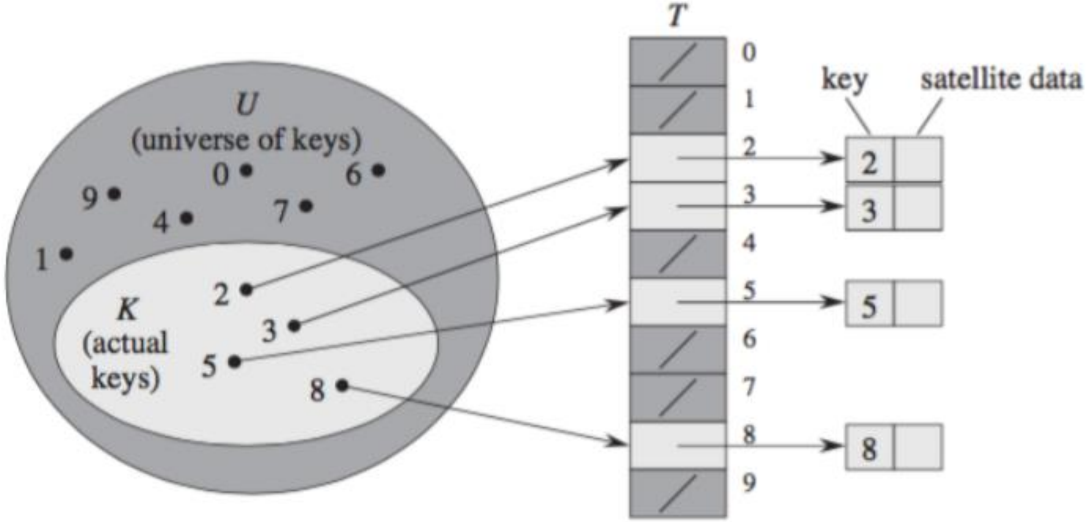
Direct Access Table

- 폴더 내 파일이 많지 않다면, 단순하게 메모리 위에 모든 파일 정보를 올려둘 수 있다
- 메모리에 모든 키의 정보(파일의 정보)를 올려두는 **Direct Access Table** 방식



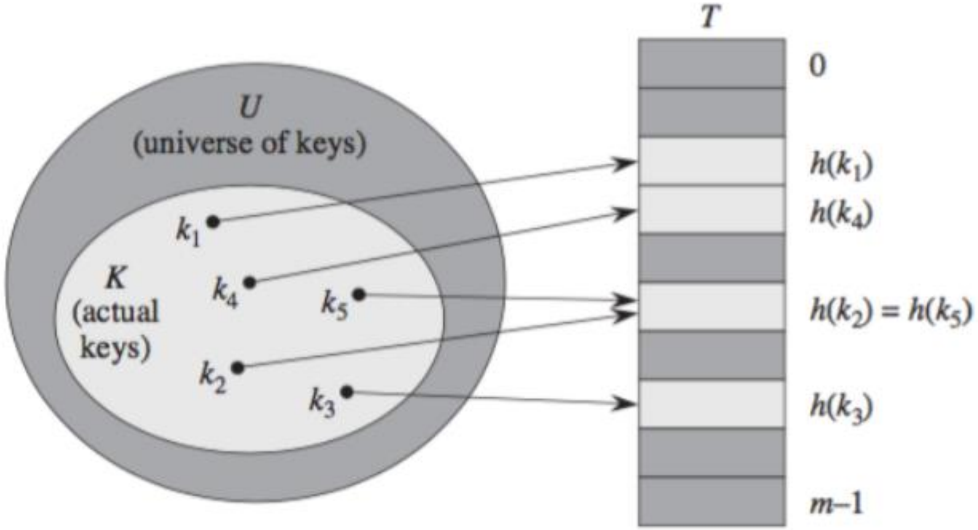
Direct Access Table

- 항상 메모리 위에 자료가 존재하므로 연산이 굉장히 빠르다
- 전체 키 집합이 메모리보다 큰 경우, 물리적으로 구현이 불가능
- 전체 키 집합은 크지만 실제 사용되는 키 집합이 작은 경우, 메모리 낭비



Hash Table

- Direct Access Table의 한계를 극복하고자 고안
- 실제 사용하는 키 집합을 메모리에 올려두고, **해당 키로 매핑**하는 해시 함수 h를 사용



Hash function

- 해시 함수 h는 **임의의 길이의 데이터**를 **고정된 길이의 데이터**로 **매핑**하는 함수

문자열, 그래프, … ⟶ `int, long long, …`

- 알고리즘 문제풀이에서는 보통 **문자열, 그래프 구조**를 **다른 문자열이나 수**로 매핑

Hash function

- 10진법으로 표현한 수

$$12\,345 = 1 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$$

- 알파벳 소문자를 정수로 변환하는 해시 함수 (26진법)

$$\mathrm{abc} = 0 \times 26^2 + 1 \times 26^1 + 2 \times 26^0$$

- 위 해시 함수는 aa 와 a를 구분할 수 없다 (둘 다 0)

Hash function

- 알파벳 소문자를 정수로 변환하는 해시 함수 (27진법)

$$\mathrm{abc} = 1 \times 26^2 + 2 \times 26^1 + 3 \times 26^0 = 731$$

- 문자의 종류는 더 많아질 수 있으므로 일반화 ($L = |s|$)

$$h(s) = s_1 \times p^{L-1} + s_2 \times p^{L-2} + \cdots + s_{L-1} \times p^1 + s_L \times p^0$$

- p는 사용되는 문자의 종류의 개수보다 큰 수여야 함

Hash function

- 문자의 종류는 더 많아질 수 있으므로 일반화 ($L = |s|$)

$$h(s) = s_1 \times p^{L-1} + s_2 \times p^{L-2} + \cdots + s_{L-1} \times p^1 + s_L \times p^0$$

문자열, 그래프, … ⟶ `int, long long, …`

- 해시 함수 h는 **임의의 길이의 데이터**를 **고정된 길이의 데이터**로 **매핑**하는 함수
- 이 함수는 일대일 대응
- 문자열의 길이가 길어질수록 **고정된 길이의 범위**를 초과할 수 있음

Hash function

$$h(s) = s_1 \times p^{L-1} + s_2 \times p^{L-2} + \cdots + s_{L-1} \times p^1 + s_L \times p^0 \ \mathrm{mod}\ M$$

- 나머지 연산을 통해 고정된 범위 내로 매핑해줄 수 있음
- 나머지 연산으로 인해 **일대일 대응이 깨지게 됨**
- 위와 같은 해싱 방법을 **Polynomial Hashing** 이라고 한다

Hash Collision

$$h(s) = s_1 \times p^{L-1} + s_2 \times p^{L-2} + \cdots + s_{L-1} \times p^1 + s_L \times p^0 \bmod M$$

- 나머지 연산으로 인해 **일대일 대응이 깨지게 됨**
- 서로 다른 두 문자열 s_1, s_2에 대해서 $h(s_1) = h(s_2)$인 경우, **해시 충돌**
- 위 방법으로 문자열을 해싱했을 때, 두 문자열이 충돌할 확률은 $\frac{1}{M}$
- 나아가 p와 M이 서로소이고, M이 소수일 때 충돌이 덜 발생

찾기 BOJ 1786

- 주어진 문자열 s에서, 문자열 p가 몇 번, 어디에서 등장하는지 알아내 보자
- ABCABCCABDCABC에서 **ABC** 찾기
- Naïve approach: s의 각 문자에서 시작해서 p가 등장하는지 확인하기
- 최악의 경우 AAAAA…AA에서 AAA…AB 찾기
- $O(|s| \times |p|)$
- 해싱을 어떻게 활용해야 할까?

찾기 BOJ 1786

- ABCABCCABDCABC에서 **ABC** 찾기
- 찾고자 하는 문자열(ABC)의 해시값을 저장
- **주어진 문자열의 모든 연속하는 부분문자열**에 대해 해시값 대조
- ABC, BCA, CAB, ABC, BCC, CCA, CAB, ABD, BDA, DCA, CAB, ABC
- 각 부분 문자열의 해시값을 매번 계산하는 데 시간이 너무 오래 걸림

Rolling hash

- 주어진 해시값은 10진법과 비슷한 형태로, 특정 수 p를 진수로 하고 있음
- ABCD에서 ABC라는 해시값을 구하면 아래와 같다

$$A \times p^2 + \boldsymbol{B \times p^1 + C \times p^0}$$

- BCD의 해시값은 아래와 같다

$$\boldsymbol{B \times p^2 + C \times p^1} + D \times p^0$$

Rolling hash

$$A \times p^2 + \boldsymbol{B} \times \boldsymbol{p^1} + \boldsymbol{C} \times \boldsymbol{p^0}$$

$$\boldsymbol{B} \times \boldsymbol{p^2} + \boldsymbol{C} \times \boldsymbol{p^1} + D \times p^0$$

- 해시값을 구할 때 부분 문자열이 이어져서 나타난다.
- 반드시 하나의 문자가 탈락하고 다른 하나의 문자가 삽입됨
- 기존의 해시 함수에 p를 곱한 뒤, 삽입할 문자의 해시를 더한다
- 이후에 삭제할 문자의 해시를 적당한 p의 거듭제곱꼴로 나타낸 뒤 뺀다

Rolling hash

- **ABCABC**CABDC**ABC**에서 **ABC** 찾기

$$A \times p^2 + B \times p^1 + C \times p^0$$

$$\times p - A \times p^3 + A$$

$$B \times p^2 + C \times p^1 + A \times p^0$$

$$\times p - B \times p^3 + B$$

$$C \times p^2 + A \times p^1 + B \times p^0$$

$$\times p - C \times p^3 + C$$

$$A \times p^2 + B \times p^1 + C \times p^0$$

$$\times p - B \times p^3 + C$$

$$B \times p^2 + C \times p^1 + C \times p^0$$

…

Rolling hash

- 왼쪽에서 삭제, 오른쪽에서 삽입하는 과정에서 착안하여 Rolling hash라고 이름이 붙여짐
- s는 주어진 문자열, pat은 찾고자 하는 문자열, p 배열은 p의 거듭제곱을 가짐

```
int n = s.length(), m = pat.length(), ret = 0;
p[0] = 1;
for(int i = 1; i < max(n, m); i++) p[i] = (p[i-1] * prime) % MOD;
for(int i = 0; i < n; i++) shash[i+1] = (shash[i] + s[i] * p[i]) % MOD;
for(int i = 0; i < m; i++) phash = (phash + pat[i] * p[i]) % MOD;
for(int i = 0; i < n-m+1; ++i){
    ll hash = (shash[i+m] - shash[i] + MOD) % MOD;
    if (hash == (phash * p[i]) % MOD) ret++, ans.push_back(i);
}
```

Rabin-Karp Algorithm

- 해시를 사용한 다른 방법도 존재
- $hash[i] \ = \ s_{1..i}$의 해시값이라고 한다면, ABCDE의 해시값을 구하는 과정은 아래와 같다
- $hash[1] = A \times p^0$
- $hash[2] = A \times p^1 + B \times p^0$
- $hash[3] = A \times p^2 + B \times p^1 + C \times p^0$
- $hash[4] = A \times p^3 + B \times p^2 + C \times p^1 + D \times p^0$
- $hash[5] = A \times p^4 + B \times p^3 + C \times p^2 + D \times p^1 + E \times p^0$

Rabin-Karp Algorithm

- 주어진 $hash$ 배열을 바탕으로 부분문자열의 해시값을 빠르게 구할 수 있다
- $s_{3..5} = \mathrm{CDE}$의 해시값은 어떻게 구할까?
- $hash[1] = A \times p^0$
- $hash[2] = A \times p^1 + B \times p^0$
- $hash[3] = A \times p^2 + B \times p^1 + C \times p^0$
- $hash[4] = A \times p^3 + B \times p^2 + C \times p^1 + D \times p^0$
- $hash[5] = A \times p^4 + B \times p^3 + C \times p^2 + D \times p^1 + E \times p^0$

Rabin-Karp Algorithm

- 누적 합과 비슷한 개념을 사용할 수 있다
- $hash[2] = A \times p^1 + B \times p^0$
- $hash[5] = A \times p^4 + B \times p^3 + C \times p^2 + D \times p^1 + E \times p^0$

$$= hash[5] - hash[2] \times p^{5-3+1}$$

- 일반화하면

$$h(s_{l..r}) = hash[r] - hash[l-1] \times p^{r-l+1}$$

Rabin-Karp Algorithm

$$h(s_{l..r}) = hash[r] - hash[l-1] \times p^{r-l+1}$$

- 해시 배열과 p의 거듭제곱 배열은 $O(|s|)$에 전처리할 수 있다
- 이후 부분 문자열의 해시값을 구하는 데에는 누적합과 같이 $O(1)$

Rabin-Karp Algorithm

```
constexpr ll BASE = 53;
constexpr ll MOD = 1e9+7;
void hash_string(string str){
    int n = str.length();
    p[0] = 1;
    for(int i = 1; i <= n; i++){
        h[i] = ((h[i-1] * BASE) % MOD + str[i-1]) % MOD;
        p[i] = (p[i-1] * BASE) % MOD;
    }
}

ll get_hash(int l, int r){
    return (h[r] - (h[l-1] * p[r-l+1]) % MOD + MOD) % MOD;
}
```

해싱할 때 사용할 p값

p의 거듭제곱 전처리

팰린드롬?? BOJ 11046

- 길이 N의 수열이 주어진다
- Q개의 쿼리를 수행해야 한다
- $l\ r$: $A[l..r]$이 팰린드롬이면 1, 아니면 0

- Naïve solution: $O(N^2)$

팰린드롬?? BOJ 11046

- 길이 N의 수열이 주어진다
- Q개의 쿼리를 수행해야 한다
- $l\ r : A[l..r]$이 팰린드롬이면 1, 아니면 0

- 부분 문자열을 $O(N)$ 전처리, $O(1)$에 판단하는 Manacher's Algorithm이 정해
- 해싱으로도 풀 수 있다?

팰린드롬?? BOJ 11046

- 주어진 수열을 해싱하고, 수열을 거꾸로 둔 것도 해싱해 두자 - 해싱 전처리 $O(N)$
- 부분 문자열의 해시값은 $O(1)$에 구할 수 있다
- 이후 주어진 구간에 대해서 해시값을 정방향에 대해서 구하고, 거꾸로 해싱한 것에서도 구간을 적절히 조절해 해시를 받아낸 뒤 둘을 비교하자

뒤집기 K BOJ 21162

- 주어진 수열을 길이 1 이상의 두 개의 부분 수열로 나눈다
- 각 부분 수열을 뒤집어서 다시 이어 붙인다
- 그런 수열들 중에서 K번째로 사전순 앞에 오는 수열은 무엇일까?
- $[5, 4, 1, 1, 2] \rightarrow [5, 4, 1, 1], [2] \rightarrow [1, 1, 4, 5, 2]$ $(K = 1)$

뒤집기 K BOJ 21162

- 문제를 잘 보면, 새롭게 만들어지는 수열에서 어떤 조작을 가했는지 알 수 있다
- 새로 생기는 수열은 기존 수열을 거꾸로 한 것의 cyclic shift
- $[1, 2, 3, 4, 5] \rightarrow [5, 4, 3, 2, 1] \rightarrow [3, 2, 1, 5, 4]$
- $[3, 2, 1, 5, 4]$는 $[1, 2, 3], [4, 5]$ 를 각각 뒤집어 붙인 것과 같다

뒤집기 K BOJ 21162

- 주어진 수열을 뒤집어 한 번 더 이어붙인 수열에서, $[i: i+N]$ 중 사전순으로 K번째 수열을 찾아야 한다 $(2 \leq i < N, 1-\text{based})$
- 하나하나 직접 비교하는 데에는 한 번의 비교에 $O(\min(|A|,|B|))$이다.
- 각 원소들에 대해서 확인하고 정렬해야 하므로, $O(N^2 \log N)$으로 시간초과이다.
- 비교를 더 빠르게 할 수 있을까?

뒤집기 K BOJ 21162

- 문자열 / 수열의 대소관계는 **가장 처음으로 같지 않은 문자/수의 위치**가 중요하다
- 같은 위치에서 서로 다른 문자가 등장했다면, 그 이후의 문자는 볼 필요가 없이 대소관계가 정해진다.
- 이 위치를 지금까지는 하나씩 확인하며 $O(N)$에 진행했다
- 해싱을 어떻게 활용할 수 있을까?

뒤집기 K BOJ 21162

- 문자열 / 수열의 대소관계는 **가장 처음으로 같지 않은 문자/수의 위치**가 중요하다
- 뒤집어서 붙인 2배의 수열을 해싱한 뒤, 특정 구간의 해시값을 구하는 데에는 $O(1)$이다
- 가장 처음으로 달라지는 위치는 **매개변수 탐색**을 통해 구할 수 있다
- 이때, 시간복잡도는 $O(\log(\min(|A|,|B|)))$이다
- 나올 수 있는 문자열의 개수는 $O(N)$개이므로, 정렬할 때 위 방법을 사용해 대소관계를 비교하면 된다

뒤집기 K BOJ 21162

```
bool cmp(int a, int b) {
    int l = 0, r = N+1;
    // l, r, mid는 같은 문자의 개수를 의미한다
    while (l + 1 < r) {
        int mid = (l + r) / 2;
        ll hash_a = get_hash(a, a+mid-1);
        ll hash_b = get_hash(b, b+mid-1);
        if (hash_a == hash_b) l = mid;
        else r = mid;
    }
    if (r == N) return false;
    return v[a+r-1] < v[b+r-1];
}
```

```
// 입력 후 배열 뒤집은 뒤 2배해준 상태
stable_sort(idx.begin(), idx.end(), cmp);
for(int i = 0; i < N; i++) {
    cout << v[idx[K-1]+i] << " ";
}
```

시간복잡도 $O(N \log^2 N)$

가장 긴 문자열 BOJ 3033

- 주어진 문자열 s에서 두 번 이상 등장한 부분 문자열 중 가장 길이가 긴 문자열 찾기
- AB**ABCA**BCA -> ABCA (4), 부분 문자열이 서로 겹쳐도 된다. $|s| \leq 200\,000$
- 해싱하는 데 $O(|s|)$, 가능한 모든 부분 문자열에 대해 확인하는 것은 시간이 오래 걸림

가장 긴 문자열 BOJ 3033

- 주어진 문자열 s에서 두 번 이상 등장한 부분 문자열 중 가장 길이가 긴 문자열 찾기
- 길이 5의 부분 문자열이 두 번 등장한다면, 길이 4의 부분 문자열도 두 번 등장함
- 길이가 길어질수록 두 번 이상 등장하는 부분 문자열의 개수는 **단조감소**
- 부분 문자열이 두 번 이상 등장하는가? 에 대해서 확인하는 결정문제로 환원

가장 긴 문자열 BOJ 3033

- 주어진 문자열 s에서 두 번 이상 등장한 부분 문자열 중 가장 길이가 긴 문자열 찾기
- 길이 5의 부분 문자열이 두 번 등장한다면, 길이 4의 부분 문자열도 두 번 등장함
- 길이가 길어질수록 두 번 이상 등장하는 부분 문자열의 개수는 **단조감소**
- **길이가 i인 부분 문자열이 두 번 이상 등장하는가?** 라는 결정문제로 환원
- 부분 문자열의 길이를 매개변수로 하는 이분탐색을 활용

가장 긴 문자열 BOJ 3033

```
int l = 0, r = N;
while (l + 1 < r) {
    memset(table, 0, sizeof table);
    int mid = (l + r) / 2;
    bool found = false;
    // 길이 mid의 두 번 이상 등장하는 부분 문자열이 존재하는가?
    for (int i = 0; i <= N-mid; i++) {
        ll hash = get_hash(i+1, i+mid); // 1-based off
        if (same_string_found_on(hash, i, mid)) {
            found = true; break;
        }
        else table[hash].push_back(i);
    }
    if (found) l = mid;
    else r = mid;
}
cout << l << '\n';
```

연습 문제

[10840](#) : 구간 성분
[1786](#) : 찾기
[3033](#) : 가장 긴 문자열 **
[21162](#) : 뒤집기 K **
[17228](#) : 아름다운 만영로 **

Reference

- https://www.acmicpc.net/blog/view/67
- https://codeforces.com/blog/entry/100027
- https://blog.naver.com/kks227/220927272165
- https://cp-algorithms.com/string/string-hashing.html
- https://ko.wikipedia.org/wiki/%ED%95%B4%EC%8B%9C_%ED%95%A8%EC%88%98
- https://github.com/justiceHui/SSU-SCCC-Study/blob/master/2022-winter-intermediate/slide/08.pdf
- https://blog.hoony.me/2023/07/06/hashing-algorithm/
- https://algoshitpo.github.io/2020/02/09/hashingtechnique/