Dynamic Programming

(CLRS Chapter 15)

김동진
(NHN NEXT)

Dynamic Programming 요약

- 용도
  - 최적화 문제 등에 적용
- 구조
  - 주어진 문제를 여러 개의 작은 문제로 분할
    - 최적화 문제의 경우 분할 가능한 모든 경우를 고려
  - 작은 문제의 해를 이용해서 주어진 문제의 해를 구함
  - 구한 해를 재사용
    - 작은 문제가 더 작은 문제로 분할되어 재귀 호출되는 과정에서 동일한 문제가 중복 발생할 경우
    - 처음에 계산된 결과를 저장한 후 재호출 시 이전에 계산된 결과를 바로 return
- 특징
  - Brute-force 방식의 접근
    - 모든 경우를 확인
    - 모든 경우를 확인할 경우 exponential time algorithm
  - 중간 계산 결과 재활용
    - exponential time을 polynomial time으로 개선

Dynamic programming algorithm 개발 절차

1. Optimal solution 구조 분석
2. Optimal solution을 재귀적으로 정의
3. 중간 계산 결과 저장 방법 수립
   - 재귀 호출 시 계산 결과 저장 혹은
   - bottom-up 방식으로 계산 수행
4. 계산된 결과를 이용해서 optimal solution 구축

목 차

- Fibonacci
- Rod-cutting problem
- Shortest Path problem
- Matrix-chain multiplication problem
- Longest common subsequence problem
- Knapsack problem

Fibonacci Sequence (1)

- Fibonacci의 정의
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ for n >= 2

- 문제
  - Recursion을 사용해서 F_n 계산하는 프로그램 구현하시오.
    - 주어진 문제: F_n
    - 작아진 문제: F_{n-1}과 F_{n-2}
    - 작아진 문제인 F_{n-1}과 F_{n-2} 를 알면 두 개를 더해서 F_n을 계산할 수 있다.
  - 각자 손코딩

Fibonacci Sequence (2)

- Solution

```
int fib(int n)
{
    if(n < 2) return n;
    return fib(n-1) + fib(n-2);
}
```

- 위 코드의 time complexity
  - $O(2^n)$
  - $\Theta(\phi^n)$, $\phi = (1+\sqrt{5})/2$

- 더 효율적인 방법은 없을까?

Fibonacci Sequence (3)

- 비효율적인 이유
  - 같은 계산을 반복하고 있다.

- 효율적 알고리즘 전략
  - 한 번 계산한 값은 저장해서 다시 사용한다.

- 방법
  - 계산 결과 값을 저장
  - Recursive function 진입 시 이미 계산했는지 확인
  - 이미 계산했으면 저장된 값을 사용

Fibonacci Sequence (4)

- Recursive 방법

```
int fib_DP_TopDown(int n, int *res)
{
	if(res[n] != -1) return res[n];

	if(n < 2)
			res[n] = n;
	else
			res[n] = fib2(n-1) + fib(n-2);

	return res[n];
}
.........
// 호출 코드
int *res = new int[n+1];
for(int id = 0; id <= n; id++) res[id] = -1;
cout << fib2(n, res) << endl;
```

time complexity: $O(n)$

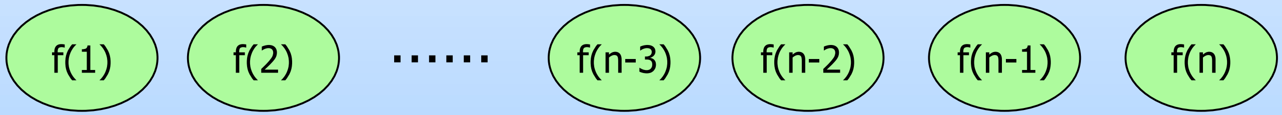
Dynamic Programming:
top-down with memoization

Fibonacci Sequence (5)

- Fibonacci의 recursive 호출 구조
  - 주어진 문제: fib(n)
  - 호출 순서
    - Recursion tree의 위에서부터 아래 방향 순서로 호출
  - 계산 순서
    - Recursion tree의 아래에서부터 위 방향 순서로 계산

Fibonacci Sequence (6)

- Topological sort 기반 방법
  - Recursion tree의 각 노드를 하나의 subproblem으로 간주
  - 각 edge의 방향을 바꾸어서 graph 생성
    - Node 집합 = {f(i) | 1<= i <= n}
    - Edge 집합 = {f(u) → f(v) | if f(v)가 f(u)를 재귀 호출}
  - Subproblem들을 topological sort



  - Topological sort 결과를 앞에서부터 뒤로 scan하면서 결과 생성

Fibonacci Sequence (7)

- 테이블 채우는 방법
  - Topological sort 결과 분석
    - 작은 값부터 계산
    - 즉, fib(1), fib(2), ..., fib(n)의 순서
  - 절차
    - 배열 생성
    - 배열의 각 원소는 subproblem에 대응됨
    - 배열의 index가 작은 값부터 차례로 계산
    - 즉, fib[1], fib[2], ..., fib[n]의 순서로 계산

Fibonacci Sequence (8)

- 테이블 채우는 방법

```
int fib_DP_BottomUp(int n)
{
	int		res;
	int		* resArr = new int[n+1];

	resArr[0] = 0;
	resArr[1] = 1;

	for(int id = 2; id <= n; id++)
		resArr[id] = resArr[id – 1] + resArr[id – 2];

	res = resArr[n];

	delete[] resArr;

	return res;
}
.........
// 호출 코드
cout << fib_DP_BottomUp(n) << endl;
```

Dynamic Programming:
Bottom-up method

실습 (1)

- fibonacci_by_recursion() 코드 구현
  - 재귀 호출 방법
- Fibonacci_top_down() 코드 구현
  - Memoization 방법
- Fibonacci_bottom_up() 코드 구현
  - Bottom-up 방법
- Topological sort를 이용한 Fibonacci 계산 코드 구현
  - 각 fibonacci number를 node로 하는 graph 구성
  - Topological sort
  - topological sort 결과 순서로 계산

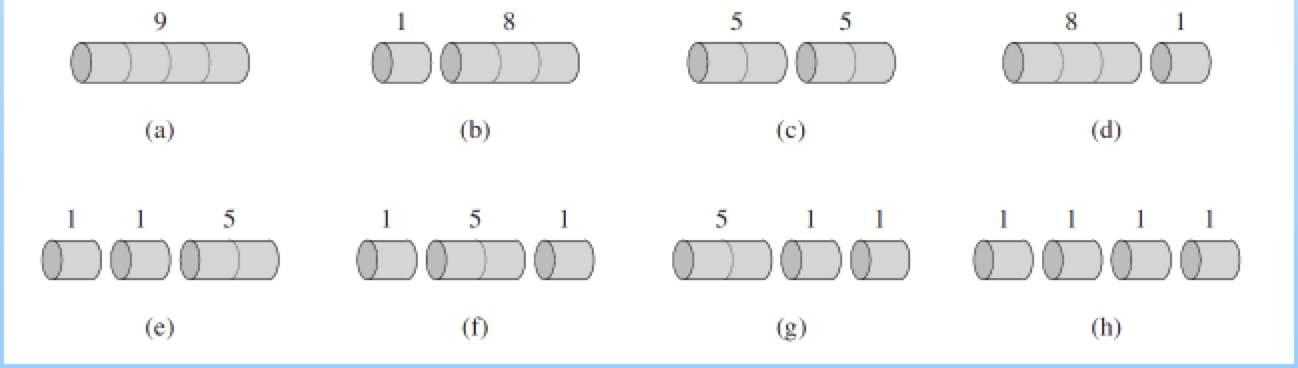
목 차

- Fibonacci
- Rod-cutting problem
- Shortest Path problem
- Matrix-chain multiplication problem
- Longest common subsequence problem
- Knapsack problem

Rod-Cutting (1)

- 문제 정의
  - Input
    - 막대 길이 n
    - 막대 길이별 가격
  - Output
    - 각 조각 가격의 합이 최대가 되도록 하는 정수 길이로 잘라진 여러 개의 조각
- 예제: 길이 4인 막대

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



Rod-Cutting (2)

- Recursive method
  - 아이디어
    - 앞에서 첫 번째 자른 위치를 정한다.
    - 나머지 부분에서 가격이 최대가 되는 결과를 구한다.
    - 위 두 개의 합을 계산한다.
    - 첫 번째 자른 위치에 따라서 계산된 결과 중 최대값이 원하는 결과값이다.
  - 각자 코딩 (손코딩부터…)

Rod-Cutting (3)

- Pseudo code

```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

Rod-Cutting (4)

◆ Code

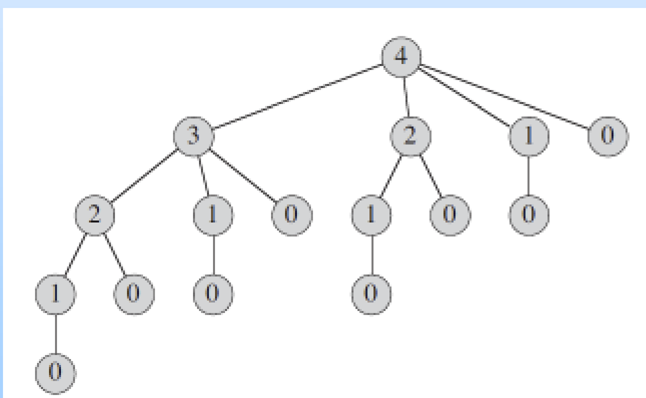
```c
int rod_cut_recursion_2(int *pieceValue, int len)
{
	int	maxValue;
	int	value;

	if(len == 0) return 0;

	maxValue = -1;
	for(int firstPieceLen = 1; firstPieceLen <= len; firstPieceLen++) {
		value = pieceValue[firstPieceLen] + rod_cut_recursion_2(pieceValue, len - firstPieceLen);

		if(value > maxValue) maxValue = value;
	}

	return maxValue;
}
```

Rod-Cutting (5)

- CUT-ROD(p, n) 함수 분석

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

  - n=4일 때의 Recursion tree



  - Time Complexity

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \Rightarrow O(2^n)$$

Rod-Cutting (6)

- $T(n) = 2^n$ 증명
  - Induction 방법 사용
  - Basis step
    - $T(0) = 1$
  - Inductive step
    - $T(k) = 2^k$이라고 가정
    - $T(k+1) = 1 + T(1) + T(2) + \cdots + T(k)$

      $= 1 + 2^1 + 2^2 + \cdots + 2^{n-1}$

      $= 2^{k+1}$
  - 따라서, $T(n) = 2^n$

Rod-Cutting (7)

- Top-down with memoization 방법
  - Recursion
  - 이미 계산된 값이 있으면 추가 recursive call 하지 않고 계산된 값 이용

- 각자 코딩(손코딩부터…)
  - Fibonacci 계산 시 memorization 방법 참고

Rod-Cutting (8)

- Top-down with memoization pseudo code

```
MEMOIZED-CUT-ROD(p, n)
let r[0..n] be a new array
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Rod-Cutting (9)

- Top-down with memoization code

```
int rod_cut_memoization(int *pieceValue, int len)
{
    int     *resValue;
    int     maxValue;

    if(len == 0) return 0;

    resValue = new int[len+1];
    for(int i = 0; i <= len; i++) resValue[i] = -1;

    maxValue = rod_cut_memoization_aux(pieceValue, len, resValue);

    delete[] resValue;

    return maxValue;
}
```

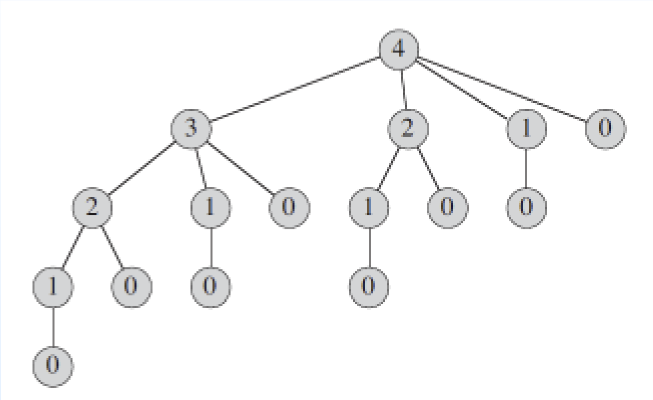
Rod-Cutting (10)

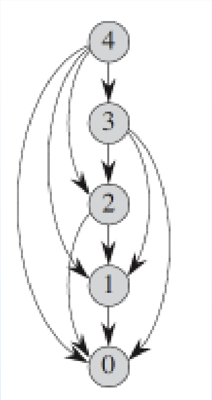
- Code

```c
int rod_cut_memoization_aux(int *pieceValue, int len, int *resValue)
{
    int     value;
    int     maxValue;

    if(resValue[len] != -1) return resValue[len];

    if(len == 0) return 0;

    maxValue = -1;
    for(int cutPos = 1; cutPos <= len; cutPos++) {
        value = pieceValue[cutPos] + rod_cut_memoization_aux(pieceValue, len - cutPos, resValue);

        if(value > maxValue) maxValue = value;
    }

    resValue[len] = maxValue;

    return maxValue;
}
```

Rod-Cutting (11)

- Top-down with memoization 방법 분석
  - Recursion tree　　　　　　Subproblem graph





  - 문제의 분할
    - parent node → 여러 개의 child nodes
    - Child nodes에 해당하는 문제를 해결하면 parent에 해당하는 문제의 해를 구할 수 있다.
    - 각 node별로 자신보다 작은 크기의 문제로 분할됨
  - 아이디어
    - Topological sort 결과의 순서로 문제 해결 : 0 → 1 → 2 → 3 → 4

Rod-Cutting (12)

- Bottom-up 방법
  - 결과 저장할 배열 사용
  - 크기가 제일 작은 문제부터 큰 문제 순서로 해결
  - 문제 해결할 때마다 배열에 결과값 저장
  - 문제를 해결할 때는 이전에 계산된 작은 문제의 결과를 이용
- 각자 손코딩

Rod-Cutting (13)

- Bottom-up 방법 pseudo code

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```

```
int rod_cut_buttomup(int *pieceValue, int totalLen)
{
    int     value;
    int     subMaxValue;
    int     maxValue;
    int     *resValue = new int[totalLen + 1];

    resValue[0] = 0;

    for(int len = 1; len <= totalLen; len++) {
        subMaxValue = -1;
        for(int subLen = 1; subLen <= len; subLen++) {
            value = pieceValue[subLen] + resValue[len - subLen];
            if(value > subMaxValue) subMaxValue = value;
        }
        resValue[len] = subMaxValue;
    }

    maxValue = resValue[totalLen];

    delete[] resValue;

    return maxValue;
}
```

Rod-Cutting (15)

- 시간 복잡도 분석
  - Top-down memoization

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

$O(n^2)$

  - Bottom-up method

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

$O(n^2)$

Rod-Cutting (16)

- 결과 생성
  - 앞의 방법들은 최대값을 찾는 방법만 기술하고 있음
  - 각 조각의 길이를 출력하는 방법 필요함
  - 아이디어
    - 1단계: 각 길이 별로 첫 번째 절단 길이 저장
    - 2단계: 저장된 정보를 이용해서 출력
- 각자 코딩(손코딩부터…)

Rod-Cutting (17)

- 조각 길이 출력하는 bottom-up 방법의 pseudo code

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] and s[0..n] be new arrays
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        if q < p[i] + r[j - i]
            q = p[i] + r[j - i]
            s[j] = i
    r[j] = q
return r and s
```

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n - s[n]
```

```
int extended_bottom_up_cut_rod(int *pieceValue, int totalLen, int *firstCutForLength)
{
	int		value;
	int		subMaxValue;
	int		maxValue;
	int		*resValue = new int[totalLen + 1];

	resValue[0] = 0;
	firstCutForLength[0] = 0;

	for(int len = 1; len <= totalLen; len++) {
		subMaxValue = -1;
		for(int subLen = 1; subLen <= len; subLen++) {
			value = pieceValue[subLen] + resValue[len - subLen];
			if(value > subMaxValue) {
				subMaxValue = value;
				firstCutForLength[len] = subLen;
			}
		}
		resValue[len] = subMaxValue;
	}

	maxValue = resValue[totalLen];

	delete[] resValue;

	return maxValue;
}
```

```
void	printSolution(int *firstCutForLength, int totalLen)
{
	int		len;

	len = totalLen;
	while(len > 0) {
		cout << firstCutForLength[len] << " ";
		len -= firstCutForLength[len];
	}
	cout << endl;
}
```

실습 (2)

- Rod-cutting의 recursive method 구현
- Rod-cutting의 memorization method 구현
- Rod-cutting의 bottom-up method 구현
- Rod-cutting의 조각 길이 출력하는 코드 구현

HW.C1

- 실습 2 완료

목 차

- Fibonacci
- Rod-cutting problem
- Shortest Path problems
- Matrix-chain multiplication problem
- Longest common subsequence problem
- Knapsack problem

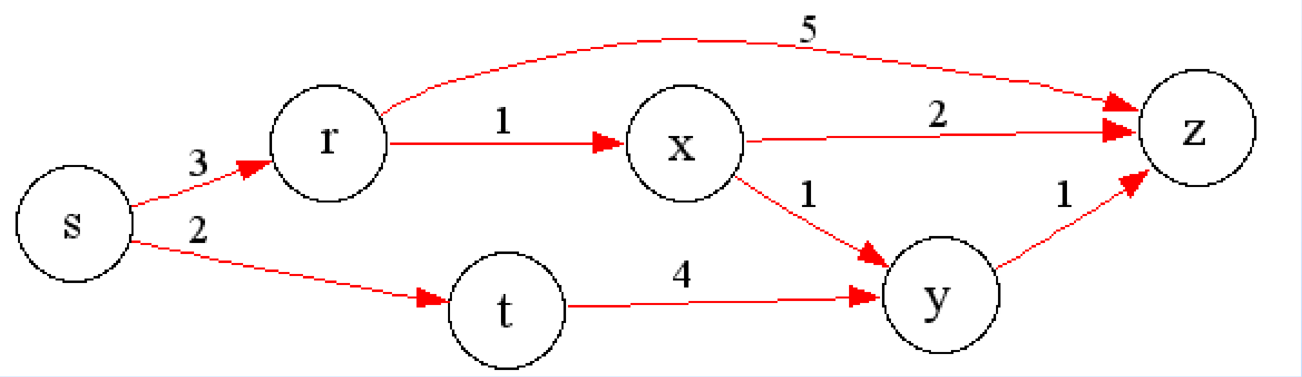
최단 경로 찾기

- 목표:
  - Dynamic programming 관점으로 최단 경로 찾는 문제 해석

- 다룰 문제
  - Directed acyclic graph (DAG)에서 최단 경로 찾기
  - Cycle이 존재하는 graph에서 최단 경로 찾기(단, negative weight cycle은 없음)

Single-source shortest paths in direct acyclic graph (1)

- 문제
  - 주어진 directed graph에 cycle이 존재하지 않는 경우
  - Source vertex가 주어지면 모든 vertex까지의 최단 경로를 구하라.

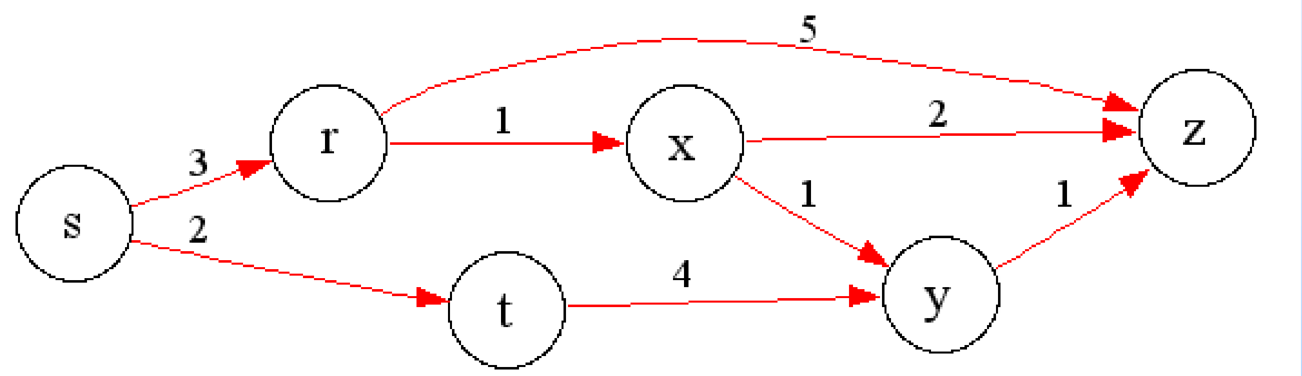


  - $\delta(s, z)$: s로부터 z까지의 최단 경로

- 아이디어
  - Dynamic programming 기법을 적용하자.

Single-source shortest paths in direct acyclic graph (2)

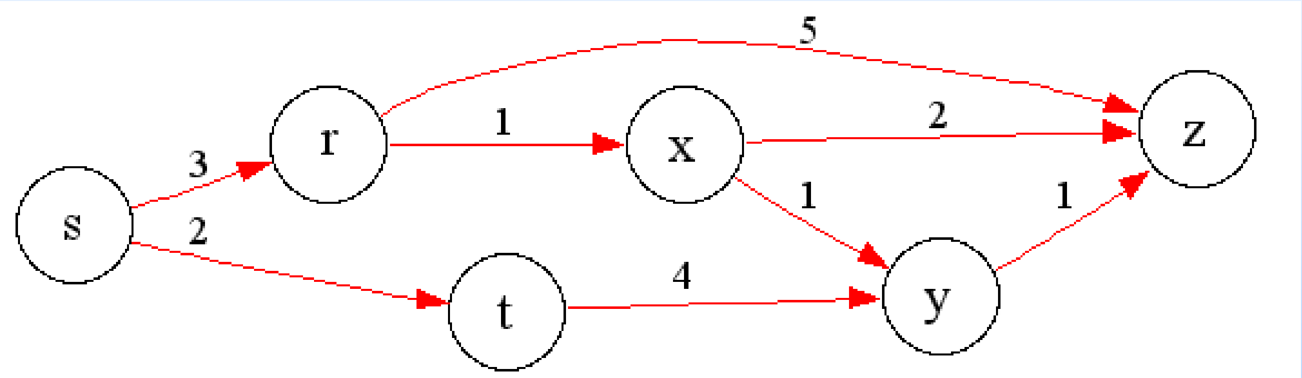
- 분석
  - Source로부터 주어진 node z까지의 최단 경로는 자신을 가리키는 edge의 시작 vertex를 경유하는 경로 중 하나임
  - 가능한 모든 경로 중 경로 weigh가 제일 작은 경로를 찾는다.
    - “인접한 vertex까지의 최단 경로 + edge weight” 중 최소값



    - $\delta(s,z) = \min(\delta(s,r) + 5, \delta(s,x) + 2, \delta(s,y) + 1)$
  - 한번 구한 최단 경로 결과는 재사용
    - $\delta(s,x)$가 구해지면 $\delta(s,y)$와 $\delta(s,z)$ 계산 시 사용 가능

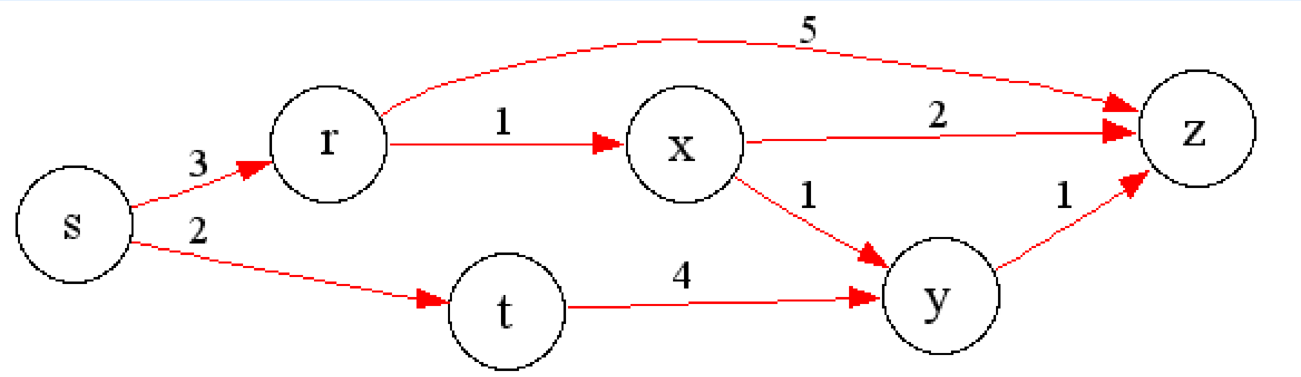
Single-source shortest paths in direct acyclic graph (3)

- 재귀적 정의
  - $\delta(s,v) = \min_{\forall(u \to v)} \{\delta(s,u) + w(u,v)\}$
    - $\delta(s,v)$: s로부터 v까지의 최단 경로
    - $w(u,v)$: $u \to v$ edge의 weight



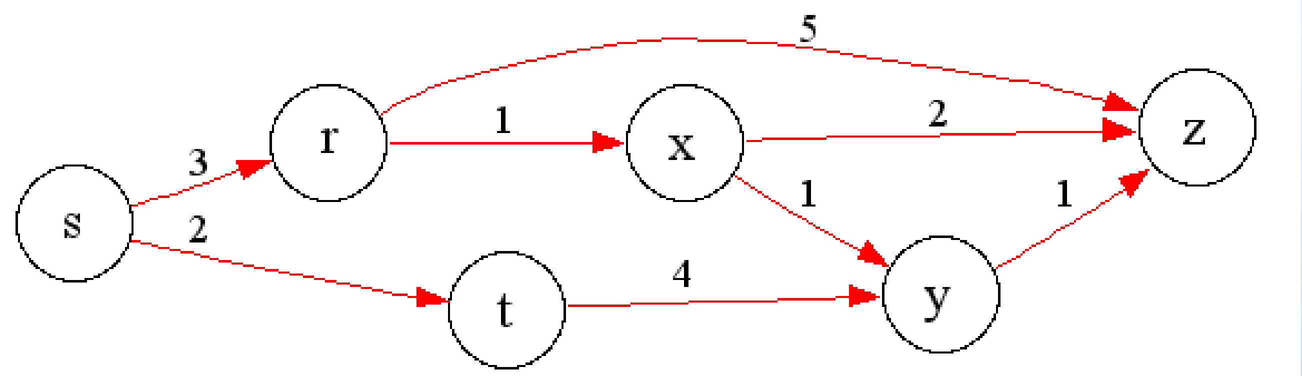
Single-source shortest paths in direct acyclic graph (4)

- Recursive method
  - 각 node 초기화
    - 최단 경로 weight를 무한대 설정 혹은 계산 미완료 표시
  - Recursive call 단계
    - 인접 노드의 최단 경로 weight가 이미 계산된 경우 저장된 값 사용
    - 인접 노드의 최단 경로 weight가 계산되지 않았으면 recursive call
    - 최단 경로 완료될 때마다 최단 경로 weight 저장



Single-source shortest paths in direct acyclic graph (5)

- Topological sort 사용
  - Node들을 topological sort한 후 앞에서부터 최단 경로 계산
  - 예제
    - 계산 순서: s → r → x → t → y → z



Single-source shortest paths in direct acyclic graph (6)

- Pseudo code

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4      for each vertex v ∈ G.Adj[u]
5          RELAX(u, v, w)
```

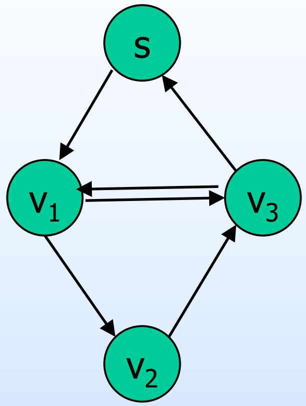
- Time complexity
  - Topological sort의 time complexity: $\Theta(V+E)$
  - lines 3-5: $\Theta(V+E)$ by aggregate analysis (chapter 17)
  - 따라서, $\Theta(V+E)$

실습 (3)

- DAG(directed acyclic graph)에서 최단 경로 찾는 코드 작성
  - Memoization 방법 사용
  - Topological sort 방법 사용

Cycle이 존재하는 경우 최단 경로 찾기 (1)

- 예제

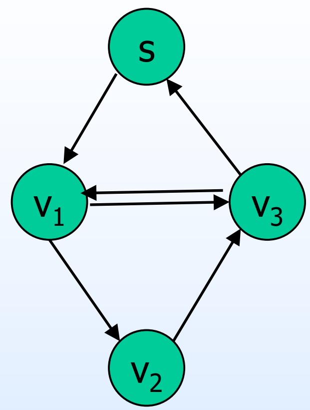


  - $\delta(s, v1) = \min(\delta(s, s) + w(s, v1), \delta(s, v3) + w(v3, v1))$
  - $\delta(s, v2) = \min(\delta(s, v1) + w(v1, v2))$
  - $\delta(s, v3) = \min(\delta(s, v1) + w(v1, v3), \delta(s, v2) + w(v2, v3))$

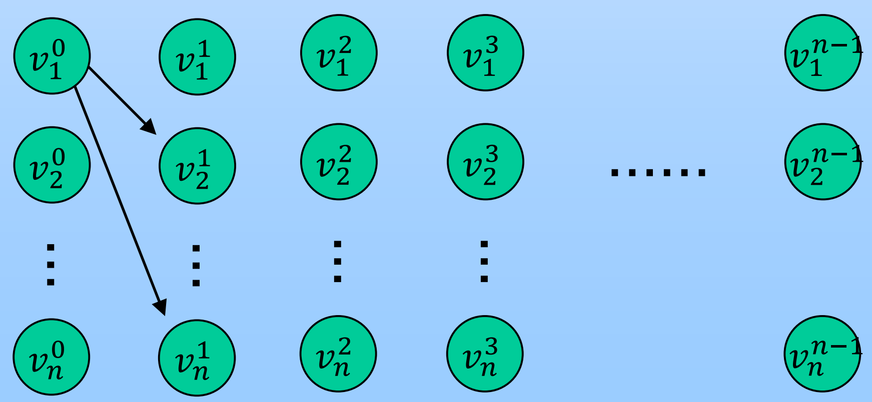
- 문제점
  - Cycle이 존재해서 무한 재귀 호출
  - 종료되지 않음

Cycle이 존재하는 경우 최단 경로 찾기 (2)

- 해결 방법
  - Cycle-break

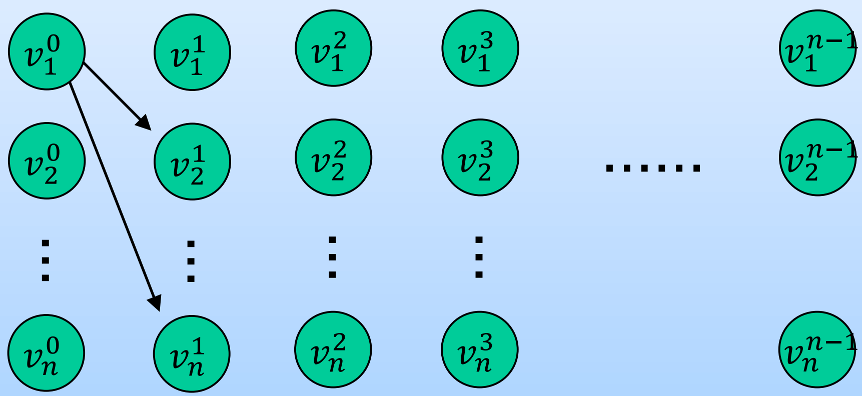


- 아이디어
  - Graph 재구성
  - Vertex 정의 변경
    - v_i를 기반으로 v_i^k 생성. k는 source로부터 v_i까지의 최대 경로 길이.
    - k마다 모든 vertex들로 하나의 layer 구성
  - Edge 정의 변경
    - (v_i, v_j)를 이용해서 (v_i^{k-1}, v_j^k) 생성. k단계의 node와 k+1 단계의 node를 연결



Cycle이 존재하는 경우 최단 경로 찾기 (3)

- 재귀적 정의
  - k단계에 있는 vertex까지의 최단 경로
    - k-1단계에 있는 vertex들을 경유한 경로 중 경로 weight가 최소인 경로
    - $\delta(s, v_i^k) = \min\{\delta(s, v_j^{k-1}) + w(v_j^{k-1}, v_i^k)\}$



  - 초기 상태에서 $s = v_i$인 경우 $v_i^0 = 0$, $v_{j \neq i}^0 = \infty$
- 주어진 문제
  - $\delta(s, v_i^{n-1})$를 구하라.

Cycle이 존재하는 경우 최단 경로 찾기 (3)

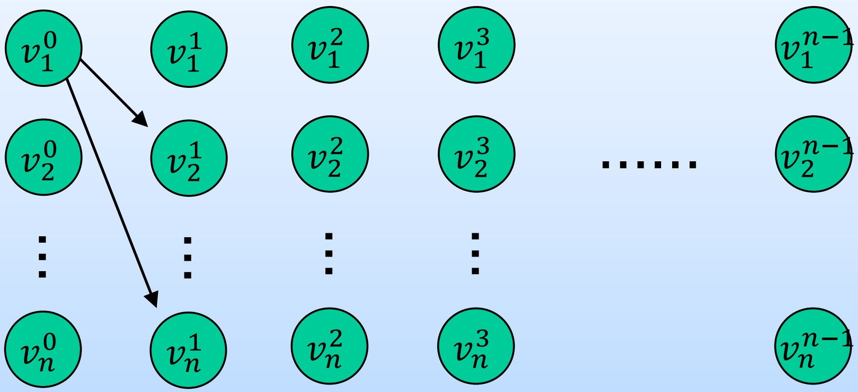
- Memoization 방법
  - 각 node가 하나의 subproblem에 해당함
  - 초기화
    - pathWeight[N][N]을 무한대로 초기화 (혹은 음수)
  - Recursive 함수
    - 종료 조건: 경로길이가 0인 경우
    - Recursive step: 모든 ingoing edge를 고려해서 인접한 vertex를 거치는 경로 weight를 계산하고 그 중 최소값을 최단 경로로 선택
    - 계산이 완료되면 결과를 pathWeight[k][i]에 저장

Cycle이 존재하는 경우 최단 경로 찾기 (3)

- Topological Sort 방법
  - 각 node가 하나의 subproblem
  - N*N개 node들을 topological sort
  - Topological sort 결과의 순서대로 최단 경로 계산

Cycle이 존재하는 경우 최단 경로 찾기 (3)

- Bottom-up의 테이블 채우기 방법
  - pathWeight[N][N] 배열 선언
  - pathWeight[k][i]의 값들을 채워 나감
    - k=0일 때 값 계산
    - k=1일 때 값 계산
    - …
    - k=N-1일 대 값 계산



- Time Complexity
  - $\mathrm{O}(V^2 + VE)$
    - V: vertex 개수
    - E: edge 개수

Bellman-Ford algorithm

실습 (4)

- Cycle이 존재하는 경우에서 주어진 source로부터 최단 경로 찾는 코드 작성
  - Memoization 방법 사용
  - Topological sort 방법 사용
  - Bottom-up method

HW.C2

- 실습 4 완료
  - 세 가지 모두 구현

목 차

- Fibonacci
- Rod-cutting problem
- Shortest Path problems
- Matrix-chain multiplication problem
- Longest common subsequence problem
- Knapsack problem

Matrix-chain 곱셈 (1)

- Input
  - A sequence of <A_1, A_2, …, A_n> of matrix chain
- Output
  - $A_1A_3 \ldots A_n$의 계산 결과
- 문제
  - 숫자의 곱셈 횟수를 최소화하여 계산하는 matrix 곱셈 순서에서 수행되는 숫자의 곱셈 회수를 계산하라.
- 문제 보조 설명
  - Matrix A[p, q]와 matrix B[q, r] 계산의 곱셈 횟수
    - p x q x r
  - 동일한 결과를 내는 다양한 곱셈 순서
    - $(A_1(A_2(A_3A_4)))$
    - $(A_1((A_2A_3)A_4))$
    - $((A_1A_2)(A_3A_4))$
    - $(((A_1A_2)(A_3)A_4)$
- 재귀적 정의

Matrix-chain 곱셈 (2)

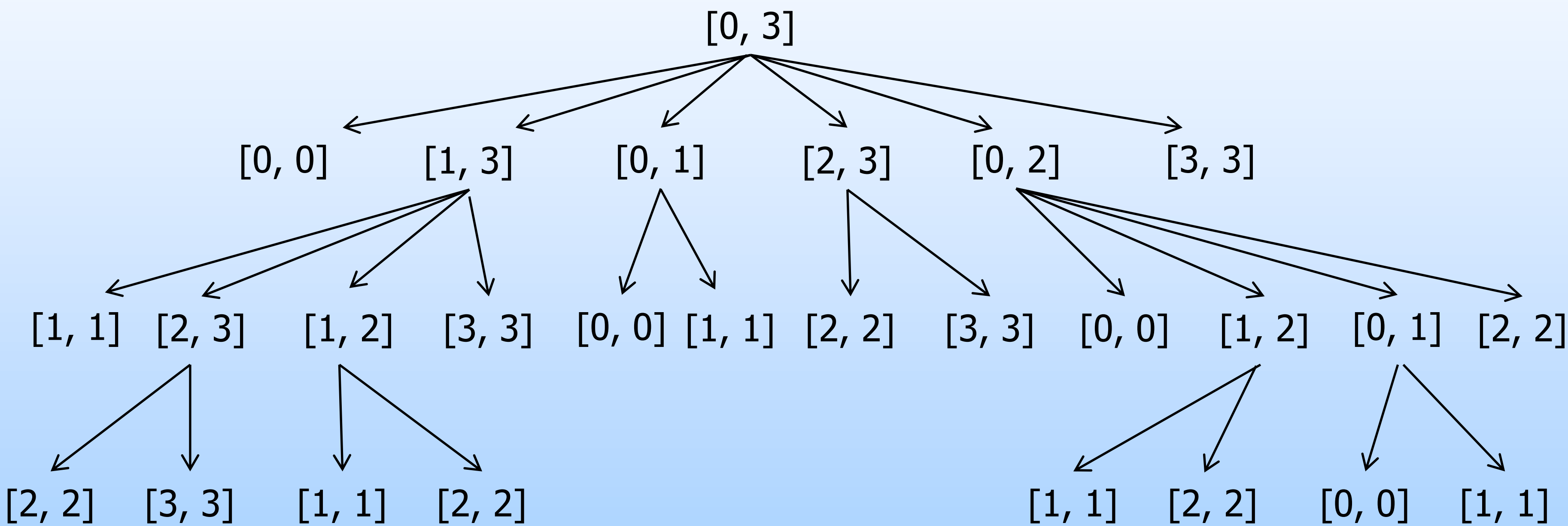
- 재귀적 정의
  - 주어진 matrix-chain을 두 개의 sub-matrix-chain으로 분할
    - 맨 마지막 matrix 곱셈 위치를 기준으로 분할
    - 두 개의 sub-matrix-chain에서 최소 계산 방법 구함
    - Sub-maxtrix-chain의 결과 matrix 두 개를 곱하는 계산 시간 계산
  - 두 개의 sub-matrix-chain으로 나누는 방법 중 계산 시간이 제일 적게 소요되는 경우를 구하자

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \,, \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j \,. \end{cases}$$

- 예제
  - $(A_1)(A_2A_3A_4)$, $(A_1A_2)(A_3A_4)$, $(A_1A_2A_3)(A_4)$ 방법 중 최소 계산 시간 추출

Matrix-chain 곱셈 (3)

- Recursion tree
  - 문제의 크기를 [startId, endId]로 표시
  - Matrix 4개로 구성된 matrix chain의 경우



- 특징
  - Leaf node: startId와 endId가 동일
  - Leaf에서 root로 갈수록 startId와 endId의 차이가 1씩 증가

Matrix-chain 곱셈 (4)

- 각 subproblem의 해가 구해지는 순서
  - Subproblem을 시작 matrix와 end matrix로 기술
    - matrix-chain(startId, endId)로 정의
  - 해가 구해지는 순서
    - 아래 매트릭스에서 원소의 값이 1 → 2 → 3 → 4 → 5 → 6 순서
    - 원소의 값이 같은 경우에는 우선 순위 없음

endId

startId	0	1	2	3	4	5
0	1	2	3	4	5	6
1		1	2	3	4	5
2			1	2	3	4
3				1	2	3
4					1	2
5						1

Matrix-chain 곱셈 (5)

- Recursive code

```
int minCostOfMatrixChain(matrix_t *matrixChain, int firstMatrixId, int lastMatrixId)
{
    int     minCost, cost;

    if(firstMatrixId == lastMatrixId) return 0;

    minCost = minCostOfMatrixChain(matrixChain, firstMatrixId + 1, lastMatrixId)
            + matrixChain[firstMatrixId].rowNum * matrixChain[firstMatrixId].colNum * matrixChain[lastMatrixId].colNum;

    for(int endOfFirstSeq = firstMatrixId + 1; endOfFirstSeq < lastMatrixId; endOfFirstSeq++) {
        cost = minCostOfMatrixChain(matrixChain, firstMatrixId, endOfFirstSeq)
             + minCostOfMatrixChain(matrixChain, endOfFirstSeq + 1, lastMatrixId)
             + matrixChain[firstMatrixId].rowNum * matrixChain[endOfFirstSeq].colNum * matrixChain[lastMatrixId].colNum;

        if(cost < minCost) minCost = cost;
    }

    return minCost;
}
```

- Time complexity

$$T(1) \geq 1 ,$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{for } n > 1 .$$

⇨ $\Omega(2^n)$

Matrix-chain 곱셈 (6)

- Time Complexity 증명

$$T(1) \geq 1,$$

$$T(n) \geq 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1) \quad \text{for } n > 1.$$

$\Rightarrow \Omega(2^n)$

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n.$$

  - Substitution method 사용해서 증명: $T(n) \geq 2^{n-1}$ for all $n \geq 1$.
  - Basis step: $T(1) \geq 1 = 2^0$
  - Inductive step:

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \\ &= 2^n - 2 + n \\ &\geq 2^{n-1}, \end{aligned}$$

Matrix-chain 곱셈 (7)

- Top-down with memoization

```
int matrixChainMemoization(matrix_t *matrixChain, int firstMatrixId, int lastMatrixId, int **resMatrix)
{
    int cost;

    if(resMatrix[firstMatrixId][lastMatrixId] != -1) return resMatrix[firstMatrixId][lastMatrixId];

    if(firstMatrixId == lastMatrixId) {
        resMatrix[firstMatrixId][lastMatrixId] = 0;
        return 0;
    }

    resMatrix[firstMatrixId][lastMatrixId] = matrixChainMemoization(matrixChain, firstMatrixId+1, lastMatrixId, resMatrix)
        + matrixChain[firstMatrixId].rowNum * matrixChain[firstMatrixId].colNum * matrixChain[lastMatrixId].colNum;

    for(int endOfFirst = firstMatrixId+1; endOfFirst < lastMatrixId; endOfFirst++) {
        cost = matrixChainMemoization(matrixChain, firstMatrixId, endOfFirst, resMatrix)
             + matrixChainMemoization(matrixChain, endOfFirst + 1, lastMatrixId, resMatrix)
             + matrixChain[firstMatrixId].rowNum * matrixChain[endOfFirst].colNum * matrixChain[lastMatrixId].colNum;

        if(cost < resMatrix[firstMatrixId][lastMatrixId]) resMatrix[firstMatrixId][lastMatrixId] = cost;
    }

    return resMatrix[firstMatrixId][lastMatrixId];
}
```

- Time complexity: $O(n^3)$

Matrix-chain 곱셈 (8)

- Time complexity 분석
  - 함수 호출 횟수. 즉, 종료 조건 수행 횟수: $O(n^2)$
  - Body 실행 횟수. 즉, resMatrix의 원소 계산 횟수: $O(n^2)$
  - Body 1번 실행할 때마다 곱셈 계산: $O(n)$
  - 총 곱셈 계산 횟수: $O(n^3)$

Matrix-chain 곱셈 (9)

- Bottom-up method
  - Recursion을 사용하지 않음
  - Memoization에서 matrix를 채우는 순서대로 계산
  - Matrix의 원소를 채울 때마다 문제 분할 방법들의 비용 비교 후 최소값을 저장
  - Subproblems 개수: n(n+1)/2 개 because of (n*n – n)/2 + n

endId

startId	0	1	2	3	4	5
0	1	2	3	4	5	6
1		1	2	3	4	5
2			1	2	3	4
3				1	2	3
4					1	2
5						1

Matrix-chain 곱셈 (10)

- Bottom-up pseudo code

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n                  // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```

Matrix-chain 곱셈 (11)

- Bottom-up code

```
int matrixChainBottomUp(matrix_t *matrixChain, int num)
{
    int cost, minCost, eId;
    int resMatrix[num][num];

    for(int sId = 0; sId < num; sId++) resMatrix[sId][sId] = 0;

    for(int diff = 1; diff < num; diff++) {
        for(int sId = 0; sId < num - diff; sId++) {
            eId = sId + diff;

            resMatrix[sId][eId] = resMatrix[sId+1][eId]
                                 + matrixChain[sId].rowNum * matrixChain[sId].colNum * matrixChain[eId].colNum;

            for(int endOfFirst = sId + 1; endOfFirst < eId; endOfFirst++) {
                cost = resMatrix[sId][endOfFirst] + resMatrix[endOfFirst + 1][eId]
                     + matrixChain[sId].rowNum * matrixChain[endOfFirst].colNum * matrixChain[eId].colNum;

                if(cost < resMatrix[sId][eId]) resMatrix[sId][eId] = cost;
            }
        }
    }

    return resMatrix[0][num-1];
}
```

Matrix-chain 곱셈 (12)

- matrix-chain optimal 계산 순서 저장 및 출력
  - 저장 정보
    - 각 subproblem별로 첫 번째 subchain의 맨 마지막 matrix의 위치 값
    - $A_1A_2A_3A_4$를 A_1과 $A_2A_3A_4$로 분할할 경우 cutPos[1][4]=1
  - 저장된 정보를 이용해서 결과 출력
    - Recursion 사용
    - 괄호로 순서 표기. 예: $(A_1(A_2(A_3A_4)))$

분할하는 위치 저장 코드

```c
int matrixChainBottomUpWithCutPos(matrix_t *matrixChain, int **cutPos, int num)
{
    int cost, minCost, eId;
    int resMatrix[num][num];

    for(int sId = 0; sId < num; sId++) resMatrix[sId][sId] = 0;

    for(int diff = 1; diff < num; diff++) {
        for(int sId = 0; sId < num - diff; sId++) {
            eId = sId + diff;

            resMatrix[sId][eId] = resMatrix[sId+1][eId]
                                + matrixChain[sId].rowNum * matrixChain[sId].colNum * matrixChain[eId].colNum;
            cutPos[sId][eId] = sId;

            for(int endOfFirst = sId + 1; endOfFirst < eId; endOfFirst++) {
                cost = resMatrix[sId][endOfFirst] + resMatrix[endOfFirst + 1][eId]
                    + matrixChain[sId].rowNum * matrixChain[endOfFirst].colNum * matrixChain[eId].colNum;

                if(cost < resMatrix[sId][eId]) {
                    resMatrix[sId][eId] = cost;
                    cutPos[sId][eId] = endOfFirst;
                }
            }
        }
    }

    return resMatrix[0][num-1];
}
```

Matrix-chain 곱셈 (14)

- 괄호를 사용한 수식 출력 방법
  - Recursion 사용
  - 저장된 결과 출력
    - 예) $(A_1(A_2(A_3A_4)))$

- 손코딩

Matrix-chain 곱셈 (15)

◆ 괄호를 사용한 수식 출력 코드

```
void    printOptimalParenthesis(int **cutPos, int sId, int eId)
{
        if(sId == eId) cout << "A" << sId;
        else {
                cout << "(";
                printOptimalParenthesis(cutPos, sId, cutPos[sId][eId]);
                printOptimalParenthesis(cutPos, cutPos[sId][eId]+ 1, eId);
                cout << ")";
        }
}
```

Memoization과 Bottom-Up의 비교

- Time Complexity
  - 동일함

- 일반적으로 bottom-up 방법이 memoization보다 효율적이다.

- Memoization의 상대적 단점
  - Recusive call에 의한 오버헤드

- Bottom-up의 상대적 단점
  - 실재로 사용하지 않는 sub-problem의 해도 다 계산해야 함.

Dynamic Programming 구성 요소

- Memoization
  - 동일한 subproblem의 여러 번 발생

- Optimal substructure
  - 주어진 문제의 optimal solution은 subproblem의 optimal solution을 포함하는 특성을 의미
  - 이 조건을 만족하지 못하면 Dynamic Programming 기법을 적용할 수 없음
    - 예: 두 지점 사이의 최대 경로 찾기

HW.C3

- Matrix-chain 곱셈 구현
  - 세 가지 방법 모두 구현
    - Recursion
    - Recursion with memorization
    - Bottom-up
- 마감: 다음 주 수업 시작 전

목 차

- Fibonacci
- Rod-cutting problem
- Shortest Path problems
- Matrix-chain multiplication problem
- Longest common subsequence problem
- Knapsack problem

Longest common subsequence 문제 (1)

- Input
  - 두 개의 sequence X, Y
    - $X_m = <x_1, x_2, \ldots, x_m>$
    - $Y_n = <y_1, y_2, \ldots, y_n>$
- Output
  - X와 Y의 common subsequence $Z_k = <z_1, z_2, \ldots, z_k>$
- 문제
  - 가장 긴 common subsequence Z를 구하라.
- 예제
  - X = <A, B, C, B, D, A, B>
  - Y = <B, D, C, A, B, A>
  - X와 Y의 common subsequence
    - <B, C, A>, <B, C, B, A>, <B, D, A, B>
  - X와 Y의 가장 긴 common subsequence
    - <B, C, B, A> 또는 <B, D, A, B>

Longest common subsequence 문제 (2)

- 재귀적 접근
  - 세 가지 경우 중 한 가지에 가장 긴 공통 subsequence가 있다.
  - $<x_1, x_2, \ldots, x_{m-1}>$과 $<y_1, y_2, \ldots, y_{n-1}>$의 결과
    - $x_m == y_m$ 이면 결과에 x_m 추가
  - $<x_1, x_2, \ldots, x_{m-1}>$과 $<y_1, y_2, \ldots, y_n>$의 결과
  - $<x_1, x_2, \ldots, x_m>$과 $<y_1, y_2, \ldots, y_{n-1}>$의 결과

- 특성 분석
  - $x_m == y_n$인 경우
    - 가장 긴 subsequence는 $<x_1, x_2, \ldots, x_{m-1}>$과 $<y_1, y_2, \ldots, y_{n-1}>$의 가장 긴 공통 subsequence에 x_m이 추가된 subsequence
  - $x_m \mathrel{!}= y_n$이면 아래 두 가지 경우 중 긴 경우
    - $<x_1, x_2, \ldots, x_{m-1}>$과 $<y_1, y_2, \ldots, y_n>$의 가장 긴 공통 subsequence
    - $<x_1, x_2, \ldots, x_m>$과 $<y_1, y_2, \ldots, y_{n-1}>$의 가장 긴 공통 subsequence

Longest common subsequence 문제 (3)

- 특성 증명
  - 가장 긴 subsequence를 Z_k = <z_1, z_2, …, z_k>라고 가정하자.
  - $x_m == y_n$인 경우
    - $x_m == z_k$이어야 한다.
    - 그렇지 않으면 <z_1, z_2, …, z_k, x_m>으로 더 긴 common subsequence가 있음
    - Z가 가장 큰 공통 subsequence라는 가정에 모순
    - 따라서 $x_m == z_k$
  - x_m != y_n인 경우
    - x_m != z_k이면 Z_k가 <x_1, x_2, …, x_{m-1}>과 <y_1, y_2, …, y_n>의 가장 긴 공통 subsequence
    - y_n != z_k이면 Z_k가 <x_1, x_2, …, x_m>과 <y_1, y_2, …, y_{n-1}>의 가장 긴 공통 subsequence

Longest common subsequence 문제 (4)

- Recurrence relation
  - c[i, j]: X_i와 Y_j의 longest common subsequence의 길이

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

- 손코딩

Longest common subsequence 문제 (5)

- Recursive code

```
int lcs(string &str1, int str1SubLen, string &str2, int str2SubLen)
{
	int	lcsLen, subseqLen;

	if(str1SubLen == 0 || str2SubLen == 0) return 0;

	if(str1[str1SubLen] == str2[str2SubLen])
		lcsLen = lcs(str1, str1SubLen - 1, str2, str2SubLen - 1) + 1;
	else
		lcsLen = max(lcs(str1, str1SubLen, str2, str2SubLen - 1),
		             lcs(str1, str1SubLen - 1, str2, str2SubLen));

	return lcsLen;
}
```

Longest common subsequence 문제 (6)

- Top-down with memoization
  - 두 문자열의 길이로 sub-problem을 정의
    - lcs(str1Len, str2Len)
  - Sub-problem의 결과를 저장할 2차원 배열 사용
    - solM[strLen][str2Len]

- 손코딩

Longest common subsequence 문제 (7)

- Code

```
int lcsWithMemo(string &str1, int str1SubLen, string &str2, int str2SubLen, int **solM)
{
    int     lcsLen, subseqLen;

    if(solM[str1SubLen][str2SubLen] != -1) return solM[str1SubLen][str2SubLen];

    if(str1SubLen == 0 || str2SubLen == 0) {
        solM[str1SubLen][str2SubLen] = 0;
        return 0;
    }

    if(str1[str1SubLen] == str2[str2SubLen])
        lcsLen = lcsWithMemo(str1, str1SubLen - 1, str2, str2SubLen - 1, solM) + 1;
    else
        lcsLen = max(lcsWithMemo(str1, str1SubLen, str2, str2SubLen - 1, solM),
                     lcsWithMemo(str1, str1SubLen - 1, str2, str2SubLen, solM));

    solM[str1SubLen][str2SubLen] = lcsLen;

    return lcsLen;
}
```

$$O(nm)$$

Longest common subsequence 문제 (8)

- Bottom-up method
  - Matrix의 원소 값을 채우기 위해서 필요한 정보
    - 행 번호가 1개 작은 원소
    - 열 번호가 1개 작은 원소
    - 행 번호와 열 번호가 1개 작은 원소

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 , \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j , \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j . \end{cases}$$

  - Table 채우는 순서
    - 행 번호 증가순
    - 각 행에서는 열 번호 증가순

j		0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Longest common subsequence 문제 (9)

- Bottom-up Dynamic Programming
  - 손코딩

$O(nm)$

```
int lcsWithBottomup(string &str1, string &str2)
{
	int		solM[str1.size() + 1][str2.size() + 1];

	for(int str1SubLen = 0; str1SubLen <= str1.size(); str1SubLen++)
		solM[str1SubLen][0] = 0;
	for(int str2SubLen = 0; str2SubLen <= str2.size(); str2SubLen++)
		solM[0][str2SubLen] = 0;

	for(int str1SubLen = 1; str1SubLen <= str1.size(); str1SubLen++) {
		for(int str2SubLen = 1; str2SubLen <= str2.size(); str2SubLen++) {
			if(str1[str1SubLen-1] == str2[str2SubLen-1])
				solM[str1SubLen][str2SubLen] = solM[str1SubLen - 1][str2SubLen - 1] + 1;
			else
				solM[str1SubLen][str2SubLen] = max(solM[str1SubLen - 1][str2SubLen],
									solM[str1SubLen][str2SubLen - 1]);
		}
	}

	return solM[str1.size()][str2.size()];
}
```

Longest common subsequence 문제 (8)

- 결과 저장 및 추출
  - 각 subproblem별로 자신이 호출하는 subproblem의 정보를 저장
  - 저장된 과정을 이용해서 결과 출력
    - x_i == y_j인 경우에 문자 출력
    - Recursion으로 처리

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

- 손코딩

```cpp
int lcsWithBottomupWithResult(string &str1, string &str2, int **seqM)
{
    int     solM[str1.size() + 1][str2.size() + 1];

    for(int str1SubLen = 0; str1SubLen <= str1.size(); str1SubLen++)
        solM[str1SubLen][0] = 0;
    for(int str2SubLen = 0; str2SubLen <= str2.size(); str2SubLen++)
        solM[0][str2SubLen] = 0;

    for(int str1SubLen = 1; str1SubLen <= str1.size(); str1SubLen++) {
        for(int str2SubLen = 1; str2SubLen <= str2.size(); str2SubLen++) {
            if(str1[str1SubLen] == str2[str2SubLen]) {
                solM[str1SubLen][str2SubLen] = solM[str1SubLen - 1][str2SubLen - 1] + 1;
                seqM[str1SubLen][str2SubLen] = BOTH;
            }
            else {
                if(solM[str1SubLen - 1][str2SubLen] > solM[str1SubLen][str2SubLen - 1]) {
                    solM[str1SubLen][str2SubLen] = solM[str1SubLen - 1][str2SubLen];
                    seqM[str1SubLen][str2SubLen] = STR1;
                }
                else {
                    solM[str1SubLen][str2SubLen] = solM[str1SubLen][str2SubLen - 1];
                    seqM[str1SubLen][str2SubLen] = STR2;
                }
            }
        }
    }

    return solM[str1.size()][str2.size()];
}
```

```
void printLCS(string &str1, int str1Len, string &str2, int str2Len, int **seqM)
{
    if(str1Len == 0 || str2Len == 0) return;

    switch(seqM[str1Len][str2Len]) {
        case BOTH:
            printLCS(str1, str1Len - 1, str2, str2Len - 1, seqM);
            cout << str1[str1Len - 1];
            break;
        case STR1:
            printLCS(str1, str1Len - 1, str2, str2Len, seqM);
            break;
        case STR2:
            printLCS(str1, str1Len, str2, str2Len - 1, seqM);
            break;
    }
}
```

```
    printLCS(str1, str1.size(), str2, str2.size(), seqM);
    cout << endl;
```

HW.C4

- Longest common subsequence 구현
  - 세 가지 방법 모두 구현
    - Recursion
    - Recursion with memorization
    - Bottom-up

- 마감: 다음 주 수업 시작 전

목 차

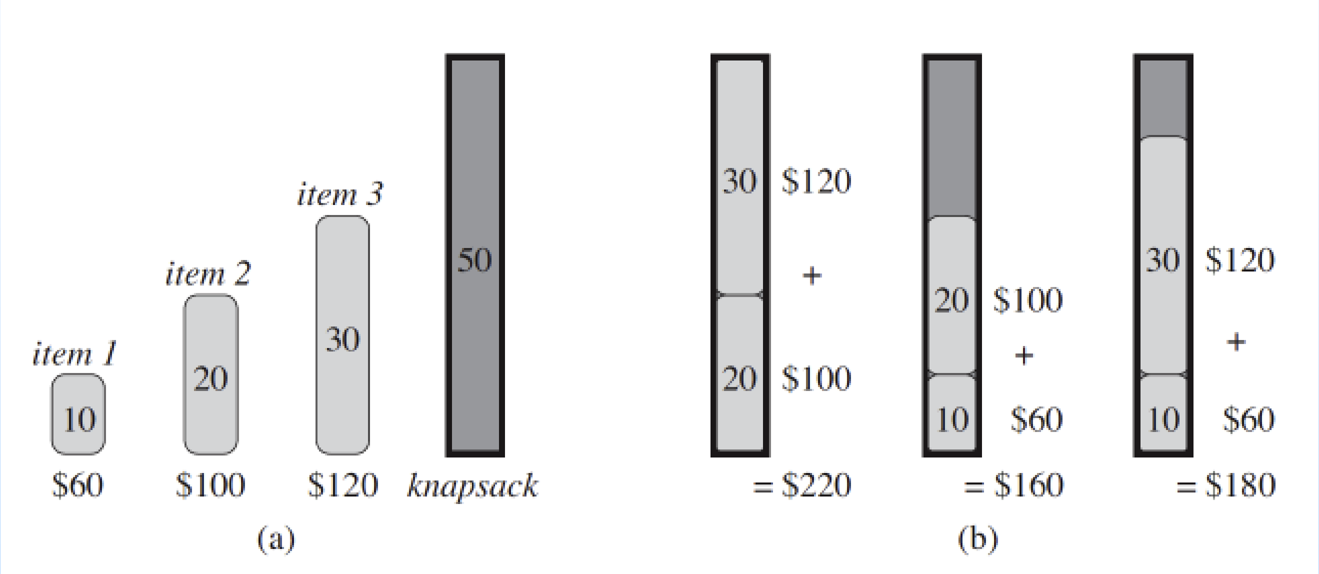
- Fibonacci
- Rod-cutting problem
- Shortest Path problems
- Matrix-chain multiplication problem
- Longest common subsequence problem
- Knapsack problem

Knapsack problem

- 상황 설명
  - 보물섬에서 보물을 찾았다.
  - 가지고 있는 배낭에 보물을 넣어서 나가려고 한다.
  - 배낭에는 W Kg까지 넣을 수 있다. 그 이상 넣으면 찢어져서 사용할 수가 없다.
  - 찾은 보물은 개수는 n개이다.
  - 보물 i의 가치는 v_i 원이다.
  - 보물 i의 무게는 w_i Kg이다.
- 0/1 knapsack problem
  - 물건을 분할해서 knapsack에 넣을 수 없다.
  - 가치가 최대가 되도록 knapsack에 채우는 방법을 제시하시오.
- Fractional knapsack problem
  - 물건을 분할해서 넣을 수 있다. 예를 들어, 금 가루인 경우 일정 무게만 넣을 수 있다.
  - 가치가 최대가 되도록 knapsack에 채우는 방법을 제시하시오.

0/1 Knapsack Problem (1)

- 아이디어
  - 가치/무게 비율이 큰 것부터 채우면 되지 않을까?
    - 안됨
    - 반례: 가치/무게 비율이 제일 큰 item 1을 채우면 최대가 안됨.



  - 각 item별로 넣을 때와 넣지 않을 때를 살펴보고 최대값을 찾아야 함.

0/1 Knapsack Problem (2)

- Recursive solution
  - Power set 출력하는 것과 유사한 방법
  - 입력
    - 아이템 배열(가치, 무게), 개수, knapsack에 넣을 수 있는 최대 용량
  - 출력
    - 넣을 수 있는 최대 가치
  - 초기 조건
    - Item이 0개일 때
    - Item이 1개일 때
  - Recursion
    - max(item이 들어가는 경우의 value, 들어가지 않는 경우의 value)

- 손코딩

0/1 Knapsack Problem (3)

- Recurrence Relation
  - 주어진 문제: $V(k, W)$
    - k는 subproblem에서 다루는 보물의 번호 최대값에 해당함.
    - 즉, V_k = $\{k | 1 \leq i \leq k \leq n$, k는 보물 번호, n은 총 보물 개수 $\}$
  - 아이디어
    - n번째 보물을 넣었을 때와 넣지 않았을 때로 구분
    - n번째 보물이 W보다 무거우면 넣을 수 없음
  - $w_n \leq W$인 경우
    - $V(n, W) = \max\{V(n-1, W), V(n-1, W-w_n) + v_n\}$
  - $w_n > W$인 경우
    - $V(n, W) = V(n-1, W)$

0/1 Knapsack Problem (4)

- Recursive solution code

```c
typedef struct item {
        int       weight;
        int       value;
} item_t;
```

```c
int knapsack01Recursion(item_t *items, int num, int capacity)
{
    if(num <= 0 || capacity <= 0) return 0;

    if(items[num - 1].weight > capacity)
        return knapsack01Recursion(items, num - 1, capacity);
    else
        return max( knapsack01Recursion(items, num - 1, capacity),
                    knapsack01Recursion(items, num - 1, capacity - items[num - 1].weight)
                    + items[num - 1].value);
}
```

0/1 Knapsack Problem (5)

- 질문 1
  - Dynamic Programming 전략을 적용할 수 있을까?
  - 적용하려면 어떤 조건이 필요할까?

- 답변
  - 보물의 무게가 정수라고 가정하면 적용 가능

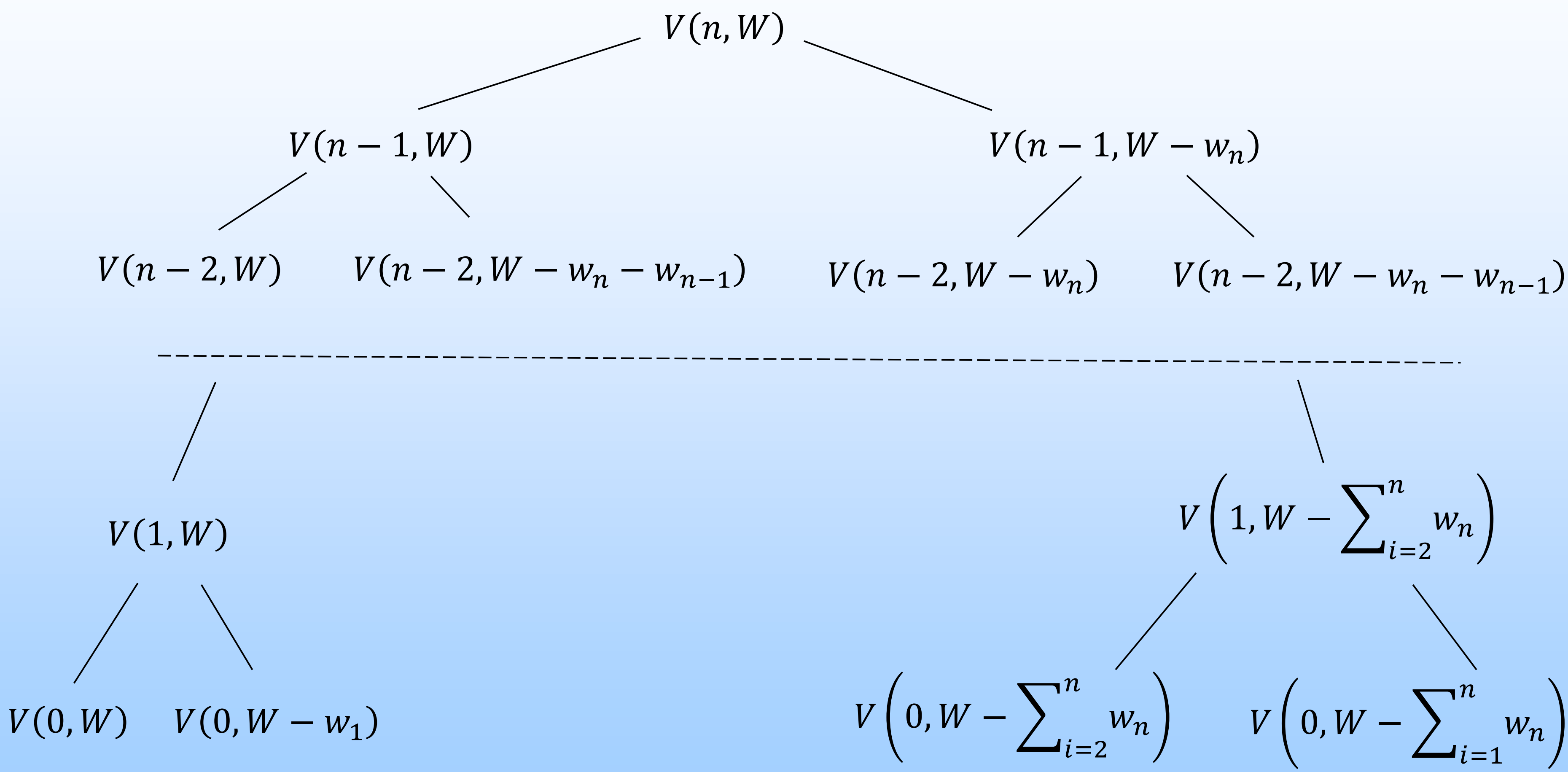
0/1 Knapsack Problem (6)

- 질문 2
  - 보물의 무게가 정수라고 가정하자.
  - Subproblem을 정의하라.

- 답변
  - $maxValue(n, W)$
    - 보물 집합 $V_n = \{k|1 \leq k \leq n,$ 여기서 k는 보물 $index\}$
    - W = knapsack에 넣을 수 있는 무게 최대값
    - 집합 V에 포함되는 보물들을 W 무게까지 넣을 수 있는 knapsack게 넣을 때 가치가 최대가 되도록 넣는 방법을 구하라.

0/1 Knapsack Problem (7)

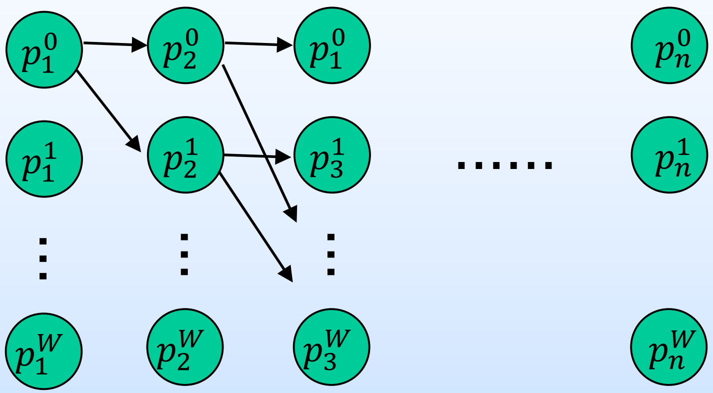
- Recursion Tree



subproblem에 주어진 W가 음수가 되면 더 이상 아래로 내려가지 않음

0/1 Knapsack Problem (7)

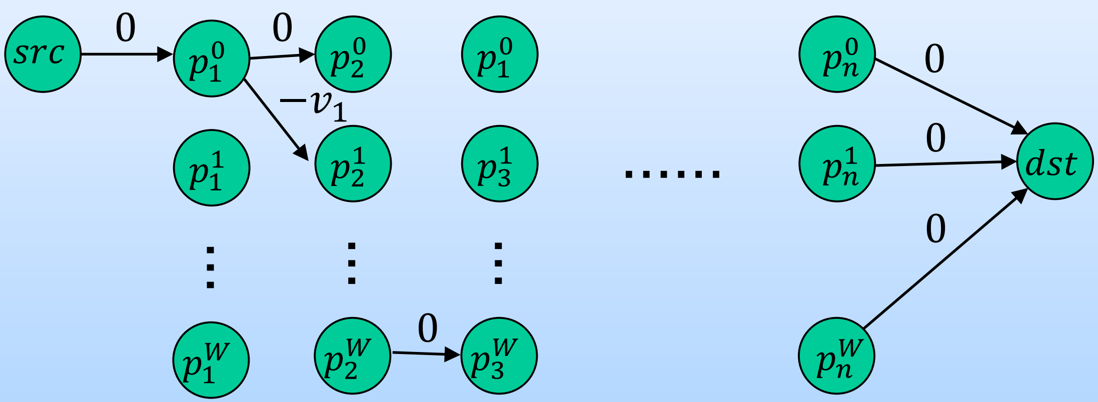
- Subproblem의 답을 구하는 순서
  - 할당된 번호가 작은 보물부터 선택
    - 보물 1 선택 여부 결정
    - 보물 2 선택 여부 결정
    - …
    - 보물 n 선택 여부 결정



- p_k^w에서 w는 knapsack에 넣을 수 있는 weight, k는 선택 여부를 결정할 보물 번호를 의미
- 1~k-1까지는 이미 선택 여부를 결정한 상태임.
  - 보물 선택 시 knapsack이 꽉 차면 knapsack에 넣을 수 없음

0/1 Knapsack Problem (8)

- Topological sort로 구하는 방법
  - 가상의 src와 dst 추가
  - 보물의 value가 v이면 edge의 weight를 –v로 지정
  - src로부터 dst까지의 최단 경로를 계산
  - 보물을 knapsack에 담지 않는 경우는 edge의 weight를 0으로 지정



0/1 Knapsack Problem (9)

- Table 채우는 방법 (Bottom-up 방법)
  - 보물 번호: 1 ~ n
  - 열 번호 $1 \le k \le n$ 의 의미: 0번부터
  - 행 번호: knapsack 용량 (0~W)

	0	1	2	…	n
0					
1					
2					
…					
W-1					
W					

$$if\ w_n \le W$$

$$V(n, W) = \max\begin{Bmatrix} V(n-1, W) \\ V(n-1, W-w_n) + v_n \end{Bmatrix}$$

$$else\ \ V(n, W) = V(n-1, W)$$

  - 코드

```
for(w=0; w <= W; w++) p[0][w] = 0; // 첫 번째 행 초기화
for(k=1; k <= n; k++)
    for(w=0; w <=W; w++)
        ......
```

0/1 Knapsack Problem (10)

- 0/1 knapsack problem의 time complexity
  - $O(nW)$