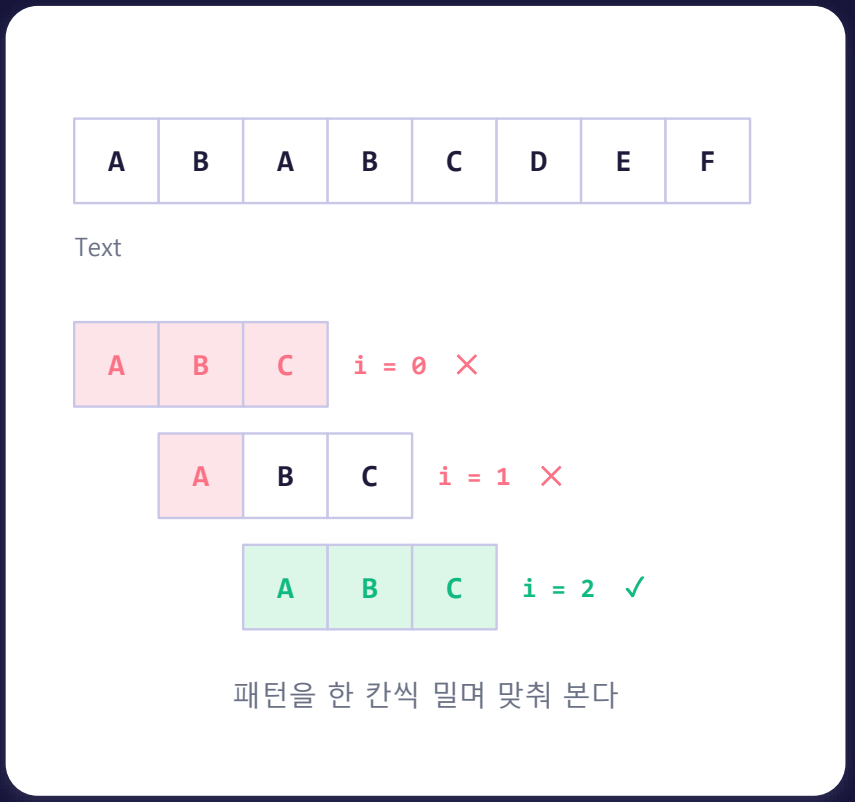
그림으로 쉽게 설명하는 파이썬 알고리즘

CHAPTER 12

텍스트 알고리즘

Text Algorithms

문자열은 길이와 패턴을 함께 가진다



CONTENTS

이번 장에서 배울 내용

12.1 **완전 탐색**

가장 단순한 문자열 검색

12.2 **KMP 알고리즘**

접두사의 정보를 활용하라

12.3 **보이어-무어**

뒤에서부터 비교하는 역발상

12.4 **라빈-카프**

문자열을 숫자로 바꾸다

12.5 **트라이 (Trie)**

접두사 트리 자료구조

12.6 **데이터 압축**

허프만 코딩

12.7 **코딩 테스트 전략**

실전 활용과 언어별 특성

문자열은 배열과 달리 "길이"와 "패턴"을 가진다 — 그 구조를 활용해 검색과 처리를 최적화한다.

PROBLEM

문자열 검색 문제 정의

텍스트 (T) 검색 대상이 되는 긴 문자열. 길이 N, 인덱스 T[0 … N-1]

패턴 (P) 텍스트 안에서 찾고자 하는 짧은 문자열. 길이 M, 인덱스 P[0 … M-1] (M ≤ N)

목표 T 안에서 P가 출현하는 모든 시작 인덱스를 찾거나, 존재 여부를 판단한다.

T = "ABABCDEF", P = "ABC" → 발견 인덱스: [2]

인덱스 2에서 정확히 겹친다

12.1

완전 탐색

첫 줄부터 한 글자씩 읽어 가며
찾는 단어와 맞는지 확인하는 방법.

완전 탐색(brute-force)이란?

책 첫 페이지 첫 줄부터 한 글자씩 읽어 가며, 내가 찾는 단어와 일치하는지 확인하는 방법이다.

동작 방식

1. 텍스트 처음부터 패턴 크기의 윈도우를 한 칸씩 옮긴다
2. 각 위치에서 패턴과 문자를 하나씩 비교한다
3. 불일치가 나면 즉시 다음 위치로 이동한다

특징

- 특별한 기교 없이 있는 그대로 비교한다
- 나이브(naïve) 알고리즘이라고도 부른다
- 구현이 가장 간단하다
- 비교 정보를 전혀 재활용하지 않는다

핵심 — "성공하든 실패하든 무조건 한 칸씩만 이동한다"

§12.1 완전 탐색

윈도우 슬라이딩 동작

T = "ABABCDEF", P = "ABC" 를 실제로 맞춰 보자. 초록은 일치, 빨강은 불일치가 난 자리다.

외부 루프 i

0부터 N-M까지 — 가능한 모든 시작 위치를 훑는다.

내부 루프 j

0부터 M-1까지 — 패턴의 각 문자와 비교한다.

불일치 시 즉시 탈출

내부 루프를 빠져나와 다음 시작 위치로 넘어간다.

왜 한 칸씩만?

겹쳐 있는 패턴을 놓치지 않기 위해서다. 두 칸씩 뛰면 답을 지나칠 수 있다.

완전 탐색 유사코드

brute_force.pseudo

```
Algorithm BruteForceSearch(T, P)

  for i from 0 to N - M do:
    match_found <- True

    for j from 0 to M - 1 do:
      if T[i + j] != P[j] then:
        match_found <- False
        break

    if match_found is True then:
      print "패턴 발견! 시작 인덱스: " + i
```

외부 루프 (i)

텍스트에서 가능한 모든 시작 위치를 순회한다. 0부터 N-M까지다.

내부 루프 (j)

현재 위치에서 패턴과 한 글자씩 비교한다.

불일치 시

즉시 break하고 다음 시작 위치로 이동한다.

이중 루프가 그대로 시간 복잡도가 된다

외부 N-M+1번, 내부 최대 M번 — 곱하면 O(N × M)이다. 코드는 짧지만 대가는 크다.

파이썬 구현 — brute_force_search

brute_force.py

```python
def brute_force_search(text, pattern):
    n, m = len(text), len(pattern)
    found_indexes = []

    if m > n or m == 0:
        return []

    for i in range(n - m + 1):
        match_found = True
        for j in range(m):
            if text[i + j] != pattern[j]:
                match_found = False
                break
        if match_found:
            found_indexes.append(i)

    return found_indexes
```

run.py

```python
brute_force_search("ABABCDEF", "ABC")
```

OUTPUT

```
[2]
```

모든 위치를 반환한다

첫 번째만 찾고 멈추지 않고 리스트로 모아 돌려준다.

예외 처리

패턴이 텍스트보다 길거나 빈 문자열이면 곧바로 빈 리스트를 반환한다.

복잡도 분석

O(N×M)

최악의 시간 복잡도

O(1)

공간 복잡도

10억 번

N=100만, M=1000 일 때

최악의 시나리오 — 패턴 뒷부분에서 불일치가 계속 발생할 때

텍스트 T : A가 100만 개 이어지고 마지막에 B （ AAAA...AB ）
패턴 P : A가 다섯 개에 마지막이 B （ AAAAB ）
→ 매번 M번을 비교한 뒤 마지막 글자에서 실패하고, 이를 약 N번 되풀이한다.

근본 원인 — "이미 알고 있는 정보를 버리고 처음부터 다시 확인하기 때문"

이미 A인 줄 아는 구간을 굳이 패턴 앞부분의 A와 다시 비교할 필요가 있을까?

완전 탐색의 한계와 다음 단계

완전 탐색의 문제점

- ✕ 비교했던 정보를 전혀 재활용하지 않는다
- ✕ 성공·실패와 무관하게 언제나 1칸만 이동한다
- ✕ 텍스트 인덱스가 뒤로 후퇴한다
- ✕ 실시간 시스템에서는 허용되기 어렵다

해결 방향 → KMP

- ✓ 실패 정보를 다음 점프에 활용한다
- ✓ 텍스트 인덱스는 절대 후퇴하지 않는다
- ✓ O(N×M)에서 O(N+M)으로 끌어내린다
- ✓ 패턴 내부의 반복 구조를 활용한다

"이미 A인 줄 아는 구간을 다시 비교할 필요가 있을까?"

12.2

KMP 알고리즘

이미 읽은 정보는 다시 읽지 않는다.
실패조차 다음 점프의 재료가 된다.

12.2

KMP — 핵심 아이디어

"이미 읽은 정보는 다시 읽지 않는다" — 1977년 Knuth · Morris · Pratt 세 사람이 제안했다.

불일치가 발생했다는 사실 그 자체가 유용한 정보라는 것이 출발점이다.

후퇴 없음 텍스트 인덱스를 절대 뒤로 돌리지 않는 획기적인 알고리즘이다.

자기 유사성 패턴 내부의 반복 구조(self-similarity)를 미리 분석해 활용한다.

실패의 재활용 실패를 단순한 실패로 끝내지 않고, 다음 점프를 위한 정보로 쓴다.

$O(N+M)$	시간 복잡도
$O(M)$	전처리 시간
$\leq 2N$	최대 비교 횟수

§12.2 KMP 알고리즘

완전 탐색의 비효율적인 뒷걸음질

패턴 P = "ABCDABD" 로 검색하다가 6글자(ABCDAB)를 맞춘 뒤 7번째에서 불일치가 났다고 하자.

완전 탐색의 반응

"에이, 틀렸네. 무조건 한 칸 이동!"

· 6글자(ABCDAB)에 대한 정보를 싹 잊어버린다
· B부터 다시 비교를 시작한다

극도로 비효율적

KMP의 반응

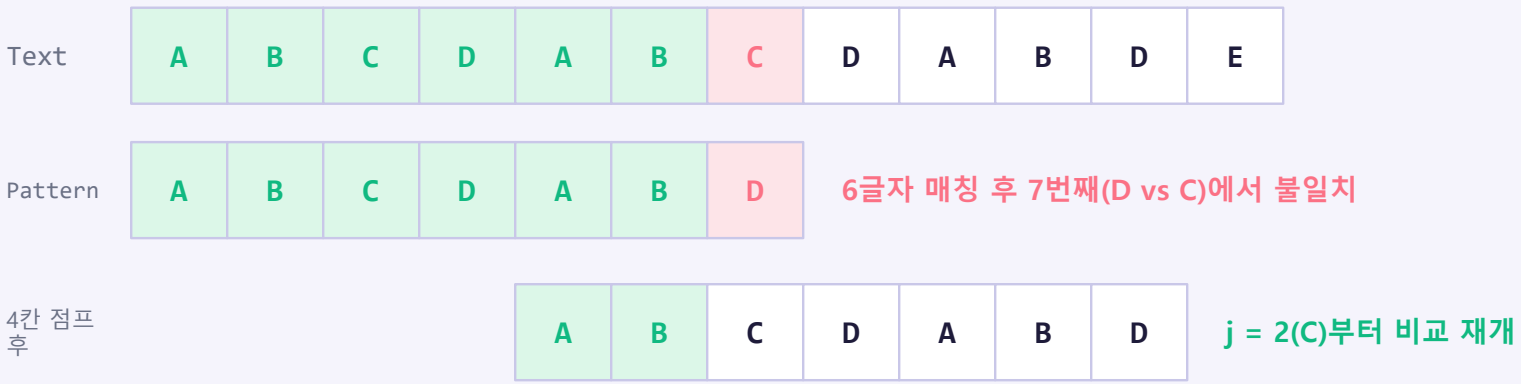
"잠깐! 방금 지나온 길을 봐!"

· ABCDAB에서 접두사 AB = 접미사 AB
· 이미 맞춘 뒤쪽 AB를 패턴 앞쪽 AB에 맞추면 된다

한 번에 4칸 점프

KMP는 "실패는 성공의 어머니" — 실패 정보를 적극적으로 활용한다.

KMP 동작 과정 시각화



점프 논리 일치 구간 ABCDAB에서 접두사 "AB"와 접미사 "AB"가 같다(길이 2). 그래서 패턴을 4칸 밀어 j=2부터 다시 본다.

텍스트 인덱스 i는 절대 후퇴하지 않는다. 움직이는 것은 패턴 인덱스 j뿐이다.

텍스트 인덱스 i는 0에서 N까지 전진만, 패턴 인덱스 j는 LPS를 따라 점프한다.

LPS 배열이란?

LPS = Longest Proper Prefix which is also Suffix — 패턴 P[0..i]에서 "접두사이면서 동시에 접미사가 되는 가장 긴 부분 문자열의 길이"

역할

불일치가 났을 때 패턴 인덱스를 어디로 옮겨야 하는지 알려 주는 내비게이션이다.

텍스트 인덱스가 아니라 패턴 인덱스를 바꾼다는 점이 핵심이다.

계산 시점

텍스트를 훑기 전에 패턴 P 자체만 분석하는 전처리(preprocessing) 과정이다.

텍스트와 무관하므로 한 번만 계산해 두면 된다. 시간 복잡도는 O(M).

이 값이 곧 점프 거리의 기준이 된다.

LPS 배열 예제 — P = "ABCDABD"

인덱스	0	1	2	3	4	5	6
패턴 문자	A	B	C	D	A	B	D
LPS 값	0	0	0	0	1	2	0

인덱스 4 (A)

부분 문자열: ABCDA
접두사 A = 접미사 A → LPS[4] = 1 (길이 1만큼 겹침)

인덱스 5 (B)

부분 문자열: ABCDAB
접두사 AB = 접미사 AB → LPS[5] = 2 (길이 2만큼 겹침)

"인덱스 5까지 맞춘 뒤 실패하면, LPS[5]=2 이므로 인덱스 2(C)로 돌아가 검사를 이어 간다."

LPS 배열 생성 알고리즘

build_lps.pseudo

```
BUILD_LPS(P):
  m <- length(P)
  LPS[0] <- 0
  len <- 0        // 현재까지 일치한 접두사=접미사 길이
  i <- 1

  while i < m:
    if P[i] = P[len]:
      len <- len + 1
      LPS[i] <- len
      i <- i + 1
    else:
      if len != 0:
        len <- LPS[len - 1]    // 더 짧은 후보로 점프
      else:
        LPS[i] <- 0
        i <- i + 1

  return LPS
```

P[i] 와 P[len] 을 비교

패턴을 자기 자신과 맞춰 보는 셈이다. 이것이 자기 유사성 분석이다.

일치하면

len을 늘리고 LPS[i]에 기록한 뒤 다음 글자로 넘어간다.

불일치하면

len을 LPS[len-1]로 되돌려 더 짧은 후보를 시도한다. i는 그대로다.

시간 복잡도 O(M)

i는 전진만 하고 len은 줄어들기만 하므로 전체 연산이 패턴 길이에 비례한다.

KMP 검색 알고리즘

kmp_search.pseudo

```
KMP_SEARCH(T, P):
  n <- length(T),  m <- length(P)
  LPS <- BUILD_LPS(P)
  i <- 0          // T의 인덱스
  j <- 0          // P의 인덱스

  while i < n:
    if T[i] = P[j]:
      i <- i + 1,  j <- j + 1
      if j = m:
        report (i - m)          // 패턴 발견
        j <- LPS[j - 1]         // 다음 탐색을 위해 점프
    else:
      if j != 0:
        j <- LPS[j - 1]         // 패턴만 이동, i는 그대로
      else:
        i <- i + 1
```

일치할 때

i와 j를 모두 전진시킨다. j가 m에 도달하면 매칭이 완성된 것이므로 위치를 보고하고, LPS로 다음 탐색을 준비한다.

불일치할 때

j가 0이 아니면 LPS를 따라 j만 점프시킨다. i는 그대로 둔다. j가 0이면 더 물러날 곳이 없으므로 i를 한 칸 전진시킨다.

이 코드 어디에도 i를 줄이는 문장이 없다 — 그것이 O(N+M)의 근거다.

KMP 파이썬 구현

kmp.py

```python
def build_lps(pattern):
    m = len(pattern)
    lps = [0] * m
    length, i = 0, 1
    while i < m:
        if pattern[i] == pattern[length]:
            length += 1; lps[i] = length; i += 1
        elif length != 0:
            length = lps[length - 1]
        else:
            lps[i] = 0; i += 1
    return lps

def kmp_search(text, pattern):
    n, m = len(text), len(pattern)
    lps = build_lps(pattern)
    i = j = 0; result = []
    while i < n:
        if text[i] == pattern[j]:
            i += 1; j += 1
        if j == m:
            result.append(i - m); j = lps[j - 1]
        elif i < n and text[i] != pattern[j]:
            if j != 0: j = lps[j - 1]
            else: i += 1
    return result
```

run.py

```python
kmp_search("ABCDABCDABDE",
           "ABCDABD")
```

OUTPUT

```
[4]
```

두 함수로 나뉜다

build_lps는 패턴만 보고, kmp_search는 그 결과를 받아 텍스트를 한 번 훑는다.

찾은 뒤에도 계속

j = lps[j-1]로 되돌려 겹치는 다음 매칭도 놓치지 않는다.

§12.2 KMP 알고리즘

KMP 시간 복잡도 분석

O(M)

전처리 단계

LPS 테이블 생성

O(N)

검색 단계

텍스트 한 번 훑기

O(N+M)

전체 시간 복잡도

두 단계의 합

텍스트 인덱스 i 0부터 N까지 전진만 한다. 절대 뒤로 돌아가지 않으므로 최대 N번 이동한다.

패턴 인덱스 j LPS를 따라 오르내리지만, 분할 상환 분석을 하면 비교 횟수는 2N 이하다.

전처리 패턴 길이만큼만 걸린다 → O(M).

N=100만, M=1000 일 때 — 완전 탐색 약 10억 번 → KMP 약 100만 번 (1000배 향상)

완전 탐색 vs KMP

항목	완전 탐색	KMP
최악 시간 복잡도	O(N × M)	**O(N + M)**
텍스트 인덱스 후퇴	Yes — 1칸씩 되돌아감	No — 절대 후퇴하지 않음
전처리 필요	없음	LPS 테이블 O(M)
비교 정보 재활용	하지 않음	LPS로 점프 거리 결정
N=100만, M=1000	약 10억 번	**약 100만 번**

KMP가 문자열 검색의 "교과서"로 불리는 이유

이론적으로 최적인 O(N+M)을 어떤 입력에서도 보장한다. 뒤에 나올 보이어-무어가 평균적으로 더 빠르더라도, 최악을 보장하는 쪽은 KMP다.

"평균이 빠른 것"과 "최악이 보장되는 것"은 다른 미덕이다.

12.3

보이어-무어 알고리즘

12.3

패턴의 뒤에서부터 비교한다.
마지막 글자 하나로 윈도우를 통째로 건너뛴다.

§12.3 보이어-무어

보이어-무어 — 역발상의 미학

1977년 Boyer & Moore — KMP와 같은 해에 나왔지만, 통계적으로는 훨씬 빠른 알고리즘이다.

핵심 — "패턴의 뒤에서부터 비교하자"

보통은 왼쪽에서 오른쪽으로 비교하지만, 보이어-무어는 오른쪽에서 왼쪽으로 비교한다. 패턴의 마지막 글자가 텍스트와 아예 다르면 앞의 글자들은 볼 필요조차 없이 윈도우 전체를 크게 이동시킬 수 있다.

나쁜 문자 규칙

Bad Character Rule — 불일치한 텍스트 문자를 기준으로 점프 거리를 결정한다.

착한 접미사 규칙

Good Suffix Rule — 이미 일치한 접미사를 패턴의 다른 곳에서 찾아 정렬한다.

나쁜 문자 규칙 (bad character rule)

Case A — 패턴에 존재함

불일치한 문자가 패턴 안에 있을 때는, 패턴 안에서 그 문자가 나오는 가장 오른쪽 위치를 찾아 텍스트의 해당 문자와 자리가 맞도록 패턴을 옮긴다.

정렬(align) 점프

Case B — 패턴에 없음

불일치한 문자가 패턴에 아예 없다면, 현재 구간에는 정답이 있을 리가 없다. 패턴 길이 M만큼 윈도우를 통째로 건너뛴다.

가장 큰 점프 — 속도의 핵심

Case B가 보이어-무어의 속도가 비약적으로 빠른 이유다

예를 들어 텍스트가 "AAAA…A"이고 패턴이 "BCDE"라면, 패턴에 A가 없으므로 매번 4칸씩 통째로 건너뛴다. 텍스트의 대부분을 아예 읽지도 않는다.

텍스트를 전부 읽지 않고도 정답을 놓치지 않는다 — 이것이 O(N/M)의 근거다.

착한 접미사 규칙 (good suffix rule)

나쁜 문자 규칙만으로 부족할 때 함께 쓰는 보조 규칙이다. 원리는 KMP의 LPS와 닮아 있다.

① 패턴의 뒤에서부터 몇 글자가 일치했다 — 이것을 "착한 접미사"라고 부른다.

② 그 앞부분에서 불일치가 발생한다.

③ 이미 일치한 접미사가 패턴의 앞부분에도 등장하는지 찾아본다.

④ 등장한다면 그 위치가 겹치도록 패턴을 옮긴다.

shift = max(bad_character_shift, good_suffix_shift)

BUILD_LAST — 나쁜 문자 테이블

build_last.pseudo

```
BUILD_LAST(P):

  for each character c in ALPHABET:
    last[c] <- -1

  for j <- 0 to m-1:
    last[P[j]] <- j       // 나중 값이 앞 값을 덮어씀

  return last
```

예 : P = "EXAMPLE"

문자	E	X	A	M	P	L
last	6	1	2	3	4	5

마지막 위치만 저장

같은 문자가 여러 번 나오면 가장 오른쪽 위치만 남긴다.

E는 0과 6에 등장

뒤쪽 값이 덮어쓰므로 last[E] = 6 이 된다.

패턴에 없는 문자

-1로 남는다. 이 경우 점프 거리가 최대가 된다.

시간 복잡도

O(M + |Σ|) — 패턴 길이와 문자 집합 크기의 합이다.

가장 오른쪽 위치를 쓰는 이유 — 패턴을 지나치게 많이 밀어 정답을 건너뛰지 않기 위해서다.

보이어-무어 검색 알고리즘

boyer_moore.pseudo

```
BOYER_MOORE_SEARCH(T, P):
  last <- BUILD_LAST(P)
  i <- 0

  while i <= n - m:
    j <- m - 1                    // 패턴 끝부터 비교
    while j >= 0 and P[j] = T[i + j]:
      j <- j - 1

    if j < 0:
      report i                    // 매칭 발견
      i <- i + m
    else:
      c <- T[i + j]
      bc_shift <- j - last[c]
      if bc_shift < 1: bc_shift <- 1
      i <- i + bc_shift
```

비교 방향 — 오른쪽에서 왼쪽

j = m-1부터 0까지 내려간다. KMP와 정반대다.

이동 계산

bc_shift = j − last[나쁜문자]. 패턴에 없으면 last가 -1이므로 j+1만큼 이동한다.

최소 1칸 보장

bc_shift가 0 이하로 나올 수 있어 1로 보정한다. 없으면 무한 루프에 빠진다.

마지막 글자 하나만 보고도 윈도우를 M칸 옮길 수 있다는 점이 이 코드의 전부다.

보이어-무어 파이썬 구현

boyer_moore.py

```python
def build_last(pattern):
    last = {}
    for i in range(len(pattern)):
        last[pattern[i]] = i      # 나중 위치가 덮어씀
    return last

def boyer_moore(text, pattern):
    n, m = len(text), len(pattern)
    last = build_last(pattern)
    result, i = [], 0

    while i <= n - m:
        j = m - 1
        while j >= 0 and pattern[j] == text[i + j]:
            j -= 1
        if j < 0:
            result.append(i); i += m
        else:
            bad_char = text[i + j]
            shift = j - last.get(bad_char, -1)
            if shift < 1: shift = 1
            i += shift

    return result
```

run.py

```python
boyer_moore(
  "HERE IS A SIMPLE EXAMPLE",
  "EXAMPLE")
```

OUTPUT

```
[17]
```

dict.get(c, -1)

패턴에 없는 문자는 -1을 반환하게 해서, Case B의 최대 점프가 자연스럽게 계산되도록 했다.

보이어-무어 성능 분석

O(N/M)
최선 · 평균

O(NM)
최악

Ctrl+F
실전 사용처

패턴이 길수록 검색이 빨라지는 기이한 특성 — 한 번에 M칸씩 건너뛰기 때문이다.

알고리즘	최선	평균	최악
완전 탐색	O(N)	O(NM)	O(NM)
KMP	O(N+M)	O(N+M)	**O(N+M)**
보이어-무어	**O(N/M)**	O(N)	O(NM)

최악은 극단적으로 반복적인 입력에서만 나타나며, 현실 데이터에서는 매우 드물다 — 그래서 대부분의 에디터와 grep이 이것을 쓴다.

왜 그렇게 과감하게 점프해도 되는가?

패턴에 없는 문자가 나올 때

텍스트 : … [Z] …
패턴 : A B C D E (Z 없음)

1칸 이동 → D와 Z 비교 → 실패
2칸 이동 → C와 Z 비교 → 실패
… 어떻게 밀어도 Z 자리에서 반드시 실패한다

패턴 전체를 통째로 옮겨도 안전

패턴에 있는 문자가 나올 때

텍스트 : … [B] …
패턴 : A B C D E

텍스트의 B 자리는 고정되어 있다.
패턴 안의 B를 그 위치에 맞춰(align) 주는 것만이
정답이 될 수 있는 유일한 가능성이다.

정렬 점프

"점프한 구간 사이에는 절대 정답이 있을 수 없다" — 수학적으로 보장된다.

12.4

라빈-카프 알고리즘

12.4

문자열을 숫자로 바꾸면
비교가 한 번의 연산으로 끝난다.

라빈-카프 — 해시값 비교 아이디어

문자열 비교를 숫자 비교로 바꾼다.

① 계산

패턴 P의 해시값을 미리 구해 둔다.

② 비교

텍스트 윈도우의 해시값과 견준다.

③ 같으면

실제 문자열도 같은지 한 번 더 확인한다.

④ 다르면

100% 다르다. 바로 다음 칸으로 넘어간다.

롤링 해시 (rolling hash)

윈도우가 한 칸 이동할 때마다 해시를 처음부터 다시 계산하면 O(M)이라 의미가 없다. "맨 앞 글자는 빼고, 맨 뒤 글자는 더하는" 방식으로 O(1)에 갱신한다. 예: 12345 → (12345 − 1×10000) × 10 + 6 = 23456

핵심 강점 — 다중 패턴 검색

금지어 100개의 해시값을 해시 테이블에 넣어 두면, 텍스트를 단 한 번만 훑어도 전부 찾아낼 수 있다. 표절 검사와 바이러스 탐지가 이 원리로 동작한다.

§12.4 라빈-카프

롤링 해시 (rolling hash) 상세

rolling_hash.py

```
# 다항식 해싱 (polynomial rolling hash)
#   H = Σ ( char[i] × base^(m-1-i) )  mod q

# 롤링 업데이트
#   new_hash = (old_hash - leftChar × base^(m-1)) × base + rightChar

# 10진수로 이해하기
#   현재 : 12345          (길이 5)
#   다음 : 23456          (앞의 1이 빠지고 뒤에 6이 붙음)
#   공식 : (12345 - 1×10000) × 10 + 6 = 23456
```

해시 충돌

서로 다른 문자열이 우연히 같은 해시값을 가질 수 있다.
→ 해시가 일치하면 반드시 실제 문자열을 비교해야 한다.

시간 복잡도

평균 O(N + M), 최악 O(N × M) — 해시 충돌이 잦을 때다.
→ mod에 큰 소수를 쓰면 충돌 확률이 크게 줄어든다.

라빈-카프 유사코드

rabin_karp.pseudo

```
RABIN_KARP(T, P):
  base <- d          // 문자 집합 크기 (예: 256)
  mod  <- q          // 큰 소수 (충돌 완화)
  h <- base^(m-1) mod q

  // 패턴 해시와 첫 윈도우 해시를 함께 계산
  pHash <- 0,  tHash <- 0
  for i <- 0 to m-1:
    pHash <- (base × pHash + code(P[i])) mod q
    tHash <- (base × tHash + code(T[i])) mod q

  // 윈도우를 한 칸씩 이동
  for s <- 0 to n - m:
    if pHash = tHash:
      if T[s..s+m-1] = P: report s        // 실제 비교

    if s < n - m:                          // 롤링 해시 갱신
      tHash <- (base×(tHash - code(T[s])×h) + code(T[s+m])) mod q
```

base와 mod

문자 집합 크기와 큰 소수. 이 두 값의 선택이 충돌 확률을 좌우한다.

해시가 같아도 확인

충돌 가능성이 있으므로 실제 문자열을 반드시 대조한다.

롤링 갱신 한 줄

앞 글자를 빼고 뒤 글자를 더한다. 이 한 줄이 O(1)을 만든다.

라빈-카프 파이썬 구현

rabin_karp.py

```python
def rabin_karp(text, pattern, base=256, mod=1000003):
    n, m = len(text), len(pattern)
    if m == 0 or m > n:
        return []

    h = pow(base, m - 1, mod)
    p_hash = t_hash = 0
    for i in range(m):
        p_hash = (base * p_hash + ord(pattern[i])) % mod
        t_hash = (base * t_hash + ord(text[i]))    % mod

    result = []
    for s in range(n - m + 1):
        if p_hash == t_hash and text[s:s+m] == pattern:
            result.append(s)
        if s < n - m:
            t_hash = (base * (t_hash - ord(text[s]) * h)
                      + ord(text[s+m])) % mod

    return result
```

run.py

```python
rabin_karp("AAAAABAAAAAB",
           "AAAAB")
```

OUTPUT

```
[1, 7]
```

pow(base, m-1, mod)

파이썬의 세 인자 pow는 거듭제곱을 모듈러 연산과 함께 빠르게 계산한다. 롤링 해시의 계수 h를 여기서 얻는다.

겹치는 매칭(1번과 7번)도 모두 찾아낸다.

12.5

트라이 (Trie)

12.5

공통된 접두사를 한 번만 저장한다.
검색 속도가 단어 수와 무관해진다.

트라이(Trie)란?

이름은 retrieval(검색)에서 왔다. 접두사 트리(prefix tree)라고도 부른다.

동기 — 자동완성

검색창에 "appl"만 쳐도 apple, application이 추천된다. 수백만 단어에서 어떻게 찰나에 찾아낼까? 배열이나 해시로는 접두사 검색이 비효율적이다.

핵심 아이디어

공통된 접두사를 공유해서 저장한다.
TEA, TED, TEN → T → E → A / D / N
T와 E는 한 번만 저장하고 마지막 글자에서 분기한다.

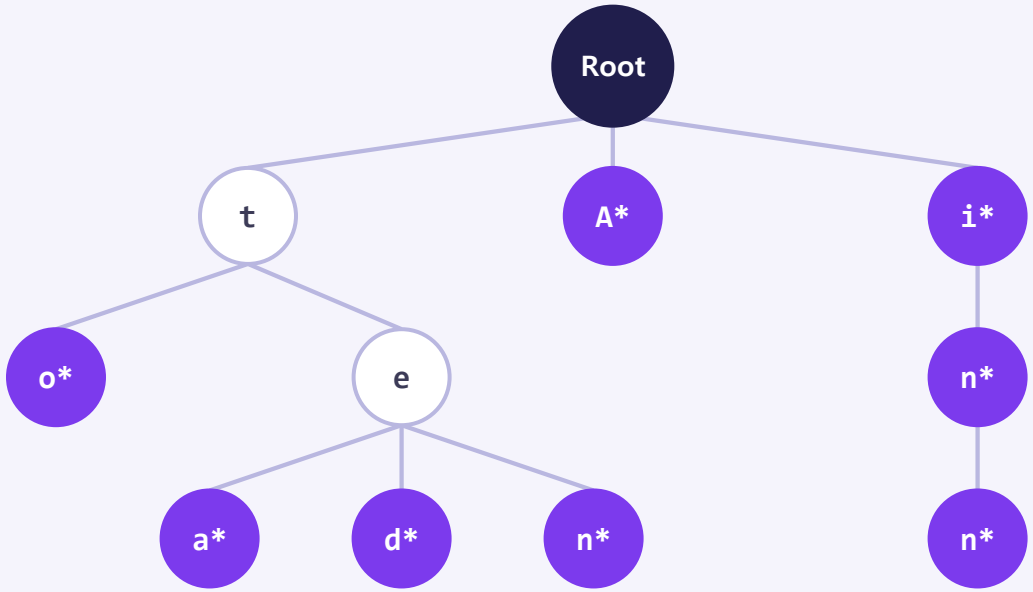
루트 노드 문자가 없는 빈 노드에서 시작한다.

간선 간선 하나가 문자 하나를 나타낸다. 경로가 곧 문자열이 된다.

단어의 끝 is_end_of_word 플래그로 표시한다. 경로 도중에도 단어가 끝날 수 있기 때문이다.

트라이 구조 예시

저장 단어 : "to", "tea", "ted", "ten", "A", "i", "in", "inn"



* = 단어의 끝 (is_end_of_word = True)

접두사를 공유한다

to·tea·ted·ten이 t를 함께 쓰고, tea·ted·ten은 te까지 함께 쓴다.

중간에서도 끝난다

i는 그 자체로 단어이면서 in·inn의 접두사이기도 하다.

경로가 곧 문자열

루트에서 특정 노드까지의 문자들을 이으면 그것이 접두사다.

트라이 자료구조와 검색

trie_find.pseudo

```
structure Node
  Children   Node[Alphabet-Size]
  Value      Data-Type
end structure

Trie-Find(x, key)
  for 0 <= i < key.length do
    if x.Children[key[i]] = nil then
      return nil
    x := x.Children[key[i]]
  return x.Value
```

① 루트에서 시작

② 각 문자를 따라 이동
자식 노드로 한 칸씩 내려간다.

③ 경로가 없으면 nil
해당 문자열은 저장되어 있지 않다.

④ 끝까지 도달하면
그 노드의 Value를 반환한다.

시간 복잡도 = O(L)

L = 찾으려는 문자열의 길이

검색 속도가 저장된 문자열 수(N)와 무관하다 — 오직 문자열 길이(L)에만 비례한다.

트라이 삽입 알고리즘

trie_insert.pseudo

```
Trie-Insert(x, key, value)

  for 0 <= i < key.length do
    if x.Children[key[i]] = nil then
      x.Children[key[i]] := Create-New-Node()
    x := x.Children[key[i]]

  x.Value := value
```

각 문자를 따라 이동
검색과 같은 경로를 밟는다.

자식이 없으면 생성
없는 갈래를 만나면 새 노드를 만들어 잇는다.

마지막 노드에 값 저장
그 자리에 단어의 끝 표시와 값을 남긴다.

연산	트라이	이진 탐색 트리
삽입	O(L)	O(L × log N)
검색	O(L)	O(L × log N)
단어 수 N의 영향	받지 않는다	log N 만큼 늘어난다

사전에 100만 개가 들어 있든 1억 개가 들어 있든 검색 속도는 일정하다.

트라이의 공간 복잡도 트레이드오프

"공짜 점심은 없다" — 속도를 얻는 대신 메모리를 희생한다.

메모리 문제

각 노드마다 알파벳 크기만큼 포인터 배열이 필요하다(영어 소문자면 26개). 단어들이 겹치지 않으면 대부분의 포인터가 NULL로 남아 심각한 낭비가 생긴다.

해결책 1 — Map / Dict

배열 대신 동적 자료구조를 써서 실제로 필요한 자식 노드만 저장한다.
장점: 메모리 절약 · 단점: 속도가 약간 느려진다

해결책 2 — 압축 트라이 (Radix Tree)

자식이 하나뿐인 경로를 하나의 노드로 합친다.
예: T → E → A 를 "TEA" 하나로 압축한다.

트라이 파이썬 구현

trie.py

```python
class TrieNode:
    def __init__(self):
        self.children = {}      # char -> TrieNode
        self.is_end = False
        self.value = None

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, key, value=None):
        node = self.root
        for ch in key:
            if ch not in node.children:
                node.children[ch] = TrieNode()
            node = node.children[ch]
        node.is_end = True
        node.value = value

    def find(self, key):
        node = self.root
        for ch in key:
            if ch not in node.children:
                return None
            node = node.children[ch]
        return node.value if node.is_end else None
```

run.py

```python
trie = Trie()
trie.insert("cat", "고양이")
trie.find("cat")
```

OUTPUT

```
"고양이"
```

dict를 자식으로 쓴 이유

알파벳 크기만 한 배열 대신 딕셔너리를 쓰면, 실제 등장한 문자에 대해서만 메모리를 쓴다. 한글처럼 문자 종류가 많은 경우에는 사실상 필수다.

12.6

데이터 압축

자주 나오는 글자는 짧게, 드문 글자는 길게.
허프만 코딩이 그 최적을 보장한다.

12.6

고정 길이 vs 가변 길이 코드

고정 길이 코드

ASCII — 모든 문자에 똑같이 8비트를 준다.

장점: 구현이 간단하고 인덱싱이 빠르다
단점: 자주 쓰는 문자와 특수문자가 같은 공간을 차지해 비효율적이다

가변 길이 코드

자주 나오는 문자는 짧게, 어쩌다 나오는 문자는 길게 준다.

예: 모스 부호 — E = ·(가장 짧음), Q = --·-(길다)
문제: 해독할 때 모호성이 생긴다

모호성 문제와 접두어 코드

'A'=0, 'B'=1, 'C'=01 이라면 01은 "AB"(0,1)인가 "C"(01)인가? → 해결책은 접두어 코드(prefix code) — 어떤 코드도 다른 코드의 접두사가 되어서는 안 된다.

허프만 코딩은 이 조건을 저절로 보장한다 — 모든 문자가 트리의 잎에만 놓이기 때문이다.

허프만 코딩 절차

1952년 데이비드 허프만(David Huffman) — 전형적인 그리디 알고리즘이다.

글자	s	i	n	t	e
빈도	4	6	8	12	15

① 빈도순으로 정렬한다 — s(4), i(6), n(8), t(12), e(15)

② 가장 작은 둘을 합친다 — s(4) + i(6) → 내부노드(10), 남은 것 [n(8), (10), t(12), e(15)]

③ n(8)+(10)→(18), t(12)+e(15)→(27), (18)+(27)→(45) 루트 완성

④ 왼쪽 간선에 0, 오른쪽 간선에 1을 붙인다.

언제나 "가장 작은 둘"을 합친다 — 최소 힙 하나면 충분하다.

허프만 코드 결과

문자	빈도	허프만 코드	비트 수
e	15	11	2
t	12	10	2
n	8	00	2
i	6	011	3
s	4	010	3

접두사 코드가 보장된다

e(11)는 t(10), n(00), i(011), s(010) 어느 것의 접두사도 아니다 → 모호성 없이 해독할 수 있다.

빈도가 높을수록 짧다

e(15)와 t(12)는 2비트, s(4)는 3비트다. 자주 나오는 글자에 짧은 코드가 자동으로 배정된다.

압축 효과

고정 길이(3비트/문자)라면 45 × 3 = 135비트. 허프만이라면 15×2 + 12×2 + 8×2 + 6×3 + 4×3 = 100비트 → 약 26% 절약된다.

빈도 차이가 클수록 압축률이 높아진다 — 모든 문자가 균등하면 이득이 없다.

허프만 코딩 핵심 파이썬 구현

huffman.py

```python
import heapq

class Node:
    def __init__(self, freq, ch=None, left=None, right=None):
        self.freq, self.ch = freq, ch
        self.left, self.right = left, right

def build_huffman_tree(freq_map):
    heap = []
    uid = 0
    for ch in freq_map:
        heapq.heappush(heap, (freq_map[ch], uid, Node(freq_map[ch], ch)))
        uid += 1

    while len(heap) > 1:
        f1, _, n1 = heapq.heappop(heap)     # 가장 작은 것
        f2, _, n2 = heapq.heappop(heap)     # 그다음 작은 것
        parent = Node(f1 + f2, None, n1, n2)
        heapq.heappush(heap, (parent.freq, uid, parent))
        uid += 1

    return heapq.heappop(heap)[2]
```

최소 힙이 전부다

가장 작은 빈도의 노드 둘을 반복해서 합친다. 그리디 전략이 그대로 코드가 된다.

uid를 넣은 이유

빈도가 같을 때 Node끼리 비교를 시도하다 오류가 난다. 그 사이에 유일한 정수를 끼워 막는다.

힙에 하나만 남으면

그것이 루트다. 여기서부터 왼쪽 0, 오른쪽 1을 붙이며 내려가면 코드가 완성된다.

12.7

코딩 테스트 전략

12.7

라이브러리를 1순위로,
알고리즘을 보험으로.

코딩 테스트에서의 텍스트 알고리즘

기본 유형 (1~2번 문제)

· 문자열 자르기(split)와 특정 문자 치환

· 팰린드롬(회문) 확인

· 진법 변환, 괄호 짝 맞추기

언어 라이브러리 숙달이 핵심

고급 유형 (변별력 문제)

· 트라이 — 자동완성, 가사 검색

· KMP — 패턴 매칭 최적화

· 모르면 시간 초과로 절대 못 푼다

많은 지원자가 포기하는 영역

파이썬 in, find() — 내부적으로 보이어-무어와 호스풀을 섞어 쓴다. C로 최적화되어 있어 직접 짠 KMP보다 빠르다.

C++ std::string::find() — 대부분 naive 방식이다. 최악 입력에서는 KMP가 필요할 수 있다.

Java indexOf() — brute-force 이중 루프다. 긴 문자열에서는 KMP 직접 구현을 고려한다.

실전 전략 — 라이브러리를 1순위로, 알고리즘을 보험으로!

실전 문제 — 가장 긴 팰린드롬

문자열 s에서 가장 긴 팰린드롬(회문) 부분 문자열을 찾아라.

palindrome.py

```python
def longestPalindrome(s: str) -> str:
    if len(s) < 2 or s == s[::-1]:
        return s

    result = ""

    def expand(left, right):
        while left >= 0 and right < len(s) \\
              and s[left] == s[right]:
            left -= 1; right += 1
        return s[left + 1 : right]

    for i in range(len(s) - 1):
        result = max(result,
                     expand(i, i + 1),   # 짝수 길이
                     expand(i, i + 2),   # 홀수 길이
                     key=len)

    return result
```

OUTPUT

```
longestPalindrome("babad")  →  "bab"
```

중심 확장 기법

각 위치에서 "내가 중심이라면 어디까지 뻗을 수 있나"를 물어 좌우로 넓혀 나간다.

짝수와 홀수를 모두

"bb"처럼 짝수 길이와 "bab"처럼 홀수 길이를 각각 확인해야 빠뜨리지 않는다.

$O(n^2)$ 시간 · $O(1)$ 공간

SUMMARY

12장 정리

완전 탐색

O(N × M) — 한 칸씩 밀며 그대로 비교한다. 비교 정보를 전혀 재활용하지 않는다.

KMP

O(N + M) — LPS 배열로 점프한다. 텍스트 인덱스가 절대 후퇴하지 않는다.

보이어-무어

평균 O(N/M) — 뒤에서부터 비교하고 나쁜 문자로 크게 건너뛴다. 에디터의 Ctrl+F가 이것이다.

라빈-카프

평균 O(N + M) — 문자열을 해시로 바꾼다. 다중 패턴 검색에서 특히 강하다.

트라이

O(L) — 접두사를 공유해 저장한다. 검색 속도가 저장된 단어 수와 무관하다.

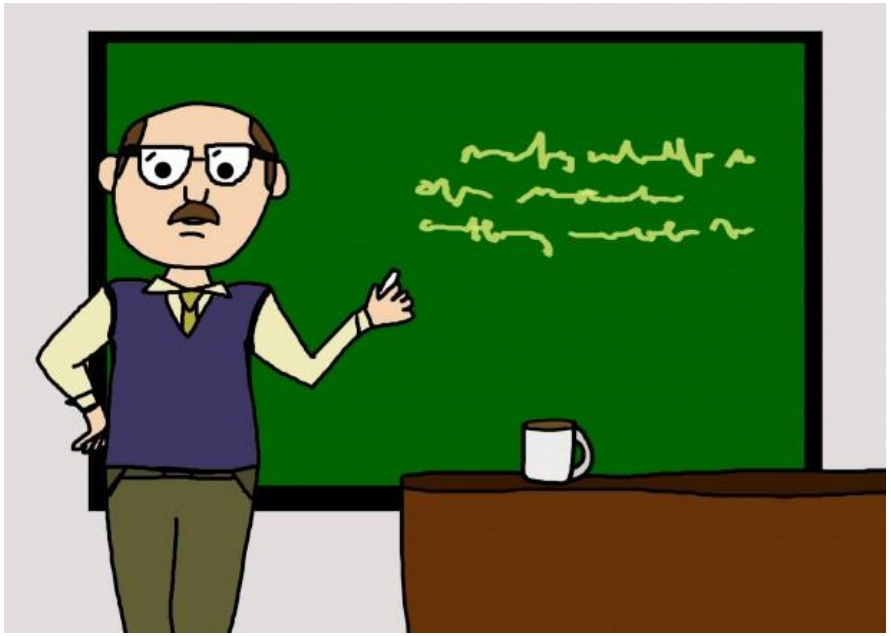
허프만 코딩

그리디로 가변 길이 접두어 코드를 만든다. 빈도 차이가 클수록 압축률이 높다.

CHAPTER 12 · 마지막 장

Q & A

"이 책은 여기서 끝나지만,
여러분의 코딩은 이제부터가 진짜 시작이다"



1장부터 12장까지 — 정렬에서 텍스트 알고리즘까지