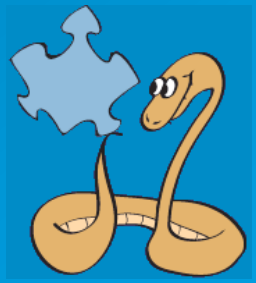
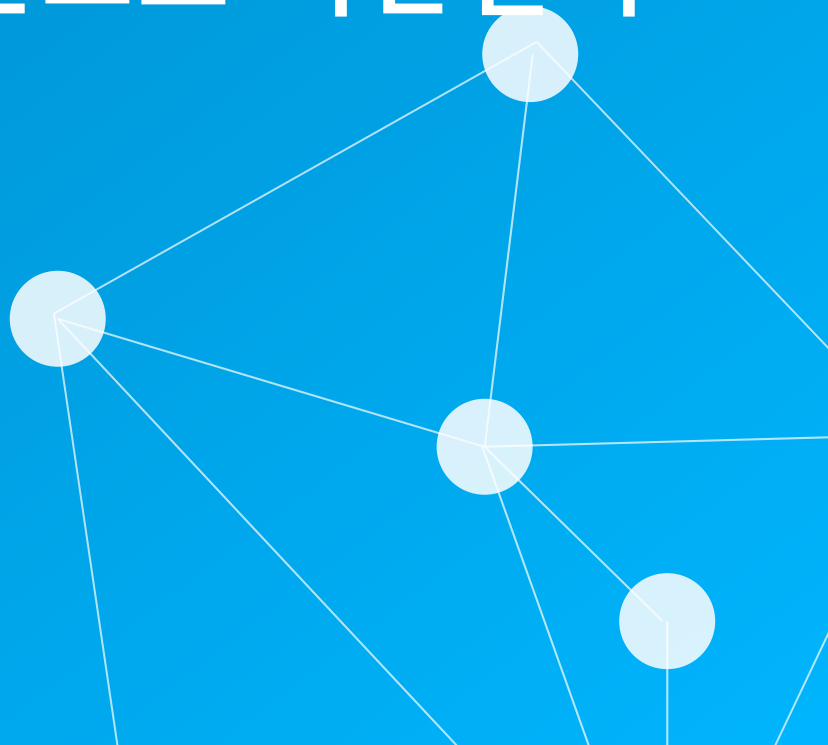
파이썬 알고리즘

ALGORITHMS IN PYTHON 최영규 지음

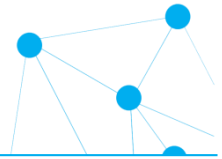


CHAPTER 06

공간으로 시간벌기

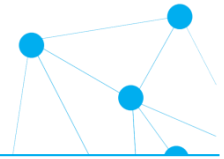


알고리즘의 설계 기법들



- 억지(brute-force) 기법과 완전 탐색: 3장
  - 문제 정의를 가장 직접 사용, 원하는 답 구할때까지 모든 경우 테스트
- 축소 정복(decrease-and-conquer): 4장
  - 주어진 문제를 하나의 좀 더 작은 문제로 축소하여 해결
- 분할 정복(divide-and-conquer): 5장
  - 주어진 문제를 여러 개의 더 작은 문제로 반복적으로 분할하여 해결 가능한 충분히 작은 문제로 분할 후 해결
- **공간을 이용해 시간을 버는 전략: 6장**
  - **추가적인 공간을 사용하여 처리시간 줄이는 전략**
- 동적 계획법(divide-and-conquer): 7장
  - 더 작은 문제로 나누는 분할정복과 유사하지만, 작은분제 먼저해결 저장하고 다음에 더 큰 문제 해결
- 탐욕적(greedy) 기법: 8장
  - 단순하고 직관적인 방법으로 모든 경우 고려하여 가장 좋은 답을 찾는 것이 아니라 "그 순간에 최적"이라고 생각되는 것을 선택
- 백트래킹과 분기 한정 기법: 9장
  - 상태공간에서 단계적 해 찾기, 현재의 최종 해가 않된다면 더 이상 탐색하지 않고 백트래킹(되돌아가서)해서 다른 후보 해 탐색

학습 내용



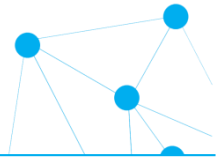
6.1 기수 정렬

6.2 카운팅 정렬

6.3 문자열 매칭

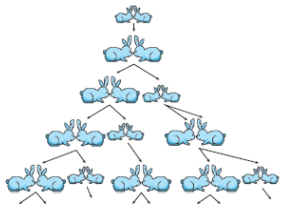
6.4 해싱(Hashing)

공간으로 시간 벌기



- 시간과 공간의 효율성은 항상 상충됨
  - 모든 답을 미리 다 계산해 저장해 놓는다면

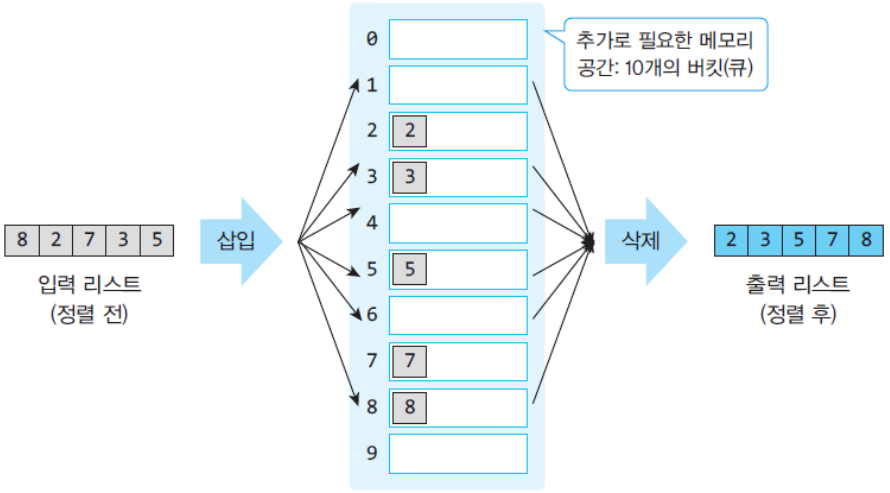
  → 모든 알고리즘은 O(1)이 가능
  - 예: 피보나치 수열

피보나치수열 문제		알고리즘 종류	시간 복잡도	공간 복잡도
	1	분할 정복 기법: 알고리즘 5.11	$O(n^2)$	$O(1)$
	2	반복 구조: 알고리즘 5.12	$O(n)$	$O(1)$
	3	축소 정복 기법의 행렬 거듭제곱 이용: 알고리즘 5.13	$O(\log_2 n)$	$O(1)$
	4	미리 답이 계산된 테이블 이용	$O(1)$	$O(n)$

- 문제는? 공간!
  - 현실적인 수준에서 공간을 희생해 시간 효율성을 높이는 방법들
  - 정렬, 문자열 매칭, 해싱 등

6.1 기수 정렬

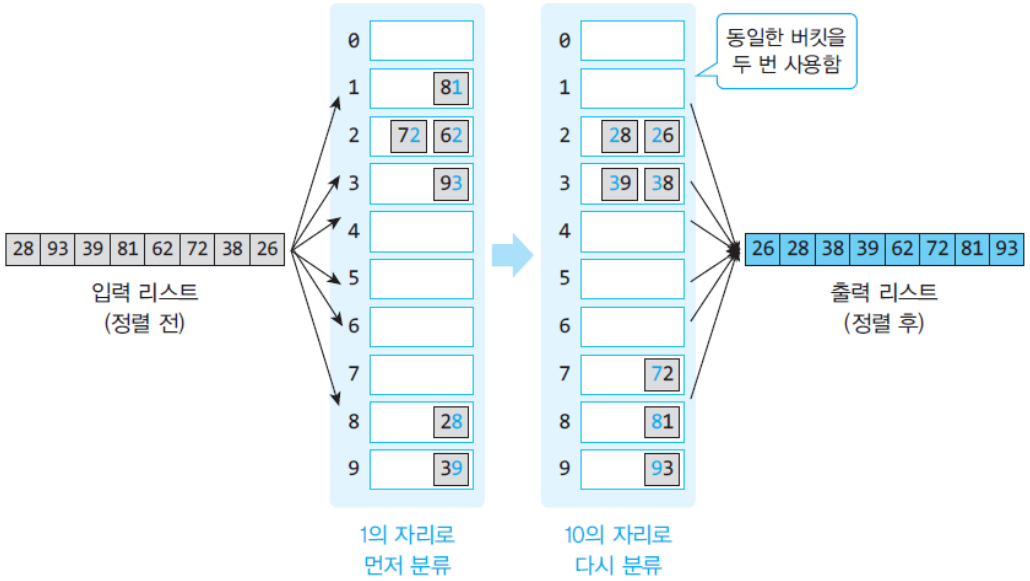
- 비교 기반 정렬
  - 적절한 배분을 사용
- 배분을 이용한 정렬 아이디어



  - 예상되는 문제들은?

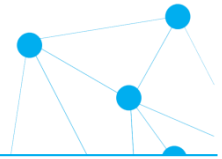
여러 자리 숫자의 정렬

- 아이디어: 낮은자리부터 정렬



- 버킷의 형태: FIFO → Queue

알고리즘



알고리즘 6.1 기수 정렬

```
01  from queue import Queue
02  def radix_sort(A) :
03      queues = []
04      for i in range(BUCKETS) :
05          queues.append(Queue())
06
07      n = len(A)
08      factor = 1
09      for d in range(DIGITS) :
10          for i in range(n) :
11              queues[(A[i]//factor) % 10].put(A[i])
12          j = 0
13          for b in range(BUCKETS) :
14              while not queues[b].empty() :
15                  A[j] = queues[b].get()
16                  j += 1
17          factor *= 10
18          print("step", d+1, A)
```

```
C:₩WINDOWS₩system32₩cmd.exe
step 1 [3790, 2850, 5162, 4122, 1043, 6894, 5425, 2706, 2267, 1679]
step 2 [2706, 4122, 5425, 1043, 2850, 5162, 2267, 1679, 3790, 6894]
step 3 [1043, 4122, 5162, 2267, 5425, 1679, 2706, 3790, 2850, 6894]
step 4 [1043, 1679, 2267, 2706, 2850, 3790, 4122, 5162, 5425, 6894]
Radix: [1043, 1679, 2267, 2706, 2850, 3790, 4122, 5162, 5425, 6894]
```

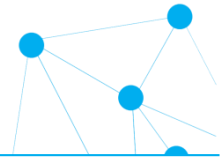
일, 십, 백, 천의 자리 순으로 정렬

최종 정렬 결과

알고리즘 테스트 기수 정렬 테스트

```
import random          # 테스트를 위한 난수
BUCKETS = 10           # 10진법으로 정렬
DIGITS  = 4            # 최대 4 자릿수
data = []
for i in range(10) :
    data.append(random.randint(1,9999))
radix_sort(data)
print("Radix: ", data)
```

복잡도 분석

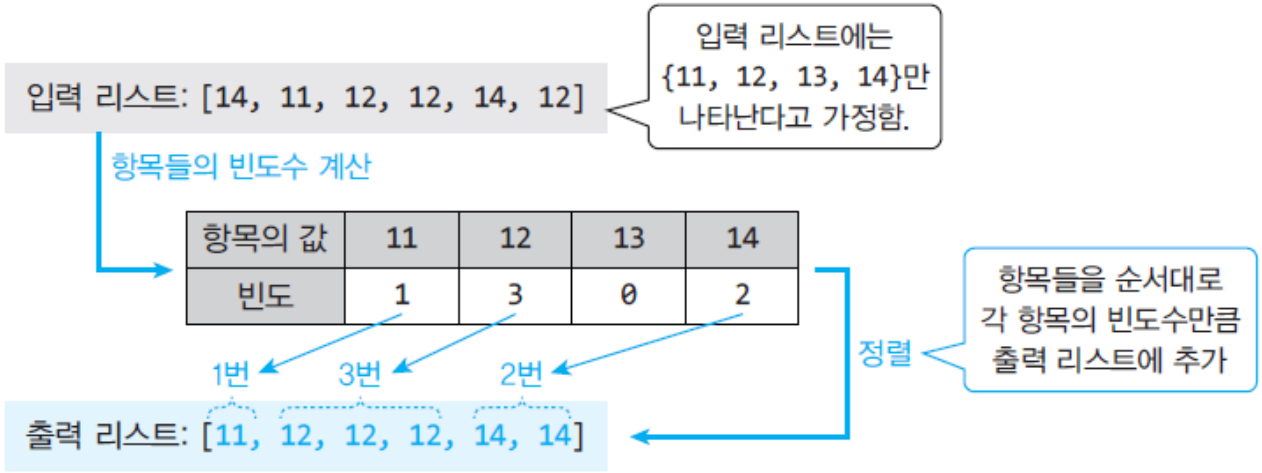


- 시간 복잡도
  - $O(dn)$
- 공간 복잡도
  - $O(n)$

- 특징
  - 킷값이 자연수로 표현되어야만 적용 가능. Why?
  - 실수나 한글, 한자 등으로 이루어진 킷값에 대해서는 거의 불가

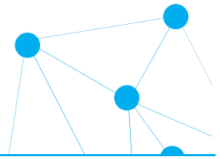
6.2 카운팅 정렬

- 리스트의 각 항목들을 단순히 세는 방법으로 정렬
- 아이디어

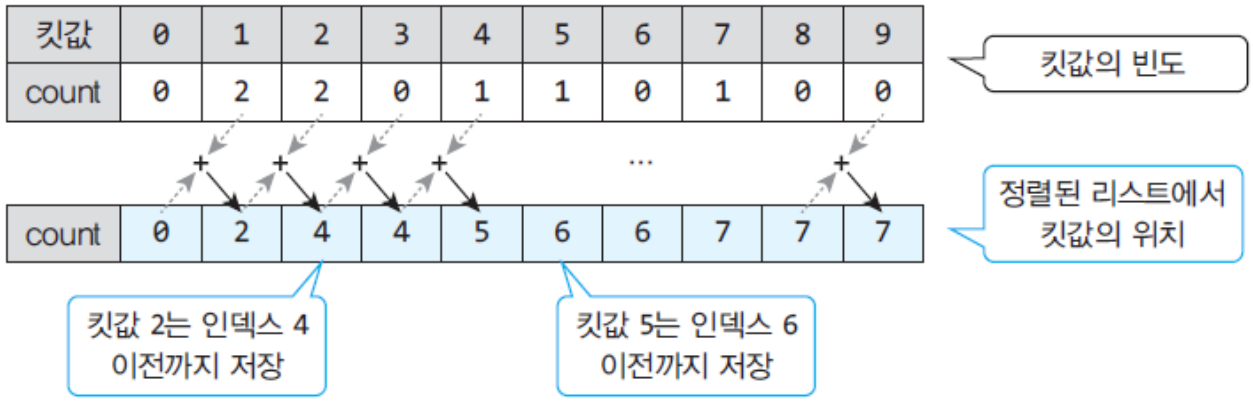


- 복잡도?
- 문제점?

알고리즘



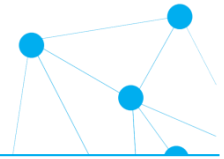
- 리스트 [1, 4, 1, 2, 7, 5, 2] 의 정렬



알고리즘 6.2 카운팅 정렬

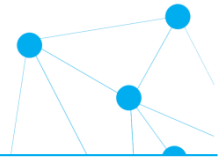
```
01  def counting_sort(A):
02      output = [0] * len(A)
03      count  = [0] * MAX_VAL
04
05      for i in A:
06          count[i] += 1
07
08      for i in range(MAX_VAL):
09          count[i] += count[i-1]
10
11      for i in range(len(A)):
12          output[count[A[i]]-1] = A[i]
13          count[A[i]] -= 1
14
15      for i in range(len(A)):
16          A[i] = output[i]
```

복잡도 분석

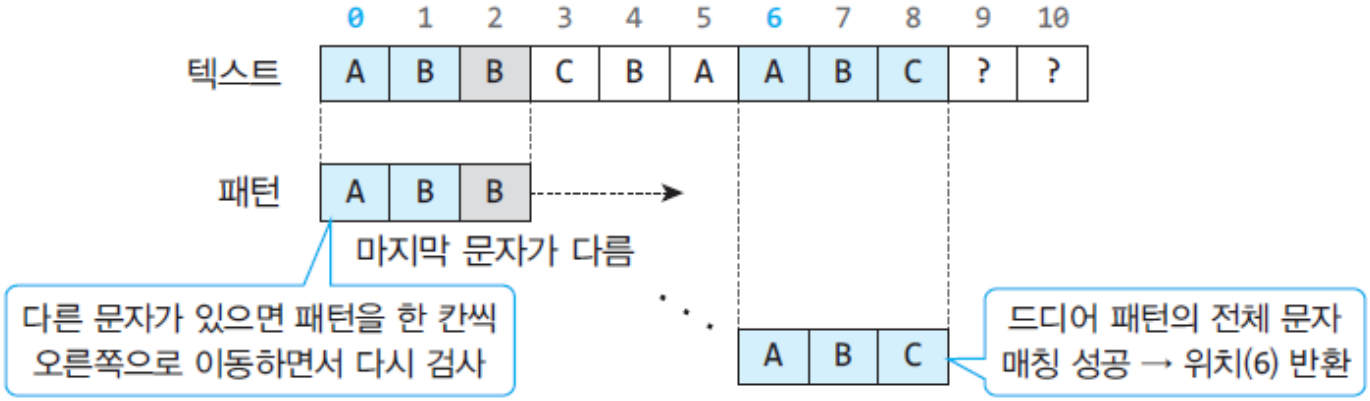


- 시간 복잡도
  - $O(k+n)$
- 공간 복잡도
  - $O(k+n)$
  - k가 매우 크다면? 예) 실수

6.3 문자열 매칭



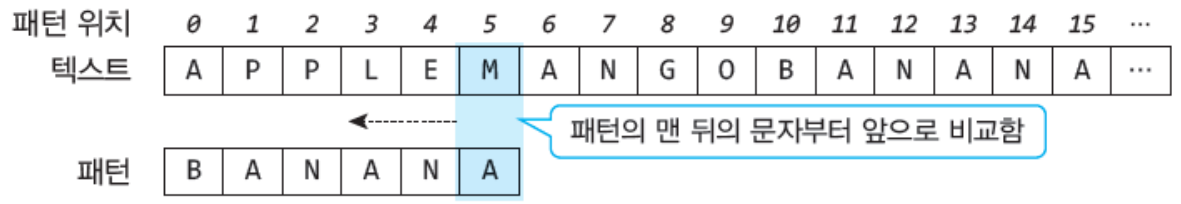
- 길이가 n인 텍스트에서 길이가 m인 패턴 찾기
- 억지 기법: 알고리즘 3.3
  - $O(mn)$



- 개선된 방법: 전처리를 이용한 정보 저장 및 활용
  - KMP(Knuth-Morris-Pratt)
  - 호스풀(Horspool) 알고리즘
  - 보이어 무어 (Boyer-Moore)

호스풀(Horspool) 알고리즘

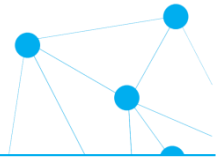
- 기본 전략



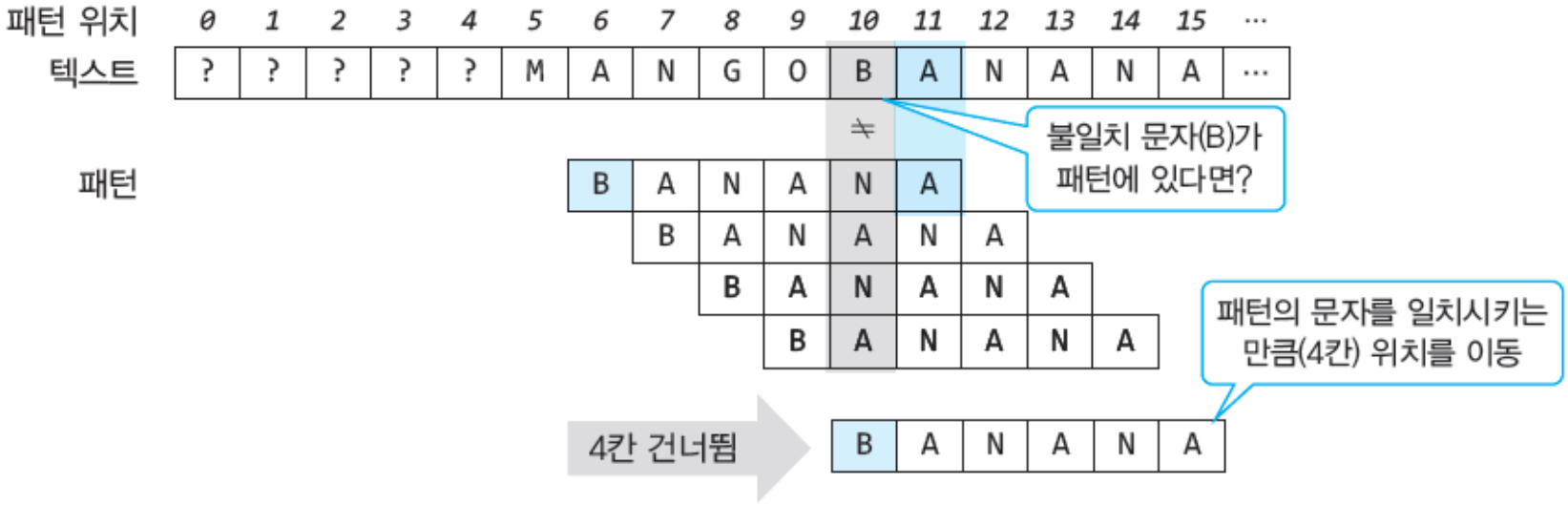
[그림 6.8] 패턴(BANANA)의 맨 뒤 문자부터 앞으로 비교함

- 불일치 문자가 패턴에 없는 문자이면?

시프트 테이블



- 불일치 문자가 패턴에 있으면?

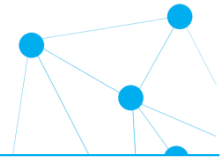


- 전처리 필요: shift_table(pat): 알고리즘 6.3
  - 모든 가능한 알파벳에 대해 패턴에서의 위치를 저장한 표

인덱스	0	…	…	…	…	64	65	66	…	78	79	…	…	…	…	…	127
알파벳	NUL	…	…	…	…	@	A	B	…	N	M	…	…	…	…	…	DEL
이동 거리	6	6	…	…	…	6	2	5	…	1	6	…	…	…	…	6	6

[그림 6.11] 텍스트가 아스키 문자인 경우의 시프트 테이블 예(패턴은 BANANA)

알고리즘과 복잡도



- 알고리즘

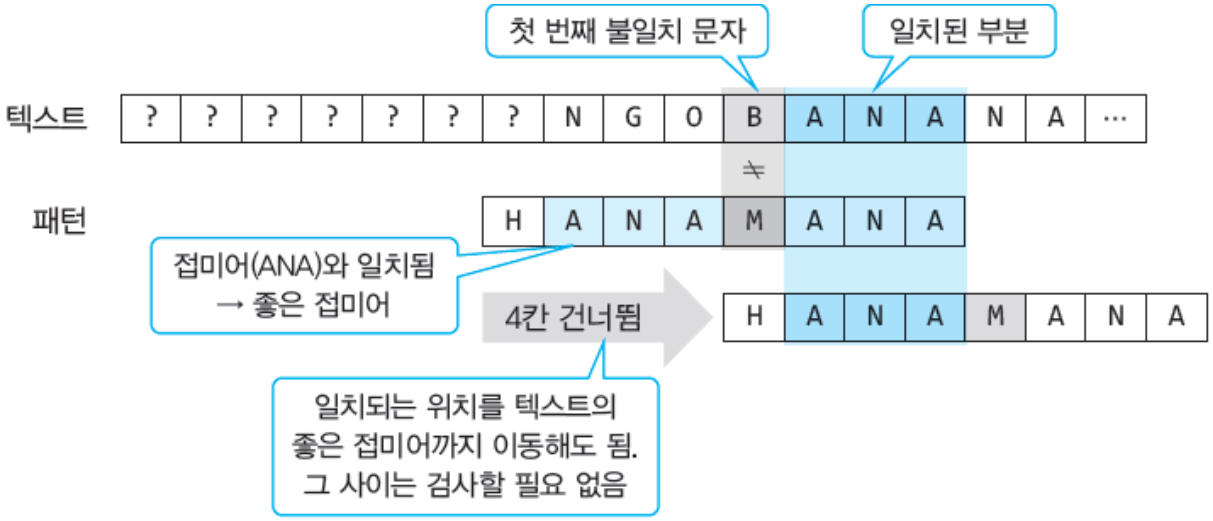
알고리즘 6.4 호스풀 알고리즘

```
01  def search_horspool(T, P):
02      m = len(P)
03      n = len(T)
04      t = shift_table(P)
05      i = m-1
06      while(i <= n-1):
07          k = 0
08          while k <= m-1 and P[m-1-k]==T[i-k]:
09              k += 1
10          if k == m :
11              return i-m+1
12          else :
13              i += t[ord(T[i])]
14      return -1
```

- 복잡도
  - $T(m,n) \in O(mn)$
  - 무작위 텍스트에 대해 거의 $O(n)$의 성능. *Why?*

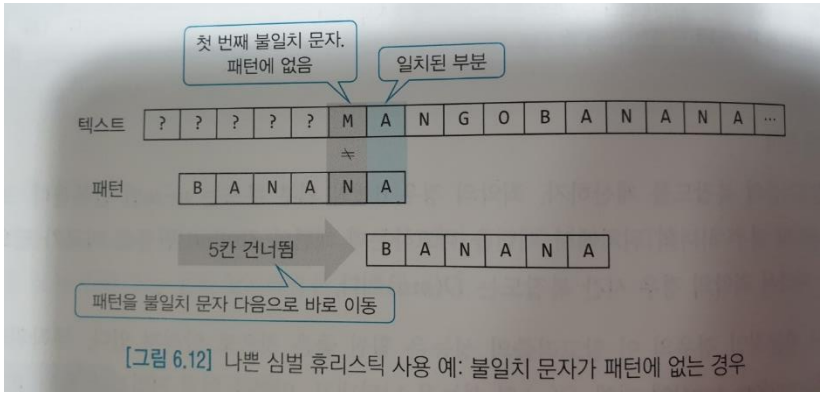
보이어 무어(Boyer-Moore) 알고리즘(심화)

- 나쁜 심벌 휴리스틱(bad symbol heuristic)
- 좋은 접미어 휴리스틱(good suffix heuristic)

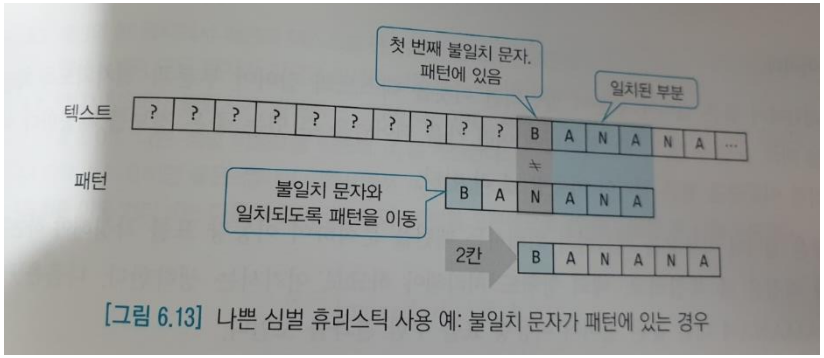


- 좋은 접미어 이동량 표

좋은 접미어의 길이	패턴	이동량(d_2)
1	N A M A N A	2
2	N A M A N A	4
3	N A M A N A	4
4	N A M A N A	4
5	N A M A N A	4

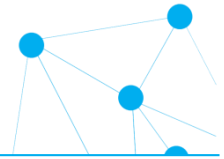


[그림 6.12] 나쁜 심벌 휴리스틱 사용 예: 불일치 문자가 패턴에 없는 경우



[그림 6.13] 나쁜 심벌 휴리스틱 사용 예: 불일치 문자가 패턴에 있는 경우

알고리즘

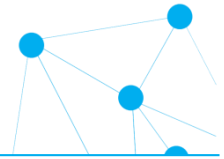


알고리즘 6.5 보이어-무어 알고리즘(자연어로 표현)

1. 주어진 패턴과 텍스트에서 사용된 알파벳을 이용해 나쁜-심벌 이동표를 만든다. 알고리즘 6.3을 이용한다.
2. 패턴을 이용해서 좋은-접미어 이동표를 만든다.
3. 패턴을 텍스트의 맨 앞에 일치시킨다.
4. 일치한 패턴을 찾거나 모든 텍스트의 검사가 끝날 때까지 다음을 반복한다.
   4.1: 패턴의 한 위치에서 패턴과 텍스트를 맨 뒤에서 앞으로 비교한다.
   4.2: 만약 m개의 모든 문자가 같으면 탐색은 성공이다. 위치를 반환하고 종료한다.
   4.3 그렇지 않으면, 불일치가 발생하는 경우이다. 이제, 처음 불일치가 발생하는 위치 k를 찾는다 ($k \geq 0$). 나쁜-심벌 이동표를 이용해 d_1 을 계산한다.
   4.4 만약 $k > 0$이면 좋은-접미어 이동표에서 d_2를 찾는다.
   4.5 최종 이동 거리 d는 다음과 같이 결정된다. 패턴을 오른쪽으로 d만큼 이동한 후 4.1을 반복한다.

$$d = \begin{cases} d_1 & \text{if } k = 0 \\ \max(d_1, d_2) & \text{if } k > 0 \end{cases}$$

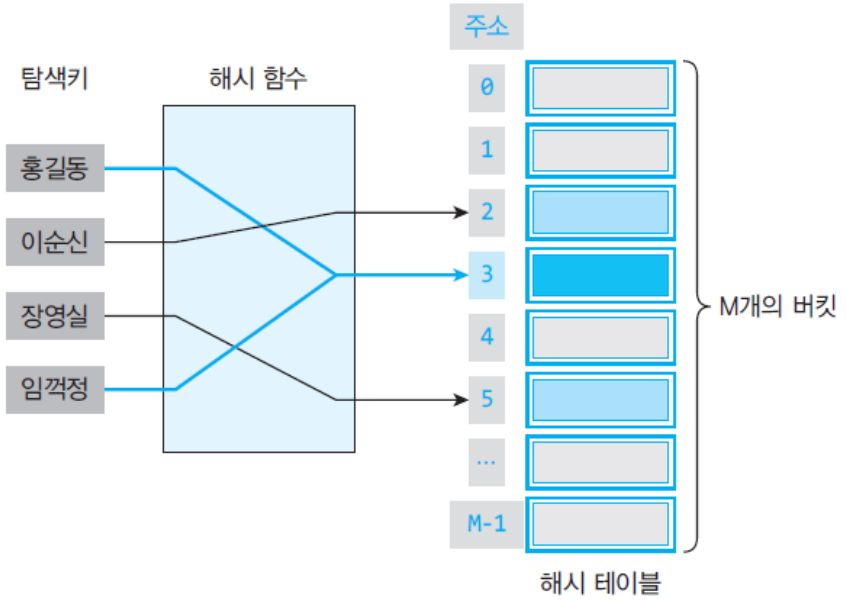
6.4 해싱(Hashing)



- 공간으로 시간 벌기 전략의 가장 대표적인 예
  - 이상적인 경우 시간 복잡도: $O(1)$!
- 아이디어

> 탐색키를 테이블에 있는 레코드와 하나씩 비교하는 것이 아니라 킷값에 산술적인 연산을 적용하여 레코드가 저장되어야 할 위치를 직접 계산한다. 즉, 탐색키로부터 레코드가 있어야 할 위치를 바로 계산하고, 그 위치에 레코드가 있는지만 확인하면 된다.

- 해싱의 구조
  - 해시 함수
  - 해시 테이블, 버킷, 슬롯
  - 해시 충돌, 동의어
  - 오버플로



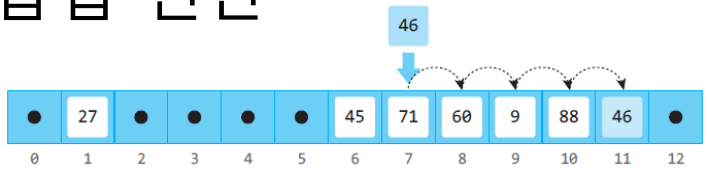
선형 조사에 의한 오버플로 처리

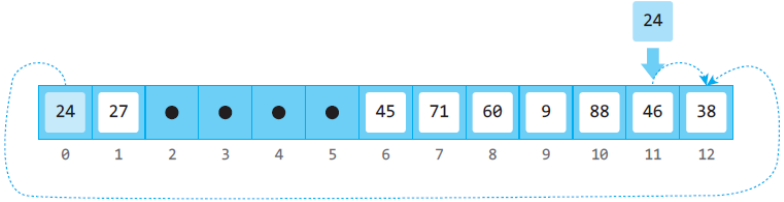
해시 함수로 계산된 버킷에 빈 슬롯이 없으면 그 다음 버킷들을 순서적으로 조사하여 빈 슬롯이 있는지를 찾는다. 이때 비어 있는 공간을 찾는 것을 **조사(probing)**라고 한다.

- 예) 킷값이 45, 27, 88, 9, 71, 60, 46, 38, 24인 레코드
  - 테이블의 크기 M=13
  - h(k) = k%M

key	45	27	88	9	71	60	46	38	24
h(key)	6	1	10	9	6	8	7	12	11

- 삽입 연산



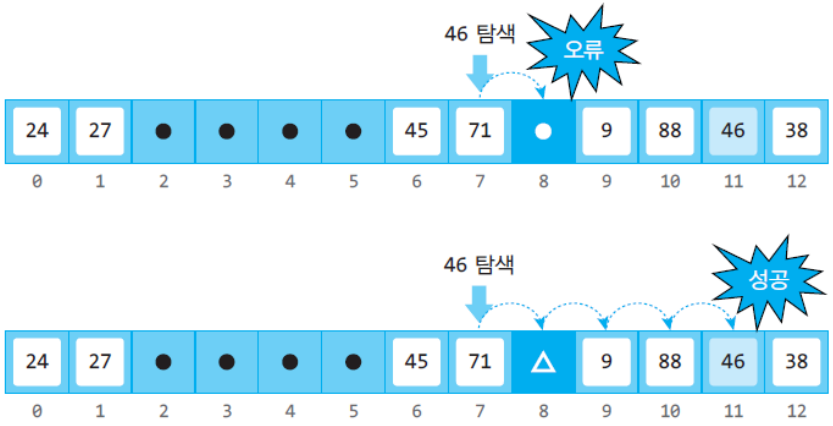


  - 군집화(clustering) 현상

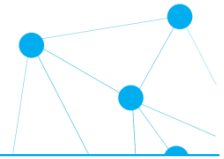
선형 조사법

- 탐색 연산

- 삭제 연산
  - 빈 버켓을 두 가지로 분류해야 함



군집화 완화 방법



- 이차 조사법(quadratic probing)

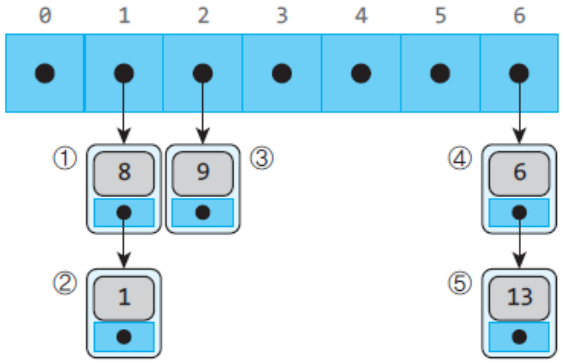
$$(h(k)+i*i)\,\%\,M \quad \text{for} \quad i=0,1,\cdots,M-1$$

- 이중 해싱법(double hashing)
  - 원래 해시 함수와 다른 별개의 해시 함수를 이용

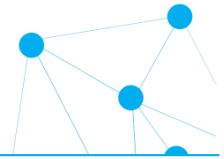
체이닝(chaining)에 의한 오버플로 처리

선형 조사법과 달리 하나의 버킷에 여러 개의 레코드를 저장할 수 있도록 하는데, 이때 버킷은 크기를 변경할 수 있는 리스트 구조로 구현한다. 따라서 하나의 버킷에서 아무리 많은 충돌이 발생하더라도 문제없이 처리할 수 있다.

- 예) 킷값이 8, 1, 9, 6, 13 삽입
  - 테이블의 크기 M=7
  - H(k) = k%M

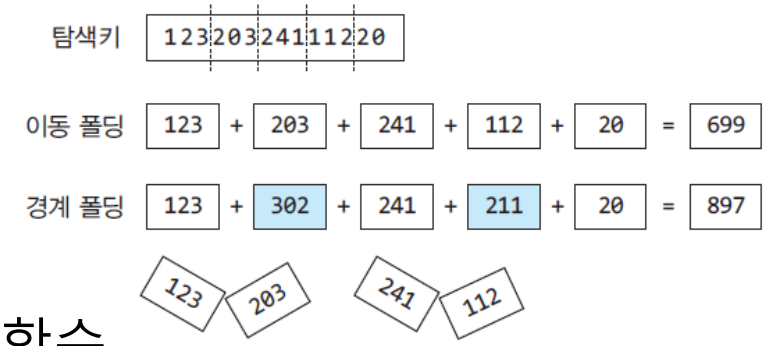


해시 함수



- 좋은 해시 함수
  - 충돌이 적고, 주소가 테이블에서 고르게 분포되며, 계산이 빠름

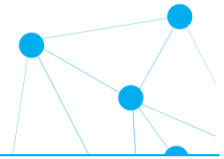
- 종류
  - 제산 함수: $h[k] = k \bmod M$
    - M은 소수(prime number)
  - 폴딩 함수



  - 중간 제곱 함수
  - 비트 추출 방법
  - 숫자 분석 방법
  - 탐색키가 문자열인 경우: 예)

```
01  def hashFn(key) :
02        sum = 0
03        for c in key :
04              sum = sum +  ord(c)
05        return sum % M
```

탐색 방법들의 성능 비교



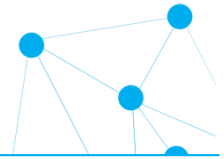
- 실제 해싱의 탐색 연산은 이상적인 *O(1)* 보다 느림
- 해시의 성능
  - 적재 밀도(loading density), 적재 비율(loading factor)

$$\alpha = \frac{\text{저장된 항목의 개수}}{\text{해시테이블의 버킷의 수}} = \frac{n}{M}$$

- 다양한 탐색 방법의 성능 비교

탐색 방법		탐색	삽입	삭제
순차 탐색		$O(n)$	$O(1)$	$O(n)$
이진 탐색		$O(\log_2 n)$	$O(\log_2 n+n)$	$O(\log_2 n+n)$
이진탐색트리	균형트리	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
	경사트리	$O(n)$	$O(n)$	$O(n)$
해싱	최선의 경우	$O(1)$	$O(1)$	$O(1)$
	최악의 경우	$O(n)$	$O(n)$	$O(n)$

실습 과제



감사합니다!