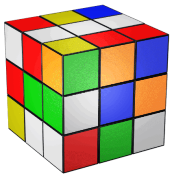
동적 프로그래밍 Part 1

한국기술교육대학교 컴퓨터공학부 김상진



```
while(!morning){
    alcohol++
    dance++
} #party
```

```
while(!sleep){
    think++
    solve++
} #cse-mode
```

교육목표

- 동적 프로그래밍(dynamic programming)
  - 같은 종류의 작은 문제를 해결한 다음 이를 이용하여 큰 문제를 해결하는 방법
  - 가장 작은 문제부터 주어진 문제까지 차례로 해결함
    - 상향식(bottom-up)
      - 비고. 분할 정복은 하향식(top-down)임
  - 동적 프로그래밍의 핵심은 소문제를 정의하는 것
    - 소문제를 정의하고 이 소문제를 이용하여 다음 크기의 문제를 해결하는 방법을 제시할 수 있어야 함
      - 재현식, 점화식(recurrence relation)
  - 기본 기법. 메모이제이션(memoization), 테뷸레이션(tabulation)
    - 메모이제이션: 하향식 형태(재귀 형태)이지만 한 번 계산한 값은 중복하여 계산하지 않음

- 전수조사 (경우의 수가 너무 많아 ㅠ.ㅠ)
- 분할정복 (다차 ⇒ 다차)
- 탐욕적 ⇒ 정확성 증명 필요
- 동적 프로그래밍

- 어렵고 직관적이지 않음
- 하지만 상대적으로 정형화된 설계 방법
- 프로그래밍 자체는 매우 간단할 수 있음
- 익히면 매우 강력한 도구임

살펴보는 문제

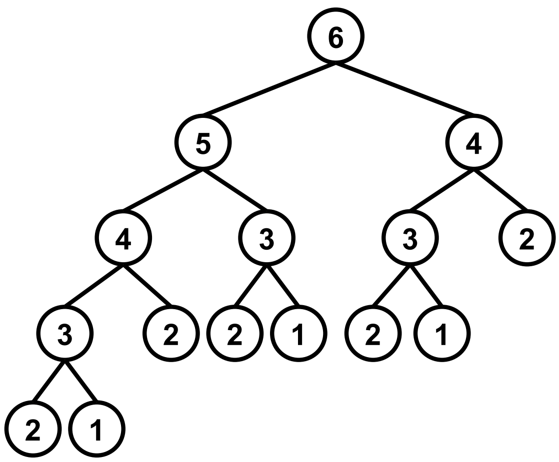
- Fibonacci: 개념 익히기 문제
- 전수조사 때 살펴본 문제를 동적 프로그래밍으로
  - CanSum
  - CountSum
  - HowSum
  - BestSum
- WIS
- 이항계수
- 합계가 가장 큰 구간 찾기: 분할 정복으로 해결한 문제
- LIS
  - 효과적인 메모이제이션 방법: 변하는 인자의 수 줄이기
- 0-1 배낭

피보나치 수

- 피보나치 수: 첫째 및 둘째 항이 1이며, 그 뒤의 모든 항은 바로 앞 두 항의 합인 수열
  - 0번째 항이 0이고, 첫째 항이 1로 표현하여 정의할 수 있음
- 입력. $n \geq 1$인 정수
- 출력. n번째 피보나치 수
- 재귀 함수로 구현

if $n <= 2$ ***then return 1***
return $f(n-1) + f(n-2)$

  - 시간 복잡도: $O(2^n)$
    - 트리 높이
  - 공간 복잡도: $O(n)$
    - 스택 공간



- 원 문제를 작은 문제를 이용하여 정의할 수 있어야 함
- 이 경우 작은 문제의 크기가 원 문제와 차이가 없음
- 계산할 때마다 작은 문제의 해가 변하지 않음

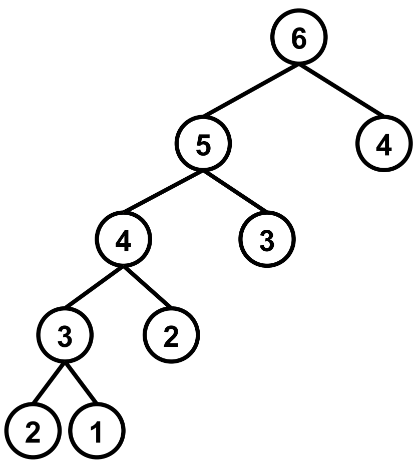
메모이제이션(memoization) (1/2)

- 메모이제이션. 재귀 호출에서 한 번 계산한 것을 기록하여 반복적으로 계산하지 않도록 하여 최적화하는 방법
- 소문제 수만큼의 공간이 필요함. 피보나치 문제: $f(1)$부터 $f(n)$까지 n개 문제
- 바뀌는 인자의 수와 범위를 분석해야 함
  - 피보나치: 바뀌는 인자는 1개이며, 정수 타입이고, 이 인자의 범위는 1 ~ n임
  - 기억해야 하는 것은 함수의 반환 값
    - 피보나치 수는 함수의 반환 값은 양의 정수임
  - 정수 배열을 이용하여 함수의 결과를 기억할 수 있음
    - 바뀌는 인자가 정수 타입이 아니면 배열 대신에 해시맵을 사용할 수 있음
- 배열의 초깃값을 통해 이미 계산하였는지 여부를 알 수 있어야 함
  - 함수의 반환 값이 양의 정수이므로 -1로 초기화하여 기억할 수 있음
  - 초깃값을 통해 구분하기 어려우면
    - 같은 크기의 bool 배열(또는 벡터)을 하나 더 사용
    - std::optional 배열(또는 벡터)

> - 해시맵을 사용하면 초깃값을 이용한 구분이 필요 없음
> - 배열을 이용하는 것보다 추가 비용
> - 공간 복잡도는 차이 없음

메모이제이션 (1/2)

- 시간 복잡도: $O(n)$
- 공간 복잡도: $O(n)$




```
fib(n, memo):
    if n ≤ 2 then return 1   // base case
    if memo[n] ≠ −1 then return memo[n]
    memo[n] ≔ fib(n − 1, memo) + fib(n − 2, memo)
    return memo[n]

computeNthFib(n):
    memo ≔ [−1]×n
    return fib(n, memo)
```

> - 재귀적으로 구현 방법을 알고 있으면 여기에 메모이제이션을 적용하는 것은 어렵지 않음

테뷸레이션(tabulation)

- 테이블을 이용하여 하향식 대신에 상향식

```
table := []
table[1] := 1
table[2] := 1
for i := 3 to n do
    table[i] := table[i − 1] + table[i − 2]
return table[n]
```

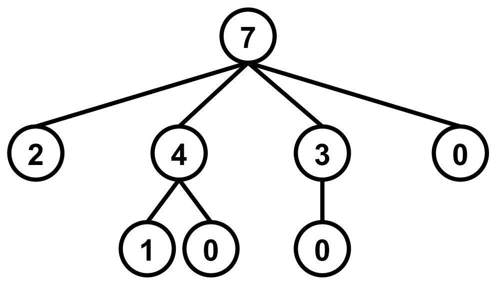
- 기저 사례에 대한 초기화
- 반복 시작 위치 결정

- 시간 복잡도: $O(n)$
- 공간 복잡도: $O(n)$
  - 같은 시간 복잡도이면 테뷸레이션이 보통 더 효과적인 방법임
  - $n+1$ 크기의 테이블을 사용하지 않고 변수 3개를 이용하여 구현 가능
  - 참고. 다른 문제는 테이블을 이용하여 해를 추가로 구해야 할 수 있음

- 이 문제처럼 점화식이 문제 자체에 주어지면 테뷸레이션으로 비교적 쉽게 해결할 수 있음

Cansum

- 입력. 목표 정수 $m(\geq 0)$, n개의 정수 $x_i > 0$
- 출력. 각 x_i을 원하는 만큼 합하여 목표 정수를 만들 수 있으면 true, 없으면 false
- 예) 7, [5, 3, 4, 7]
  - 답: true ⟹ (3, 4), (4, 3), (7)
- 트리로 표현




```
cansum(m, A[]):
    if m < 0 then return false
    if m = 0 then return true
    for all x ∈ A do
        if cansum(m − x, A) then return true
    return false
```

- 시간 복잡도: $O(n^m)$
- 공간 복잡도: $O(m+n)$

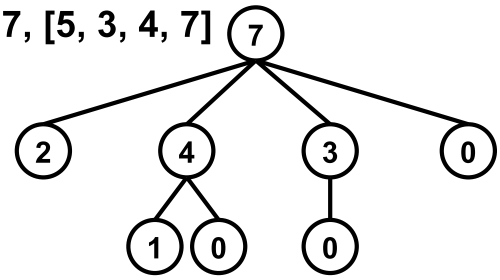
Cansum

- bool 배열로는 주어진 인자를 이용하여 계산했는지 여부를 알 수 없음
- 슬라이드 5에 제시한 3가지 방법 중 하나를 사용
- 이 예는 해시 집합 자료구조를 사용하고 있음
- -1, 0, 1을 유지하는 정수 배열을 이용할 수도 있음
- 사용하는 방법에 따라 공간 복잡도에 영향을 줌

- 메모이제이션

```
cansum(m, A[], memo):
    if m < 0 then return false
    if m = 0 then return true
    if m ∈ memo then return memo.get(n)
    for all x ∈ A do
        if cansum(m − x, A) then
            memo.set(m, true)
            return true
    memo.set(m, false)
    return false
```

7, [5, 3, 4, 7]



- 시간 복잡도: $O(mn)$
- 공간 복잡도: $O((2+\alpha)m+n)$

- m 때문에 여전히 지수 시간 알고리즘
- nums, memo, 스택 공간을 고려

$f(m) = \textbf{true if } \exists x \textbf{ in } A, f(m-x) = \textbf{true}$
$f(0) = \textbf{true}$

Cansum

$f(m) = \textbf{true if } \exists x \textbf{ in } A, f(m-x) = \textbf{true}$
$f(0) = \textbf{true}$

- 테뷸레이션
  - 예) 7, [5, 3, 4, 7]

T	F	F	F	F	F	F	F

```
canSum(m, A[])
    table := [false]×(m + 1)  // 0 색인
    table[0] := true
    for i := 0 to m − 1 do
        if table[i] then
            for all x ∈ A do
                if i + x ≤ m then table[i + x] := true
    return table[m]
```

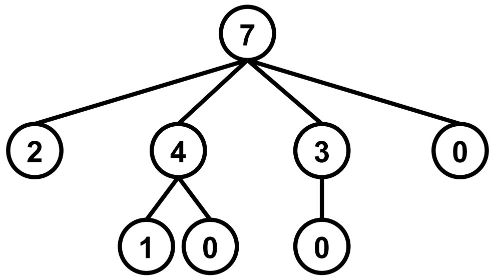
- 시간 복잡도: $O(mn)$
- 공간 복잡도: $O(m+n)$

- 시간 복잡도는 같지만 성능은 보통 테뷸레이션이 더 우수함
- Cansm은 해를 하나만 찾으면 중단할 수 있기 때문에 오히려 메모이제이션이 더 좋은 성능을 보일 수 있음

Countsum

- **입력.** 목표 정수 $m(\geq 0)$, n개의 정수 $x_i > 0$
- **출력.** 각 x_i을 원하는 만큼 합하여 목표 정수를 만들 수 있는 조합의 수
- **예)** 7, [5,3,4,7]
  - 답: 3 ⟹ (3,4), (4,3), (7)

cansum(m, $A[]$):
    if $m < 0$ *then return false*
    if $m = 0$ *then return true*
    for all $x \in A$ *do*
        if cansum($m - x$, A) *then return true*
    return false



- 시간복잡도: $O(n^m)$
- 공간복잡도: $O(m+n)$

- CanSum과 비교하면 중간에 중단할 수 없음

countsum(m, $A[]$)
    if $m < 0$ *then return 0*
    if $m = 0$ *then return 1*
    $count \coloneqq 0$
    for all $x \in A$ *do*
        $count$ += *countsum*($m - x$, A)
    return count

Countsum

countsum(m, $A[]$, *memo*)
    if $m < 0$ *then return 0*
    if $m = 0$ *then return 1*
    if $m \in memo$ *then return* *memo.get*(m)
    $count \coloneqq$ *0*
    for all $x \in A$ *do*
        $count$ += *countsum*($m - x$, A, *memo*)
    memo.put(m, *count*)
    return count

시간 복잡도: $O(mn)$
공간 복잡도: $O(2m+n)$

countsum(m, $A[]$)
    $table \coloneqq [0]\times(m+1)$ *// 0 색인*
    $table[0] \coloneqq 1$
    for $i \coloneqq 0$ *to* $m - 1$ *do*
        if $table[i] \neq 0$ *then*
            for all $x \in A$ *do*
                if $i + x \leq m$ *then* $table[i+x]$ += $table[i]$
    return $table[m]$

시간 복잡도: $O(mn)$ // 함수 호출 없음
공간 복잡도: $O(m+n)$

Howsum

- 입력. 목표 정수 $m(\geq 0)$, n개의 정수 $x_i > 0$
- 출력. 각 x_i을 원하는 만큼 합하여 목표 정수를 만들 수 있는 조합
- 예) 7, [5, 3, 4, 7]
  - 답: (3, 4), (4, 3), (7) 중 하나 출력

```
howsum(m, A[])
    if m < 0 then return null
    if m = 0 then return []
    for all x ∈ A do
        list ≔ howsum(m − x, A)
        if list ≠ null then
            add x to list
            return list
    return null
```

- 시간 복잡도: $O(n^m)$
- 공간 복잡도: $O(m + n)$

Howsum

- howSum은 하나의 답만 찾으면 됨

```
howsum(m, A[], memo)
    if m < 0 then return null
    if m = 0 then return []
    if m ∈ memo then memo.get(m)
    for all x ∈ A do
        list ≔ howsum(m − x, A, memo)
        if list ≠ null then
            next ≔ list.clone()
            add x to next
            memo.put(m, next)
            return next
    memo.put(m, null)
    return null
```

시간 복잡도: $O(m^2 n)$
공간 복잡도: $O(m^2 + n)$

- memo: $O(m^2)$
- 각 언어에서 구현하는 방법에 따라 추가 공간이 소요될 수 있음

```
howsum(m, A[])
    table ≔ [null]×(m + 1) // 0색인
    table[0] ≔ []
    for i ≔ 0 to m − 1 do
        if table[i] ≠ null then
            for all x ∈ A do
                if i + x ≤ m and table[i + x] = null then
                    table[i + x] ≔ table[i].clone()
                    add x to table[i + x]
                    // if i + x = m then break
    return table[m]
```

시간 복잡도: $O(m^2 n)$
공간 복잡도: $O(m^2 + n)$

- 동작 방식 때문에 메모이제이션은 clone을 하지 않아도 되지만 테뷸레이션은 clone이 필요함

Bestsum

- **입력.** 목표 정수 $m(\geq 0)$, n개의 정수 $x_i > 0$
- **출력.** 각 x_i을 원하는 만큼 합하여 목표 정수를 만들 수 있는 가장 짧은 조합 (같은 길이의 조합이 있으면 그들 중 임의로 출력)
- **예)** 7, [5,3,4,7]
  - 답: (7)
- 시간 복잡도: $O(n^m)$
- 공간 복잡도: $O(m+n)$

```
bestsum(m, A[])
    if m < 0 then return null
    if m = 0 then return []
    best := null
    for all x ∈ A do
        list := bestsum(m − x, A)
        if list ≠ null and (best = null or
            len(best) > len(list) + 1) then
            best := list.clone()
            add x to best
    return best
```



```
bestsum(m, A[], memo)
    if m < 0 then return null
    if m = 0 then return []
    if m ∈ memo then return memo.get(m)
    best := null
    for all x ∈ A do
        list := bestsum(m − x, A, memo)
        if list ≠ null then
            if best  = null or len(best) > len(list) + 1 then
                best := list.clone()
                add x to best
    memo.put(m, best)
    return best
```

시간 복잡도: $O(m^2n)$
공간 복잡도: $O(m^2+n)$

```
bestsum(m, nums)
    table := [null]×(m + 1) // 0색인
    table[0] := []
    for i := 0 to m − 1 do
        if table[i]  ≠ null then
            for all x ∈ A do
                if i + x ≤ m and (table[i + x] = null or
                    len(table[i + x]) > len(table[i]) + 1) then
                    table[i + x] := table[i].clone()
                    add x to table[i + x]
    return table[m]
```

시간 복잡도: $O(m^2n)$
공간 복잡도: $O(m^2+n)$

동적 프로그래밍의 원리

- 동적 프로그래밍 설계 방법의 3가지 단계 (테뷸레이션)
  - 단계 1. 부분 문제 식별하기
  - 단계 2. 작은 문제의 해로부터 큰 문제의 해를 구하는 효율적 방법 찾기
  - 단계 3. 모든 부분 문제의 해로부터 원 문제의 해를 구하는 효율적 방법 찾기
- 또 다른 방법: 전수조사 방법을 구현한 다음 메모이제이션 적용
- 단계 1에서는 부분 문제의 수가 중요함: 성능을 결정함
- 단계 2는 점화식 찾기
- 부분 문제를 식별하는 것과 점화식을 구하는 것은 훈련이 필요함!!!
- 피보나치 문제의 예
  - 단계 1: $n-1$개의 부분 문제를 해결해야 함. $f(2), f(3), \ldots, f(n)$
  - 단계 2: $f(n)=f(n-1)+f(n-2)$
    - 문제에 주어진 식임. 하지만 다른 문제는 이 식을 찾는게 어려울 수 있음
  - 단계 3: 가장 큰 부분 문제의 해가 원 문제의 해임
    - 항상 이와 같은 형태는 아님

동적 프로그래밍의 시간 복잡도

- 동적 프로그래밍 해결책의 성능

$$f(n)\times g(n)+h(n)$$

  - $f(n)$: 부분 문제의 수
  - $g(n)$: 부분 문제를 해결하는 비용
  - $h(n)$: 모든 부분 문제의 해로부터 원 문제의 해를 구하는 비용
- CountSum의 예)
  - $f(n): O(m), g(n): O(n), h(n): O(1)$
  - 전체 비용: $O(mn)$
    - 다중 변수 시간 복잡도 해석에 주의할 필요가 있음

메모이제이션

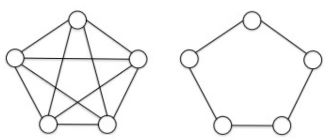
- 항상 적용 가능한 것은 아님
  - 참조적 투명성(referential transparency)을 제공하는 함수만 가능
    - 함수의 반환 값이 그 입력 값만으로 결정되는 경우
      - 입력이 같으면 항상 출력이 같은 함수
- 캐시의 크기는 소문제 수에 비례함
  - 보통 배열을 사용함
  - 해시 맵을 사용할 수 있음
- 보통 적절한 값으로 초기화해야 함
  - 적절한 값으로 초기화하기 어려운 경우 캐시 형태에 대한 고려 필요
- 메모이제이션의 시간복잡도
  - 대략 계산법
    - (존재하는 부분 문제의 수)x (한 문제를 풀 때 필요한 비용)

```
int cache[2500][2500];
int func(int a, int b){
    if(...) return ..; // base case
    int& ret = cache[a][b];
    if(ret != -1) return ret;
    ...
    return ret;
}

int computeFunc(int a, int b){
    memset(cache, -1, sizeof(cache));
    return func(a, b);
}
```


독립 집합

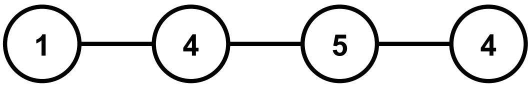
- 무방향 그래프 $G = (V, E)$가 주어졌을 때, V의 부분집합 S에 있는 노드들이 서로 인접한 노드가 아니면 S를 G의 독립 집합(independent set)이라 함
  - $v, w \in S$이면 $(v, w) \notin E$임
- 예) 다음 그래프에서 독립 집합의 개수는?



  - 완전 그래프는 모든 노드가 이웃 노드임
    - 따라서 각 노드로 구성된 부분집합과 공집합이 독립집합임 (6개)
  - 두 번째 그래프는?
    - {}, {1}, {2}, {3}, {4}, {5}, {1, 3}, {1, 4}, {2, 4}, {2, 5}, {3, 5}

Path Graph

- **입력.** 각 노드가 음수가 아닌 가중치 값을 가진 무방향 그래프 $G = (V, E)$



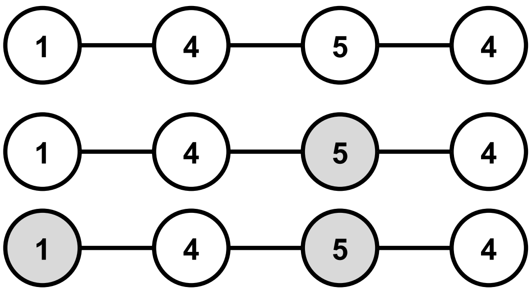
- **출력.** 가중치 합이 최대가 되는 인접하지 않는 노드 집합
  - 가중치 독립 집합(WIS, Weighted Independent Set)
- **경로 그래프**(path graph)는 WIS 문제의 단순 버전
  - 경로 그래프에서 경로는 단순 경로임. 따라서 경로 그래프는 선형 트리 형태임
- 주어진 예의 답은?
  - 독립 집합의 종류
    - {}, {1}, {4}, {5}, {4}, {1, 5}, {1, 4}, {4, 4}
  - [4, 4]: MWIS(maximum weighted independent set)
- 독립 집합의 수는 노드 수의 지수 비용으로 증가함
  - n개 원소로 구성된 집합의 부분집합의 수는?

탐욕적 기법

- How? 가장 가중치가 높은 노드부터 선택

S := empty set
sort V by weight
for all $v \in V$ ***do*** // (내림차순으로 처리)
    if $S \cup \{v\}$ *is an independent set of G* ***then*** $S := S \cup \{v\}$
return S

- 예)

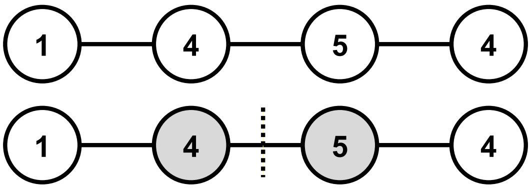


분할 정복법

- 전형적인 분할 정복법에 적용

G_1 := *first half of* G
G_2 := *second half of* G
S_1 := *recursively solve the WIS problem on* G_1
S_2 := *recursively solve the WIS problem on* G_2
combine S_1 *and* S_2 *into a solution for* G
return S

- 예)



  - 하지만 반으로 나눌 경우 왼쪽에서 4, 오른쪽에서는 5을 선택하게 되지만 두 노드는 인접 노드이기 때문에 두 답을 결합할 수 없음
    - 이 문제를 해결하는 방법은?

Optimal Substructure

- 최적 해결책의 구조를 생각하여 보자
  - 동적 프로그래밍의 핵심
  - 답의 구조로부터 알고리즘을 찾는 방법
  - 어떤 구조? 부분 문제의 최적 해를 이용하여 그보다 큰 문제의 해를 나타낼 수 있을까?
    - 재귀?
      - yes $\Longrightarrow$ top-down: 메모이제이션
      - no $\Longrightarrow$ bottom-up: 테블레이션
  - 문제가 매우 작으면 그 문제는 전수조사 방법으로 해결할 수 있음

WIS의 optimal substructure (1/2)

- $S \subseteq V$가 최대 가중치 독립 집합이라 하자. (우리가 구하고자 하는 답)
- v_n: 경로 그래프에 있는 마지막 노드
- Case Analysis
  - 경우 1. $v_n \notin S$
    - $G' = G - \{v_n\}$
    - S는 G'의 MWIS임 (why?)
      - S는 G'의 IS임. G'의 MWIS가 $S^* \neq S$이고, 이것의 가중치 합이 S보다 크다면 S가 G의 MWIS가 될 수 없음 (S^*은 G의 IS임)
  - 경우 2. $v_n \in S$
    - $v_{n-1} \notin S$
    - $G'' = G - \{v_{n-1}, v_n\}$
      - $S - \{v_n\}$은 G''의 MWIS임 (Why?)
        - $S^* \neq S - \{v_n\}$이고, 이것의 가중치 합이 $S - \{v_n\}$보다 크다면 S가 G의 MWIS가 될 수 없음

WIS의 optimal substructure (2/2)

- 최종해는 경우 1과 경우 2 중 하나
  - G의 MWIS = max(G'의 MWIS, G''의 MWIS $+ v_n$)
    - WIS 점화식: $G_n = max(G_{n-1}, G_{n-2} + w_n)$
  - 직관적으로 구현하면 재귀 알고리즘

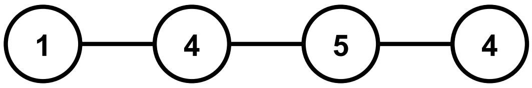
```
if n = 0 then return empty set
if n = 1 then return {v_1}
S_1 := recursively compute an MWIS of G_{n-1}
S_2 := recursively compute an MWIS of G_{n-2}
return max(S_1, S_2 ∪ v_n)
```

    - 하지만 중복 계산이 많음 $\Rightarrow$ 메모이제이션
    - 총 소문제의 수는? $n + 1$개 $(G_0, G_1, \ldots, G_n)$
      - 이 문제들만 해결하면 선형???

G_{n-1}, G_{n-2}를 해결하면 G_n을 해결할 수 있음
Bottom-up

테뷸레이션을 이용한 최적값 찾기

- 보통 bottom-up 방식으로 계산하여야 하며, 소문제 수만큼의 공간이 필요함
  - WIS 문제: $n+1$
- 초깃값
  - $table[\,]$: 용량이 $n+1$인 정수 배열
  - $table[0]=0$
  - $table[1]=w_1$
  - $table[i]=\max(table[i-1], table[i-2]+w_i)$
- 시간 복잡도: $O(n)$
  - 최적값만 얻음
    - 실제 해는? 다음 슬라이드

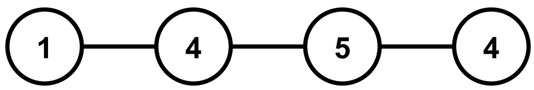


0	1	2	3	4
0	1	4	6	8

Path Graph 최적해 찾기

- 동적 프로그래밍 과정에서 배열에 값 뿐만 아니라 값을 구할 때 사용한 이전 요소를 배열에 저장
  - 답을 구할 수 있지만 좋은 방법은 아님
- 답을 구한 후 되돌아가면서 해를 구성
  - 어떻게?
    - v_i가 해에 포함될 필요충분조건
      - $w_i + max(G_{i-2}) \geq max(G_{i-1})$
  - 소요 비용: 보통 선형 비용 (시간 복잡도 그대로, 공간 복잡도 그대로)

0	1	2	3	4
0	1	4	6	8
-	-	-	1	2



$S := \{\}$
$i := n$
while $i \geq 2$ ***do***
  if $table[i-1] \geq table[i-2]+w_i$ ***then*** $i := i-1$
  else
    add v_i *to* S
    $i := i-2$
if $i = 1$ ***then*** *add* v_1 *to* S

$i=4$, $A[3] < A[2]+4$: add v_4 to S
$i=2$, $A[1] < A[0]+4$: add v_2 to S

분할정복과 비교

- 차이점 1. 분할 정복은 분할한 것을 결합하여 해를 구하는 반면에 동적 프로그래밍은 분할한 것 중 하나를 선택함
- 차이점 2. 분할 정복의 재귀 호출은 보통 중복되지 않음
- 차이점 3. 분할 정복은 직관적인 다차 시간 알고리즘의 성능을 개선함. 동적 프로그래밍은 직관적인 지수 시간 알고리즘을 다차 시간으로 개선함
  - 항상 이와 같이 획기적으로 개선하는 것은 아님. 예) 합계 문제
- 차이점 4. 분할 정복은 성능을 개선하기 위해 부분 문제를 선택함. 동적 프로그래밍은 답을 찾기 위해 부분 문제를 선택함
  - 빠른 정렬은 어떤 피봇을 선택하여도 올바르게 정렬함
- 차이점 5. 분할 정복은 부분 문제의 크기가 상수 비율로 축소되지만 동적 프로그래밍은 그렇지 않을 수 있음
  - WIS는 n에서 $n-1$

이항계수 (1/3)

- 이항계수(binomial coefficient)

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}, \qquad 0 \le k \le n$$

  - 2가지 용도
    - $(a+b)^n$을 전개할 때 $a^k b^{n-k}$의 계수 $\boxed{(a+b)^2 = \binom{2}{0} a^2 + \binom{2}{1} ab + \binom{2}{2} b^2}$
    - n개에서 k를 선택하는 조합의 수
  - 이 식을 이용하여 이 값을 바로 계산하기 힘듦
    - $n!$은 매우 큰 값임
  - 그러면 어떻게?
- 재귀적 정의

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0, k = n \end{cases}$$

이항계수 (2/3)

- 이 정의를 이용하여 분할 정복 방식으로 알고리즘을 구현할 수 있음

bin(n, k)
if $k = 0$ ***or*** $n = k$ ***then return*** *1* ***else return*** *bin*$(n-1, k-1)$ + *bin*$(n-1, k)$

  - 중복 계산을 너무 많이 함
- 동적 프로그래밍?
  - top-down이 아니라 bottom-up

이항계수 (3/3)

- 알고리즘

bin(n, k)

$B \coloneqq [[0]\times(n+1)]\times(n+1)$ *// 0 색인*
for $i \coloneqq 0$ ***to*** n ***do***
    for $j \coloneqq 0$ ***to*** *min(i, k)* ***do***
        if $j = 0$ ***or*** $j = i$ ***then*** $B[i][j] \coloneqq 1$
        else $B[i][j] \coloneqq B[i-1][j-1] + B[i-1][j]$
return $B[n][k]$

  - 시간 복잡도: $O(nk)$
  - 개선 방법
    - 2차원 배열 대신에 일차원 배열로…
    - $\binom{n}{k} = \binom{n}{n-k}$를 이용. 즉, $k > n - k$이면 바꾸어 계산

합계가 가장 큰 구간 찾기

- 입력. n개의 정수 배열
- 출력. 합계가 가장 큰 연속된 부분 구간의 합계
- 예) [-2, 1, -3, 4, -1, 2, 1, -5, 4] ⟹ 6 [4, -1, 2, 1]
- 3장에서 분할 정복으로 해결하였음 ⟹ $O(n \log n)$
- 이번에는 동적 프로그래밍으로
- WIS처럼 분석하여 점화식을 찾자
  - S: 가장 큰 부분 구간
  - L_n: n번째 요소로 끝나는 가장 큰 부분 구간
    - 다음과 같은 2가지 경우만 존재
      - 경우 1. v_n이 홀로 가장 큰 부분 구간을 형성하는 경우
      - 경우 2. $L_{n-1} + v_n$이 가장 큰 부분 구간을 형성하는 경우
    - $L_n = \max(v_n, L_{n-1} + v_n)$
  - $S = \max(L_1, L_2, \dots, L_n)$

합계가 가장 큰 구간 찾기

- 알고리즘

```
L := v[1]
S := v[1]
for i := 2 to n do
    L := max(L + v[i], v[i])
    S := max(S, L)
return S
```

- 시간복잡도: $O(n)$
- 해를 구하고 싶으면?
  - $S = \max(L_1, L_2, \dots, L_n)$에서 $S = L_i$일 때 색인 i를 알아야 함

LIS(Longest Increasing Sequence)

- 최대 증가 부분 수열 문제
  - 주어진 수열에서 얻을 수 있는 가장 긴 증가 부분 수열의 길이 구하기
    - 부분 수열이란: 수열의 요소로 구성된 수열
      - 예) [1 5 2 4 7 4]: [1 5] [5 2 7] [2 4 4] 등
    - 증가 부분 수열이란 (순 증가 수열을 말함)
      - 예) [1 5 2 4 7 4]: [1 5] [1 2 4 7] 등, [2 4 4]는 아님
  - LeetCode 300
- 완전 탐색
  - 경우의 수: 어떤 요소를 포함할 수 있고, 포함하지 않을 수 있음
    - 특정 요소를 포함한 경우 증가 부분 수열이기 때문에 다음에 포함하는 요소는 제약이 있음
    - 최악의 경우는 제외할 것이 없는 경우: 2^n

LIS (2/7)

- [1 5 2 4 7 4]: [1 5 7] [1 2 4 7]
- 답을 보아도 소문제를?
- WIS와 같은 방법으로 한번 접근?
  - 마지막 값이 최종해에 포함될 수 있고 포함되지 않을 수 있음
    - 포함되면 어떤 의미? 포함되지 않으면 어떤 의미?
  - 마지막 값이 LIS에 포함되기 위한 조건은?
    - 4을 생각하면 4는 그 앞에 있는 2와 1만 올 수 있음
    - 7을 생각하면 4, 2, 5, 1이 앞에 올 수 있음
      - 어떤 규칙???
      - 7로 끝나는 LIS의 길이는 4, 2, 5, 1로 끝나는 LIS 중 가장 긴 것 + 1
  - $\mathrm{LIS}(i) = \max\{LIS(k) \mid \forall k(1 \le k < i)\ s.t.\ A[k] < A[i]\} + 1$
    - 이것은 점화식이 아닌 것 같음???

소문제의 답을 알면 대문제의 답을 구할 수 있음
Bottom-up

LIS (3/7)

- 소문제를 못 찾고 점화식을 구하지 못하면 그냥 전수조사로???
  - 첫 번째 값은 LIS에 포함될 수 있고, 포함되지 않을 수 있음
  - 이전에 포함한 요소를 알아야 현재 요소를 포함할 수 있는지 판단할 수 있음

```
lis(A[], prevPos, pos) :
    if pos > len(A) then return 0
    // 현재 요소를 포함하지 않는 경우
    ret := lis(A, prevPos, pos + 1)
    // 현재 요소를 포함하는 경우
    if prevPos = 0 or A[prevPos] < A[pos] then
        ret := max(lis(A, pos, pos + 1) + 1, ret)
    return ret
```

lis(A, 0, 1)

  - 중복 계산이 많음 ⟹ memoization
  - 재귀 호출 과정에서 바뀌는 값: prevPos, pos
    - 2차원 배열이 필요함

LIS (4/7)

```
lis(A[], prevPos, pos, memo[][]) :
    if pos > len(A) return 0
    if memo[prevPos][pos] ≠ −1 then return memo[prevPos][pos]
    ret := lis(A, prevPos, pos + 1, memo)
    if prevPos = 0 or A[prevPos] < A[pos] then
        ret := max(lis(A, pos, pos + 1, memo) + 1, ret)
    memo[prevPos][pos] := ret
    return ret
```

lis(A, 0, 1, memo)

- 2차원 배열을 사용하고 있어 공간 복잡도가 너무 큼
- 공간복잡도를 줄이기 위해서는 재귀 호출에서 바뀌는 값을 줄이면 가능. 어떻게?

LIS (5/7)

- 전수조사 방식을 개선할 수 없나?
  - LIS의 시작 색인은 1, 2, … n일 수 있음

```
lis(A[]) :
    ret := 0
    for pos := 1 to n do
        ret := max(lis(A, pos), ret)
    return ret
```

  - 1부터 시작한 LIS의 그 다음 요소의 색인은?

```
lis(A[], pos) :
    ret := 1
    for next := pos + 1 to n do
        if A[pos] < A[next] then ret := max(lis(A, next) + 1, ret)
    return ret
```

  - 지난 번 시도와 달리 재귀 호출과정에서 변하는 것은 pos 하나

LIS (6/7)

```
lis(A[], pos, memo[]) :
    if memo[pos] ≠ −1 then return memo[pos]
    ret := 1
    for next := pos + 1 to n do
        if A[pos] < A[next] then ret := max(lis(A, next, memo) + 1, ret)
    memo[pos] := ret
    return ret
```

LIS (7/7)

- DP 알고리즘

$\mathrm{LIS}(i) = \max\{LIS(k) \mid \forall k(1 \le k < i)\ s.t.\ A[k] < A[i]\} + 1$

LIS(A[])

```
dp := [0]×n
dp[1] := 1
ret := 1
for i := 2 to n do
    maxSequence := 0
    for j := 1 to i − 1 do
        if A[i] > A[j] then
            maxSequence := max(maxSequence, dp[j])
    dp[i] := maxSequence + 1
    ret := max(ret, dp[i])
return ret
```

$dp[i]$는 i까지 요소들만 이용한 LIS

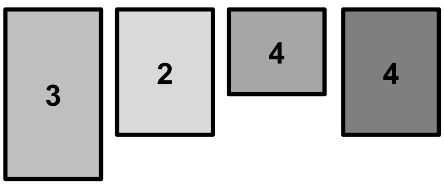
현재 요소가 이전 요소보다 크면
해당 요소가 마지막인 IS에 추가 가능

  - 시간 복잡도: $O(n^2)$
    - 이진 탐색을 이용하는 $O(n \log n)$ 시간 복잡도 알고리즘도 있음
    - LeetCode 300 solution 참고

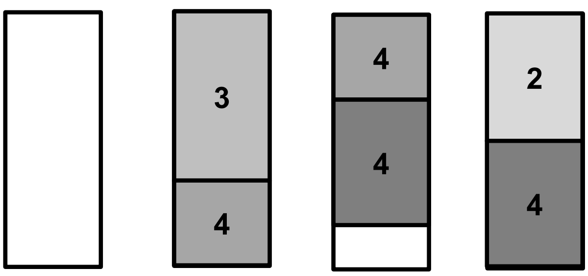
0-1 배낭 채우기 문제

- 입력. n개의 item과 가방 하나
  - 각 item은 음의 아닌 값 v_i와 음의 아닌 크기 w_i를 가짐
  - 가방의 용량 W
- 출력. $\sum_{i \in S} v_i$가 가장 큰 부분집합 $S \subseteq \{1, 2, \ldots, n\}$
  - 단, $\sum_{i \in S} w_i \le W$를 만족해야 함

(v_i, w_i)



- 예) $n = 4, W = 6, \{(3,4), (2,3), (4,2), (4,3)\}$



- 0-1? 특정 물건은 포함하거나 포함하지 않음
  - 배낭 빈틈없이 채우기 문제: 물건을 잘라서 담을 수 있음

0-1 배낭 채우기 문제

- 탐욕적 기법으로 가능하지 않을까? (v_i, w_i)
  - 가장 가치가 높은 순으로 정렬한 후 채우는 방법
    - 예) {(10, 25), (9, 10), (9, 10)}, 30
      - 알고리즘 결과: [1], 10 ⟹ 최적해: [2,3], 18
  - 가장 무게가 적은 순으로 정렬한 후 채우는 방법
    - 예) {(4, 10), (4, 10), (10, 25)}, 30
      - 알고리즘 결과: [1,2], 8 ⟹ 최적해: [3], 10
  - 무게당 가치 순으로 정렬한 후 채우는 방법
    - 예) {(50, 5), (60, 10), (140, 20)}, 30
      - score: 10, 6, 7
    - 알고리즘 결과: [1, 3], 190 ⟹ 최적해: [2, 3], 200
    - 빈틈없이 채우기 문제는 이 알고리즘으로 최적해를 얻을 수 있음
      - 알고리즘 결과: [1, 3, 2의 반], 220

동적 프로그래밍 접근

- 경우 1. $n \in S$
  - $w_n \le W$
  - $S - \{n\}$은 $n - 1$개 item으로 구성된 용량이 $W - w_n$인 가방 문제의 최적해
    - 증명) 모순. S^*이 $S - \{n\}$보다 더 최적해이면 $S^* \cup \{v_n\}$이 S보다 최적해임
- 경우 2. $n \notin S$
  - S는 $n - 1$개 item으로 구성된 용량이 W인 가방 문제의 최적해
    - 모순 방법을 이용하여 증명 가능
- 점화식으로 구성해 보자
  - $V_{i,x}$: 가방 용량이 x일 때, 첫 i개 item만 사용하는 최적해
  - $V_{i,x} = \max(V_{i-1,x}, v_i + V_{i-1,x-w_i})$
    - edge case: $w_i > x$이면 $V_{i,x} = V_{i-1,x}$
    - base case: $i = 0$ 또는 $x = 0$이면 $V_{i,x} = 0$

알고리즘

- 두 개의 변수로 점화식이 정의됨
  - $(n+1)\times(W+1)$ 2차원 배열이 필요함
- 알고리즘

```
knapsack(n, W)
A := [[0]×(n+1)]×(W+1)      // 0 색인
for i := 0 to W do
    A[0][i] := 0
for i := 1 to n do
    for x := 1 to W do
        if w_i > x then A[i][x] := A[i−1][x]
        else A[i][x] := max(A[i−1][x], A[i−1][x−w_i] + v_i)
return A[n][W]
```

  - 시간 복잡도: $O(nW)$

예) $n = 4, W = 6$

- 입력 (v_i, w_i): (3, 4), (2, 3), (4, 2), (4, 3)
  - 핵심 알고리즘: $A[i][x] = max(A[i-1][x], A[i-1][x-w_i]+v_i)$

가방용량

$x = 6$	0	3	3	7	8
$x = 5$	0	3	3	6	8
$x = 4$	0	3	3	4	4
$x = 3$	0	0	2	4	4
$x = 2$	0	0	0	4	4
$x = 1$	0	0	0	0	0
$x = 0$	0	0	0	0	0
	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$

$x - w_1 = 1 - 4 < 0$

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

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

$\max(A[3][5] = 6, A[3][2] + 4 = 8) = 8$

예) $n = 4, W = 6$

- 입력 (v_i, w_i): (3, 4), (2, 3), (4, 2), (4, 3)
  - 해 재구성

$w = 6$	0	3	3	7	8
$w = 5$	0	3	3	6	8
$w = 4$	0	3	3	4	4
$w = 3$	0	0	2	4	4
$w = 2$	0	0	0	4	4
$w = 1$	0	0	0	0	0
$w = 0$	0	0	0	0	0
	$i = 0$	$i = 1$	$i = 2$	$i = 3$	$i = 4$

$A[3][6-3] + 4 \geq A[3][6]$

$A[2][3-2] + 4 \geq A[2][3]$

S := *empty set*
$w := W$
for $i := n$ ***downto 1 do***
    if $w_i \leq w$ ***and*** $A[i-1][w-w_i] + v_i \geq A[i-1][w]$ ***then***
        $S := S \cup \{i\}$
        $w := w - w_i$
return S