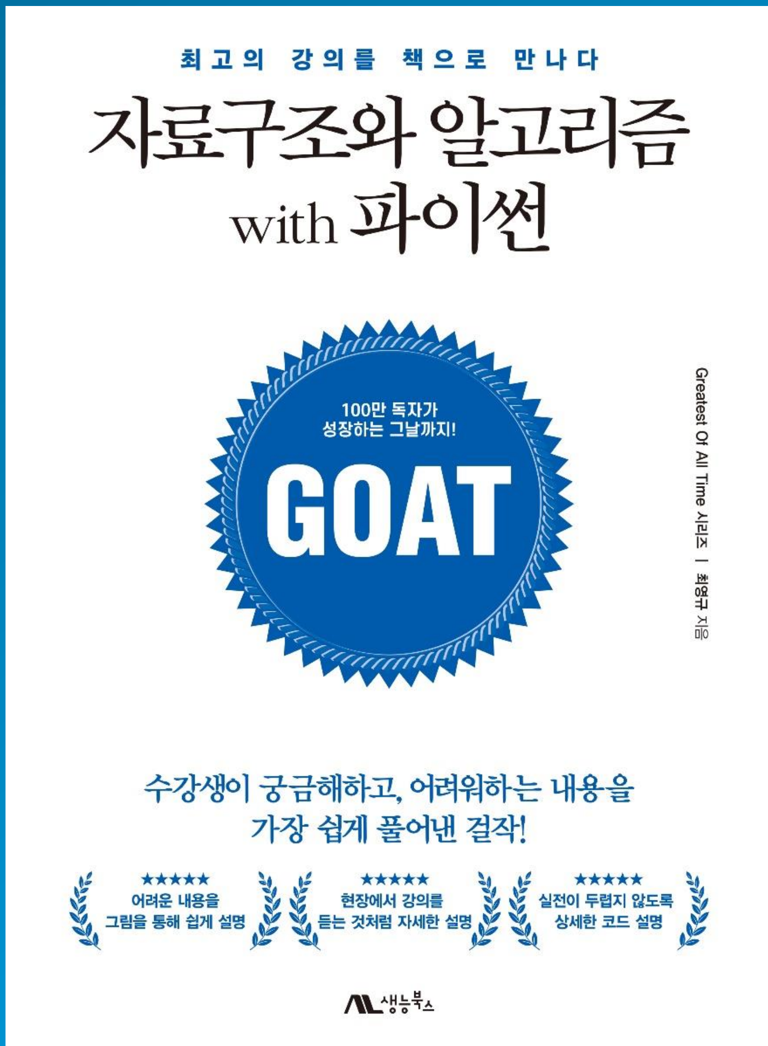
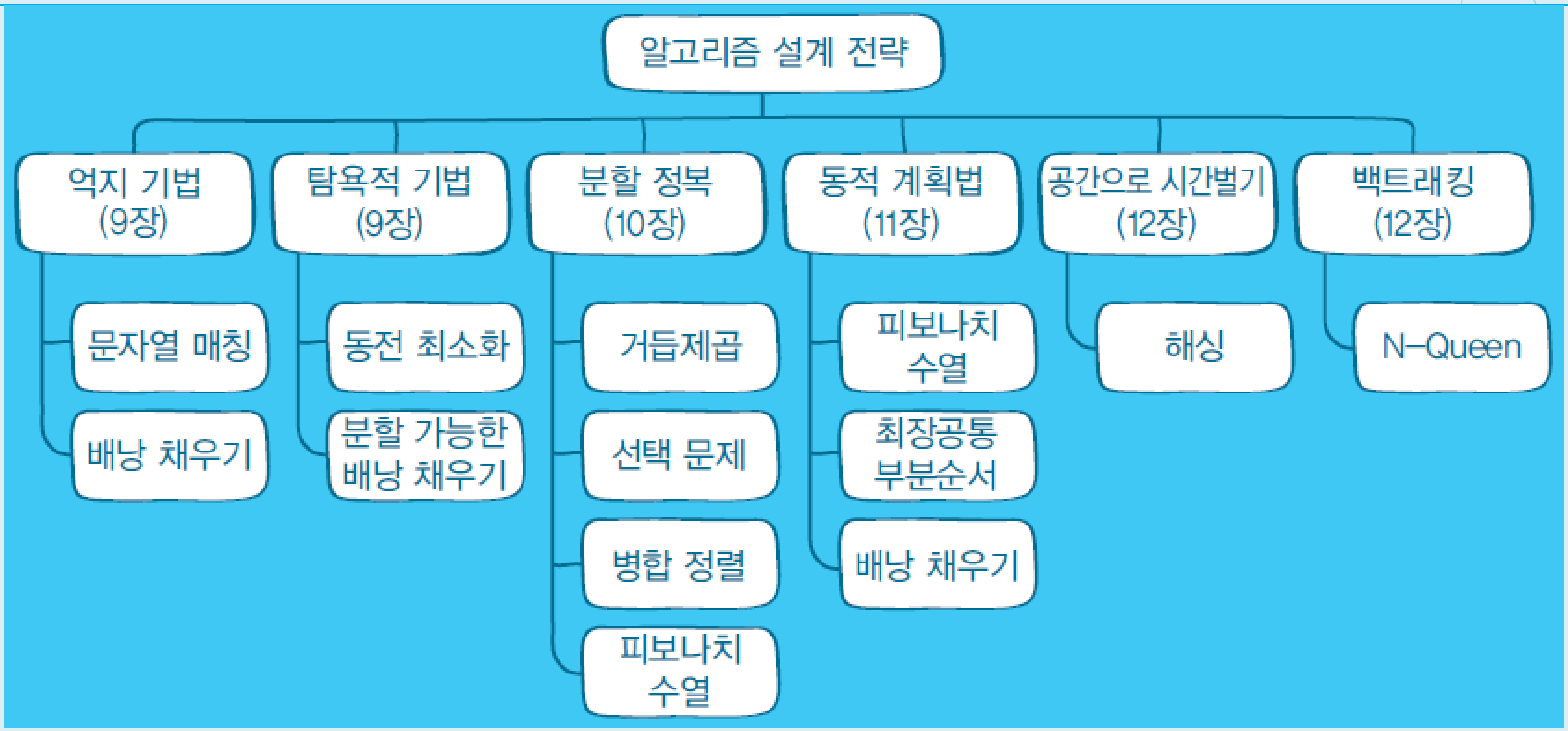


PART 3

알고리즘 설계전략

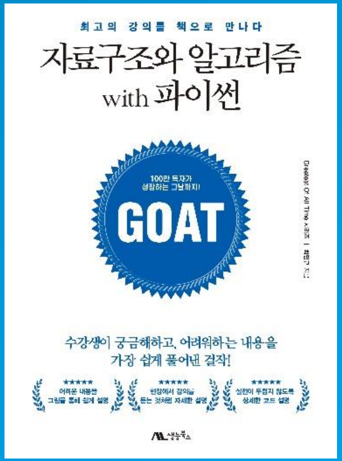
Part3. 알고리즘 설계 전략



- **목표**: 알고리즘 설계의 전반적인 흐름과 전략을 간략히 소개.
- **내용**:
  - 알고리즘 설계 전략의 분류: 탐욕적 기법, 분할 정복, 동적 계획법, 백트래킹.
  - 문제 유형에 따라 적합한 전략을 선택하는 방법.
  - 알고리즘 설계 시 고려할 시간복잡도와 공간복잡도.

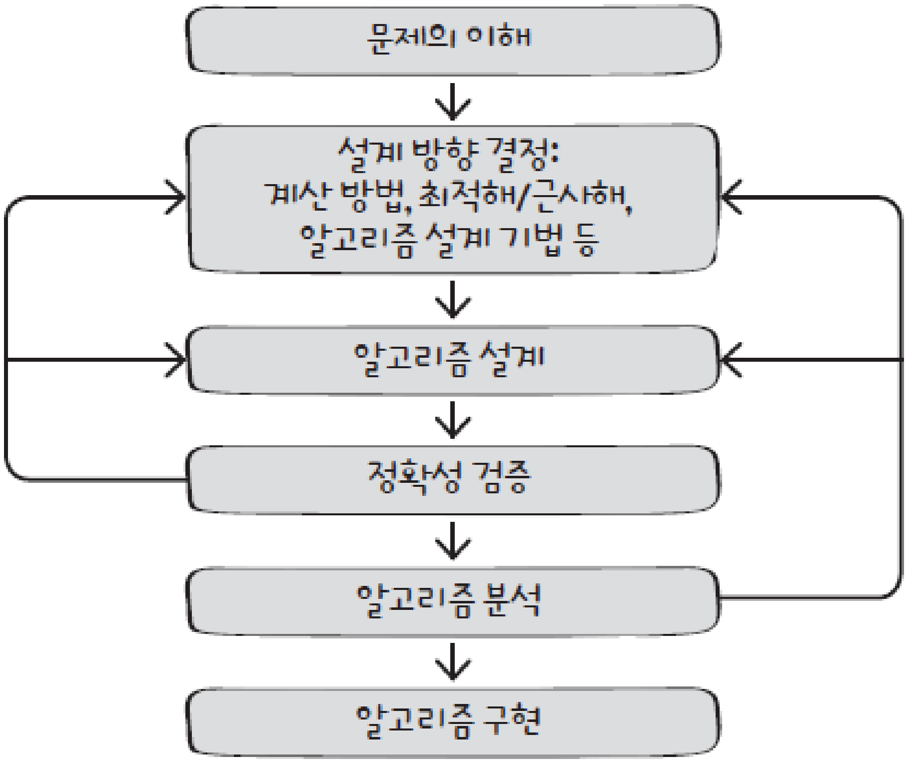
CHAPTER 09

억지 기법과 탐욕적 전략



9.1 문제 해결 과정

- 알고리즘 개발 과정



개발자로서 꼭 기억해야 할 사항

1. **문제 이해**: 문제의 입력, 출력, 제약 조건을 명확히 파악.
2. **설계 전략 선택**: 문제 유형에 따라 가장 적합한 알고리즘 설계 전략을 선택.
3. **최적화 고려**:
   - 단순 구현보다 효율성을 고려.
   - 테스트 케이스를 다양하게 설정하여 검증.
4. **성능 분석**:
   - 시간과 공간 복잡도를 계산하여 최적화 필요성을 판단.

알고리즘 개발 과정

- **문제의 이해**
  - 핵심 질문: 문제의 입력과 출력, 제약 조건
  - 간단한 입력에 대한 해답을 구해보고, 문제의 구조를 파악
  - 특수한 경우(예외 상황)에 대해 생각
  - 예: 거스름돈 문제에서 동전의 종류와 목표 금액, 최소 동전 개수

- **설계 방향 결정**
  - 순차적(sequential) : 순차적으로 실행
  - 병렬처리(parallel) : 여러 연산을 동시에 수행
  - 최적해 : 모든 조건을 만족하는 가장 좋은 답
  - 근사해 : 실행 속도를 높이기 위해 최적에 가까운 답

알고리즘 개발 과정

- 알고리즘 설계 전략들
  - 억지(brute-force) 기법
    - 가능한 모든 경우를 전부 탐색
    - 장점: 항상 정확한 답을 찾음
    - 단점 : 느림
    - 예: 순열, 조합 계산, 배낭 채우기
  - 탐욕적(greedy) 기법
    - 현재 단계에서 최선의 선택을 반복
    - 장점: 간단, 빠름
    - 단점: 모든 문제에 적용할 수 없음
    - 예: 동전 거스름돈 문제(X), 배낭 채우기(x), 분할 가능한 배낭 채우기
  - 분할 정복(divide-and-conquer)
    - 문제를 작은 문제로 나누고 결과를 병합
    - 장점: 재귀적으로 해결 가능, 병렬 처리 가능
    - 단점: 재귀 오버헤드 발생
    - 예: 병합 정렬

알고리즘 개발 과정

- 알고리즘 설계 전략들
  - 동적 계획법(dynamic programming)
    - 문제를 작은 부분 문제로 나눠 결과 저장
    - 장점: 중복 계산 제거, 최적해 보장
    - 단점: 메모리 사용량이 큼
    - 예: 피보나치 수열
  - 공간으로 시간을 버는 전략
    - 속도를 위해 메모리를 더 쓰거나, 메모리를 아끼기 위해 계산을 더 수행
    - 공간 사용과 계산 시간 균형 고려 (공간/시간 Trade-off)
    - 장점: 효율적 설계
    - 예: 해시 테이블을 이용한 해싱
  - 백트래킹과 분기한정 기법
    - 모든 경우를 탐색하면서 조건에 맞지 않으면 가지치기 사용
    - 장점: 최적해 보장
    - 단점: 탐색 공간이 크면 느림
    - 예: N-Queen

알고리즘 개발 과정

- 알고리즘의 정확성
  - 알고리즘이 항상 올바른 결과를 반환하는지 확인하는 과정
  - 실험적 분석 : 다양한 입력값으로 테스트
  - 증명적 분석 : 수학적 귀납법 등을 사용해 논리적으로 증명
- 알고리즘 성능 분석 : 점근 표기법
  - 시간 효율성 :
    - 알고리즘이 실행되는 데 걸리는 시간
  - 공간 효율성
    - 알고리즘이 사용하는 메모리 양
    - 예: 재귀호출에서 사용하는 시스템 스택
- 알고리즘의 구현
  - 특정 프로그래밍 언어
  - 컴파일(compile) 기반 : 실행 속도가 빠름(C++)
  - 인터프리터(interpreter) 기반: 개발 속도가 빠름(python)

9.2 억지 기법(Brute-force)

- 문제의 정의를 바탕으로 한 가장 직접적인 해결 방법
  - 단순한 또는 순진한(naive) 전략
  - 예: 순차 탐색, 선택 정렬 등
- 억지 기법의 의의
  - 해결하지 못하는 것보다는 단순하게라도 해결하는 것이 좋음
  - 매우 광범위한 문제에 적용할 수 있는 알고리즘 설계 기법
  - 입력의 크기가 작은 경우 충분히 빠를 수 있고, 심지어 점근적으로 더 효율적인 알고리즘보다 실제로는 더 빠를 수도 있음

배낭 채우기 문제(Knapsack Problem)

- 0-1 배낭 채우기 문제(0-1 knapsack problem)

> 무게가 각각 wgt_i이고 가치가 val_i인 n개의 물건이 있습니다. 이것을 넣을 배낭의 용량(최대 무게)은 W인데, 이를 초과해서 넣을 수는 없습니다. 물건들의 가치의 합이 최대가 되도록 배낭을 채웠을 때, 배낭의 최대가치를 구해보세요. 단, 하나의 물건을 잘라서 일부만 넣을 수는 없습니다.

  – 예) W=50, (무게, 가치)가 (10, 60), (20, 100), (30,120)인 물건들

무게 합이 용량을 넘지 않으면서 가치 합은 최대인 최적해

가치 합은 최대이지만 무게 합이 용량을 넘어 불가능한 해답

넣는 물건	A	B	C	A, B	B, C	A, C	A, B, C
무게 합	10	20	30	30	50	40	60
가치 합	60	100	120	160	220	180	280

억지기법 알고리즘 설계

- 완전 탐색
  - n개의 물건의 집합에 대한 모든 부분 집합을 만들고, 무게 합이 배낭 용량을 넘지 않으면서 가치가 최대인 것을 찾으면 됨
  - 가능한 부분 집합의 수 : 2^n

1. **시간 복잡도**: $O(2^n \times n)$
   - 모든 조합(2^n)을 탐색.
   - 각 조합에서 무게와 가치를 계산하기 위해 n번 반복.
2. **공간 복잡도**: $O(n)$
   - 부분 집합 표시를 위한 리스트(`subset`)가 필요.
3. **단점**:
   - 물건의 개수가 많아질수록(즉, n이 증가할수록) 계산량이 기하급수적으로 증가.
4. **적합한 경우**:
   - 물건의 개수(n)가 매우 작을 때 적합한 풀이 방식.

```python
def Knapsack01_BF(wgt, val, W):
    n = len(wgt)          # 전체 물건의 수
    bestVal = 0           # 배낭의 최대 가치

    for i in range(2**n) :
        s = [0]*n
        for d in range(n) :
            s[d] = i%2
            i = i//2

        sumval = 0
        sumWgt = 0
        for d in range(n):
            if s[d] == 1 :
                sumWgt += wgt[d]
                sumVal += val[d]

        if sumWgt <= W :
            if sumVal > bestVal :
                bestVal = sumVal

    return bestVal  # 최대 가치 반환
```

부분집합의 수는 2^n이므로, i에 0부터 2^n-1까지를 순서대로 대입함.

i를 이진수로 변환했을 때, 각 자리의 수를 리스트에 저장(역순으로). 예를 들어, 6은 이진수로 110인데, [0, 0, 1]로 저장함. 리스트에서 1이 포함되는 물건을 가리키도록 함.

현재 경우(i)에 대한 물건의 총 무게와 총 가치를 구함.

가능한 부분 집합이고, 가치합이 최대 가치보다 크면 최대 가치 갱신

9.3 탐욕적 기법

- 모든 경우를 고려해 보고 가장 좋은 답을 찾는 것이 아니라 "그 순간에 최적"이라고 생각되는 답을 선택
- 억지 기법과 달리 멀리 내다보지 않고 앞에 있는 가까운 것들만 보고 결정하기 때문에 근시안적인 알고리즘
- 결정 순간에 가능한 해 중에 지역적으로 최적인 것을 선택하고, 이러한 선택은 이후의 단계에서 다시 변경될 수 없다.
- 순간에 최적이라고 판단했던 선택들을 모아 만든 최적해가 항상 그렇다는 보장은 없음
- 최적의 해답을 주는지 반드시 검증이 필요

거스름돈 동전 최소화

- 거스름돈 동전 최소화 문제

> 액면가가 서로 다른 m 가지의 동전 $\{C_1, C_2, \cdots, C_m\}$ 이 있습니다. 거스름돈으로 V원을 동전으로만 돌려주어야 한다면 최소 몇 개의 동전이 필요한지를 구하세요. 단, 모든 동전은 무한히 사용할 수 있고, 액수가 큰 것부터 내림차순으로 순서대로 정렬되어 있습니다.

- 예: 우리나라 동전

  - 액면가가 가장 높은 동전부터 탐욕적으로 최대한 사용하면서 거스름돈을 맞추면 동전 개수를 최소화
    - 거스름돈 620원: 500원 + 100원 + 10원 × 2 → 동전 4개
    - 거스름돈 345원: 100원 × 3 + 10원 × 4 + 5원 → 동전 8개
    - 거스름돈 572원: 500원 + 50원 + 10원 × 2 + 1원 × 2 → 동전 6개

최적해를 구할까?

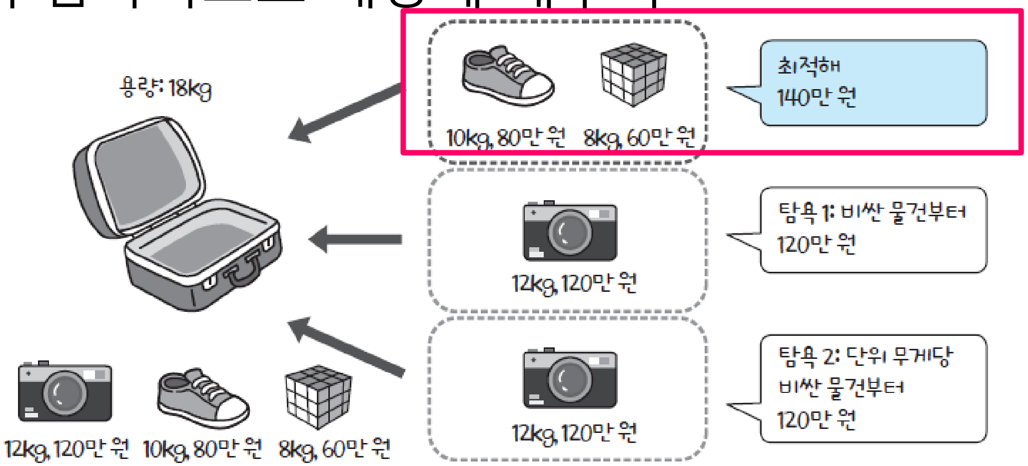
- 탐욕적 기법이 항상 최적해를 만들지는 못함
  - 예) 기념 주화로 60원 추가

- 다음과 같은 동전 체계를 갖는다면 최적해 보장

동전의 액면가 중에서 어떤 두 개를 고르더라도 큰 액면가를 작은 액면가로 나누어 떨어지는 동전 체계를 갖는다면 최적해를 보장합니다. 작은 액면가를 여러 개 모으면 반드시 큰 액면가를 만들 수 있기 때문입니다.

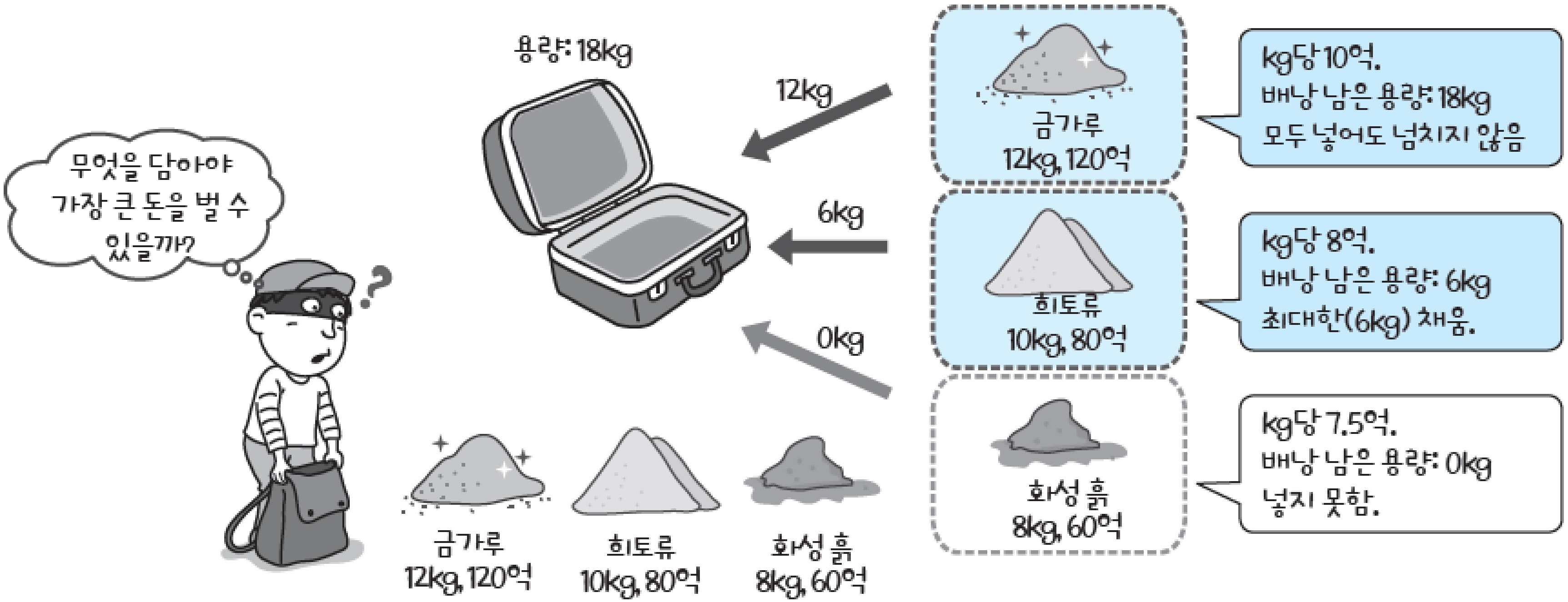
분할 가능한 배낭 채우기(Fractional Knapsack)

- **0-1 배낭 문제 :** 탐욕적 기법으로 최적해를 구하지 못함
  - 탐욕 1: 무게와 상관없이 가장 비싼 물건부터 넣어보는 방법
  - 탐욕 2: 단위 무게당 가격이 가장 높은 물건부터 넣어보는 방법
- **분할 가능한 배낭 채우기 문제 :**
  - 만약 물건들을 나누어 **일부분만을 배낭에 넣을 수 있다면 가능**
  - 항상 배낭을 최대 용량으로 채울 수 있음
    - **단위 무게당 가치를 기준으로 내림차순으로 정렬**
    - 넣을 수 있는 모든 공간을 항상 단위 무게당 가격이 가장 높은 것부터 최대한 많이 탐욕적으로 배낭에 채우기
  - 시간복잡도: $O(n)$



분할 가능한 배낭 채우기 문제

각각 무게가 wgt_i이고 가치가 val_i인 n개의 물건들이 있고, 이것을 배낭에 넣으려고 합니다. 배낭에는 용량(최대 무게) W까지만 넣을 수 있습니다. 물건들의 가치의 합이 최대가 되도록 배낭을 채우고, 이때 배낭의 최대가치를 구해 보세요. 단, 물건들은 나누어 일부분만을 넣을 수도 있습니다.



분할 가능한 배낭 채우기(탐욕적 기법)

```python
def KnapSackFrac(wgt, val, W):
    bestVal = 0                # 최대가치
    for i in range(len(wgt)):     # 단가가 높은 물건부터 처리
        if W <= 0 :            # 용량이 다 찼으면 채우기 종료
            break
        if W >= wgt[i]:
            W -= wgt[i]
            bestVal += val[i]
        else:
            fraction = W / wgt[i]
            bestVal += val[i] * fraction
            break

    return bestVal  # 최대 가치 반환

# 테스트 프로그램
weight = [12,  10, 8]      # (정렬됨)
value  = [120, 80, 60]     # (정렬됨)
W = 18                 # 배낭의 제한 용량
print("Fractional Knapsack(18):", KnapSackFrac(weight, value, W))
```

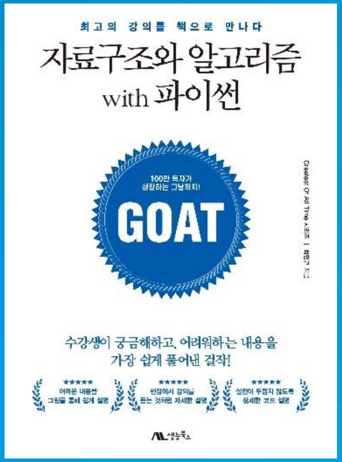
물건들은 단위 무게당 가격의 내림차순으로 정렬되어 있어야 함.

물건 전체를 넣을 수 있으면, 넣고(최대 가치 증가시킴), 남은 용량 W를 갱신

일부만 넣을 수 있으면, 최대 비율을 계산하고, 최대 한 채움(최대가치 증가) 채우기 종료

실행 결과

```
Fractional Knapsack(18): 168.0
```



CHAPTER 10

분할 정복

10장. 분할 정복

10-1 분할 정복이란?

~~10-2 거듭제곱 구하기~~

~~10-3 선택 문제: k-번째로 작은 수 찾기~~

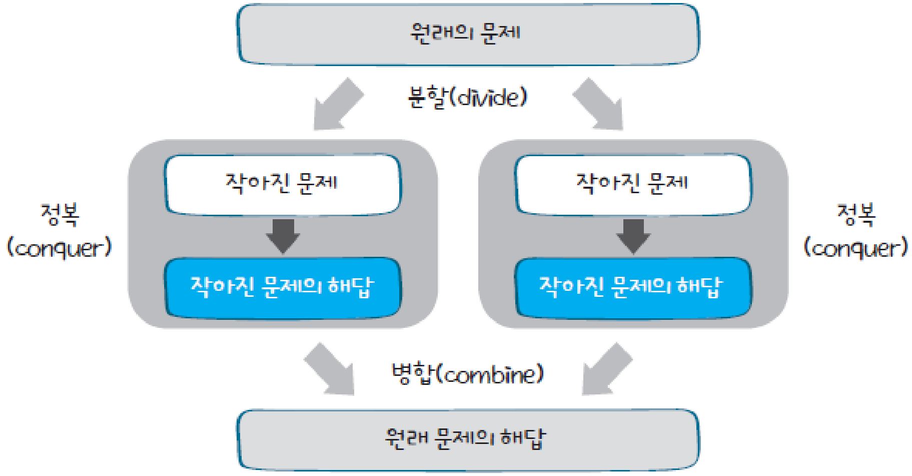
10-4 병합 정렬

10-5 피보나치 수열과 분할 정복의 주의점

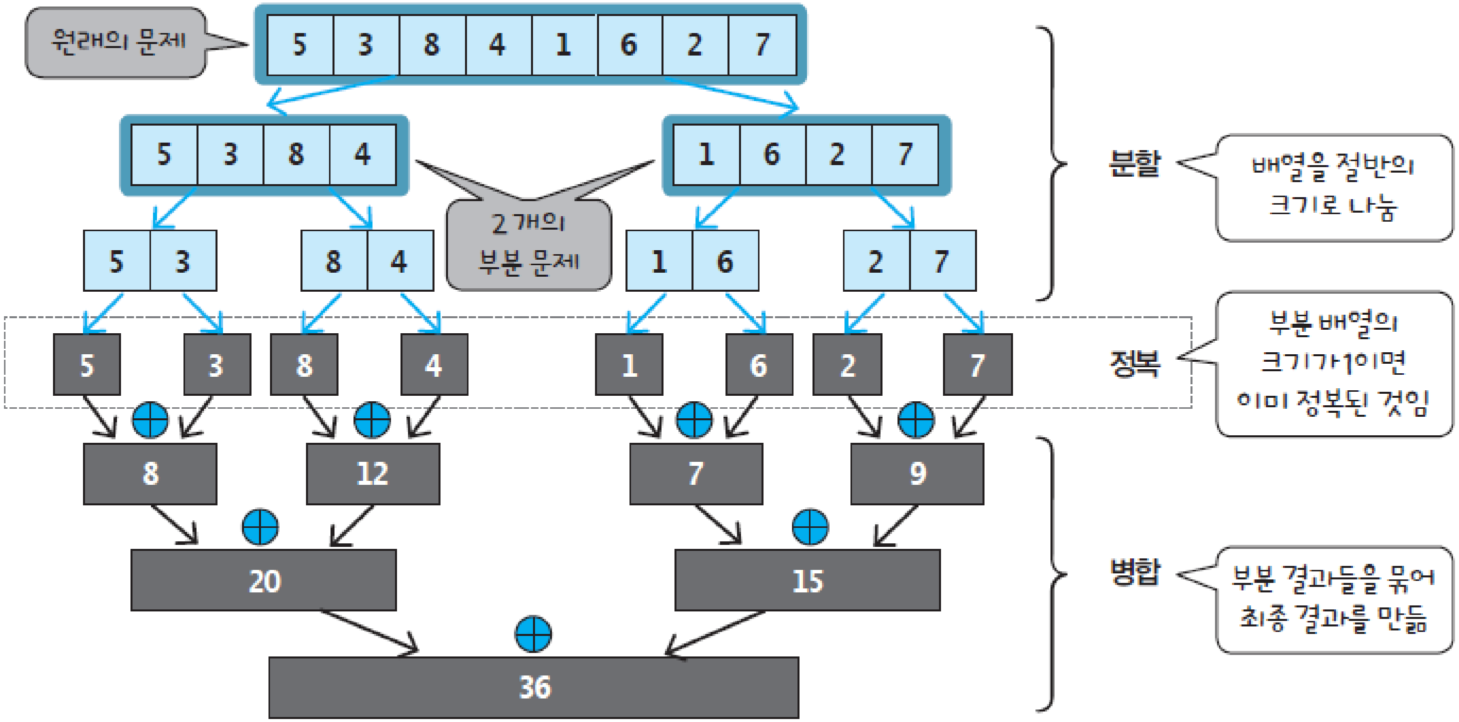
"적군이 강하면 먼저 적을 작게 나눈 다음, 작아진 상대들을 개별적으로 정복 "

10.1 분할 정복(Divide and Conquer)이란?

- 분할 정복
  - 주어진 문제를 여러 개의 작은 부분 문제들로 나누고, 각 부분 문제를 독립적으로 해결한 뒤 결과를 모아서 원래의 문제를 해결하는 전략
  - 예: 퀵 정렬, 이진 탐색 문제에서는 효율적
- 분할 정복의 구조
  - 분할(Divide): 문제를 더 작은 하위 문제로 나눈다.
  - 정복(Conquer): 나눠진 하위 문제를 재귀적으로 해결
  - 병합(combine): 하위 문제의 결과를 결합하여 최종 해를 만듦
- 분할정복은 같은 문제가 여러 번 반복되어 나타나지 않을 때 사용



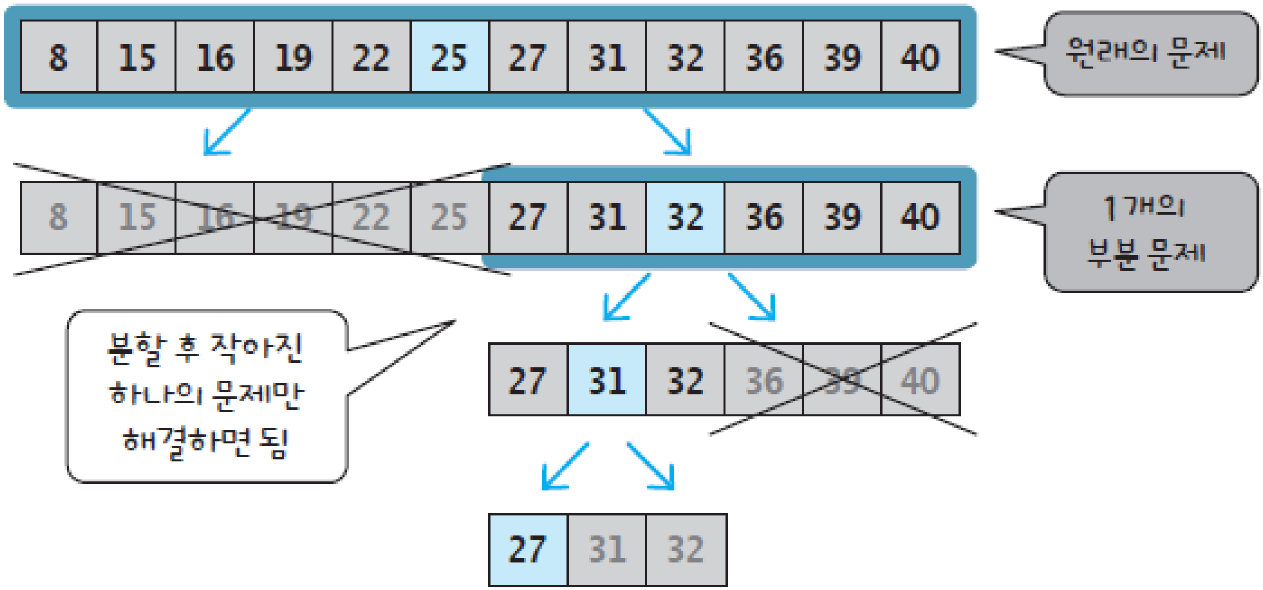
예: 배열의 합 구하기



- 단순 방법: 반복문을 이용하여 첫 번째 숫자부터 하나씩 모든 숫자를 누적하여 더하기 – $O(n)$
- 분할 정복 방법: $O(\log n) + O(n) = O(n)$
- 분할 정복이 모든 문제에서 더 효율적인 것은 아님
  - 그렇지만 정렬이나 탐색과 같은 문제에서 상당한 효과를 발휘

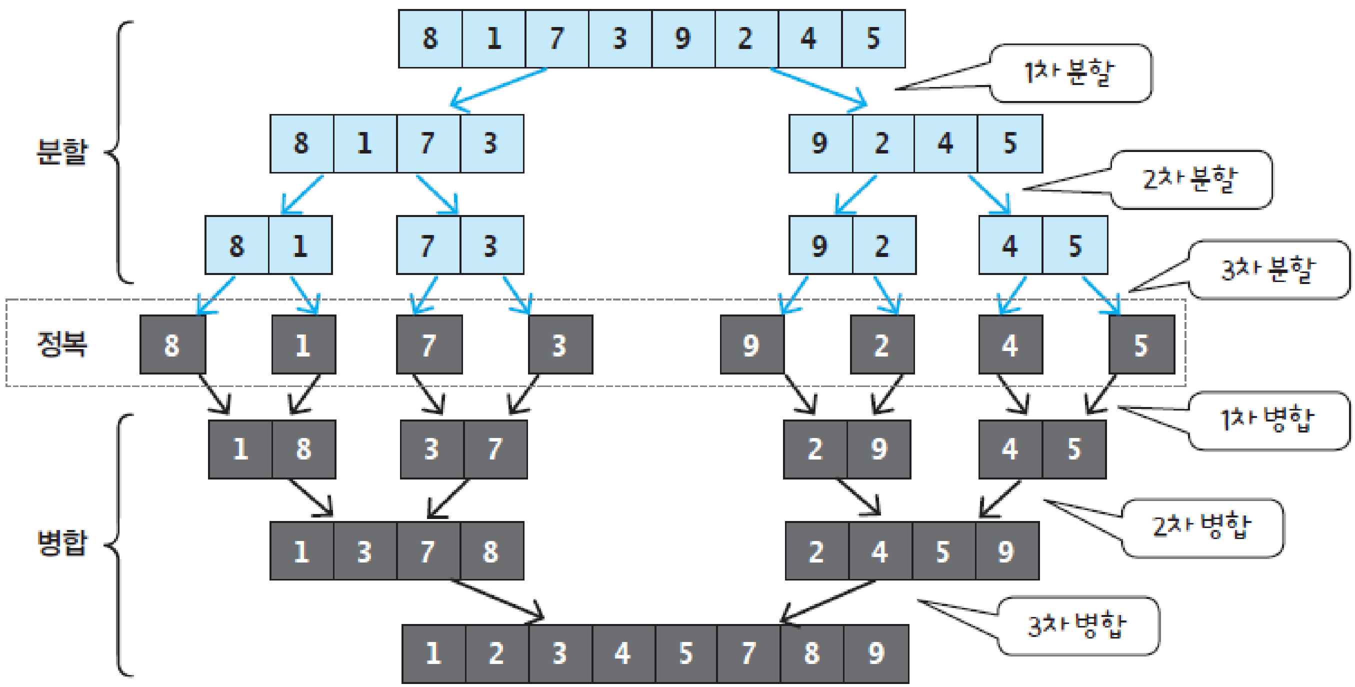
축소 정복(Decrease-and-conquer)

- 축소 정복(decrease-and-conquer)
  - 원래의 문제를 나눈 후에 해결해야 할 부분 문제가 하나만 남는 분할 정복문제의 특별한 경우
- 이진 탐색 : 분할된 두 부분 배열에서 한쪽은 더는 고려하지 않지않고, 결국 원래의 문제가 절반 이하로 작아진 하나의 부분 문제로 축소



10.4 병합 정렬(Merge Sort)

- 병합 정렬
  - 분할 단계 : 입력 리스트를 균등하게 두 부분으로 분할
    - 이 과정은 부분 리스트의 크기가 1이 될 때까지 반복
  - 정복 단계 : 크기가 1이면 그 부분 리스트는 이미 정렬
  - 병합 단계 : 2개의 정렬된 리스트를 병합해 하나의 정렬된 리스트를 만드는 과정

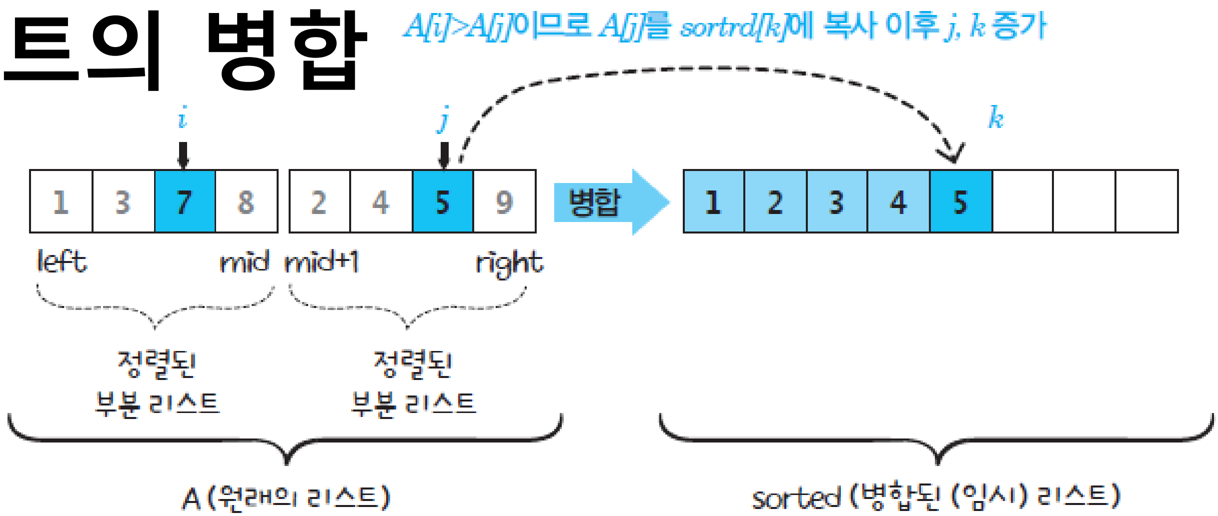


병합 정렬 알고리즘

```
def merge_sort(A, left, right) : # A[left..right]를 오름차순으로 정렬
    if left<right :              # 항목이 2개 이상인 경우
        mid = (left + right) // 2
        merge_sort(A, left, mid)
        merge_sort(A, mid + 1, right)
        merge(A, left, mid, right)
    # else: 항목이 1개인 경우. 자동으로 정복되었음(하나이므로)
```

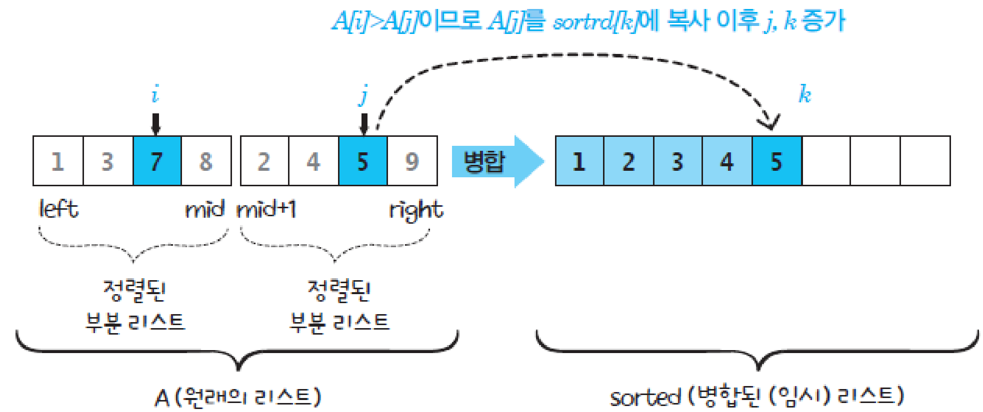
리스트를 균등하게 둘로 나누고,
왼쪽 부분(A[left~mid])과 오른쪽
부분(A[mid+1~right])을 각각 병합정렬.
마지막으로 정렬된 두 부분 리스트를 병합함.

정렬된 두 리스트의 병합


A[i]>A[j]이므로 A[j]를 sortrd[k]에 복사 이후 j, k 증가
i
j
k
1 3 7 8
2 4 5 9
병합
1 2 3 4 5
left
mid
mid+1
right
정렬된 부분 리스트
정렬된 부분 리스트
A (원래의 리스트)
sorted (병합된 (임시) 리스트)


왼쪽 리스트
오른쪽 리스트
병합 결과를 위한 임시 리스트

병합 알고리즘




```python
def merge(A, left, mid, right) :
    k = left                # 병합을 위한 임시 리스트의 인덱스
    i = left                # 왼쪽 리스트의 인덱스
    j = mid + 1             # 오른쪽 리스트의 인덱스
    while i<=mid and j<=right :
        if A[i] <= A[j] :
            sorted[k] = A[i]
            i, k = i+1, k+1
        else:
            sorted[k] = A[j]
            j, k = j+1, k+1

    if i > mid :
        sorted[k:k+right-j+1] = A[j:right+1]
    else :
        sorted[k:k+mid-i+1] = A[i:mid+1]

    A[left:right+1] = sorted[left:right+1]
```

값이 작은 부분 리스트의 요소를 sorted에 복사하고, 그 리스트의 인덱스를 증가시킴. 이 과정은 어느 한쪽 부분이 모두 처리될 때까지 진행.

남은 부분 리스트의 모든 요소를 sorted로 복사. 슬라이싱을 이용함.

임시 리스트에 저장된 결과를 원래의 리스트 A에 복사

병합 정렬의 특징

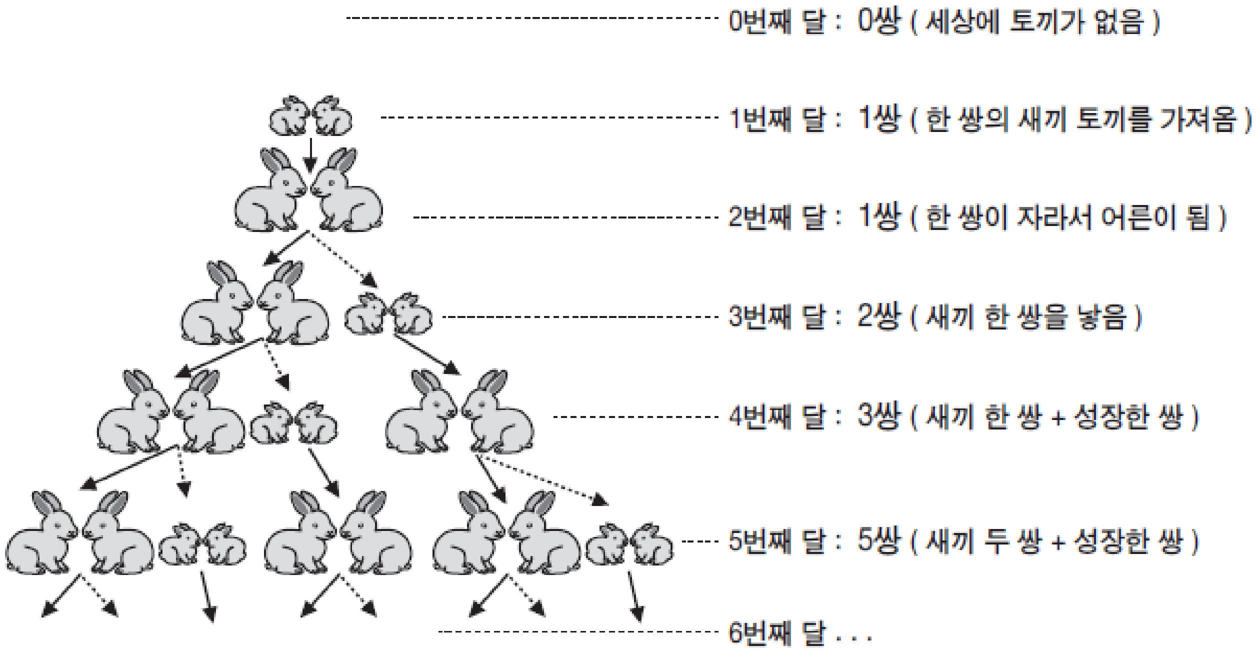
- 시간 복잡도: $O(n\, log_2 n)$
  - 분할 작업: $O(log_2 n)$
  - 병합 작업 : $O(n)$
  - 전체 시간 복잡도 : $O(n\, log_2 n)$

병합 정렬의 이동 횟수 $= k \times 2n = \log_2 n \times 2n = 2n log_2 n \in O(n log_2 n)$

- 공간 복잡도 :
  - 추가 공간 사용 (임시 배열): $O(n)$
- 특징
  - 효율적인 정렬 방법
  - 입력의 구성과 상관없이 동일한 시간에 정렬
  - 안정성을 만족
  - 제자리 정렬이 아님

10.5 피보나치 수열과 분할 정복의 주의점

첫 번째 달에 한 쌍의 새끼 토끼를 세상에 가져왔습니다. 토끼는 한 달이 지나면 어른으로 성장하고, 어른 토끼는 매달 새끼 토끼를 한 쌍씩 낳습니다. 그리고 한번 태어난 토끼는 절대 죽지 않습니다. n번째 달에는 몇 쌍의 토끼가 있을까요?



– 피보나치 수열: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

순환 관계식

- 1달 전의 토끼 $fib(n\text{-}1)$은 모두 그대로 살아 있습니다. 모두 어른 토끼입니다.
- 2달 전에 있었던 토끼들은 1달 전에는 모두 어른이므로, 이번 달에 무조건 새끼를 낳습니다. 따라서 새로 태어나는 토끼 수는 $fib(n\text{-}2)$입니다.

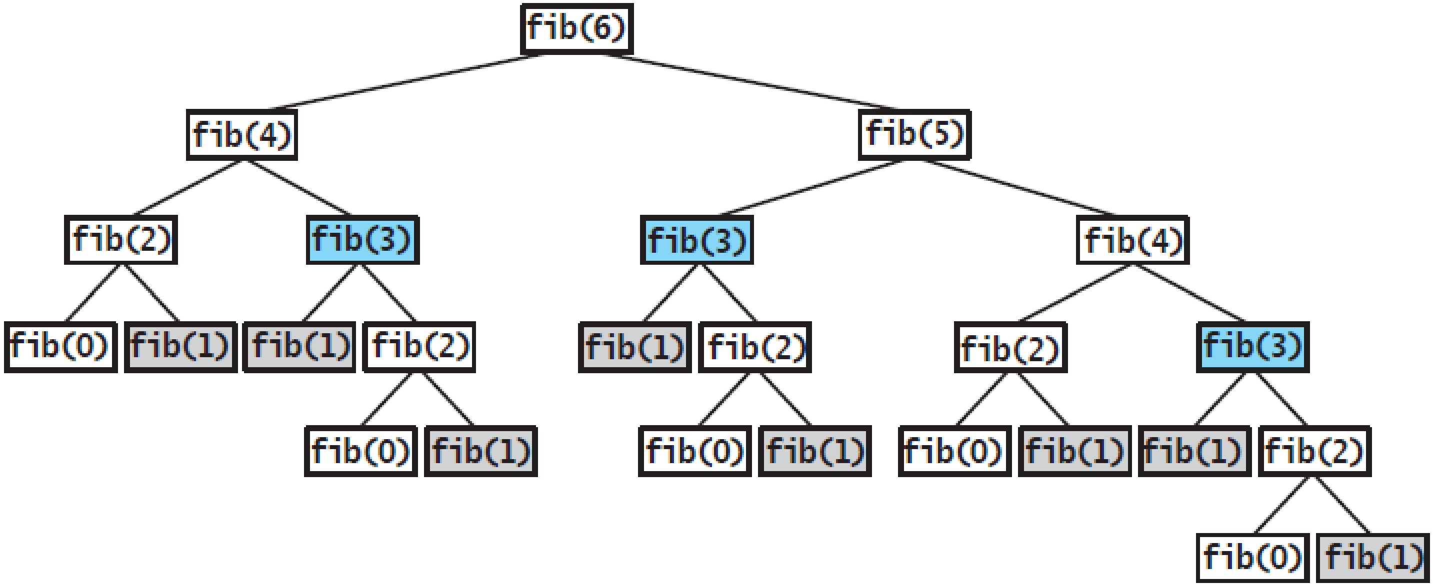
$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-2)+fib(n-1) & otherwise \end{cases}$$

- 분할 정복을 이용한 피보나치 수열 알고리즘

```
def fib(n) :
    if n == 0 : return 0          # 정복: 0번째 달
    elif n == 1 : return 1        # 정복: 1번째 달
    else :
        return fib(n - 1) + fib(n - 2)
```

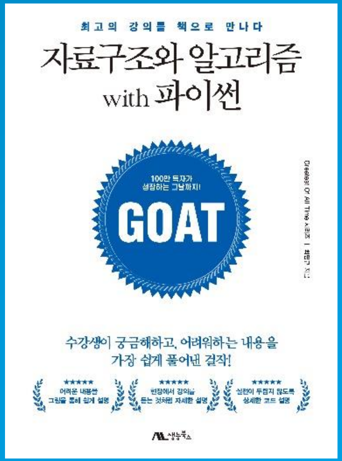
중복 계산 문제

- 예: fib(6)



  – 한번 문제를 나눌 때마다 해결해야 할 전체 부분 문제의 크기가 거의 두 배로 늘어남

$$\boxed{O(2^n)}$$



CHAPTER 11

동적 계획법

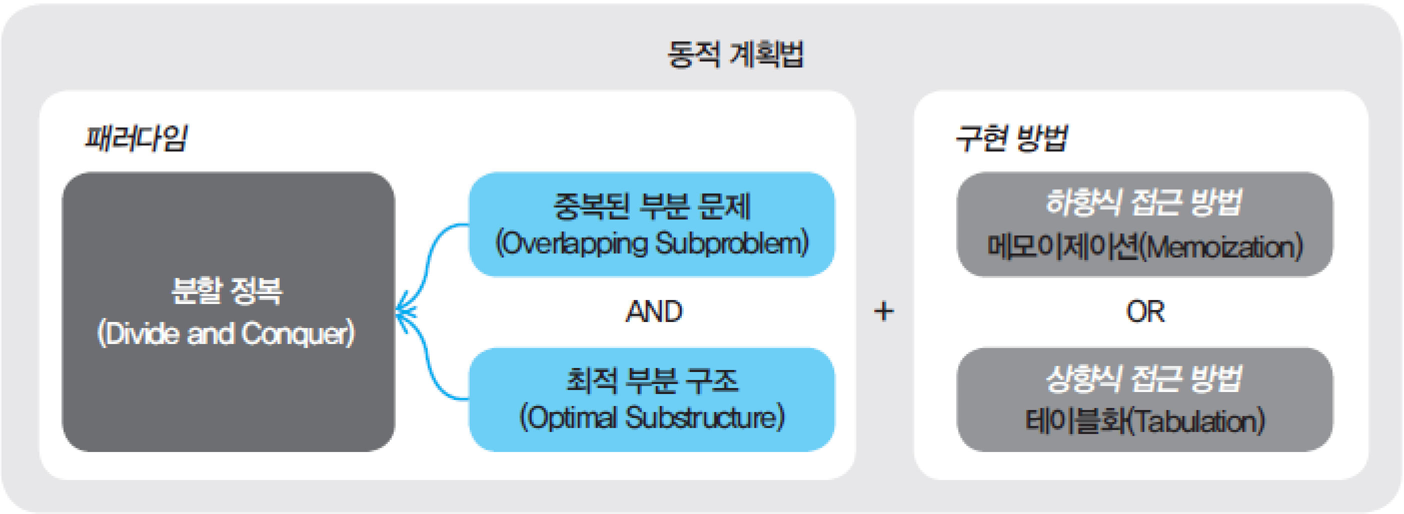
11장. 동적 계획법

11-1 동적 계획법이란?

배낭 채우기, 피보나치 수열

동적 계획법을 이용한 문제 해결 전략

- 동적 계획법의 패러다임과 구현 방법



  - 최적 부분 구조 특성(중복된 문제를 반복해서 해결)을 만족하지 않는 예
    - 이진 탐색
    - 병합 정렬

11.1 동적 계획법이란?

- 동적 프로그래밍(dynamic programming; DP)
  - 문제를 작은 부분 문제로 나누어 해결하고, 그 결과를 저장하여 동일한 계산을 반복하지 않도록 하는 방법
  - 분할 정복과 유사
  - 부분 문제들의 답을 어딘가에 저장해 놓고 필요할 때 다시 꺼내서 사용하는 전략
  - 같은 부분 문제를 다시 풀지 않도록 함
  - 주로 중복되는 부분 문제를 가진 문제에 사용

동적 계획법의 조건

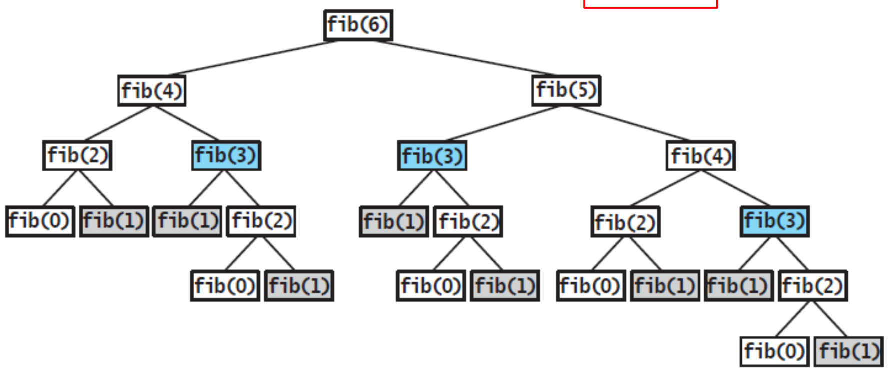
1. **Optimal Substructure (최적 부분 구조)**:
   - 문제를 작은 부분 문제로 나누고, 이들을 조합해 전체 최적 해를 구할 수 있어야 함.
2. **Overlapping Subproblems (중복되는 부분 문제)**:
   - 동일한 하위 문제가 여러 번 등장하는 경우, 결과를 저장하여 효율적으로 처리.

피보나치 수열 문제

- 순환 관계식

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & otherwise \end{cases}$$

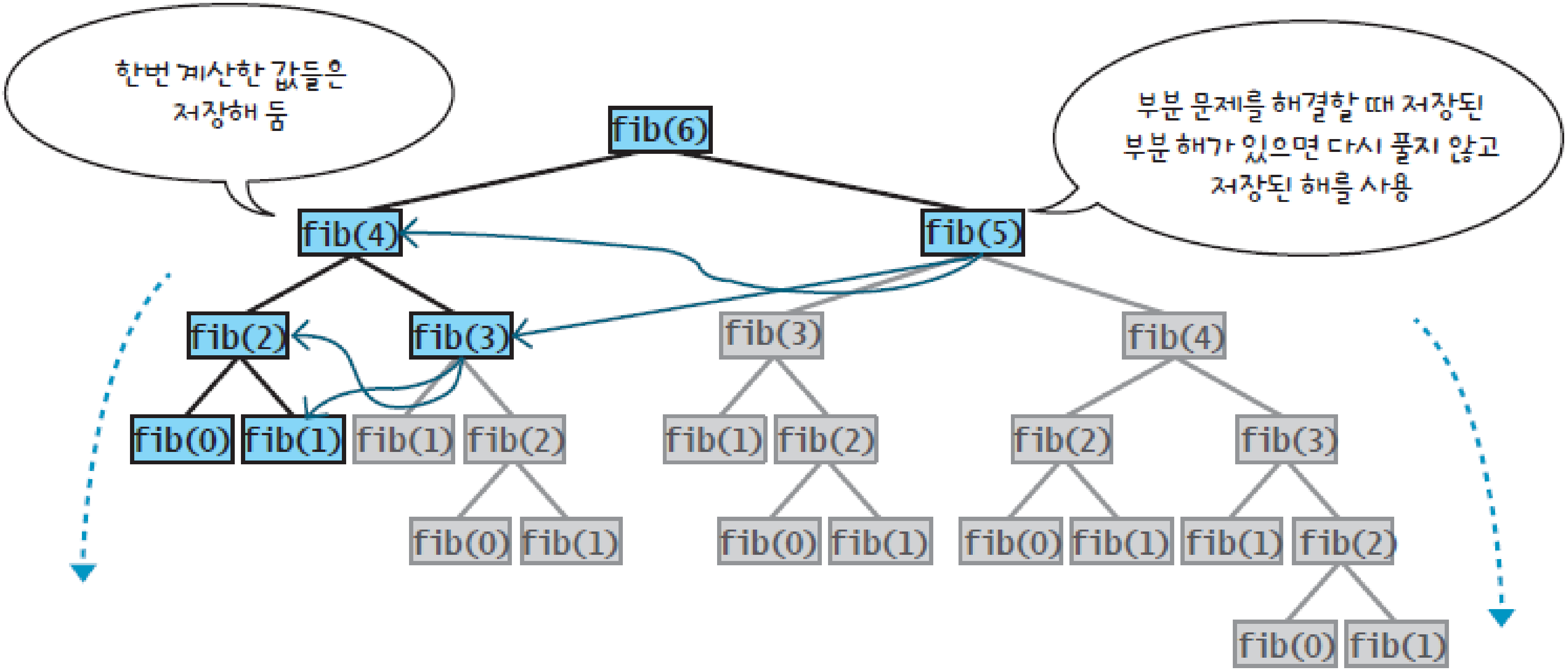
  – 분할 정복 코드 → 중복 계산 문제 $\boxed{O(2^n)}$



- 동적 계획법
  – 부분 문제의 해를 저장하여 재사용 방법
  – 메모이제이션(memoization), 테이블화(tabulation)

(1) 메모이제이션을 이용한 피보나치 수열

- **메모이제이션(memoization)**
  – 함수의 결과를 저장할 메모리를 미리 준비해서 한번 계산한 값을 저장해 두었다가 재활용하는 방법
  – 하향식(top-down) 접근



피보나치 수열(메모이제이션 이용)

```
def fib_dp_mem(n) :
    if( mem[n] == None ) :
        if n < 2 :
            mem[n] = n
        else:
            mem[n] = fib_dp_mem(n-1) + fib_dp_mem(n-2)
    return mem[n]
```

처음 푸는 문제이면, 풀어서 메모리에 저장함.

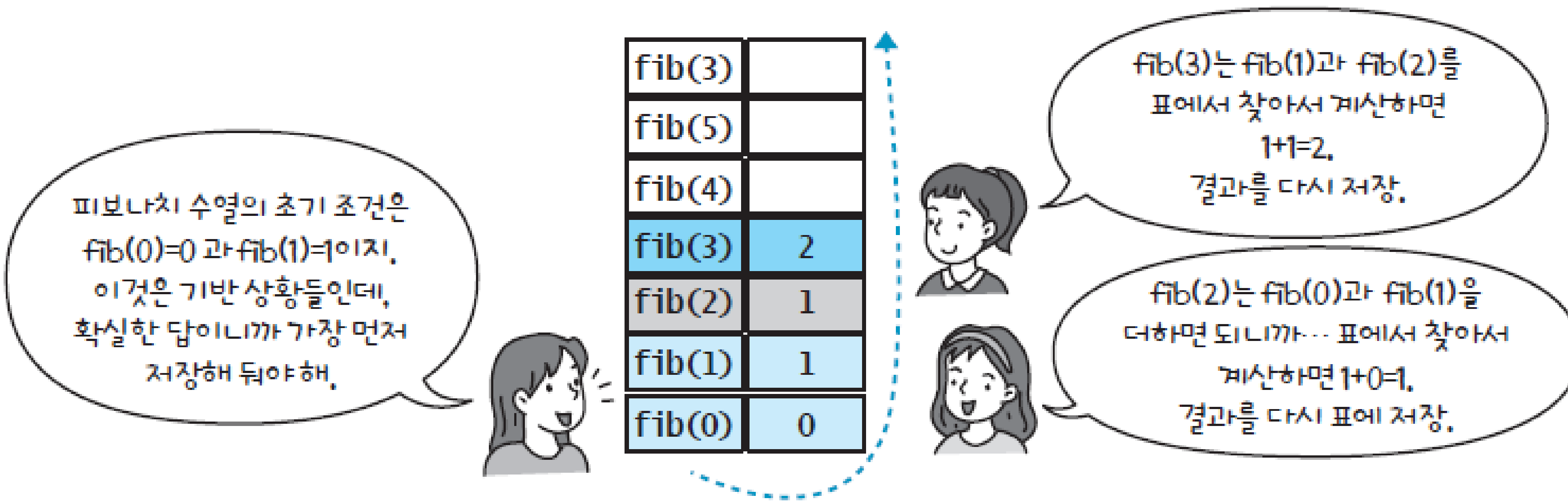
저장된 답을 반환

리스트로 사용하면 리스트의 크기를 넘어가게 되면
오류가 발생하기에 리스트 대신 딕셔너리를 사용한다.

- 결과 저장을 위한 메모리:
  - mem (크기가 n+1인 리스트)
  - mem[k]에는 k번째 피보나치 수가 저장
- 성능
  - 시간 복잡도: $O(n)$
  - 공간 복잡도: $O(n)$

(2) 테이블화를 이용한 피보나치 수열

- 테이블화(tabulation, 타뷸레이션)
  - 부분 문제의 해를 메모리에 저장
  - 테이블 항목들을 순서대로 채워나가는 것에 초점
  - 상향식(bottom-up)으로 문제를 해결
    - 가장 작은 부분 문제부터 순서대로 해를 구해 테이블을 채워서 올라감



피보나치 수열(테이블화 이용)

```python
def fib_dp_tab(n) :
    f = [None] * (n+1)
    f[0] = 0
    f[1] = 1
    for i in range(2, n + 1):
        f[i] = f[i-1] + f[i-2]
    return f[n]        # 결과 반환
```

답을 저장할 테이블을 만들고, 알려진 답을 저장합니다.
나머지는 모두 None으로 초기화합니다.

순서대로(상향식) 문제를 풀고,
결과를 테이블에 저장합니다.

- 시간 복잡도 $O(n)$, 공간 복잡도 $O(n)$

실행 결과

```
Fibonacci(8) 분할 정복     =  21
Fibonacci(8) 테이블화      =  21
Fibonacci(8) 메모이제이션  =  21
Fibonacci(8) 반복 구조     =  21
```

1. 테이블 초기화:

$$f = [None] \times 9 \quad (\text{크기 } n+1)$$

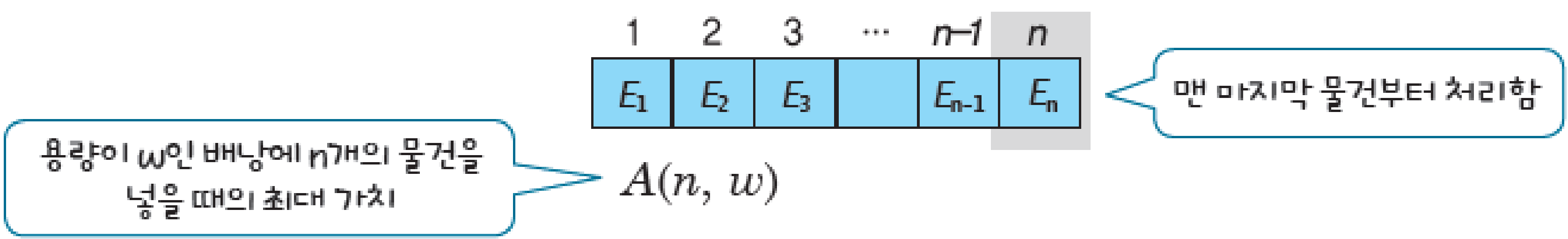
초기값 설정:

$$f[0] = 0, \quad f[1] = 1$$

단계	$f[0]$	$f[1]$	$f[2]$	$f[3]$	$f[4]$	$f[5]$	$f[6]$	$f[7]$	$f[8]$
초기화	0	1	None	None	None	None	None	None	None
$i = 2$	0	1	$f[1] + f[0] = 1$	None	None	None	None	None	None
$i = 3$	0	1	1	$f[2] + f[1] = 2$	None	None	None	None	None
$i = 4$	0	1	1	2	$f[3] + f[2] = 3$	None	None	None	None
$i = 5$	0	1	1	2	3	$f[4] + f[3] = 5$	None	None	None
$i = 6$	0	1	1	2	3	5	$f[5] + f[4] = 8$	None	None
$i = 7$	0	1	1	2	3	5	8	$f[6] + f[5] = 13$	None
$i = 8$	0	1	1	2	3	5	8	13	$f[7] + f[6] = 21$

11.3 배낭 채우기

- 01-배낭 채우기
  - 억지 기법(완전 탐색)
    - 원소의 개수가 n인 집합의 부분집합의 수가 2^n → $O(2^n)$
  - 탐욕적 기법: 물건을 잘라 넣을 수 없으니 사용할 수 없음
  - 동적 계획 전략 적용
- **배낭 문제의 순환 관계식 만들기**
  - 배낭 용량 : W
  - n개의 물건: E_1, E_2, …, E_n
  - 각 물건의 무게와 가치 : $E_i = (wt_i, val_i)$
  - 배낭의 최대 가치 : $A(n, W)$
    - n개 모든 물건($\mathrm{E_1} \sim \mathrm{E_n}$)를 용량이 W인 배낭에 넣는 경우의 배낭 최대 가치



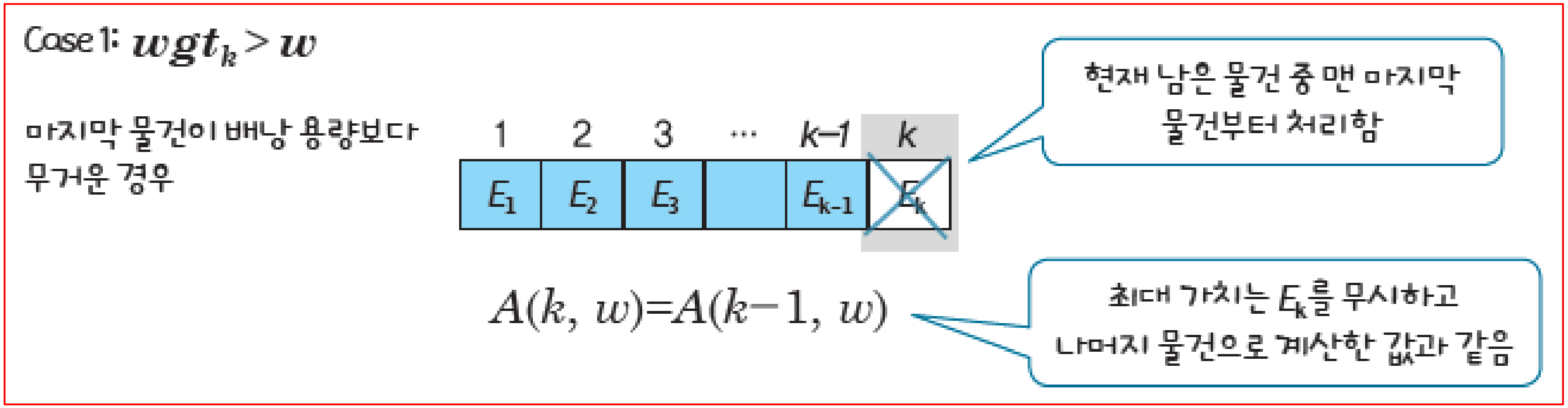
배낭 문제의 순환 관계식

- $A(k, w)$의 기반 상황
  - $A(0, w) = 0$, $A(k, 0) = 0$

남은(넣을) 물건이 없음

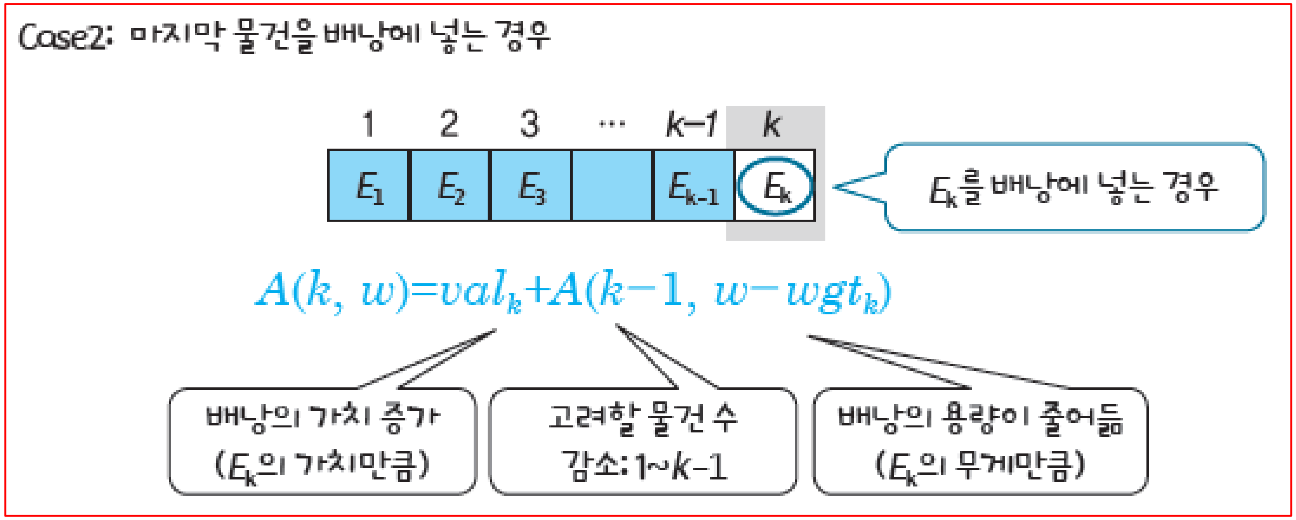
남은 용량이 0

- $A(k, w)$의 일반 상황
  - k개의 물건 $E_i \sim E_k$ 을 용량이 w인 배낭에 넣을 때의 최대 가치
  - **Case1**: $wgt_k > w$ → 어차피 넣을 수 없음
    - $A(k, w) = A(k-1, w)$

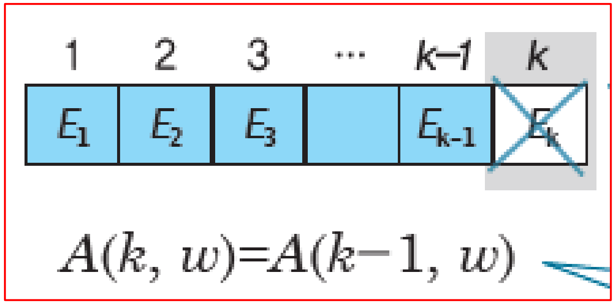


배낭 문제의 순환 관계식

- $A(k, w)$의 일반 상황
  - Case2: $\boldsymbol{wgt_k \le w}$ → 넣거나, 안 넣거나
    - 넣은 경우: $A(k-1, w-wgt_k) + val_k$



    - 넣지 않은 경우: $A(k-1, w)$



배낭 문제의 순환 관계식

$$A(n, W) = \begin{cases} 0 & \text{if} \quad W = 0 \; or \; n = 0 \\ A(n-1, W) & \text{if} \quad wgt_n > W \\ \max\left(val_{n-1} + A(n-1, W - wgt_{n-1}), A(n-1, W)\right) & otherwise \end{cases}$$

동적 계획법에 의한 0-1 배낭 문제 해법

- **적용 조건**
  - 물건의 무게와 배낭의 용량을 모두 정수로 제한
- **테이블 설계 :**
  - 답을 저장해 둘 테이블
  - $A(n, W)$
  - $(n+1)\ X\ (W+1)$의 2차원 배열

길이

		배낭 용량							
		0	1	…	$w-wt_i$	…	w	…	W
	0	0	0	…	0	…	0	…	0
	1	0							
	…	…							
물건	$i-1$	0			$A(i-1, w-wt_i)$	…	$A(i-1, w)$		
	i	0					$A(i, w)$		
	…	…							
	n	0							$A(n, W)$

0-1 배낭 채우기(동적 계획법)

```
def knapSack_dp(W, wt, val, n):
    A = [[0 for x in range(W + 1)] for x in range(n + 1)]
```

(n+1)x(W+1) 크기의 2차원 배열을 생성하고 모든 요소를 0으로 초기화

```
    for i in range(1, n + 1):                   # 위에서 아래로 진행
        for w in range(1, W + 1):               # 좌에서 우로 진행
            if wt[i-1] > w:                     # i번째 물건이 용량 초과
                A[i][w] = A[i-1][w]
            else :                              # i번째 물건을 넣을 수 있음
                valWith = val[i-1] + A[i-1][w-wt[i-1]]  # 넣는 경우
                valWithout = A[i-1][w]                  # 빼는 경우
                A[i][w] = max(valWith, valWithout)      # 더 큰 값 선택

    return A[n][W]
```

i번째 물건을 넣는 경우와 빼는 경우를 구해 더 큰 값을 선택.
계산을 위한 값들은 표에 이미 모두 계산되어 있음

- 복잡도 분석
  - 시간 복잡도 / 공간 복잡도 : $O(nW)$
  - 배낭의 용량이 매우 크거나 물건의 종류가 많다면 추가 공간과 처리시간이 모두 크게 늘어남

문제: DP 프로그래밍 : knapsack problem

- 예제 ([weights: 2, 3, 4], [values: 3, 4, 5], W=5)

1. 초기 테이블 생성

2차원 DP 테이블 $A[i][w]$을 만들고, i는 물건의 개수 (0부터 시작)이며 w는 배낭의 용량 (0부터 시작)입니다.

초기화:

- $A[i][0] = 0$ (배낭 용량이 0인 경우)
- $A[0][w] = 0$ (물건이 없는 경우)

초기 테이블 (값은 모두 0으로 초기화):

$$A = \begin{array}{|c|c|c|c|c|c|c}
i \backslash w & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}$$

물건 1 ($weights[0] = 2, values[0] = 3$):

조건: 무게가 2이고 가치가 3.

- $w = 0, 1$: 배낭 용량이 물건의 무게보다 작으므로 포함할 수 없음. $A[1][w] = A[0][w]$.
- $w = 2$: 물건을 포함 가능.

$$A[1][2] = \max(A[0][2], A[0][2-2] + 3) = \max(0, 0+3) = 3$$

- $w = 3$: 물건을 포함 가능.

$$A[1][3] = \max(A[0][3], A[0][3-2] + 3) = \max(0, 0+3) = 3$$

- $w = 4$: 물건을 포함 가능.

$$A[1][4] = \max(A[0][4], A[0][4-2] + 3) = \max(0, 0+3) = 3$$

- $w = 5$: 물건을 포함 가능.

$$A[1][5] = \max(A[0][5], A[0][5-2] + 3) = \max(0, 0+3) = 3$$

$$A = \begin{array}{|c|c|c|c|c|c|c|} \hline i \backslash w & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 3 & 3 & 3 & 3 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

물건 2 ($weights[1] = 3, values[1] = 4$):

조건: 무게가 3이고 가치가 4.

- $w = 0, 1, 2$: 배낭 용량이 물건의 무게보다 작으므로 포함할 수 없음. $A[2][w] = A[1][w]$.
- $w = 3$: 물건을 포함 가능.

$$A[2][3] = \max(A[1][3], A[1][3-3] + 4) = \max(3, 0+4) = 4$$

- $w = 4$: 물건을 포함 가능.

$$A[2][4] = \max(A[1][4], A[1][4-3] + 4) = \max(3, 0+4) = 4$$

- $w = 5$: 물건을 포함 가능.

$$A[2][5] = \max(A[1][5], A[1][5-3] + 4) = \max(3, 3+4) = 7$$

$$A = \begin{array}{|c|c|c|c|c|c|c|} \hline i \backslash w & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 3 & 3 & 3 & 3 \\ 2 & 0 & 0 & 3 & 4 & 4 & 7 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

물건 3 ($weights[2] = 4, values[2] = 5$):

조건: 무게가 4이고 가치가 5.

- $w = 0, 1, 2, 3$: 배낭 용량이 물건의 무게보다 작으므로 포함할 수 없음. $A[3][w] = A[2][w]$.
- $w = 4$: 물건을 포함 가능.

$$A[3][4] = \max(A[2][4], A[2][4-4] + 5) = \max(4, 0+5) = 5$$

- $w = 5$: 물건을 포함 가능.

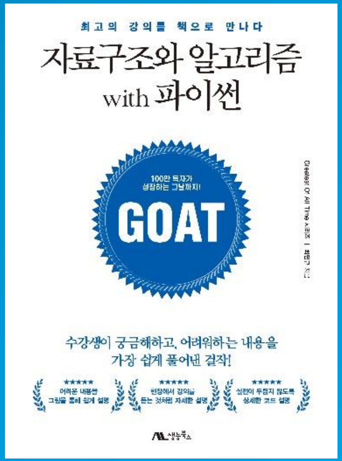
$$A[3][5] = \max(A[2][5], A[2][5-4] + 5) = \max(7, 0+5) = 7$$

최종 테이블:

$$A = \begin{array}{|c|c|c|c|c|c|c|} \hline i \backslash w & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 3 & 3 & 3 & 3 \\ 2 & 0 & 0 & 3 & 4 & 4 & 7 \\ 3 & 0 & 0 & 3 & 4 & 5 & 7 \\ \hline \end{array}$$

최종 선택된 물건

- **물건 2 ($weights[1] = 3, values[1] = 4$)**
- **물건 1 ($weights[0] = 2, values[0] = 3$)**
- 무게의 합: $3 + 2 = 5$
- 총 가치: $4 + 3 = 7$



12 CHAPTER

공간으로 시간벌기와 백트래킹

12장. 공간으로 시간벌기와 백트래킹

12-1 공간으로 시간을 살 수 있나요?

12-2 해싱

~~12-3 백트래킹~~

12.1 공간으로 시간을 살 수 있나요?

- sin(x) 계산

- 공간으로 시간을 버는 알고리즘 예
  - 기수 정렬
  - 동적 계획법
  - 해싱

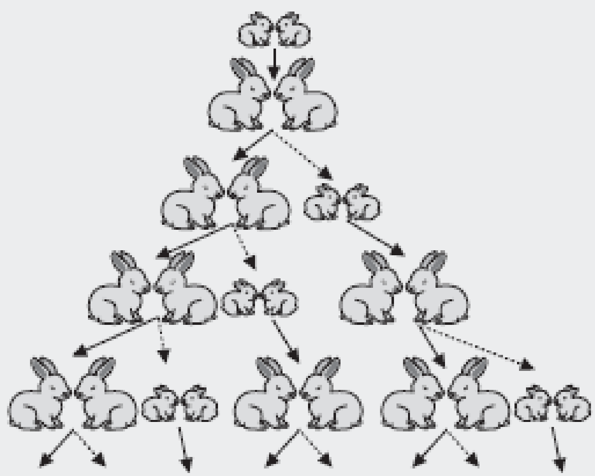
공간과 시간의 Trade-off

- 기본 개념
  - 문제를 해결할 때 공간(메모리)과 시간(속도) 간의 균형을 고려
  - 일반적으로 시간을 줄이기 위해 더 많은 메모리를 사용하거나, 메모리를 줄이기 위해 더 많은 계산을 수행

- 사례
  - 해싱 : 메모리를 사용하여 빠르게 데이터를 검색
  - 캐싱 : 자주 사용하는 데이터를 저장하여 계산 속도를 높임
  - DP의 공간 최적화: 필요하지 않은 메모리를 제거하여 공간 사용량을 줄임

피보나치 수열 문제

- n번째 피보나치 수를 구하는 다양한 알고리즘들

피보나치 수열 문제	알고리즘 종류		시간 복잡도	공간 복잡도
	1	분할 정복 기법: 코드 10.9	$O(2^n)$	$O(1)$
	[redacted]			
	3	동적 계획법: 코드 11.1, 11.2	$O(n)$	$O(n)$
	4	미리 답이 계산된 테이블 이용	$O(1)$	$O(n)$

12.2 해싱

- 순차탐색, 이진탐색과 해싱의 비교

아파트 전체 세대에 대한 하나의 우편함

아파트의 각 세대별 우편함

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
순차 탐색으로 32 탐색:	35	45	8	43	15	27	22	19	25	32	36	33	39	40	16	5
	❶	❷	❸	❹	❺	❻	❼	❽	❾	❿						

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
이진 탐색으로 32 탐색:	3	5	8	15	16	19	22	25	27	31	32	36	39	40	43	45

❶ ❷ ❸ ❹

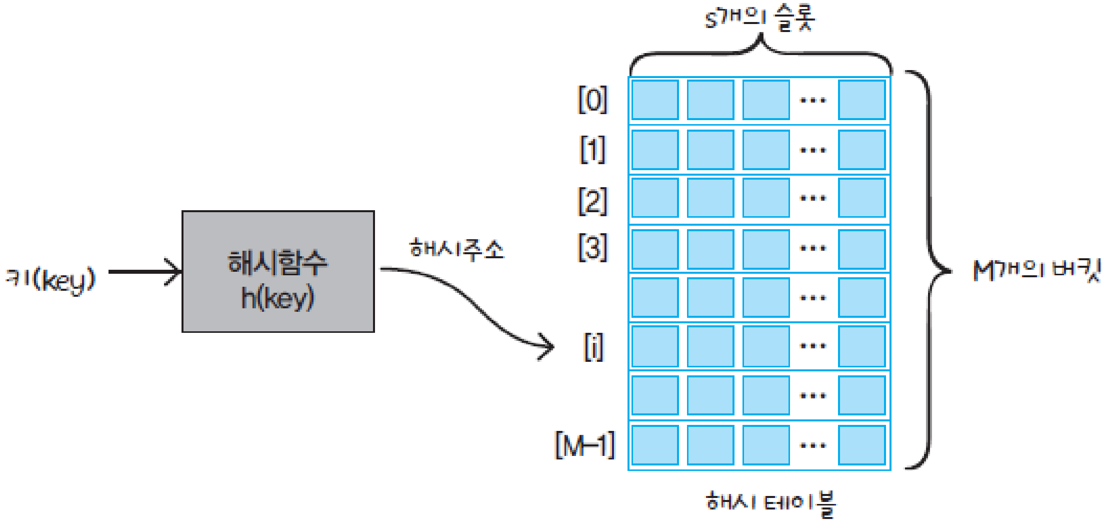
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
해싱으로 32 탐색:	31	45	8	43	15	27	22	19	25	3	36	32	39	40	16	5

❶

32에 대한 해시 주소 계산: 예) hash_function(32) = 11
해당 주소에서 탐색: 32가 테이블에 있다면 반드시 11번지에 있어야 함

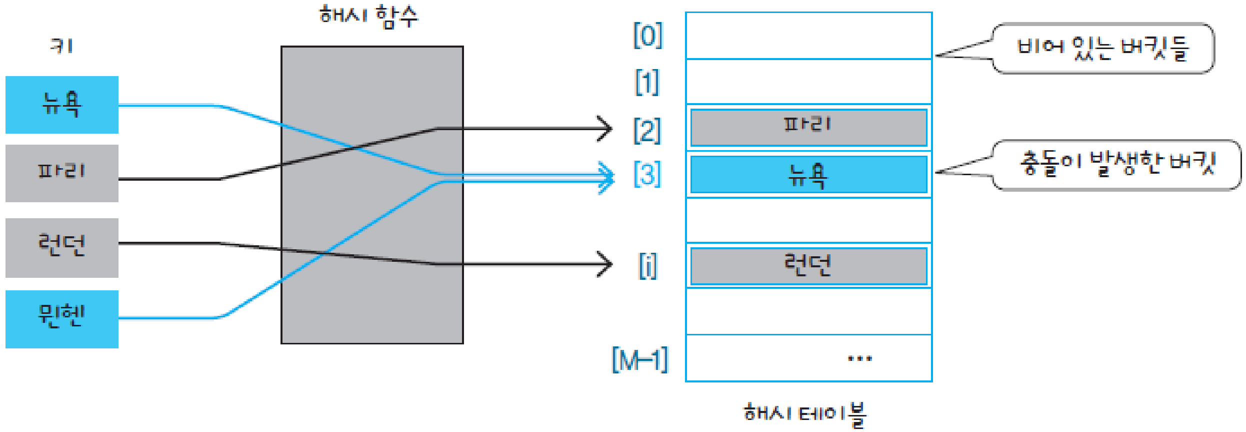
해싱의 구조

- 해시 테이블(hash table)
  - 레코드를 저장한 테이블
  - M개의 버킷(bucket)으로 구성
  - 각 버킷은 보통 여러 개의 슬롯(slot)으로 나누어지는데, 각 슬롯에 하나의 레코드가 저장됨
  - 단순화: 버킷에는 하나의 슬롯이 있다고 가정
- 해시 함수
  - 킷값에서부터 레코드의 위치인 해시 주소(hash address)을 계산



이상적인 해싱과 실제의 해싱

- 이상적인 해싱:
  - 충돌이 절대 일어나지 않는 경우
  - h(key) = key
  - 탐색, 삽입, 삭제 연산 : $O(1)$
  - 지나치게 많은 메모리가 필요해 비현실적
- 실제의 해싱
  - 테이블의 크기를 적절히 줄이고, 해시 함수를 이용해 주소 계산
  - 충돌(collision)이 발생
  - 오버플로 처리 필요: 충돌이 슬롯 수보다 더 많이 발생하는 경우



해시 함수

- 해시 함수 : 임의의 길이를 갖는 데이터를 고정된 길이의 데이터로 변환
  - 충돌이 적게 발생
  - 해시 결과가 테이블의 주소 영역 내에서 고르게 분포
  - 계산이 빨라야 함
- 예: 영어 단어의 첫 문자만을 취해 해시 주소를 만드는 것은 좋지 않음
- 제산 함수(나머지 연산 함수)
  - $h(k) = k\ mode\ M$
  - 테이블의 크기 M을 소수(prime number)로 선택
- 탐색키가 문자열인 경우

```
def hashFnStr(key) :
    sum = 0
    for c in key :                  # 문자열의 모든 문자에 대해
        sum = sum + (ord(c))        # 그 문자의 아스키 코드값을 sum에 더함
    return sum % M
```

문자를 아스키 코드로 바꿔주는 파이썬 내장함수

아스키 코드의 합에 제산함수를 적용

오버플로 처리하기

- **개방 주소법(open addressing)**
  - 오버플로가 일어나면 그 항목을 해시 테이블의 다른 위치(주소)에 저장.
  - 선형 조사법, 이차 조사법, 이중 해싱법 등
- 체이닝(chaining)
  - 해시 테이블의 하나의 위치에 여러 개의 항목을 저장할 수 있도록 테이블의 구조를 변경

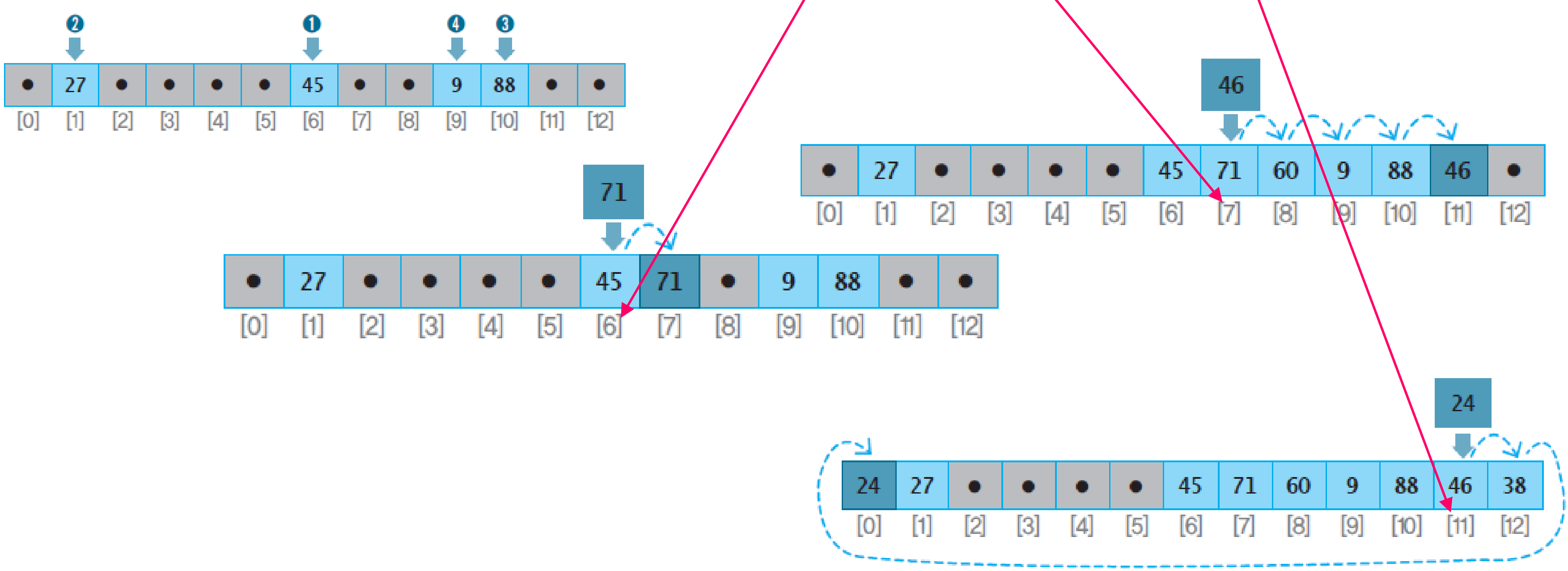
선형 조사법(linear probing)

- 선형 조사법:
  - 계산된 주소에 빈 슬롯이 없으면 다음 주소(버킷)들을 순서에 따라 조사하여 빈 슬롯을 찾는다.
  - 이때 비어 있는 공간을 찾는 것을 조사(probing)이라 함
  - 예:
    - 해시 테이블의 k번째 위치인 $h[k]$에서 충돌이 발생했다면 다음 위치인 $h[k+1]$부터 순서대로 비어 있는지를 살피고, 빈 곳이 있으면 저장
- 삽입 연산
  - 군집화(clustering) 발생
    - 한번 충돌이 발생한 위치 부근에서 연속적으로 충돌이 발생하는 경향
  - 오버플로우가 자주 발생하면 군집화에 따라 탐색 효율이 크게 저하

선형 조사법(linear probing)

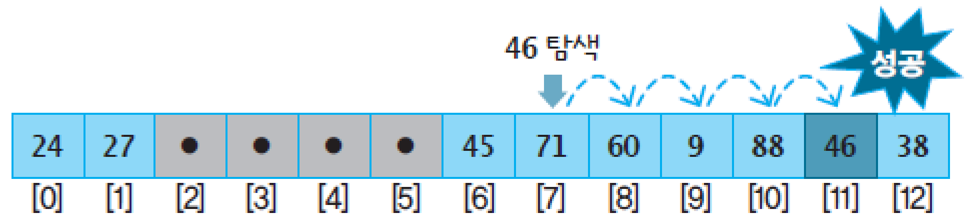
- 삽입 연산 : 군집화 발생
  - $h(k) = k \% M,\ M = 13$

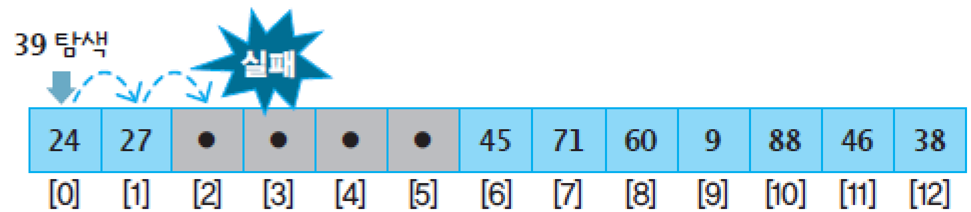
key	45	27	88	9	71	60	46	38	24
h(key)	6	1	10	9	6	8	7	12	11



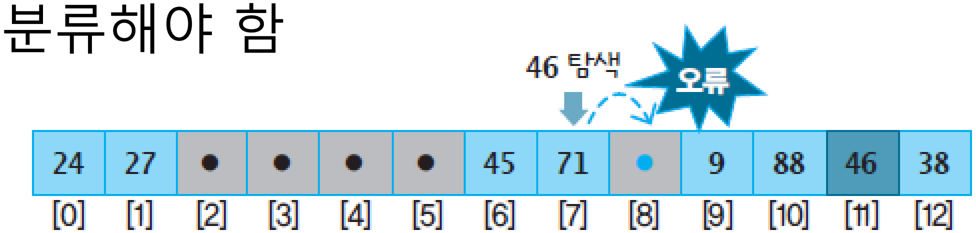
선형 조사법

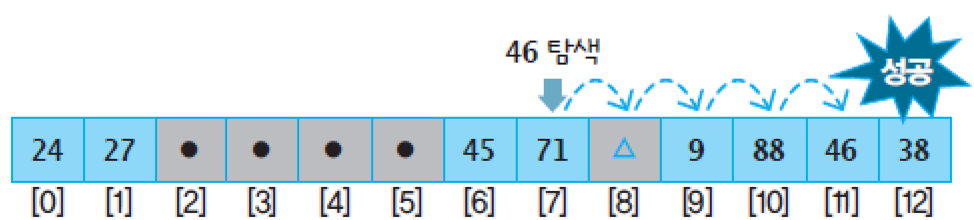
- 탐색 연산





- 삭제 연산
  – 빈 버킷을 두 가지로 분류해야 함





테스트 프로그램

```
print("   최초:", table )
lp_insert(45);  print("45 삽입:", table )
lp_insert(27);  print("27 삽입:", table )
lp_insert(88);  print("88 삽입:", table )
lp_insert(9);   print(" 9 삽입:", table )
lp_insert(71);  print("71 삽입:", table )
lp_insert(60);  print("60 삽입:", table )
lp_insert(46);  print("46 삽입:", table )
lp_insert(38);  print("38 삽입:", table )
lp_insert(24);  print("24 삽입:", table )
lp_delete(60);  print("60 삭제:", table )
print("46 탐색:", lp_search(46) )
```

실행 결과

```
   최초: [None, None, None, None, None, None, None, None, None, None, None, None, None]
45 삽입: [None, None, None, None, None, None, 45, None, None, None, None, None, None]
27 삽입: [None, 27, None, None, None, None, 45, None, None, None, None, None, None]
88 삽입: [None, 27, None, None, None, None, 45, None, None, None, 88, None, None]
 9 삽입: [None, 27, None, None, None, None, 45, None, None, 9, 88, None, None]
71 삽입: [None, 27, None, None, None, None, 45, 71, None, 9, 88, None, None]
60 삽입: [None, 27, None, None, None, None, 45, 71, 60, 9, 88, None, None]
46 삽입: [None, 27, None, None, None, None, 45, 71, 60, 9, 88, 46, None]
38 삽입: [None, 27, None, None, None, None, 45, 71, 60, 9, 88, 46, 38]
24 삽입: [24, 27, None, None, None, None, 45, 71, 60, 9, 88, 46, 38]
60 삭제: [24, 27, None, None, None, None, 45, 71, -1, 9, 88, 46, 38]
46 탐색: 46
```

탐색 성공