Competitive
Programming and
Mathematics
Society

Programming Workshop 4

Dynamic Programming II

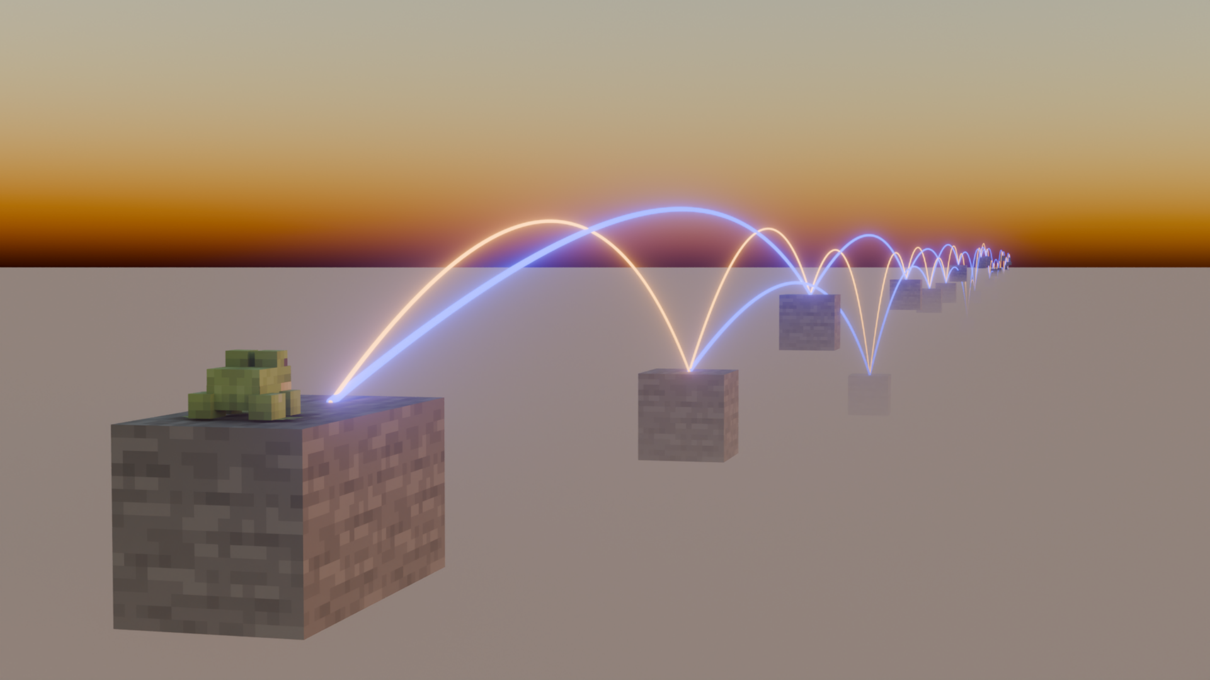
Bharat, Yiheng, Miles

Table of contents

Frog Jump

Source: https://atcoder.jp/contests/dp/tasks/dp_a.

- Consider a frog jumping along a sequence of n stones. Stone i has a height of h_i. The cost of jumping between two stones is the difference in heights between the stones.
- The frog starts on stone 1. On each turn, they can jump to either the next stone or the stone after the next stone.
- What is the minimum cost to jump to the last stone?



Frog Jump

- Let $\text{DP}[i]$ = the minimum cost to get from stone 1 to stone i.

Frog Jump

- Let $\text{DP}[i]$ = the minimum cost to get from stone 1 to stone i.
- There are two ways we could have arrived at stone i: from stone $i-1$ or from stone $i-2$.

Frog Jump

- Let DP$[i]$ $=$ the minimum cost to get from stone 1 to stone i.
- There are two ways we could have arrived at stone i: from stone $i-1$ or from stone $i-2$.
- Therefore, we can compute DP$[i] = \min($DP$[i-1] + |h_i - h_{i-1}|,$ DP$[i-2] + |h_i - h_{i-2}|)$.

Frog Jump

Consider the input where $n = 6$ and the heights are:

3	1	4	1	5	9

We initialise the DP table:

0	2				

Frog Jump

Consider the input where $n = 6$ and the heights are:

3	1	4	1	5	9

We initialise the DP table:

0	2				

Then we compute the DP:

0	2	1			

Frog Jump

Consider the input where $n = 6$ and the heights are:

3	1	4	1	5	9

We initialise the DP table:

0	2				

Then we compute the DP:

0	2	1			
0	2	1	2		

Frog Jump

Consider the input where $n = 6$ and the heights are:

3	1	4	1	5	9

We initialise the DP table:

0	2				

Then we compute the DP:

0	2	1			
0	2	1	2		
0	2	1	2	2	

Frog Jump

Consider the input where $n = 6$ and the heights are:

3	1	4	1	5	9

We initialise the DP table:

0	2				

Then we compute the DP:

0	2	1			
0	2	1	2		
0	2	1	2	2	
0	2	1	2	2	6

Frog Jump

Consider the input where $n = 6$ and the heights are:

3	1	4	1	5	9

We initialise the DP table:

0	2				

Then we compute the DP:

0	2	1			
0	2	1	2		
0	2	1	2	2	
0	2	1	2	2	6

Thus, the answer is 6.

Frog Jump Implementation

```
int cache[MAX_N];
int best(int n, int *h) {
    cache[0] = 0;
    cache[1] = abs(h[1] - h[0]);
    for (int i=2; i<n; i++) {
        cache[i] = min(cache[i-1] + abs(h[i] - h[i-1]),
                       cache[i-2] + abs(h[i] - h[i-2]));
    }
    return cache[n-1];
}
```

Frog Jump Implementation

UNSW CPMSoc

```
int h[MAX_N];
bool seen[MAX_N];
int cache[MAX_N];
int dp(int i) {
    if (i==0) return 0;
    if (i==1) return abs(h[i] - h[i-1]);
    if (seen[i]) return cache[i];
    seen[i] = true;
    return cache[i] = min(dp(i-1) + abs(h[i] - h[i-1]),
                          dp(i-2) + abs(h[i] - h[i-2]));
}
```

Frog Jump

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

Extension: what if we could jump more than 2 steps at once? Assume the frog can jump between 1 and k steps each turn. How would this change the DP? What would be the new time complexity?

Longest Increasing Subsequence

- Given an array a of length n, find the longest subsequence of the array such that the elements in the subsequence are strictly increasing.

Longest Increasing Subsequence

- Given an array a of length n, find the longest subsequence of the array such that the elements in the subsequence are strictly increasing.
- Let l_i denote the longest increasing subsequence of the first i elements of the array that includes a_i.

Longest Increasing Subsequence

UNSW CPMSoc

- Given an array a of length n, find the longest subsequence of the array such that the elements in the subsequence are strictly increasing.
- Let l_i denote the longest increasing subsequence of the first i elements of the array that includes a_i.
- Assuming we know $l_1, l_2, ..., l_{i-1}$, to calculate l_i, we can iterate over all indices $0 \leq j < i$ such that $a_j < a_i$, and try extending the sequence ending at j by adding i.
- The maximum of these will be the value of l_i.

Longest Increasing Subsequence

- By computing this naively, we end up with a time complexity of $\mathcal{O}(n^2)$.

Longest Increasing Subsequence

- By computing this naively, we end up with a time complexity of $\mathcal{O}(n^2)$.
- Extension: if we use a range tree or sorted stack, we can calculate each step in $\mathcal{O}(\log(N))$, resulting in an overall time complexity of $\mathcal{O}(N \log(N))$.

LIS Implementation

```
int cache[MAX_N];
int lis(int *data, int n) {
  for (int i=0; i<N; i++) {
    int best = 1;
    for (int j=0; j<i; j++) {
       if (data[j] < data[i]) best = max(best, cache[j]+1);
    }
    cache[i] = best;
  }
  int best = 0;
  for (int i=0; i<n; ++i) best = max(best, cache[i]);
  return best;
}
```

Longest Common Subsequence

- Given two arrays a and b, find the longest sequence that is a subsequence of both a and b.

Longest Common Subsequence

- Given two arrays a and b, find the longest sequence that is a subsequence of both a and b.
- Currently, we have only considered problems where our DP state is one-dimensional.
- Let's define DP$[i][j]$ to be the longest common subsequence of the first i elements of a and the first j elements of b.

Calculating the DP Recurrence

- Let's define DP$[i][j]$ to be the longest common subsequence of the first i elements of a and the first j elements of b.

Calling the DP Recurrence

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

- Let's define DP$[i][j]$ to be the longest common subsequence of the first i elements of a and the first j elements of b.
- Two cases:

Calculating the DP Recurrence

- Let's define DP$[i][j]$ to be the longest common subsequence of the first i elements of a and the first j elements of b.
- Two cases:
  - $a_i = b_j$, so DP$[i][j] = 1 +$ DP$[i-1][j-1]$.

Calculating the DP Recurrence

- Let's define DP$[i][j]$ to be the longest common subsequence of the first i elements of a and the first j elements of b.
- Two cases:
  - $a_i = b_j$, so DP$[i][j] = 1 +$ DP$[i-1][j-1]$.
  - $a_i \neq b_j$, so DP$[i][j] = \max($DP$[i-1][j],$ DP$[i][j-1])$.

Calculating the DP Recurrence

- Let's define DP$[i][j]$ to be the longest common subsequence of the first i elements of a and the first j elements of b.
- Two cases:
  - $a_i = b_j$, so DP$[i][j] = 1 +$ DP$[i-1][j-1]$.
  - $a_i \neq b_j$, so DP$[i][j] = \max($DP$[i-1][j],$ DP$[i][j-1])$.
- The recurrence is $\mathcal{O}(1)$, and there are $\mathcal{O}(NM)$ states, so our total complexity is $\mathcal{O}(NM)$.

LCS Implementation

```
int data_a[MAX_N];
int data_b[MAX_M];
bool seen[MAX_N][MAX_M];
int cache[MAX_N][MAX_M];
int dp(int i, int j) {
    if (i<0||j<0) return 0;
    if (seen[i][j]) return cache[i][j];
    seen[i][j] = true;
    if (i==j) return cache[i][j] = 1+dp(i-1, j-1);
    return cache[i][j] = max(dp(i-1, j), dp(i, j-1));
}
```

0/1 Knapsack

- You are robbing a house with N items, of which you select a subset to steal. Each item has a value v_i and a volume w_i. Given that the sum of the volumes of the items you steal cannot exceed V (the volume of your knapsack), what is the maximum sum of the values of the items that you can steal?
- Can we use a greedy algorithm?

0/1 Knapsack

- Define DP$[i][j]$ as the best value we can get from the first i items without exceeding the volume j.

0/1 Knapsack

- Define DP$[i][j]$ as the best value we can get from the first i items without exceeding the volume j.
- For each item, we either steal it or ignore it.

0/1 Knapsack

- Define $\text{DP}[i][j]$ as the best value we can get from the first i items without exceeding the volume j.
- For each item, we either steal it or ignore it.
- Thus, $\text{DP}[i][j] = \max(\text{DP}[i-1][j], \text{DP}[i-1][j-w_i])$.
- Remember to consider base cases!
- Our total time complexity becomes $\mathcal{O}(NV)$.

0/1 Knapsack Implementation

```
bool seen[N][V+1];
int cache[N][V+1];
int value[N];
int volume[N];
int dp(int i, int j) {
   if (i==-1) return 0;
   if (j<0) return INT_MIN;
   if (seen[i][j]) return cache[i][j];
   seen[i][j] = true;
   return cache[i][j] = std::max(dp(i-1,j), value[i] + dp(i-1, j-volume[i]));
}
```

Knapsack

Challenge: how can the DP be modified if there are unlimited copies of every object?

Subset Sum

- Given an array a of positive integers with length n and a positive integer k, can we select a subset of a with a sum exactly equal to k?
- This is actually very similar to knapsack! The only difference is that **we cannot leave empty space in our knapsack**

Subset Sum

- Given an array a of positive integers with length n and a positive integer k, can we select a subset of a with a sum exactly equal to k?
- This is actually very similar to knapsack! The only difference is that **we cannot leave empty space in our knapsack**
- By slightly modifying our knapsack algorithm, we can solve this in $\mathcal{O}(nk)$.
- This is often considered to be exponential time, because the size of the input is $\mathcal{O}(\log(k))$, so $\mathcal{O}(k)$ is exponential in the size of the input. Knapsack and Subset Sum are both NP-hard, so it is conjectured that there is no better-than-exponential algorithm to solve them.

Subset Sum Implementation

Note the difference between the two boolean arrays `seen` and `cache`. `seen` stores whether we have calculated a particular result, whereas `cache` stores whether we can fill our knapsack perfectly at that point.

```
bool seen[N][K+1];
bool cache[N][K+1];
int volume[N];
bool dp(int i, int j) {
    if (j==0) return true;
    if (i==-1) return false;
    if (j<0) return false;
    if (seen[i][j]) return cache[i][j];
    seen[i][j] = true;
    return cache[i][j] = dp(i-1,j) || dp(i-1, j-volume[i]);
}
```

Attendance form :D

SCAN ME

Further events

Please join us for:

- Computer Pizza Making Situated Online Contest (tomorrow!)
- DP III Workshop (11 April)
- Poker Night (11 April)