ICPC Workshop 2

Dynamic Programming

Angus Ritossa and Isaiah Iliffe

Table of contents

Maximum Non-Adjacent Subarray Sum

There is an array $[a_0, ..., a_{N-1}]$ of integers.
You can select some of these integers, but they must not be adjacent.
What is the maximum sum you can select?

Sample Input 1

```
7
5 3 1 4 1 2 1
```

Sample Output 1

```
11
```

Explanation

5 3 1 4 1 2 1

Sample Input 2

```
5
-1 -2 3 2 -5
```

Sample Output 2

```
3
```

Explanation

-1 -2 3 2 -5

Constraints

$N \leq 200\,000$

Solution Ideas

- We could try to tackle this with a greedy algorithm. For example, keep taking the biggest value if it isn't adjacent to something we have already taken.
  This doesn't work: consider the case `2 3 2`
- What about alternating (i.e. take every second element)?
  Also doesn't work, consider `5 1 1 5`
- There are many other solution ideas like these, and we can come up with breaking cases for all of them - we need a different approach.

Dynamic Programming

- We can solve this problem using a technique called **Dynamic Programming** (DP).
- The key aspect of DP is breaking down a problem into smaller subproblems which are easy to solve.

DP Solution

- Let $f(i)$ be the answer to the problem only considering the array $[a_i, ..., a_{N-1}]$.
  Example: `5 3 1 4 1 2 1`
  - $f(6) = 1$: `5 3 1 4 1 2 1`
  - $f(5) = 2$: `5 3 1 4 1 2 1`
  - $f(4) = 2$: `5 3 1 4 1 2 1` or `5 3 1 4 1 2 1`
  - $f(3) = 6$: `5 3 1 4 1 2 1`
  - $f(2) = 6$: `5 3 1 4 1 2 1`
  - $f(1) = 9$: `5 3 1 4 1 2 1`
  - $f(0) = 11$: `5 3 1 4 1 2 1`
- By definition, $f(0)$ is the answer to the problem
- $$f(i) = \begin{cases} 0 & \text{if } i \geq n \\ \max(a_i + f(i+2), f(i+1)) & \text{otherwise} \end{cases} \tag{1}$$

Code

```
#include <algorithm>
#include <cstdio>
using namespace std;
#define MAXN 200010
int N, a[MAXN];
int f(int i) {
  if (i >= N) return 0;
  return max(a[i] + f(i+2), f(i+1));
}
int main() {
  scanf("%d", &N);
  for (int i = 0; i < N; i++) scanf("%d", &a[i]);
  printf("%d\n", f(0));
}
```

Time Complexity

- What is the time complexity of this solution?
- The function $f(i)$ recursively calls $f(i+1)$ and $f(i+2)$, which leads to an exponential time solution.
- We can upper bound the time complexity by $O(2^n)$ (in fact, its $O(1.618^n)$, but the exact value isn't too important)
- Each time we call $f(i)$ for some fixed i it returns the same answer, so we can save the answer (this is called *memoisation*). This avoids doing the same thing multiple times and makes the complexity $O(N)$.

$O(N)$ Recursive DP

```
#include <algorithm>
#include <cstdio>
using namespace std;
#define MAXN 200010
int N, a[MAXN], memo[MAXN];
bool done[MAXN];
int f(int i) {
  if (i >= N) return 0;
  if (done[i]) return memo[i];
  done[i] = true;
  return memo[i] = max(a[i] + f(i+2), f(i+1));
}
int main() {
  scanf("%d", &N);
  for (int i = 0; i < N; i++) scanf("%d", &a[i]);
  printf("%d\n", f(0));
}
```

Recursion vs Iteration

- The code so far has been recursive, but you can write a DP iteratively as well.
- In most DP problems, recursive and iterative solutions both work and its a matter of preference. There are a few cases where one is preferred over the other.
- In problems where memory optimisations are needed, iterative is generally better
- In problems where there are many unreachable states, recursive is generally better because it will not visit these states.

Iterative DP

```
#include <algorithm>
#include <cstdio>
using namespace std;
#define MAXN 200010
int N, a[MAXN], dp[MAXN];
int main() {
  scanf("%d", &N);
  for (int i = 0; i < N; i++) scanf("%d", &a[i]);
  for (int i = N-1; i >= 0; i--) {
     dp[i] = max(a[i] + dp[i+2], dp[i+1]);
  }
  printf("%d\n", dp[0]);
}
```

Grid Walk

There is a $N \times N$ grid of integers.
You start at the top-left, and wish to walk to the bottom right. You can only walk down and right to an adjacent cell (not diagonal). The score of a walk is the sum of the values in all the cells (including start and end). What is the maximum score of any walk from the top-left to bottom right?

Sample Input

```
3
3 4 2
1 -1 1
1 6 4
```

Sample Output

```
16
```

Explanation

```
3
3 4 2
1 -1 1
1 6 4
```

Constraints $N \leq 2\,000$

Solution

- We will use DP.
- Let $f(i,j)$ be the best path to the end $(n-1,n-1)$ if we start at (i,j). The top-left is cell $(0,0)$, so the answer is $f(0,0)$.
- $$f(i,j) = \begin{cases} -\infty & i \geq N \text{ or } j \geq N \\ a_{n-1,n-1} & i = N-1 \text{ and } j = N-1 \\ a_{i,j} + \max(f(i+1,j), f(i,j+1)) & \text{otherwise} \end{cases} \tag{2}$$
- The time complexity is $O(N^2)$, because there are $O(N^2)$ states each with $O(1)$ recurrence.

```cpp
#include <algorithm>
#include <cstdio>
using namespace std;
#define MAXN 2020
int N, a[MAXN][MAXN], memo[MAXN][MAXN], done[MAXN][MAXN];
int f(int i, int j) {
  if (i >= N || j >= N) return -1e9;
  if (i == N-1 && j == N-1) return a[i][j];
  if (done[i][j]) return memo[i][j];
  done[i][j] = true;
  return memo[i][j] = a[i][j] + max(f(i+1, j), f(i, j+1));
}
int main() {
  scanf("%d", &N);
  for (int i = 0; i < N; i++) for (int j = 0; j < N; j++)
    scanf("%d", &a[i][j]);
  printf("%d\n", f(0, 0));
}
```

Knapsack

You have N items, each with a weight w_i and value v_i.
Your backpack has a weight limit of W. What is the maximum value of items you can fit into your backpack, without exceeding the weight limit?
Input format is N W on the first line, followed by N lines of the form w_i v_i.

Sample Input

```
4 10
4 3
3 2
6 3
2 2
```

Sample Output

```
7
```

Explanation
Take the first, second and fourth items for a total weight of $4 + 3 + 2 = 9 \leq 10$ and value $3 + 2 + 2 = 7$

Constraints $N \leq 2\,000$. $W \leq 5\,000$.

Solution

- Our DP state needs to consider the weight of the items, and which items we have taken.
- Let $dp(i, w)$ be the maximum value of items if their total weight is w and we have only selected items from item 0 to item $i - 1$.
- We will use a forward-pushing dp. So far, we have used backwards dp (we calculate the result of a state based on already calculated states). In a forwards dp, you update the results of future states using current states. This only works for iterative DPs.

Solution

- Initially, every $dp[i][w]$ is set to $-\infty$, except $dp[0][0]$ which is set to 0.
- When we process a state, its value is correct. We update future states as follows
  - $dp[i+1][w] = \max(dp[i+1][w], dp[i][w])$. This represents not adding item i to our backpack.
  - $dp[i+1][w+w_i] = \max(dp[i+1][w+w_i], dp[i][w]+v_i)$. This represents adding item i to our backpack.
- The answer is $\max(dp[N][0], dp[N][1], ..., dp[N][W])$.
- The time complexity is $O(NW)$, because there are $O(NW)$ states each with $O(1)$ recurrence.
- We reduce the memory usage by storing $dp[w]$ rather than $dp[i][w]$. This works because $dp[i+1][w] = \max(dp[i+1][w], dp[i][w])$. We need to iterate backwards (from $W-1$ to 0) in the inner loop to avoid taking an item twice.

```
#include <algorithm>
#include <cstdio>
using namespace std;
#define MAXN 5010
int N, W, w[MAXN], v[MAXN], dp[MAXN], ans;
int main() {
  scanf("%d%d", &N, &W);
  for (int i = 0; i < N; i++) scanf("%d%d", &w[i], &v[i]);
  for (int i = 0; i < N; i++) {
    for (int weight = W-1; weight >= 0; weight--) {
      if (weight+w[i] <= W) { // check we won't overflow the array
        dp[weight+w[i]] = max(dp[weight+w[i]], dp[weight]+v[i]);
      }
    }
  }
  for (int weight = 0; weight <= W; weight++) ans = max(ans, dp[weight]);
  printf("%d\n", ans);
}
```

Palindromes

There is a string $s = s_1 s_2 ... s_N$.
You must answer Q queries. In each query, you are given two integers l_i and r_i ($l_i \leq r_i$) and must answer how many substrings $s_x s_{x+1} ... s_y$ where $l_i \leq x \leq y \leq r_i$ are palindromes (a palindrome is the same forwards and backwards).

Sample Input

```
caaaba
5
1 1
1 4
2 3
4 6
4 5
```

Sample Output

```
1
7
3
4
2
```

Explanation

Fourth query: `a` (4, 4), `b` (5, 5), `a` (6, 6), `aba` (4, 6).

Constraints $N \leq 5\,000$. $Q \leq 1\,000\,000$.

Solution

- First: for each substring, how do we know if its a palindrome?
- A simple way is to check each substring on its own. The time complexity of this $O(N^2) \times O(N) = O(N^3)$ which is too slow.
- A faster way uses DP

$$\text{is\_palindrome}(i, j) = \begin{cases} \text{true} & i = j \\ s_i == s_j & i + 1 = j \\ s_i == s_j \text{ AND is\_palindrome}(i+1, j-1) & \text{otherwise} \end{cases} \tag{3}$$

- This is $O(N^2)$, which is fast enough.

```cpp
// include <algorithm>, <cstdio> and <cstring>
using namespace std;
#define MAXN 5010
char s[MAXN];
bool is_palindrome[MAXN][MAXN];
int dp[MAXN][MAXN], N, Q;
int main() {
  scanf(" %s", s+1); // str+1 1-indexes the string
  N = strlen(s+1);
  // Calculate is_palindrome
  // We need to process substrings in order of length
  for (int len = 0; len < N; len++) {
    for (int i = 1; i <= N-len; i++) {
      int j = i+len;
      if (!len) is_palindrome[i][i] = true;
      else if (len == 1) is_palindrome[i][j] = s[i] == s[j];
      else is_palindrome[i][j] = s[i] == s[j] && is_palindrome[i+1][j-1];
    }
  }
  // TODO: Rest of the solution
```

Solution

- Now, we will look at the actual problem - counting the number of palindromes in a range.
- $O(N^2)$ per query: use is_palindrome on every substring. This is too slow.
- We can do a DP which utilises inclusion-exclusion. $dp(i, j)$ is the number of palindromes in the range (i, j).
- $dp(i, j) =$

$$\begin{cases} 0 & j < i \\ \text{is\_palindrome}(i, j) + dp(i+1, j) + dp(i, j-1) - dp(i+1, j-1) & \text{otherwise} \end{cases} \tag{4}$$

```
  // The first half of the code is on an earlier slide

  // Calculate dp
  // We need to process substrings in order of length
  for (int len = 0; len < N; len++) {
    for (int i = 1; i <= N-len; i++) {
      int j = i+len;
      dp[i][j] = is_palindrome[i][j] + dp[i+1][j] + dp[i][j-1] - dp[i+1][j-1];
    }
  }

  // Print answers to queries
  scanf("%d", &Q);
  for (int i = 0; i < Q; i++) {
    int l, r;
    scanf("%d%d", &l, &r);
    printf("%d\n", dp[l][r]);
  }
}
```

Lab

- Join the vjudge group: https://vjudge.net/group/unswicpc
- Go to the contest for this workshop
- If you need help, or don't know what to do, message me or Isaiah
- **A: A+B** — solve this first if you haven't used vjudge before
- **B: Frog 1** — a simple DP, similar to the subarray sum problem
- **C: Vacation** — another DP problem, different to the problems from today
- **D: The least round way** — similar to the grid problem from today, but with a twist
- **E: Antimatter** — A harder DP problem
- **Photo** — Extension Problem. Available here: http://ceoi.inf.elte.hu/probarch/09/photo.pdf