Dynamic Programming

다이나믹 프로그래밍?

- 단순한 문제들로 나누고, 단순한 문제들의 답을 적절히 조합해 원래 문제의 답을 구함
- 큰 문제의 최적해가 작은 문제의 최적해를 포함하는 **Optimal Substructure**
- 부분 구조가 아니라면, 단순한 문제들의 합을 조합해서 원래 문제의 합을 구할 수 없다

다이나믹 프로그래밍?

- 단순한 문제들로 나누고, 단순한 문제들의 답을 적절히 조합해 원래 문제의 답을 구함
- 작은 문제의 답을 여러 번 다시 계산하는 **Overlapping Subproblem**
- 한 번 계산한 다음에 이를 저장해 두면, 다시 계산할 필요 없이 바로 가져다 쓰면 된다

다이나믹 프로그래밍?

- 문제를 보고 바로 다이나믹 프로그래밍(**DP**) 를 떠올리기까지는 많은 경험이 필요
- 문제를 나눠 보고, 그 문제를 풀 수 있다면 지금 문제도 풀 수 있는지 확인해야 함 (Optimal Substructure)
- **상태**가 어떻게 변화하는지에 대한 관찰 필요
- 상태의 흐름을 잘 파악하면, **DP 점화식**을 세우기 쉽다

피보나치 수열

- 다음과 같이 정의된 피보나치 수열의 n번째 항 구하기

$$f_i = \begin{cases} 1 \ (i = 1, 2) \\ f_{i-1} + f_{i-2} (i \geq 2) \end{cases}$$

피보나치 수열

- 재귀적으로 쓰였으니 그대로 함수를 작성하면?

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

- 이 코드의 시간복잡도는 어떻게 될까?

피보나치 수열

- 트리의 형태에서, 겹치는 부분은 한 번만 계산, 한 번 계산한 값은 바뀌지 않음

```
long long fibonacci(int n){
    int& ret = f[n];
    if (ret != -1) return ret;
    ret = fibonacci(n-1) + fibonacci(n-2);
    return ret;
}
```

- 이 코드의 시간복잡도는 어떻게 될까?

피보나치 수열

- 점화식을 이용해서 아래에서 계산을 하면서 채워도 된다

```
long long fibonacci(int n){
    dp[0] = dp[1] = 1;
    for(int i = 2; i <= n; i++) dp[i] = dp[i-1] + dp[i-2];
    reutrn dp[i];
}
```

- 이 코드의 시간복잡도는 어떻게 될까?

Memoization과 Tabulation

- 흔히들 말하는 Top-down과 Bottom-up

```
long long fibonacci(int n){
    int& ret = f[n];
    if (ret != -1) return ret;
    ret = fibonacci(n-1) + fibonacci(n-2);
    return ret;
}
```

```
long long fibonacci(int n){
    dp[0] = dp[1] = 1;
    for(int i = 2; i <= n; i++) dp[i] = dp[i-1] + dp[i-2];
    reutrn dp[i];
}
```

Memoization과 Tabulation

- Tabulation(Bottom-up)에서는 **테이블을 채우는 순서**가 중요
- 프로그램에게 건네야 하는 정보가 더 많음 (순서 강제 등)

- Memoization(Top-down)에서는 이미 채운 경우, **즉시 리턴**하는 것이 중요
- 재귀 코드를 그대로 사용하는 대신, 시간복잡도를 계산하는 것이 보다 까다로움

1로 만들기 BOJ 1463

- 주어진 수를 아래 연산을 사용해 1로 만드는 연산 횟수의 최솟값 구하기

- 3으로 나누어 떨어지면, 3으로 나눈다
- 2로 나누어 떨어지면, 2로 나눈다
- 1을 뺀다

1로 만들기 BOJ 1463

- 항상 나누면 1보다 더 많은 값이 떨어져나가기는 하는데... 이것이 최선일까?
- 10을 1로 만들려면 몇 번의 연산이 필요할까?

$$10 \rightarrow 5 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

1로 만들기 BOJ 1463

- 항상 나누면 1보다 더 많은 값이 떨어져나가기는 하는데… 이것이 최선일까?
- 10을 1로 만들려면 몇 번의 연산이 필요할까?

$$10 \rightarrow 9 \rightarrow 3 \rightarrow 1$$

- 세 번의 연산이면 충분하다!

1로 만들기 BOJ 1463

- 항상 나누면 1보다 더 많은 값이 떨어져나가기는 하는데… 이것이 최선일까?
- 10을 1로 만들려면 몇 번의 연산이 필요할까?

$$10 \rightarrow 9 \rightarrow 3 \rightarrow 1$$

- 세 번의 연산이면 충분하다!

다시 돌아보는 수학적 귀납법

- Base case 찾기, 가정하고 다음 Step이 기계적으로 계산된다면, 증명 완료
- $f(n) = n$을 1로 만드는데 드는 최소 연산 횟수
- Base: $n = 1$ 일 때, 연산이 필요하지 않다 $\rightarrow 0$.
- Step: $n = i$일 때,
- i가 3의 배수라면, $f(\frac{i}{3})$이 올바른 값을 구해준다면, 3으로 나누는 비용 1을 더한다
- 2의 배수, 1을 뺄 때에도 마찬가지

1로 만들기 BOJ 1463

- 이미 기록된 값이라면 **곧바로 리턴**하는 것을 잊지 말자!

```
int f(int x){
    if (x == 1) return 0;
    int& ret = d[x];
    if (ret != INF) return ret;
    if (x % 3 == 0) ret = min(ret, f(x/3) + 1);
    if (x % 2 == 0) ret = min(ret, f(x/2) + 1);
    ret = min(ret, f(x-1) + 1);
    return ret;
}
```

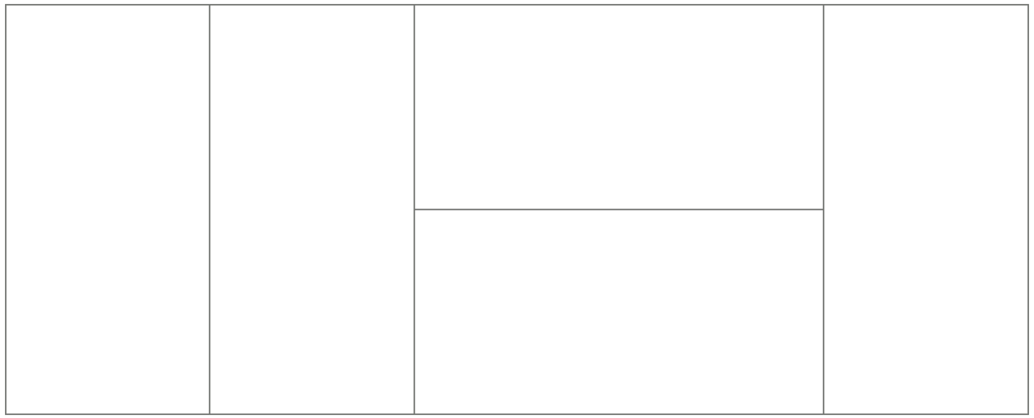
1로 만들기 BOJ 1463

- 바닥부터 올라갈 수 있을까? 계산 순서는 어디에서부터 해야 할까?
- $i = 1$일 때 연산이 필요없다.
- 2 또는 3으로 나누어떨어지는 경우, 해당 최솟값에 연산을 1회 하면 된다.
- 1을 뺄 때에도 마찬가지이다.

```
dp[0] = dp[1] = 0;
for(int i = 2; i <= N; i++) {
    if (i%2 == 0) dp[i] = min(dp[i/2] + 1, dp[i]);
    if (i%3 == 0) dp[i] = min(dp[i/3] + 1, dp[i]);
    dp[i] = min(dp[i-1] + 1, dp[i]);
}
```

2 x n 타일링 BOJ 11726

- $f(i)$ 가 어떤 값을 반환한다고 생각할까?
- 이 함수의 Base case는 어떻게 될까?
- 해당 값을 구하기 위해 어떤 가정을 세워야 할까?



2 x n 타일링 BOJ 11726

- $f(i) = 2 \times i$ 의 타일을 채우는 경우의 수
- Base case: $i = 2$, 한 가지
- Step: 세워진 것 하나를 놓는 경우, 눕혀진 것 두 개를 놓는 경우.
- 세워진 것 하나를 놓기 위해서 $f(i-1)$을 알아야 하고,
- 눕혀진 것 두개를 놓기 위해서 $f(i-2)$를 알아야 함
- 이 두 개는 가정하고, 두 개를 더해 주면 현재 타일의 경우의 수를 얻을 수 있다!

2 x n 타일링 BOJ 11726

- 피보나치 수와 같은 방식으로 값을 채워나갈 수 있다
- Top-down, Bottom-up 중 자신이 편한 방법으로 문제를 해결하자
- 일반적으로 Bottom-up이 조금 더 짜기 어려울 때가 많지만, 함수의 재귀호출 오버헤드가 없어 더 빠르게 동작한다

포도주 시식 BOJ 2156

- 놓여진 포도주 잔을 **연속으로 세 잔** 마실 수 없다
- 마실 수 있는 포도주의 양을 **최대화**하자
- **6**, **10**, 13, **9**, **8**, 1
- 항상 많은 것을 마시는 것이 최선이 아니다

포도주 시식 BOJ 2156

- $f(i) = i$번째 잔까지 먹을 수 있는 양의 최댓값?
- 단순히 날짜의 정보보다, 내가 지금까지 몇 잔을 마셨는지에 대한 정보도 필요하다
- $f(i, j) = i$번째 포도주까지, j번 연속해서 마셨을 때, 먹을 수 있는 양의 최댓값
- 이렇게 정의한 뒤, 정답을 구하기 위해서 어떻게 해야 할까?

포도주 시식 BOJ 2156

- $f(i, j) = i$번째 포도주에서 시작해서 현재 j번 연속해서 마셨을 때, 먹은 양의 최댓값
- Base: $x = N$인 경우, j가 2면 세 잔을 연속해서 마실 수 없으므로 0, 그렇지 않다면 a_i
- step: 만약 $j < 2$라면, 현재 잔을 마실 수 있으므로, $f(i + 1, j + 1) + a_i$
- 그렇지 않거나, 안 먹는게 이득일 수 있으므로 $f(i + 1, 0)$
- $f(1, 0)$을 부르면 알아서(?) 해 준다!

포도주 시식 BOJ 2156

```
int f(int i, int j) {
    if (i == N) {
    if (j < 2) return a[i];
    else return 0;
}
    int& ret = dp[i][j];
    if (ret != -1) return ret;
    if (j < 2) ret = max(ret, f(i+1, j+1) + a[i]);
    ret = max(ret, f(i+1, 0));
    return ret;
}
```

포도주 시식 BOJ 2156

- 귀납법을 이용한 Top-down이 아닌, Bottom-up으로도 해결해 보자!
- $d_{i,j}$: 방금 정의한 것과 같다고 하자
- $j = 0$일 때에는 지금 마시지 않는 것이므로 $i - 1$일 때 모든 j에 대해서 가져온다
- $j = 1$일 때에는 이전 잔의 $j = 0$를 가져와서 마신다
- $j = 2$일 때에는 이전 잔의 $j = 1$을 가져와서 마신다
- 연산의 순서를 적절히 정한 뒤, 답을 구해 보자!

포도주 시식 BOJ 2156

```
dp[1][1] = a[1];
for(int i = 2; i <= N; i++) {
    dp[i][0] = max(dp[i-1][0], max(dp[i-1][1], dp[i-1][2]));
    dp[i][1] = dp[i-1][0] + a[i];
    dp[i][2] = dp[i-1][1] + a[i];
}
cout << max(dp[N][0], max(dp[N][1], dp[N][2])) << '\n';
```

LCS BOJ 9251

- 두 문자열의 최장 공통 부분 수열을 구하는 문제, 문자열의 길이는 최대 $1\,000$
- **ACA**Y**K**P
- C**A**P**CAK**
- $f(i) = ?$
- $f(i, j) = ?$

LCS BOJ 9251

- 두 문자열의 최장 공통 부분 수열의 길이를 구하는 문제, 문자열의 길이는 최대 $1\,000$
- $f(i, j)$: $a[1..i], b[1..j]$의 LCS 길이

LCS BOJ 9251

- 두 문자열의 최장 공통 부분 수열의 길이을 구하는 문제, 문자열의 길이는 최대 $1\,000$
- $f(i,j)$: $a[1..i], b[1..j]$의 LCS 길이
- Base : $i \leq 0$ 또는 $j \leq 0$, LCS는 0

LCS BOJ 9251

- 두 문자열의 최장 공통 부분 수열의 길이을 구하는 문제, 문자열의 길이는 최대 $1\,000$
- $f(i,j)$: $a[1..i], b[1..j]$의 LCS 길이
- Base : $i \leq 0$ 또는 $j \leq 0$, LCS는 0
- Step: $a[i] = b[j]$인 경우, $f(i-1, j-1)$에 1을 더한 값
- 그렇지 않은 경우, $f(i-1, j)$, $f(i, j-1)$중 더 큰 값

LCS BOJ 9251

- $d[i][j]$: $a[1 \dots i], b[1 \dots j]$ 의 LCS 길이

	-	A	C	A	Y	K	P
-	0	0	0	0	0	0	0
C	0						
A	0						
P	0						
C	0						
A	0						
K	0						

LCS BOJ 9251

- a[i]==b[j] 라면, 각자 한 글자 전에서 LCS 길이를 1씩 더한 것과 같다.

	-	A	C	A	Y	K	P
-	0	0	0	0	0	0	0
C	0	0	1				
A	0						
P	0						
C	0						
A	0						
K	0						

LCS BOJ 9251

- 같지 않다면, a[i-1], b[j]의 LCS 길이와, a[i], b[j-1]의 LCS 길이 중 큰 것을 취한다.

	-	A	C	A	Y	K	P
-	0	0	0	0	0	0	0
C	0	0	1	**1**			
A	0						
P	0						
C	0						
A	0						
K	0						

LCS BOJ 9251

- 구하고자 하는 값은, a[1..N], b[1..M]의 LCS 길이이다.

	-	A	C	A	Y	K	P
-	0	0	0	0	0	0	0
C	0	0	1	1	1	1	1
A	0	1	1	2	2	2	2
P	0	1	1	2	2	2	3
C	0	1	2	2	2	2	3
A	0	1	2	3	3	3	3
K	0	1	2	3	3	4	**4**

평범한 배낭 BOJ 12865

- 배낭에 최대 K 무게만큼 담을 수 있다
- 각 물건은 무게와 가치를 가지고 있다
- 배낭에 넣을 수 있는 물건들의 가치의 최댓값
- $N \leq 100, K \leq 100\,000$

평범한 배낭 BOJ 12865

- 간단하게 생각하면, 하나의 물건을 넣거나, 넣지 않거나 두 가지 상태
- 모두 고려하면 2^N으로 시간초과
- 적절한 DP 식을 어떻게 세울까?
- $f(i) \rightarrow ?$
- $f(i, j) \rightarrow ?$

평범한 배낭 BOJ 12865

- $f(i, cap)$: i번째 물건까지 고려했을 때, 현재 남은 배낭의 크기가 cap일 때 담을 수 있는 최대 가치

Base: $i > N$이거나 $\text{cap} = 0$, 더 담을 수 있는 가치는 없다.

Step: 물건을 고르거나, 고르지 않을 수 있다.

- 고르지 않는 경우, $f(i + 1, cap)$, 배낭의 남은 크기는 변하지 않는다
- 고르기 위해서는 현재 남은 배낭의 크기가 물건 크기보다 크거나 같아야 한다.
- 가능한 경우, $f(i + 1, cap - w[i]) + v[i]$ 가 답이 **될 수** 있다.

평범한 배낭 BOJ 12865

```
int f(int i, int cap) {
    if (i > N || cap < 0) return 0;
    int& ret = dp[i][cap];
    if (ret != -1) return ret;
    ret = f(i+1, cap);
    if (cap >= w[i])
        ret = max(ret, f(i+1, cap-w[i]) + v[i]);
    return ret;
}
```

Prefix Sum

간단한 문제

- 길이 N인 배열 arr이 주어진다
- $l, r\ (1 \leq\ l, r \leq N)$이 주어졌을 때,
- `arr[l] + arr[l+1] + … + arr[r-1] + arr[r]` 의 값을 구하자

1	4	2	6	7	9	2	6	8	4
		l				r			

- Naïve (Bruteforce)

```
for(int i = l; i <= r; i++) ans += arr[i];
```

오늘 해결할 문제 BOJ 11659

- 길이 N인 배열 arr이 주어진다
- $l, r\ (1 \le\ l, r \le N)$이 주어졌을 때,
- `arr[l] + arr[l+1] + … + arr[r-1] + arr[r]` 의 값을 ***Q*번** 구하자

1	4	2	6	7	9	2	6	8	4
		l				r			

- $N, Q \le 100\,000$
- Naïve (Bruteforce) : $O(NQ)$

누적 합 아이디어

1	4	2	6	7	9	2	6	8	4
		l				r			

- 미리 모든 경우의 수를 전처리한다고 하더라도 $O(N^2)$
- 구간의 기반해서 문제를 다시 확인하자
- $[l, r]$ 구간은 어떻게 나타낼 수 있을까?
- l, r을 각각 독립적으로 보자

누적 합 아이디어

1	4	2	6	7	9	2	6	8	4
		l				r			

- 왼쪽과 오른쪽 사이의 간격은 얼마나 될까?
- $r - l + 1$, 이것을 우리가 원하는 쿼리에 적용할 수 있을까?
- $p[r] - p[l-1]$, p를 어떻게 구축해야 할까?

누적 합 아이디어

1	4	2	6	7	9	2	6	8	4
1	5	7	13	20	29	31	37	45	49
		l				r			

- $p[r] - p[l-1]$, p를 어떻게 구축해야 할까?
- 구간의 사이 거리를 구할 때에도 암묵적으로 하나의 위치(0)를 상대적으로 가진다
- 가장 처음부터 **누적해서 합**을 구해나간다면, 원하는 값은 얼마만에 구할 수 있을까?

누적 합 아이디어

1	4	2	6	7	9	2	6	8	4

1	5	7	13	20	29	31	37	45	49
		l				r			

- 구현의 편의를 위해서 새로 구한 p는 1-based로 작성하기도 한다
- 전처리 $O(N)$, 쿼리 하나 당 $O(1)$
- $O(N+Q)$에 문제를 해결할 수 있다

누적 합

```
for(int i = 1; i <= N; i++) {
    cin >> arr[i];
    p[i] = p[i-1] + arr[i];
}
for(int i = 0; i < Q; i++) {
    cin >> l >> r;
    cout << p[r] - p[l-1] << '\n';
}
```

블로그 BOJ 21921

- 블로그의 방문자 수가 주어지면, X일 동안 방문한 사람의 수가 가장 많은 구간 구하기
- $X = 3$이고 방문자 수가 $1\ 1\ 1\ 1\ 1\ 5\ 1$ 이라면, $1\ 1\ 5, 1\ 5\ 1$ 두 가지 구간이 7
- 단순하게 먼저 접근해 보자, 모든 구간의 경우를 계산한다면, 시간이 얼마나 걸릴까?

블로그 BOJ 21921

- 블로그의 방문자 수가 주어지면, X일 동안 방문한 사람의 수가 가장 많은 구간 구하기
- $X = 3$이고 방문자 수가 $1\ 1\ 1\ 1\ 1\ 5\ 1$ 이라면, $1\ 1\ 5, 1\ 5\ 1$ 두 가지 구간이 7
- 단순하게 먼저 접근해 보자, 모든 구간의 경우를 계산한다면, 시간이 얼마나 걸릴까

- $O(N^2)$

블로그 BOJ 21921

- 블로그의 방문자 수가 주어지면, X일 동안 방문한 사람의 수가 가장 많은 구간 구하기
- 누적 합을 사용해서 $O(N)$개의 쿼리가 있는 누적 합 문제로 환원
- $O(N)$에 문제를 해결할 수 있다!

벌레컷 BOJ 27651

- 배열이 주어질 때, 다음 식을 만족하는 X, Y, Z 쌍의 개수 찾기

$$\sum_{i=1}^{X} A_i < \sum_{i=Y+1}^{N} A_i < \sum_{i=X+1}^{Y} A_i,\ 1 \le X < Y < N$$

벌레컷 BOJ 27651

- 배열이 주어질 때, 다음 식을 만족하는 X, Y, Z 쌍의 개수 찾기

$$\sum_{i=1}^{X} A_i < \sum_{i=Y+1}^{N} A_i < \sum_{i=X+1}^{Y} A_i,\ 1 \le X < Y < N$$

3	4	12	1	8

- 세 부분으로 나누었을 때, 앞쪽 합이 제일 작고, 가운데 합이 제일 큰 구간 구하기

벌레컷 BOJ 27651

- 배열이 주어질 때, 다음 식을 만족하는 X, Y, Z 쌍의 개수 찾기
- 합을 구해야 하니 누적 합을 사용하는 것까지는 생각할 수 있지만, 움직이는 점이 두 개
- 결국 모든 점에 대해서 따져보면 $O(N^2)$

벌레컷 BOJ 27651

- 배열이 주어질 때, 다음 식을 만족하는 X, Y, Z 쌍의 개수 찾기
- 점이 여러 개일 때에는, 하나를 고정하는 테크닉을 사용해 보자
- X를 고정하고, Y에 해당할 수 있는 점의 개수를 빠르게 구해야 한다
- 누적 합의 특징은 무엇이 있을까?

벌레컷 BOJ 27651

- 배열이 주어질 때, 다음 식을 만족하는 X, Y, Z 쌍의 개수 찾기
- 누적 합 배열이 **단조증가한다는 성질**을 활용하면, 이분 탐색을 사용할 수 있다
- X를 하나 고정하는 데 $O(N)$
- X에 대해서, 문제 식에 해당하는 Y의 범위를 찾는 데 두 번의 이분 탐색으로 $O(logN)$
- 총 $O(NlogN)$에 문제를 해결할 수 있다!
- $O(N)$에 문제를 해결할 수 있을까?

누적 합을 2차원에서 사용하기 BOJ 11660

- 2차원 배열이 주어질 때, (x_1, y_1)을 왼쪽 위 꼭짓점으로, (x_2, y_2)를 오른쪽 아래 꼭짓점으로 하는 직사각형에 적힌 수의 합을 구하기
- 1차원 누적 합을 사용하면 하나의 행 또는 열에 대해서 $O(1)$에 해결할 수 있음
- 하지만, 쿼리 한 번에 모든 행/열을 계산해야 하므로 결국 $O(N)$
- $O(NQ)$는 시간초과

누적 합을 2차원에서 사용하기 BOJ 11660

- 1차원 누적 합에서는 p_i를 $[1, i]$의 합으로 정의했다
- 이를 확장해서 $p_{i,j}$를 $[1,1]$을 왼쪽 위 꼭짓점으로, $[i, j]$를 오른쪽 아래 꼭짓점으로 가지는 배열을 만들 수 있을까?
- $p_{i,j}$는 $p_{i-1,j}$에 해당하는 사각형과 $p_{i,j-1}$에 해당하는 사각형과의 관계를 살펴보자

누적 합을 2차원에서 사용하기 BOJ 11660

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

1	3	6	10
3	8	15	24
6	15	27	?

1	3	6	10
3	8	15	24
6	15	27	42

낚시 BOJ 30461

- 그림과 같이 직각삼각형에 적힌 수들의 합을 구하는 문제
- $N, M \leq 2\,000, \quad Q \leq 300\,000$

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

낚시 BOJ 30461

- $p_{i,j}$: (i, j)를 오른쪽 아래 꼭짓점으로 하는 직각삼각형에 적힌 모든 수의 합
- 계산의 순서가 중요하다

1	2	3	4
2	3	4	5
3	4	5	6
4	5	6	7

낚시 BOJ 30461

- 다음과 같이 모형을 바꾸면, 단순한 2차원 누적 합으로도 가능

1	2	3
5	2	1
0	2	4

→

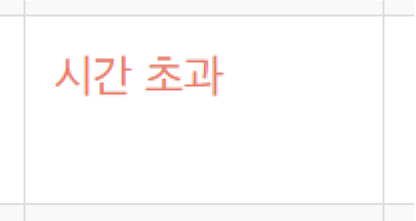
0	0	3
0	2	1
1	2	4
5	2	0
0	0	0

나중에 생각할 문제 (BOJ 2042)

- 길이 N인 배열 arr이 주어진다
- l, r $(1 \leq l, r \leq N)$이 주어졌을 때, 다음 두 가지 연산 중 하나를 Q번 수행한다.
- `arr[l] + arr[l+1] + … + arr[r-1] + arr[r]` 의 값을 구하기
- `arr[i]`의 값을 k로 변경하기

1	4	2	6	~~7~~2	9	2	6	8	4
		l				r			

- Prefix sum with update? $O(NQ)$
- 최악의 경우는, $i = 1$인 변경 쿼리가 계속해서 들어올 때



References

- https://github.com/justiceHui/SSU-SCCC-Study/blob/master/2023-summer-basic/slide/06-1-dynamic-programming.pdf