UoA CP Lecture 4
Dynamic Programming

Qingchuan(Sam) Zhang
qzha536@aucklanduni.ac.nz

April 24, 2020

Contents

1 What is Dynamic Programming

Dynamic Programming(DP) is technique in which we reduce the problem to subproblem(s) retaining the same nature but of smaller size(s).

2 Procedures of applying DP

1. What is the problem asking? What's the thing we are optimizing/calculating?
2. What are the parameters/constraints(money, volume, etc.)? Define the **state**.
3. What is the relation between states? Define the **transfer**.
4. (Usually trivial but) what is the **base** case?

3 Trivial Examples

3.1 Factorial

Thinking process

1. Calculate $f(n)$
2. Single parameter: n
3. Transfer: $f(n) = n \times f(n-1)$
4. Base case: $f(0) = 1$

Code

```
int f(int n){
    if(n == 0) return 1;
    else return n * f(n-1);
}
```

3.2 Fibonacci

Thinking process

1. Calculate $f(n)$
2. Single parameter: n
3. Transfer: $f(n) = f(n-1) + f(n-2)$
4. Base case: $f(0) = 0, f(1) = 1$

Here we need to apply **Memorization** to aide our calculation, otherwise the complexity would be exponential, (approximately $O(1.618^n)$).

Code

```
const int maxn = 100 + 10;
int fib_cache[maxn];
const int UNCALCULATED = -1;
int fib(int n){
    if(fib_cache[n] != UNCALCULATED) return fib_cache[n];
    if(n == 0) return 0;
    else if(n == 1) return 1;
    else return fib_cache[n] = fib(n-1) + fib(n-2);
}
int main(){
    fill(fib_cache,fib_cache+maxn,UNCALCULATED);
    cout<<fib(10)<<endl;
}
```

4 Not Too Trivial Examples

4.1 Dice Combinations

Thinking process

1. Calculate the number of ways to construct sum n. Denote as $\texttt{dp}[n]$.
2. Single parameter: n
3. Transfer: $\texttt{dp}[n] = \texttt{dp}[n-1] + \texttt{dp}[n-2] + \cdots + \texttt{dp}[n-6]$
4. Base case: $\texttt{dp}[0] = 1, \texttt{dp}[-n] = 0$

Code

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 1000000+10;
int dp[maxn];
const int UNCALCULATED = -1;
const int MOD = 1000000007;
int calc(int n){
    if(n<0) return 0;
    if(n==0) return 1;
    if(dp[n] != UNCALCULATED)return dp[n];
    else{
        dp[n] = 0;
        for(int i = 1;i<=6;i++){
            dp[n] += calc(n-i);
            dp[n] %= MOD;
        }
        return dp[n];
    }
}
int main(){
    fill(dp,dp+maxn,UNCALCULATED);
    int n;cin>>n;
    cout<<calc(n)<<endl;
}
```

4.2 Maximum path sum II

Thinking process

1. Find the maximum value path from a given position (r, c). Denote as $\texttt{dp}[r][c]$.
2. Parameters: r, c
3. Transfer: $\texttt{dp}[r][c] = v[r][c] + \max(\texttt{dp}[r+1][c], \texttt{dp}[r+1][c+1])$
4. Base case: $\texttt{dp}[n+1][\ldots] = 0$

Code

```cpp
#include <bits/stdc++.h>
using namespace std;

const int ROW = 100;
const int MAXN = 100 + 10, UNCALCULATED = -1;
int dp[MAXN][MAXN];
int v[MAXN][MAXN];

int calc(int r, int c) {
    if(r == ROW + 1)return 0;
    if(dp[r][c] != UNCALCULATED)return dp[r][c];
    dp[r][c] = v[r][c] + max(calc(r + 1, c), calc(r + 1, c + 1));
    return dp[r][c];
}

int main() {
    for(int r = 0; r < MAXN; r++) {
        for(int c = 0; c < MAXN; c++) {
            dp[r][c] = UNCALCULATED;
        }
    }
    for(int r = 1; r <= ROW; r++) {
        for(int c = 1; c <= r; c++) {
            cin >> v[r][c];
        }
    }
    cout << calc(1, 1) << endl;
    return 0;
}
```

5 Knapsacks

5.1 0/1 Knapsack

Thinking process

1. Find the maximum sum value can be picked for the first i items under capacity constraint of j. Denote as $\texttt{dp}[i][j]$.
2. Parameters: i, j
3. Transfer: $\texttt{dp}[i][j] = \max(v[i] + \texttt{dp}[i-1][j-w[i]], \texttt{dp}[i-1][j])$
4. Base case: $\texttt{dp}[0][\ldots] = 0, \texttt{dp}[\ldots][-j] = -\infty$

code

```
#include <bits/stdc++.h>
using namespace std;

const int UNCALCULATED = -1;
const int MAXN = 100 + 10, MAXW = 100000 + 10;
long dp[MAXN][MAXW];
long v[MAXN], w[MAXN];

long calc(int i, int j) {
    if(j < 0)return -1e9;
    if(i == 0)return 0;
    if(dp[i][j] != UNCALCULATED)return dp[i][j];
    return dp[i][j] = max(v[i] + calc(i - 1, j - w[i]), calc(i - 1, j));
}

int main() {
    for(int i = 0; i < MAXN; i++) {
        for(int j = 0; j < MAXW; j++) {
            dp[i][j] = UNCALCULATED;
        }
    }
    int n, total;
    cin >> n >> total;
    for(int i = 1; i <= n; i++)cin >> w[i] >> v[i];
    cout << calc(n, total) << endl;
    return 0;
}
```

5.2 Another Knapsack

Thinking process

1. Minimum capacity required for the first i items to achieve a sum of j. Denote as $\texttt{dp}[i][j]$.
2. Parameters: i, j
3. Transfer: $\texttt{dp}[i][j] = \min(w[i] + \texttt{dp}[i-1][j-v[i]], \texttt{dp}[i-1][j])$
4. Base case: $\texttt{dp}[0][0] = 0, \texttt{dp}[0][* \setminus \{0\}] = \infty$

code

```
#include <bits/stdc++.h>
using namespace std;

const int UNCALCULATED = -1;
const int MAXN = 100 + 10, MAXV = 100 * 1000 + 10;
long dp[MAXN][MAXV];
long v[MAXN], w[MAXN];
long calc(int i, int j) {
    if(j < 0)return 2e9;
    if(i == 0)
        if(j == 0)return 0;
        else return 2e9;
    if(dp[i][j] != UNCALCULATED)return dp[i][j];
    return dp[i][j] = min(w[i] + calc(i - 1, j - v[i]), calc(i - 1, j));
}

int main() {
    for(int i = 0; i < MAXN; i++) {
        for(int j = 0; j < MAXV; j++) {
            dp[i][j] = UNCALCULATED;
        }
    }
    int n, total;
    cin >> n >> total;
    for(int i = 1; i <= n; i++)cin >> w[i] >> v[i];
    for(int j = MAXV-1;j>=0;j--){
        if(calc(n,j) <= total){
            cout<<j<<endl;
            return 0;
        }
    }
    return 0;
}
```