6

0/1_Knapsack

```
void knapsack(int n, int x, int weight[], int value[]) {
  int dp[n + 1][x + 1];

  for (int i = 0; i <= n; i++)
    for (int j = 0; j <= x; j++)
      dp[i][j] = 0;

  for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= x; j++) {
      int pick = 0;
      if (j >= weight[i - 1]) // 注意：weight/value 是 0
-based
        pick = value[i - 1] + dp[i - 1][j - weight[i -
1]];
      int skip = dp[i - 1][j];
      dp[i][j] = max(pick, skip);
    }
  }
  cout << dp[n][x];
}
```

unbounded_knapsack_reconstruct

```
// reconstructItems：重建 Unbounded Knapsack 中物品的選取
次數
vector<int> reconstructItems(int n, int W, int weights[],
 int values[]) {
    // 建立 dp 陣列：dp[w] 表示容量為 w 時所能達到的最大價
值
    vector<int> dp(W + 1, 0);

    // 建立 choice 陣列：choice[w] 記錄在容量為 w 時最後一
次選的物品 index
    vector<int> choice(W + 1, -1);

    // 遍歷所有物品
    for (int i = 0; i < n; i++) {
        // 對每個物品，從 weights[i] 開始（因為比它還小裝不
下），一直到 W
        for (int w = weights[i]; w <= W; w++) {
            // 如果在容量 w - weights[i] 的情況下加入這個
物品會變得更好
            int newValue = dp[w - weights[i]] + values
[i];
            if (newValue > dp[w]) {
                // 更新最大價值
                dp[w] = newValue;
                // 記錄是由物品 i 選進來的
                choice[w] = i;
            }
        }
    }

    // 初始化結果陣列，resultCount[i] 表示第 i 個物品選了幾
次
    vector<int> resultCount(n, 0);
    // 從背包的最終容量 W 開始回推
    int w = W;

    // 當還有容量且有選過的物品紀錄時，持續回推
    while (w > 0 && choice[w] != -1) {
        // 找出這個容量是由哪個物品選進來的
        int item = choice[w];
        // 該物品使用次數 +1
        resultCount[item]++;
        // 回推剩餘容量
        w -= weights[item];
    }

    // 回傳每個物品的選用次數
    return resultCount;
}
```

---

bounded_knapsack_brute_reconstruct (時間：O(n * W * c_max))

```
int boundedKnapsack_brute(int n, int W, int weights[], i
nt values[], int counts[], vector<vector<int>> &choice)
{
    // dp[i][w]: 前 i 個物品在容量 w 下的最大總價值
    vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));
    // choice[i][w]: 在 dp[i][w] 的最佳選法中，選了第 i 個
物品幾次
    choice.assign(n + 1, vector<int>(W + 1, 0));

    for (int i = 1; i <= n; ++i) {
        int w_i = weights[i - 1];
        int v_i = values[i - 1];
        int c_i = counts[i - 1];
        for (int w = 0; w <= W; ++w) {
            // 預設不選第 i 個物品
            dp[i][w] = dp[i - 1][w];
            choice[i][w] = 0;

            // 嘗試選 1~c_i 個物品 i（只要裝得下）
            for (int k = 1; k <= c_i; ++k) {
                if (w >= k * w_i) {
                    int candidate = dp[i - 1][w - k * w_
i] + k * v_i;
                    if (candidate > dp[i][w]) {
                        dp[i][w] = candidate;
                        choice[i][w] = k; // 紀錄這狀態是
選了幾個 i
                    }
                } else break;
            }
        }
    }

    return dp[n][W]; // 回傳最大價值
}

vector<int> reconstruct_bounded_brute(int n, int W, cons
t vector<vector<int>> &choice, int weights[]) {
    vector<int> resultCount(n, 0);
    int w = W;

    for (int i = n; i >= 1; --i) {
        int k = choice[i][w];       // 選了幾個第 i 個物
品（對應 weights[i-1]）
        resultCount[i - 1] = k;     // 對應到原始物品陣列
 index
        w -= k * weights[i - 1];    // ❗正確地扣除重量
（這是關鍵）
    }

    return resultCount;
}
```

---

knapsack Binary Optimization + reconstruct

```
int boundedKnapsack_binary(int n, int W, int weights[],
int values[], int counts[],
                           vector<int> &dp, vector<int>
&choice,
                           vector<int> &itemRecord, vect
or<int> &virtualWeights) {
    dp.assign(W + 1, 0);               // dp[w]: 容量 w
下的最大價值
    choice.assign(W + 1, -1);          // choice[w]: 最
後放入哪個虛擬物品
    itemRecord.clear();
    virtualWeights.clear();

    for (int i = 0; i < n; ++i) {
        int count = counts[i];
        int w = weights[i], v = values[i];

        for (int k = 1; count > 0; k <<= 1) {
            int actual = min(k, count);
```

```
            int tw = actual * w, tv = actual * v;

            itemRecord.push_back(i);              // 原始物品編號
            virtualWeights.push_back(tw);         // 虛擬物品重量
            int virtualIndex = itemRecord.size() - 1;

            for (int j = W; j >= tw; --j) {
                if (dp[j] < dp[j - tw] + tv) {
                    dp[j] = dp[j - tw] + tv;
                    choice[j] = virtualIndex;     // 紀錄這次選的虛擬物品
                }
            }
            count -= actual;
        }
    }
    return dp[W];
}

vector<int> reconstruct_binary(int W, const vector<int> &choice,
                               const vector<int> &itemRecord,
                               const vector<int> &virtualWeights) {
    vector<int> resultCount(*max_element(itemRecord.begin(), itemRecord.end()) + 1, 0);

    int w = W;
    while (w > 0 && choice[w] != -1) {
        int virtualIndex = choice[w];
        int item = itemRecord[virtualIndex];
        resultCount[item]++;
        w -= virtualWeights[virtualIndex];
    }

    return resultCount;
}
```

方法	時間複雜度	優點	缺點
Brute Force	O(n * W * c_max)	容易理解、實作直覺	遇到大 c 容易超時
Binary Opt.	O(n * W * log c)	快速、省空間	稍微複雜，要做虛擬物品轉換
重建方法	O(W)	可取得每個物品使用次數	額外記錄需要準備完整

7

Basketball

```
#include <iostream>
#include <vector>
using namespace std;
#define tp top_down
#define bu bottom_up

int tp(int n) {
  static vector<int> dp(1, -1);
  if (n < 0)
    return 0;
  if (n == 0)
    return 1;
  if (n >= dp.size())
    dp.resize(n + 1, -1);
  if (dp[n] != -1)
    return dp[n];

  return dp[n] = tp(n - 2) + tp(n - 3);
}

int bu(int n) {
  vector<int> dp(n + 1, 0);
  for (int i = 1; i <= n; ++i) {
    if (i >= 2)
      dp[i] += dp[i - 2];
    if (i >= 3)
      dp[i] += dp[i - 3];
  }
  return dp[n];
}

int main() {
  int n;
  cin >> n;
  cout << tp(n);
  cout << '\n';
  cout << bu(n);
  return 0;
}
```

Broken Stair

```
#include <iostream>
#include <unordered_set>
#include <vector>
using namespace std;
#define tp top_down
#define bu bottom_up

unordered_set<int> broken;

int tp(int n) {
  static vector<int> dp = {-1};
  if (n < 0)
    return 0;
  if (broken.count(n))
    return 0;
  if (n == 0)
    return 1;
  if (n >= dp.size())
    dp.resize(n + 1, -1);
  if (dp[n] != -1)
    return dp[n];

  return dp[n] = tp(n - 1) + tp(n - 2);
}

int bu(int n) {
  vector<int> dp(n + 1, 0);
  dp[0] = 1;

  for (int i = 1; i <= n; ++i) {
    if (broken.count(i))
      continue;
    if (i >= 1)
      dp[i] += dp[i - 1];
    if (i >= 2)
      dp[i] += dp[i - 2];
  }
  return dp[n];
}

int main() {
  int n;
  cin >> n;

  broken = {3, 5};

  cout << tp(n);
  cout << '\n';
  cout << bu(n);
  return 0;
}
```

Coin Change

```
#include <iostream>
#include <vector>
using namespace std;

int coin_change_brute(const vector<int> &coins, int x) {
  int n = coins.size();
```

```
  vector<vector<int>> dp(n, vector<int>(x + 1, 0));

  // 初始化：金額為 0 時都有一種方式（什麼都不拿）
  for (int i = 0; i < n; ++i)
    dp[i][0] = 1;

  // 處理只有第一種硬幣時的情況
  for (int j = 1; j <= x; ++j)
    dp[0][j] = (j % coins[0] == 0) ? 1 : 0;

  // 主體：對每一種硬幣、每一個金額，試試看拿幾個
  for (int i = 1; i < n; ++i) {
    for (int j = 1; j <= x; ++j) {
      dp[i][j] = 0;
      for (int count = 0; count * coins[i] <= j; ++count)
{
        dp[i][j] += dp[i - 1][j - count * coins[i]];
      }
    }
  }

  return dp[n - 1][x];
}

int coin_change(const std::vector<int> &coins, int x) {
  int n = coins.size();
  int dp[n][x + 1]; // declaration

  for (int i = 0; i < n; ++i)
    dp[i][0] = 1; // init

  for (int j = 1; j <= x; ++j)
    dp[0][j] = (j % coins[0] == 0) ? 1 : 0;
  // 如果金額剛好能被第一種硬幣整除，那就有一種方式。否則沒有
解。

  for (int i = 1; i < n; ++i)
    for (int j = 1; j <= x; ++j) {
```

```
      dp[i][j] = dp[i - 1][j];
      if (j >= coins[i])
        dp[i][j] += dp[i][j - coins[i]];
    }
  return dp[n - 1][x];
}

int coin_change_1D(const std::vector<int> &coins, int am
ount) {
  int n = coins.size();
  std::vector<int> dp(amount + 1, 0);

  dp[0] = 1; // init
  for (int ik = 0; ik < n; ++ik)
    for (int im = coins[ik]; im <= amount; ++im)
      dp[im] += dp[im - coins[ik]];

  return dp[amount];
}

int main() {
  int n, x;
  std::cin >> n >> x;
  vector<int> a(n);
  for (int &i : a)
    std::cin >> i;

  std::cout << coin_change_brute(a, x) << '\n';
  std::cout << coin_change(a, x) << '\n';
  std::cout << coin_change_1D(a, x) << '\n';
  return 0;
}
```

Climb Stair

```
#define ll long long
ll climb(int n) {
  if (n < 0)
    return 0;
  static std::vector<ll> dp(1, -1);
  dp.resize(n + 1, -1);
  if (n == 0)
    return 1;
  if (dp[n] != -1)
    return dp[n];
  return dp[n] = climb(n - 1) + climb(n - 2) + climb(n -
 3);
}

// ##### 學生的實作區域 START #####
ll count_ways_to_climb(int n) {
  if (n < 0)
    return 0;
  ll ways = 0;
  std::vector<ll> dp(n + 1, 0);
  dp[0] = 1;
  for (int i = 1; i <= n; i++) {
    if (i >= 1)
      dp[i] += dp[i - 1];
    if (i >= 2)
      dp[i] += dp[i - 2];
    if (i >= 3)
      dp[i] += dp[i - 3];
  }
  return ways = dp[n];
}
```

Longest Increasing Subsequence

```
int longest_increasing_subsequence(const std::vector<int
> &nums) {
  if (nums.empty())
    return 0;
```

```
  int n = nums.size();
  std::vector<int> dp(n, 1);

  for (int i = 1; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
      if (nums[j] < nums[i]) {
        dp[i] = std::max(dp[i], dp[j] + 1);
      }
    }
  }

  return *std::max_element(dp.begin(), dp.end());
}
```

Number Triangle

```
#include <iostream>
#include <vector>
using namespace std;

int n;
vector<vector<int>> grid;
vector<vector<int>> dp;

int solve(int i, int j) {
  if (i == n - 1)
    return grid[i][j];
  if (dp[i][j] != -1)
    return dp[i][j];
  dp[i][j] = grid[i][j] + max(solve(i + 1, j + 1), solve
(i + 1, j));

  return dp[i][j];
}

int solve1(int i, int j) {
  if (j < 0 || j > i)
    return 0;
```

```
  if (i == 0 && j == 0)
    return grid[i][j];
  if (dp[i][j] != -1)
    return dp[i][j];

  return dp[i][j] = max(solve(i - 1, j - 1), solve(i - 1,
 j)) + grid[i][j];
}

int main() {
  cin >> n;
  grid.resize(n);
  dp.resize(n, vector<int>(n, -1));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < i + 1; j++) {
      int x;
      cin >> x;
      grid[i].push_back(x);
    }
  }

  // 1
  // 1 2
  // 1 2 3
  // 1 2 3 4

  int Max = 0;
  // for (int j = n - 1; j >= 0; --j)
  //   Max = max(Max, solve1(n - 1, j));
  //
  Max = solve(0, 0);

  cout << Max << '\n';

  for (auto row : grid) {
    for (int col : row) {
      cout << col << ' ';
    }
    cout << '\n';
  }
  return 0;
}
```

Stock holder

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main() {
  int n, fee;
  cin >> n >> fee;
  vector<int> prices(n);
  for (int i = 0; i < n; ++i)
    cin >> prices[i];

  /*
  vector<int> cash(n), hold(n);
  cash[0] = 0;
  hold[0] = -prices[0];

  for (int i = 1; i < n; ++i) {
    cash[i] = max(cash[i - 1], hold[i - 1] + prices[i] -
 fee);
    hold[i] = max(hold[i - 1], cash[i - 1] - prices[i]);
  }
  */

  // 初始化兩個狀態：
  // cash 表示當前不持有股票的最大利潤
  // hold 表示當前持有股票的最大利潤
  int cash = 0;
  int hold = -prices[0]; // 第一天買入，手上有股票，利潤為
負

  for (int i = 1; i < n; ++i) {
    // 更新狀態：
    // 如果今天賣出股票：我們昨天手上有股票 hold，加上今天股
價，扣除手續費
    // 和昨天就沒持股的 cash 比較，取較大者
    cash = max(cash, hold + prices[i] - fee);

    // 如果今天買入股票：我們昨天手上沒股票 cash，扣今天股價
    // 和昨天就有持股的 hold 比較，取較大者
    hold = max(hold, cash - prices[i]);
  }

  // 最後答案是我們**最後一天不持股時的最大利潤**
  cout << cash << endl;

  return 0;
}
```