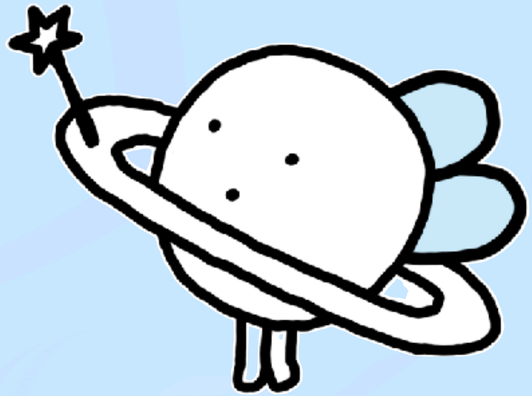
활동 추천 알고리즘 개선하기

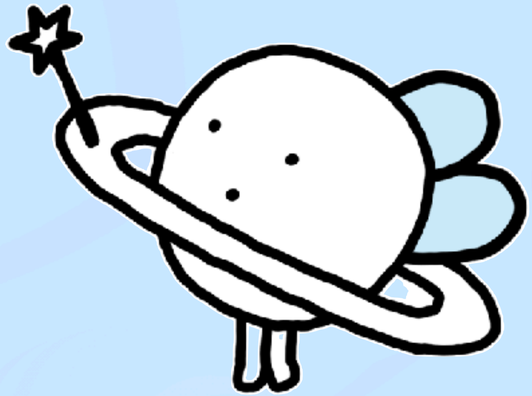
밍트

개요



1. 문제 정의
2. 알고리즘 비교 분석
3. 실제 구현 코드
4. 개선 방향

개요



1. 문제 정의
2. 알고리즘 비교 분석
3. 실제 구현 코드
4. 개선 방향

비즈니스 요구사항

사용자에게 제한 시간 안에서

개인 우선순위에 맞는 최적의 활동 조합 추천하기

1. 문제 정의

비즈니스 요구사항

사용자에게 제한 시간 안에서

개인 우선순위에 맞는 최적의 활동 조합 추천하기

제한 시간: 100분

활동명	시간	점수 (우선순위)	효율 (가치/시간)
A	10	11	1.1
B	50	50	1.0
C	50	50	1.0
D	95	99	1.04

1. 문제 정의

비즈니스 요구사항

사용자에게 제한 시간 안에서

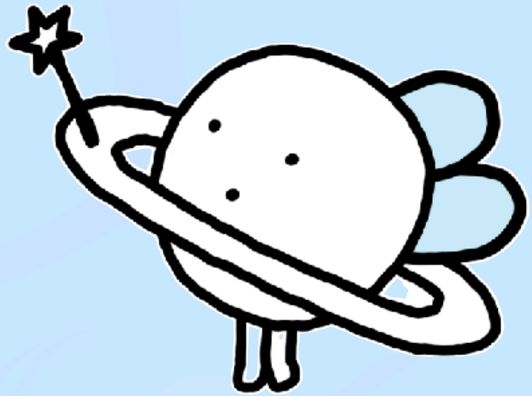
개인 우선순위에 맞는 최적의 활동 조합 추천하기

Greedy

Brute Force

DP

개요

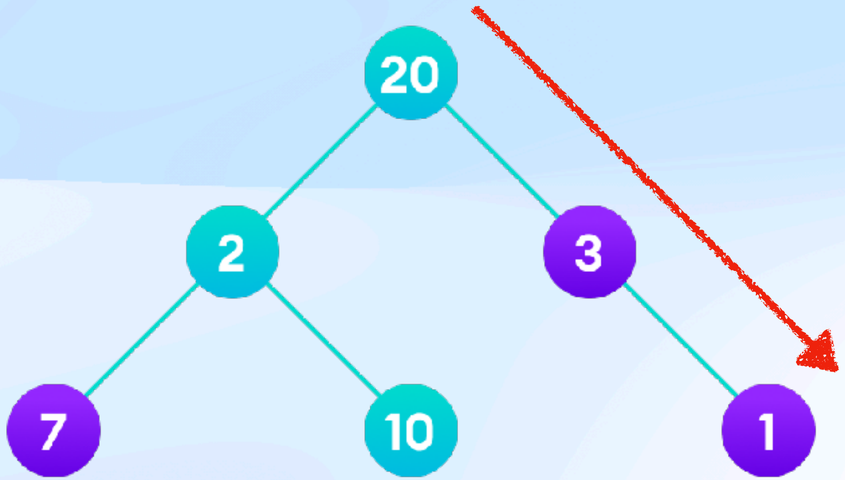


1. 문제 정의
2. 알고리즘 비교 분석
3. 실제 구현 코드
4. 개선 방향

2. 알고리즘 비교 분석

1. Greedy

현재 시점에서 가장 좋아 보이는 선택을 반복하는 방법



- when : 현재의 최선이 전체 최선으로 이어지는 경우
- why: 구현이 간단하고 직관적임
- 시간 복잡도 : O(n logn) ← 정렬
- 공간 복잡도 : O(n)

2. 알고리즘 비교 분석

1. Greedy

점수 순으로 추천 활동 선택하기

제한 시간: 100분

활동명	시간	점수 (우선순위)	효율 (가치/시간)
A	10	11	1.1
B	50	50	1.0
C	50	50	1.0
D	95	99	1.04

1. D 선택 (95분, 99점)
   → 남은 시간: 5분
2. 그 후 선택 불가

선택: D
시간 : 95분
점수 : 99점

← 점수가 높지만, 시간 고려 X

2. 알고리즘 비교 분석

1. Greedy

효율 순으로 추천 활동 선택하기

제한 시간: 100분

활동명	시간	점수 (우선순위)	효율 (가치/시간)
A	10	11	1.1
B	50	50	1.0
C	50	50	1.0
D	95	99	1.04

1. A 선택 (10분, 11점)
   → 남은 시간: 90분
2. D 선택 불가 (시간 부족)
3. C 선택 (50분, 50점)
   → 남은 시간: 40분
4. B 선택 불가 (시간 부족)

선택: A, C
시간 : 60분
점수 : 61점

효율이 높은 것을 선택해도
높은 점수를 보장하지 않음

2. 알고리즘 비교 분석

1. Greedy

실패 원인 : “ 가장 점수 / 효율이 높은 것부터 선택 “

제한 시간: 100분

활동명	시간	점수 (우선순위)	효율 (가치/시간)
A	10	11	1.1
B	50	50	1.0
C	50	50	1.0
D	95	99	1.04

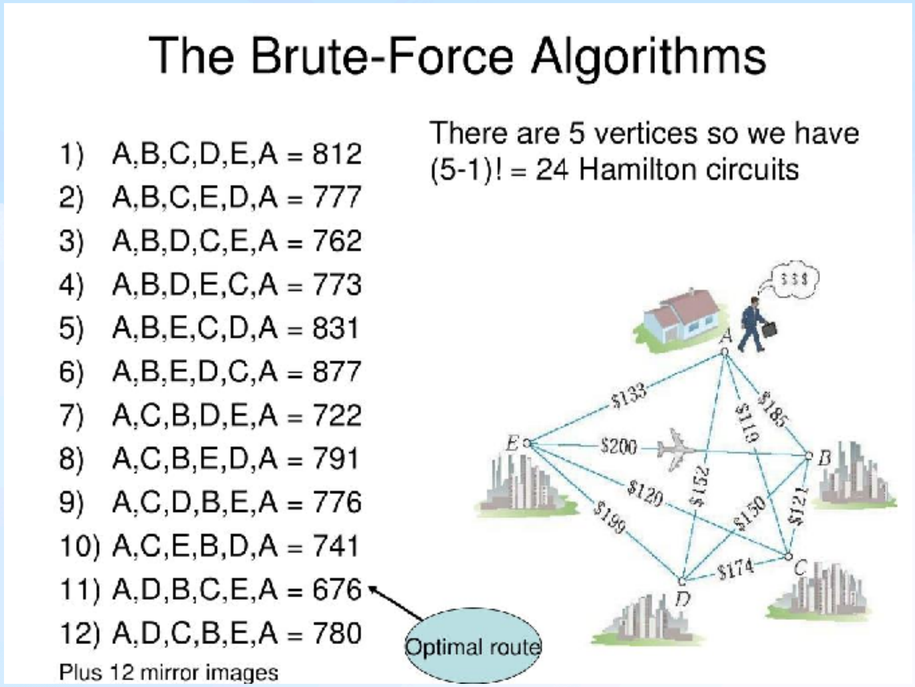
선택: B, C
시간 : 100분
점수 : 100점

큰 것 하나 < 작은 것 여러개

작은 차이에 집착하여
큰 그림을 보지 못함

2. Brute Force

가능한 모든 경우의 수를 직접 다 확인하는 방법



- when : 경우의 수가 적고, 정확성이 최우선일때
- why: 정답을 100% 보장받을 수 있음
- 시간 복잡도 : O(2^n) ← n <= 20 (100ms)
- 공간 복잡도: O(n)

https://www.slideserve.com/cecily/6-4-the-brute-force-algorithms

2. 알고리즘 비교 분석

2. Brute Force

가능한 모든 경우의 수를 직접 다 확인하는 방법

제한 시간: 100분

활동명	시간	점수 (우선순위)	효율 (가치/시간)
A	10	11	1.1
B	50	50	1.0
C	50	50	1.0
D	95	99	1.04

모든 조합

1. {} → 0분, 0점
2. {A} → 10분, 11점
3. {B} → 50분, 50점
4. {C} → 50분, 50점
5. {D} → 95분, 99점
6. {A,B} → 60분, 61점
7. {A,C} → 60분, 61점
8. {B,C} → 100분, 100점 ✅

선택: B, C
시간 : 100분
점수 : 100점

최적의 조합 선택!

2. 알고리즘 비교 분석

2. Brute Force

한계 : 활동이 많아지면 사용 불가

```
for (int mask = 0; mask < (1 << n); mask++) { // O(2^n)
    for (int i = 0; i < n; i++) {  // 활동별 가치 계산
    }
}
```

시간 복잡도: O(n * 2^n)

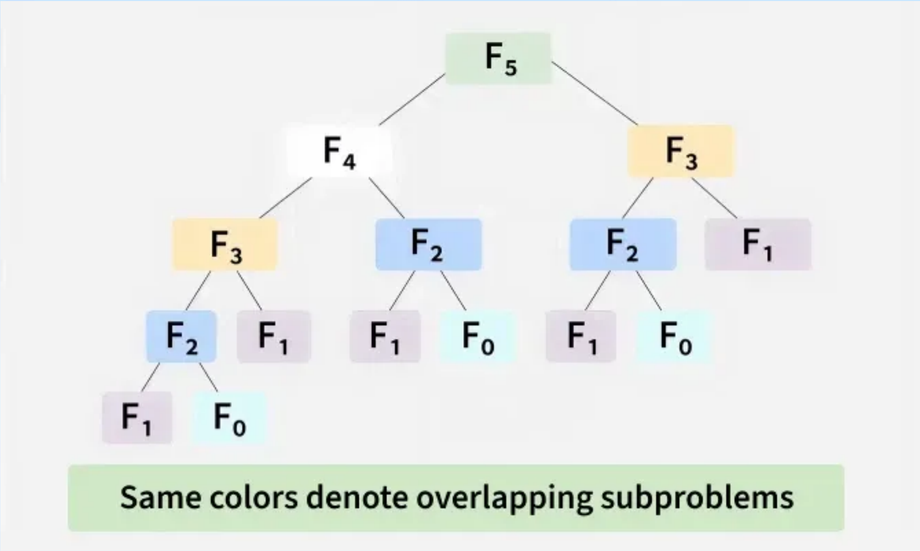
가정) 1초에 10억 연산

n	2^n	n * 2^n	예상 시간
10	1,024	10,240	0.00016ms
20	1,048,576	20,971,520	21ms
25	33,554,432	838,860,800	839ms
30	1,073,741,824	32,212,254,720	32초

n>=25부터 1초 초과 가능

3. DP (Dynamic Programming)

큰 문제를 작은 문제로 나누어 해결하고,
그 결과를 저장하여 재사용하는 방법



- when : 같은 부분 문제가 반복적으로 등장할 때
- why: 중복 계산을 제거하여 시간 단축
- 시간 복잡도 : dp 테이블 칸 개수 * 각 칸 계산 시간
- 공간 복잡도 : dp 테이블이 차지하는 메모리 크기

2. 알고리즘 비교 분석

3. DP (Dynamic Programming)

dp[i][w] = i 개 활동을 보고, w분 사용했을 때 최대 가치

점화식: dp[i][w] = max(dp[i-1][w], dp[i-1][w-time[i]] + v[i])

활동 선택 X　　활동 선택 O

제한 시간: 100분

활동명	시간	점수 (우선순위)	효율 (가치/시간)
A	10	11	1.1
B	50	50	1.0
C	50	50	1.0
D	95	99	1.04

DP 테이블

	w=0	w=10	w=50	w=60	w=95	w=100	
A	0	11	11	11	11	11	A 까지 고려
B	0	11	50	61	61	61	A,B 까지 고려
C	0	11	50	61	61	100	A,B,C 까지 고려
D	0	11	50	61	99	100	A,B,C,D 까지 고려

dp[4][100] = 100

2. 알고리즘 비교 분석

3. DP (Dynamic Programming)

dp[i][w] = i 개 활동을 보고, w분 사용했을 때 최대 가치

DP 테이블

	w=0	w=10	w=50	w=60	w=95	w=100
A	0	11	11	11	11	11
B	0	11	50	61	61	61
C	0	11	50	61	61	100
D	0	11	50	61	99	100

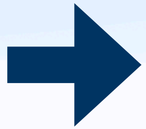
A 까지 고려

A,B 까지 고려

A,B,C 까지 고려

A,B,C,D 까지 고려

dp[4][100] = 100



최대 가치 : 100
선택: ?

Backtracking으로 활동 역추적

2. 알고리즘 비교 분석

3. DP (Dynamic Programming)

제한 시간: 100분

활동명	시간	점수	효율
A	10	11	1.1
B	50	50	1.0
C	50	50	1.0
D	95	99	1.04

Backtracking으로 활동 역추적

$$dp[i][w] = \max(dp[i-1][w],\ dp[i-1][w-time[i]] + v[i])$$

- 값이 변했다면 = i번째 활동 선택 / 값이 같다면 = i번째 활동 선택 안함

DP 테이블

	w=0	w=10	w=50	w=60	w=95	w=100
A	0	11	11	11	11	11
B	0	11	50	61	61	61
C	0	11	50	61	61	100
D	0	11	50	61	99	100

dp[4][100] = 100
dp[3][100] = 100
dp[2][100] = 61
C 선택
D 선택 안함
dp[2][50] = 50
dp[1][50] = 11
B 선택
dp[1][0] = 0 (종료)

선택: B, C
시간 : 100분
점수 : 100점

3. DP (Dynamic Programming)

채택 : 최적의 해 를 보장하면서도 성능 충분

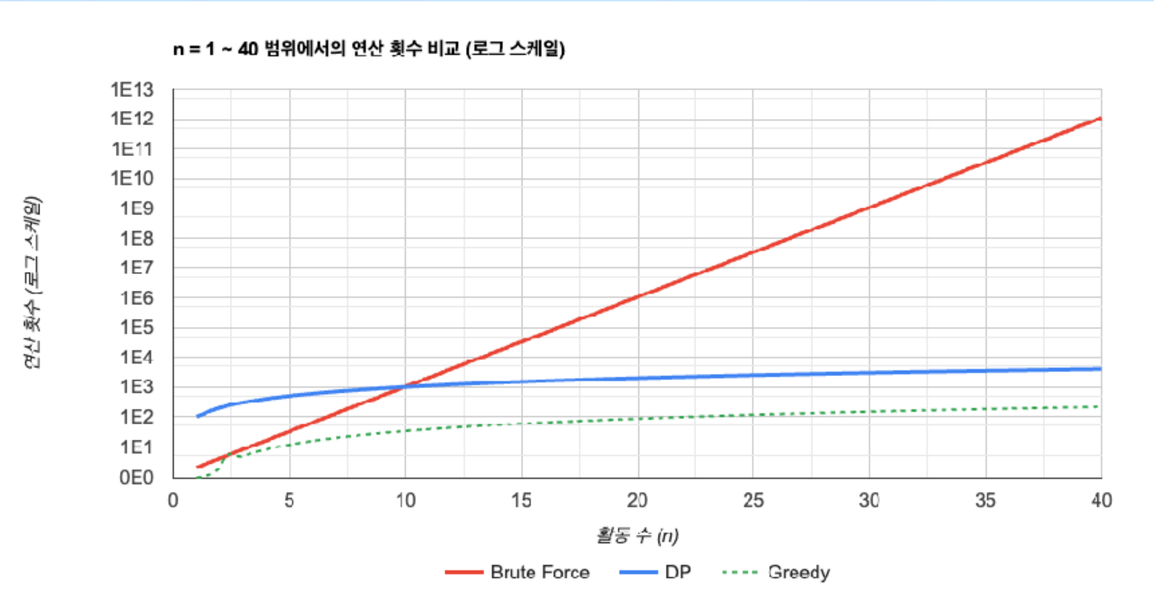
선택: B, C
시간 : 100분
점수 : 100점

시간 복잡도: O(n * W) (n : 활동 개수, W : 제한 시간)

공간 복잡도: O(n * W) (n : 활동 개수, W : 제한 시간)

2. 알고리즘 비교 분석

DP vs Brute Force

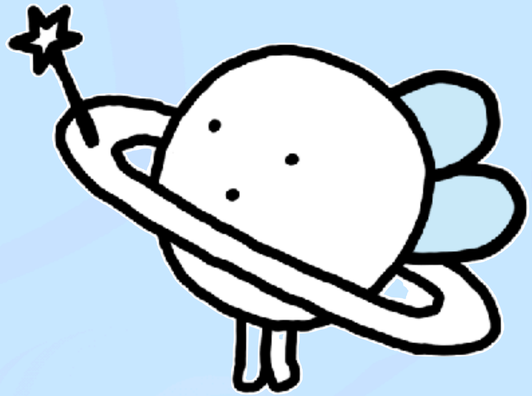


활동 수 (n)	Brute Force (2^n)	DP (n × 100)	Greedy (n log n)	가장 빠른 알고리즘
1	2	100	0.00	Brute Force
2	4	200	2.00	Brute Force
3	8	300	4.75	Brute Force
4	16	400	8.00	Greedy/DP
5	32	500	11.61	Greedy/DP
10	1,024	1,000	33.22	Greedy/DP
15	32,768	1,500	58.60	DP
20	1,048,576	2,000	86.44	DP
25	33,554,432	2,500	116.10	DP
30	1,073,741,824	3,000	147.21	DP

O(n * W) vs O(2^n)

- n <= 4 : Brute Force
- n >= 20 : DP가 압도적으로 빠름

개요



1. 문제 정의
2. 알고리즘 비교 분석
3. 실제 구현 코드
4. 개선 방향

3. 실제 구현 코드

OptimalActivityFinder

DP 테이블 생성

```java
private int[][] buildDpTable(List<SelectedActivity> activities, int maxMinutes) {
        int n = activities.size();
        int[][] dp = new int[n + 1][maxMinutes + 1];

        for (int i = 1; i <= n; i++) {
            SelectedActivity cur = activities.get(i - 1);
            int duration = cur.getActivity().getDurationMinutes();
            int value = cur.getValue();

            for (int w = 0; w <= maxMinutes; w++) {
                // 선택하지 않는 경우
                dp[i][w] = dp[i - 1][w];

                // 선택하는 경우
                if (w >= duration) {
                    int valueIfSelected = dp[i - 1][w - duration] + value;
                    dp[i][w] = Math.max(dp[i][w], valueIfSelected);
                }
            }
        }

        return dp;
    }
```

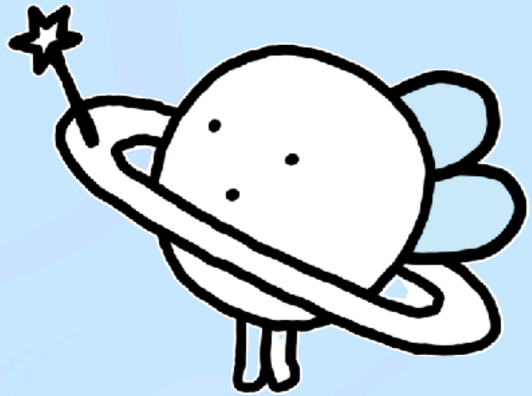
백트래킹 (최적 활동 추적)

```java
private List<SelectedActivity> backtrack(
            List<SelectedActivity> activities,
            int[][] dp,
            int maxMinutes
    ) {
        List<SelectedActivity> selected = new ArrayList<>();
        int n = activities.size();
        int w = maxMinutes;

        for (int i = n; i > 0 && w > 0; i--) {
            if (dp[i][w] != dp[i - 1][w]) {
                SelectedActivity cur = activities.get(i - 1);
                selected.add(cur);
                w -= cur.getActivity().getDurationMinutes();
            }
        }

        Collections.reverse(selected);
        return selected;
    }
```

개요



1. 문제 정의
2. 알고리즘 비교 분석
3. 실제 구현 코드
4. 개선 방향

4. 개선 방향

매번 DP 테이블이 재계산되는 문제

활동 추가/ 삭제 (n)

시간 (w)

재계산

1. 캐싱 : 우선 순위별로 결과 캐시
2. 증분 계산 : 변경된 부분만 재계산

n과 w 자체가 늘어날때
실시간으로 추천해야하는 경우

재계산

근사 알고리즘 (Greedy + 후처리)

꼿