최적화 문제와 동적계획법

주요 내용

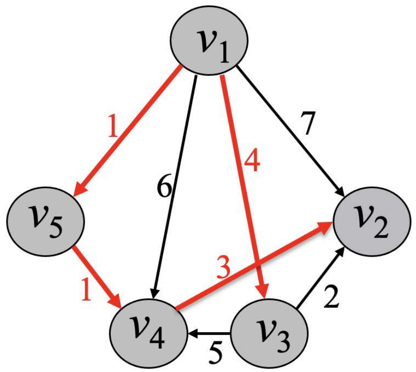
- 최적화 문제
- 메모이제이션
- 동적계획법
- 잔돈 지불 문제
- 이항 계수

최적화 문제

- 여러 개의 해답 중에서 주어진 조건을 만족하는 최적의 해답을 찾는 문제
- 최적의 기준: 특정 기준에 맞는 최댓값 또는 최솟값

예제: 두 지점 사이의 최단 경로 찾기 문제

- v_1에서 다른 지점으로 이동하는 가장 짧은 경로 찾기
- 숫자는 두 지점 사이의 경로의 길이



잔돈 지불 문제

- 63원을 지불하기 위해 필요한 최소한의 동전 수 계산하기
- 1원, 5원, 10원, 25원짜리 동전만 이용

기법 1: 탐욕 기법

- 정해진 기준에 따라 **매 선택 순간에 가장 좋은 것**을 선택하는 기법

- 잔돈 지불 문제의 경우: 가능한 가장 큰 단위의 동전 먼저 사용: 동전 최소 6개 필요
    - 25원 동전: 2개
    - 10원 동전: 1개
    - 1원 동전: 3개

- 탐욕 알고리즘이 항상 최선의 해답을 제공하지는 않음
    - 잔돈 30원 지불 방법: 10원 동전 3개
    - 탐욕 기법: 25원 동전 1나와 1원 동전 5개, 총 6개 필요

기법 2: 완전 탐색

- 가능한 모든 경우를 고려하는 기법
- **부르트 포스**brute force 기법이라고도 불림

$$
\text{동전개수} = min \begin{cases} 1 + \text{동전개수}(\text{지불액} - 1) & (1\text{원 동전을 최소 하나 사용한 경우}) \\ 1 + \text{동전개수}(\text{지불액} - 5) & (5\text{원 동전을 최소 하나 사용한 경우}) \\ 1 + \text{동전개수}(\text{지불액} - 10) & (10\text{원 동전을 최소 하나 사용한 경우}) \\ 1 + \text{동전개수}(\text{지불액} - 25) & (25\text{원 동전을 최소 하나 사용한 경우}) \end{cases}
$$

In [8]:

```
def make_change_1(coin_value_list, change):
    min_coins = change                              # 최소 동전 개수 초기화

    if change in coin_value_list:                   # 종료 조건
        return 1
    else:
        # 하나의 동전을 사용한 각각의 경우: 재귀 적용
        available_coins = [c for c in coin_value_list if c <= change] # 사용 가
        for i in available_coins:
            num_coins = 1 + make_change_1(coin_value_list, change - i)

            if num_coins < min_coins:                # 최소 동전 수 업데이트
                min_coins = num_coins

    return min_coins
```

In [9]:

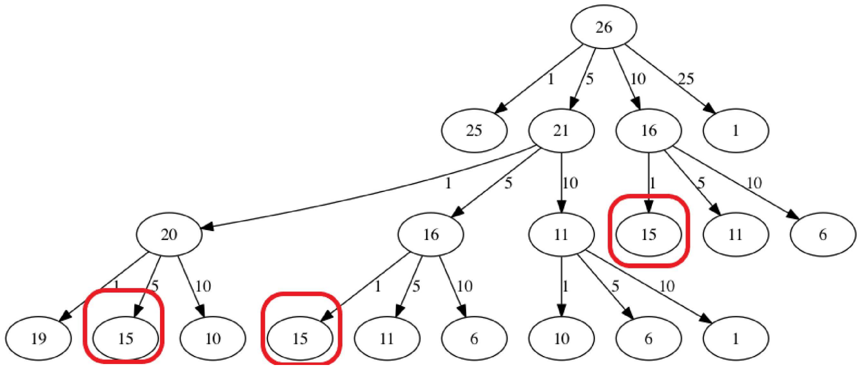
```
make_change_1([1, 5, 10, 25], 63)
```

Out[9]:

```
6
```

재귀 알고리즘의 기본 문제

- 재귀 알고리즘의 실행에 많은 시간 소요
- `make_change_1([1, 5, 10, 25], 63)` 실행: 67,716,925 번의 재귀 호출 발생
- 아래 그림: `make_change_1([1, 5, 10, 25], 63)`를 실행할 때 `make_change_1([1, 5, 10, 25], 63)`가 최소 3번 이상 호출됨을 보여줌. 실제로는 52번 호출됨.



기법 3: 메모이제이션

- 작은 입력크기에 대한 반환값을 기억해두고 필요한 경우 재활용하는 기법
- 여기서는 **디폴트 딕트**(`defaultdict`) 활용

In [3]:
```
from collections import defaultdict

aDict = defaultdict(int)
aDict[10]
```

Out[3]:
```
0
```

- `make_change_2()` 함수: 메모이제이션 기법 활용
- `make_change_2([1, 5, 10, 25], 63)`를 실행 할 때의 재귀 호출 횟수: 206회

In [6]:
```
def make_change_2(coin_value_list, change, known_results=defaultdict(int)):
    min_coins = change # 가장 큰 값으로 시작

    if change in coin_value_list:
        known_results[change] = 1
        return 1

    # 기존에 호출되었다면 저장된 값 재활용
    # 키의 값이 0보다 크면, 이미 계산된 경우를 가리킴
    elif known_results[change] > 0:
        return known_results[change]

    else:
        available_coins = [c for c in coin_value_list if c <= change] # 사용 가능
        for i in available_coins:
            num_coins = 1 + make_change_2(coin_value_list, change - i, known_resu
            if num_coins < min_coins:
                min_coins = num_coins
            known_results[change] = min_coins
    return min_coins
```

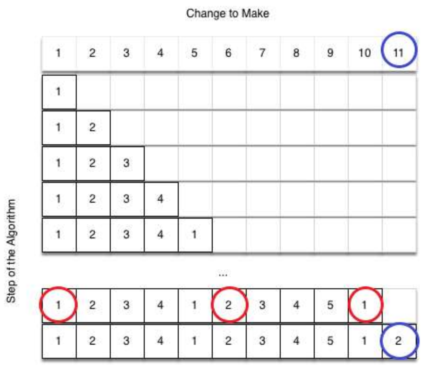
기법 4: 동적계획법

- 재귀를 사용하지 않는 대신 재귀 알고리즘의 종료조건에서 출발하여 차례대로 필요한 인자에 해당하는 값까지 쌓아가는 기법
- 잔돈 63원을 지불해야 하는 경우: 1원부터 출발해서 63원까지 각각의 경우에 필요한 최소 동전 수 계산
- 메모이제이션 기법을 거꾸로 적용하는 것과 유사
- 1원, 2원, 3원 등부터 63원까지 **모든 경우**에 대해 차례대로 필요한 최소 동전 수를 저장하여 재활용

예제

11원을 지불하고자 하는 경우 아래 세 경우를 확인한 다음에 최솟값을 선택

- 1원 동전 사용: 나머지 10원을 지불할 때 필요한 최소 동전 수에 1을 더한 값
- 5원 동전 사용: 나머지 6원을 지불할 때 필요한 최소 동전 수에 1을 더한 값
- 10원 동전 사용: 나머지 1원을 지불할 때 필요한 최소 동전 수에 1을 더한 값



In [12]:

```python
from collections import defaultdict

def make_change_3(coin_value_list, change):
    min_coins = defaultdict(int)

    # 1원부터 차례대로 최소 동전 수 계산
    for changeToMake in range(1, change + 1):
        coin_count = changeToMake # 가장 큰 값으로 시작

        available_coins = [c for c in coin_value_list if c <= changeToMake] # 사
        for j in available_coins:
            if min_coins[changeToMake - j] + 1 < coin_count:
                coin_count = min_coins[changeToMake - j] + 1

        min_coins[changeToMake] = coin_count

    # 최종적으로 계산된 값 반환
    return min_coins[change]
```

잔돈 지불 방법 출력

In [14]:
```
from collections import defaultdict

def make_change_4(coin_value_list, change):
    min_coins = defaultdict(int)
    coins_used = defaultdict(int)  # 특정 액수의 잔돈을 지불할 때 사용되는 마지막

    # 1원부터 차례대로 최소 동전 수 계산
    for changeToMake in range(1, change + 1):
        coin_count = changeToMake # 가장 큰 값으로 시작
        new_coin = 1              # 마지막으로 사용된 코인 저장 용도. 1원부터 시작

        available_coins = [c for c in coin_value_list if c <= changeToMake] # 사
        for j in available_coins:
            # 하나의 동전을 사용한 각각의 경우: 이전에 계산되어 저장된 값 활용
            if min_coins[changeToMake - j] + 1 < coin_count:
                coin_count = min_coins[changeToMake - j] + 1
                new_coin = j

        min_coins[changeToMake] = coin_count
        coins_used[changeToMake] = new_coin    # changeToMake 를 지불할 때 사용되

    # 최종적으로 계산된 값 반환
    return min_coins[change], coins_used
```

In [15]:
```python
def print_coins(coins_used, change):
    coin = change
    while coin > 0:
        this_coin = coins_used[coin]
        print(this_coin, end=" ")
        coin = coin - this_coin
    print()
```

In [16]:
```python
amount = 63
coin_list = [1, 5, 10, 25]

num_coins, coins_used = make_change_4(coin_list, amount)

print(f"잔돈 {amount} 센트를 지불하기 위해 다음 {num_coins} 개의 동전 필요:", enc
print_coins(coins_used, amount)
```

```
잔돈 63 센트를 지불하기 위해 다음 6 개의 동전 필요: 1 1 1 10 25 25
```

이항 계수

아래 다항 등식에서 사용되는 계수 $\binom{n}{k}$를 **이항 계수**binomial coefficients라 한다.

$$\begin{aligned}(a+b)^n &= a^n + \binom{n}{1}a^{n-1}b + \binom{n}{2}a^{n-2}b^2 + \cdots + \binom{n}{n-1}ab^{n-1} + b^n \\ &= \sum_{k=0}^{n}\binom{n}{k}a^{n-k}b^k\end{aligned}$$

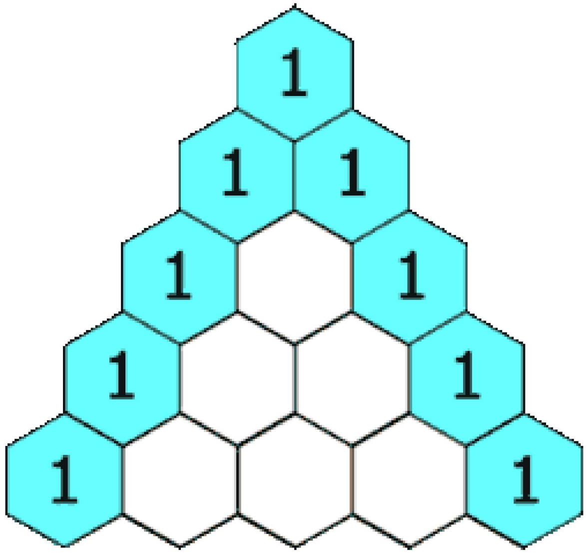
$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

조합

서로 다른 n 개의 구슬에서 임의로 서로 다른 k개의 구슬을 선택하는 방법을 의미하기도 한다.

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

파스칼의 삼각형



이항 계수 계산: 재귀 활용

In [22]:
```python
def bin_coeff(n, k):
    # 종료조건
    if k == 0 or k == n:
        return 1
    else: # 재귀
        return bin_coeff(n-1, k-1) + bin_coeff(n-1, k)
```

- `bin_coeff(n, k)` 계산을 위한 `bin_coeff()` 함수 재귀 호출 횟수

$$2\binom{n}{k} - 1$$

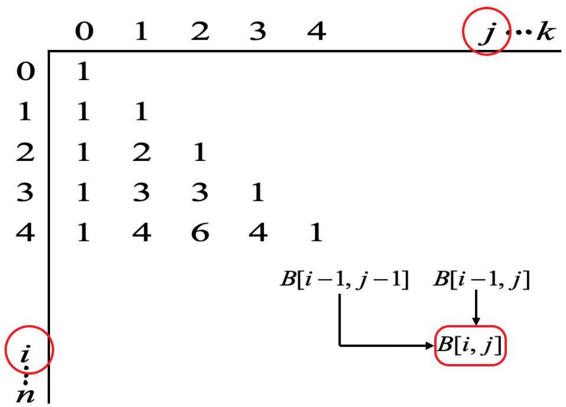
- 기본적으로 지수함수 정도의 나쁜 시간복잡도

$$\binom{n}{k} \approx \frac{n^k}{k!}$$

이항 계수 계산: 동적계획법 적용

2차원 행렬 B의 항목을 파스칼의 삼각형이 만들어지는 과정과 동일한 방식으로 채움.

- 숫자가 표시되지 않는 영역은 모두 0으로 채워졌다고 가정
- $B[0][0]$ 에서 시작
- 위에서 아래로 재귀 관계식을 적용하여 파스칼의 삼각형을 완성해 나감



In [19]:
```python
def bin_coeff2(n, k):
    # (n, k) 모양의 2차원 행렬 준비. 리스트 조건제시법 활용

    B = [[0 for _ in range(k+1)] for _ in range(n+1)]

    # 동적계획법으로 행렬 대각선 이하 부분 채워나가기
    for i in range(n+1):
        for j in range(min(i, k) + 1):
            if j == 0 or j == i:
                B[i][j] = 1
            else:
                B[i][j] = B[i-1][j-1] + B[i-1][j]

    return B[n][k]
```

`bin_coeff2(n, k)` 함수의 시간복잡도

- 입력 크기: n과 k
- 계산단위: `j` 변수에 대한 `for` 반복문 실행횟수

i 값	반복횟수
0	1
1	2
2	3
...	...
k-1	k
k	k+1
k+1	k+1
...	...
n	k+1

$$
\begin{aligned}
T(n,k) &= 1 + 2 + 3 + \cdots + k + (k+1)\cdot(n-k+1) \\
&= \frac{(2n-k+2)(k+1)}{2} \\
&\in O(n\,k)
\end{aligned}
$$