Dynamic Programming

Emmanuel Anaya Gonzalez

Meeting 8. May 28, 2024

Competitive Programming Club @ UCSD

What is DP?

- Wikipedia: “...simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner."

General form of a DP solution

1. Define subproblems (original problem usually a particular case)
2. Formulate recursive relation between subproblems
3. Solve base cases
4. Cache subproblem solutions

Dynamic programming formulation

```
map<problem, value> memory;

value dp(problem P) {
    if (is_base_case(P)) {
        return base_case_value(P);
    }

    if (memory.find(P) != memory.end()) {
        return memory[P];
    }

    value result = some value;
    for (problem Q in subproblems(P)) {
        result = combine(result, dp(Q));
    }

    memory[P] = result;
    return result;
}
```

The staircase problem

Alice is at the bottom of a staircase with n steps. She is able to climb either 1 or 2 steps at the time.

Problem: How many different ways are there for Alice to get to the top of the staircase?

For $\mathtt{n} = 5$, she could step on $[1, 3, 5]$, or $[2, 4, 5]$, or $[2, 3, 4, 5]$. All of these count as different solutions.

1. Define subproblems (original problem should be particular case)

Let $\mathtt{ways(i)} = \#$ of ways to get to step $\mathtt{i}$

Then the solution to the original task is $\mathtt{ways(n)}$

The staircase problem

2. Formulate recursive relation between subproblems
    - Let's say Alice is at step $\mathtt{i}$
    - The previous step must have been either $\mathtt{i}-1$ or $\mathtt{i}-2$
    - If we know $\mathtt{ways}(\mathtt{i}-1)$ and $\mathtt{ways}(\mathtt{i}-2)$, then we can obtain $\mathtt{ways}(\mathtt{i})$

$$\mathtt{ways}(\mathtt{i}) = \mathtt{ways}(\mathtt{i}-1) + \mathtt{ways}(\mathtt{i}-2)$$

3. Solve base cases

$$\mathtt{ways}(1) = \mathtt{ways}(2) = 1$$

The staircase problem

```c
int ways(int n) {
    if (n <= 2) {
        return 1;
    }

    int res = ways(n - 2) + ways(n - 1);

    return res;
}
```

The Fibonacci sequence

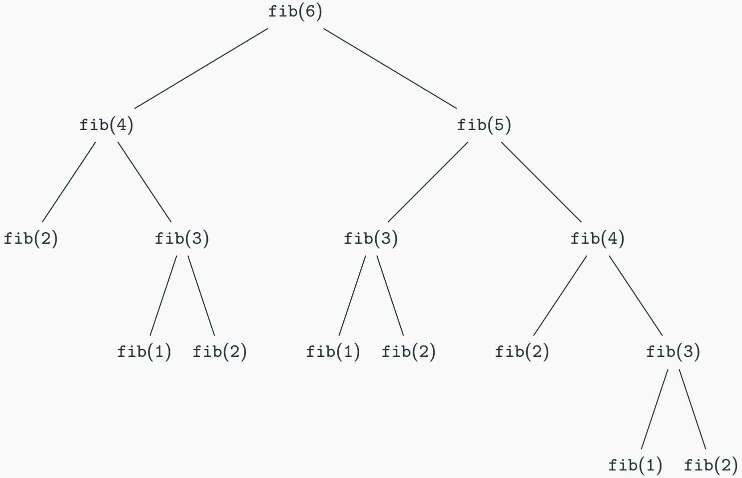
```c
int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

    int res = fibonacci(n - 2) + fibonacci(n - 1);

    return res;
}
```

The Fibonacci sequence

- What is the time complexity of this? Exponential, almost $O(2^n)$



The Fibonacci sequence

4. Cache subproblem solutions

```
int dp[1000];
for (int i = 0; i < 1000; i++)
    dp[i] = -1;

int fibonacci(int n) {
    if (n <= 2) return 1;

    if (dp[n] != -1) return dp[n];

    dp[n] = fibonacci(n - 2) + fibonacci(n - 1);

    return dp[n];
}
```

Total time complexity is `O(n)`

Longest increasing subsequence

Given a sequence of numbers $\mathtt{a} = \mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n}$, we say $\mathtt{a}'$ is a *subsequence* of a if it can be obtained by deleting some (possible zero) elements from a.

- Example: $\mathtt{a} = [5, 1, 8, 1, 9, 2]$
- $[5, 8, 9]$ is a subsequence
- $[1, 1]$ is a subsequence
- $[5, 1, 8, 1, 9, 2]$ is a subsequence
- $[]$ is a subsequence
- $[8, 5]$ is **not** a subsequence
- $[10]$ is **not** a subsequence

Longest increasing subsequence

An increasing sequence is such that the elements are in (strictly) increasing order

Problem: What is the length the Longest Increasing Subsequence (LIS) of `a`.

- $[5, 8, 9]$ and $[1, 8, 9]$ are the longest increasing subsequences of $\texttt{a} = [5, 1, 8, 1, 9, 2]$

Naive algorithm: There are $2^{\texttt{n}}$ subsequences, check if each is increasing. Worst case complexity is $\texttt{O}(\texttt{n}2^{\texttt{n}})$

What about dynamic programming?

Longest increasing subsequence

1. Define subproblems

Let $\mathtt{dp(i)}$ = length of LIS ending at element $\mathtt{i}$.

Our original task is then $\mathtt{max_i}\{\mathtt{dp(i)}\}$

2. Formulate recurrence relation

- Let's say we want to use elements $\mathtt{a_i}$ as the last element of an IS.
- We can do so for previous ISs that end in an element **smaller** than $\mathtt{a_i}$.

$\mathtt{dp(i)} = 1 + \mathtt{max}_{\mathtt{j}<\mathtt{i},\mathtt{a_j}<\mathtt{a_i}}\{\mathtt{dp(j)}\}$

3. Solve base cases

$\mathtt{dp(0)} = 0$

4. Cache subproblem results.

Longest increasing subsequence

```
int a[1000], dp[1000];
memset(dp, -1, sizeof(dp));

int lis(int i) {
    if (dp[i] != -1) return dp[i];

    int res = 1;
    for (int j = 0; j < i; j++)
        if (a[j] < a[i])
            res = max(res, 1 + lis(j));

    return dp[i] = res;
}
int main(){
    int mx = 0;
    for (int i = 0; i < n; i++)
        mx = max(mx, dp(i));
    printf("%d\n", mx);
}
```

New time complexity $O(n^2)$

0-1 Knapsack

We have knapsack with maximum capacity $\mathtt{W}$.

There are $\mathtt{n}$ available items. The $\mathtt{i}$-th item has weight $\mathtt{w_i}$ and gives us a value $\mathtt{v_i}$.

Problem: What is the maximum value we can hold in our knapsack?

- Example: $\mathtt{n} = 4, \mathtt{W} = 10, \mathtt{w} = [20, 5, 50, 40], \mathtt{v} = [1, 3, 7, 8]$
- $[3, 4]$ doesn't work, weight is $15 > 10$.
- $[2, 3]$ fits in the knapsack, but value 55 is not optimal.
- $[1, 4]$ is the solution with value 60.

0-1 Knapsack

Naive algorithm: For each subset, check weight $\leq \texttt{W}$, keep max. Complexity $\texttt{O}(\texttt{n}2^{\texttt{n}})$.

We can do better with DP.

1. Define subproblems

Let `dp(i, j)` = maximum value we can obtain with the first `i` items and maximum weight `j`.

The solution to the original task is `dp(n, W)`.

0-1 Knapsack

2. Formulate recursive relation
    - For given $\mathtt{i}, \mathtt{j}$, we can decide to take object $\mathtt{i}$ or not to.
    - If we don't take it, we directly reuse the solution of $\mathtt{i} - 1$.
    - If we do take it, we can improve our solution by $\mathtt{v}_\mathtt{i}$, but we now have to query for $\mathtt{j} - \mathtt{w}_\mathtt{i}$.

$$\mathtt{dp}(\mathtt{i}, \mathtt{j}) = \max(\mathtt{dp}(\mathtt{i} - 1, \mathtt{j}), \quad \mathtt{v}_\mathtt{i} + \mathtt{dp}(\mathtt{i} - 1, \mathtt{j} - \mathtt{w}_\mathtt{i}))$$

3. Solve base cases

$$\mathtt{dp}(0, \mathtt{j}) = \mathtt{dp}(\mathtt{i}, 0) = 0$$

4. Cache subproblem solutions

0-1 Knapsack

```
int n, W, w[1000], v[1000], dp[1000][1000];
memset(dp, -1, sizeof(dp));

int ks(i, j) {

    if (!i || !j) return 0;

    if (dp[i][j] != -1) return dp[i][j];

    dp[i][j] = max(
        ks(i - 1, j),
        ks(i - 1, j - w[i]) + v[i]
    );

    return dp[i][j];
}
```

New complexity is `O(nW)`.

One issue: stack size

Top-down vs Bottom-up

- Top-down
    - Direct from recursive definition
    - Stack usage impacts performance
- Bottom-up
    - Implementation can get non-trivial
    - `O(1)` stack usage

0-1 Knapsack

```c
int n, W, w[1000], v[1000], dp[1000][1000];
memset(dp, 0, sizeof(dp));

int ks() {
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= W; j++) {
            dp[i][j] = max(
                ks(i - 1, j),
                ks(i - 1, j - w[i]) + v[i]
            );
        }
    }

    printf("%d\n", dp[i][j]);
}
```

Longest common subsequence

Problem: Given 2 sequences $\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{m}$ and $\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{m}$, find the lenth of the longest subsequence they have in common.

$\mathtt{a} =$"$\mathtt{b\underline{an}an\underline{inn}}$"

$\mathtt{b} =$"$\mathtt{k\underline{anin}a\underline{n}}$"

The longest common subsequence of `a` and `b`, `"aninn"`, has length 5.

1. Define subproblems

Let $\mathtt{lcs}(\mathtt{i}, \mathtt{j}) =$ length of LCS of $\mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{i}$ and $\mathtt{b}_1, \ldots, \mathtt{b}_\mathtt{j}$.

Our original task is exactly $\mathtt{lcs}(\mathtt{n}, \mathtt{m})$.

Longest common subsequence

2. Formulate recursive relation
   - When looking at elements $\mathtt{i}, \mathtt{j}$ we can decide to match them or not, if they coincide.
   - If they don't we just reuse the subproblem solution.

$$\mathrm{lcs}(\mathtt{i}, \mathtt{j}) = \max \begin{cases} \mathrm{lcs}(\mathtt{i}, \mathtt{j} - 1) \\ \mathrm{lcs}(\mathtt{i} - 1, \mathtt{j}) \\ \mathrm{lcs}(\mathtt{i} - 1, \mathtt{j} - 1) + \mathtt{a}[\mathtt{i}] == \mathtt{b}[\mathtt{j}] \end{cases}$$

3. Solve base cases

$$\mathtt{lcs}(0, \mathtt{j}) = \mathtt{lcs}(\mathtt{i}, 0) = 0$$

4. Cache subproblem solutions

Longest common subsequence

```
string a = "bananinn",
       b = "kaninan";
int dp[1000][1000];
memset(dp, 0, sizeof(dp));

int lcs() {
    for(int i = 1; i <= a.size(); i++) {
        for(int j = 1; j <= b.size(); j++) {
            dp[i][j] = max(
                max(
                    dp[i - 1][j],
                    dp[i][j - 1]
                )
                dp[i - 1][j - 1] + a[i] == b[j]
            )
        }
    }
    printf("%d\n", dp[a.size()][b.size()]);
}
```

Time complexity is $O(nm)$

DP on trees

The diameter of a graph is the length of the longest simple path in it.
Problem: Given a rooted tree with `n` nodes, compute it's diameter.

1. Define subproblems

Let `f(i) = ` length of longest path from `i` to a descendant.

Let `g(i) = ` length of longest path rooted at `i`.

Since the longest path is rooted at some node, then the solution of the original task is $\max_i g(i)$

DP on trees

2. Formulate recursive relation

Let `c_1, ..., c_m` be the children if node `i`.

Then, $\mathtt{f}(\mathtt{i}) = 1 + \max_{\mathtt{j}}\{\mathtt{f}(\mathtt{c_j})\}$

Also, $\mathtt{g}(\mathtt{i}) = 1 + \max_{\mathtt{j} \neq \mathtt{k}}\{\mathtt{f}(\mathtt{c_j}) + \mathtt{f}(\mathtt{c_k})\}$

3. Solve base cases

`f(leaf) = g(leaf) = 1`

You should learn DP

- Subset sum
- Coin change
- DP on DAGs
- Edit distance
- Graph distances
- DP with bitmasks
- DP on digits