Dynamic Programming

Bjarki Ágúst Guðmundsson
Tómas Ken Magnússon

School of Computer Science
Reykjavík University

Árangursrík forritun og lausn verkefna

Today we're going to cover

- Dynamic Programming

What is dynamic programming?

- A problem solving paradigm
- Similar in some respects to both divide and conquer and backtracking
- Divide and conquer recap:
  - Split the problem into *independent* subproblems
  - Solve each subproblem recursively
  - Combine the solutions to subproblems into a solution for the given problem
- Dynamic programming:
  - Split the problem into *overlapping* subproblems
  - Solve each subproblem recursively
  - Combine the solutions to subproblems into a solution for the given problem
  - *Don't compute the answer to the same problem more than once*

Dynamic programming formulation

1. Formulate the problem in terms of smaller versions of the problem (recursively)
2. Turn this formulation into a recursive function
3. Memoize the function (remember results that have been computed)

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[Q] = result;
    return result;
}
```

The Fibonacci sequence

The first two numbers in the Fibonacci sequence are 1 and 1. All other numbers in the sequence are defined as the sum of the previous two numbers in the sequence.

- Task: Find the *n*th number in the Fibonacci sequence
- Let's solve this with dynamic programming

1. Formulate the problem in terms of smaller versions of the problem (recursively)

$$\text{fibonacci}(1) = 1$$
$$\text{fibonacci}(2) = 1$$
$$\text{fibonacci}(n) = \text{fibonacci}(n-2) + \text{fibonacci}(n-1)$$

The Fibonacci sequence

2. Turn this formulation into a recursive function

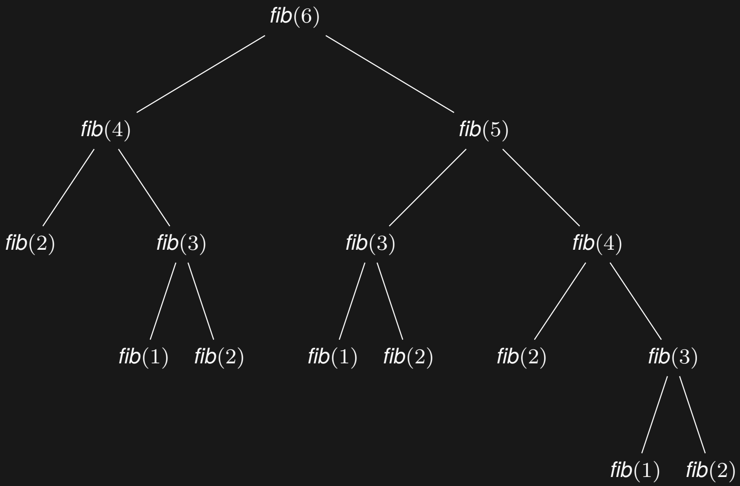
```
int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }

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

    return res;
}
```

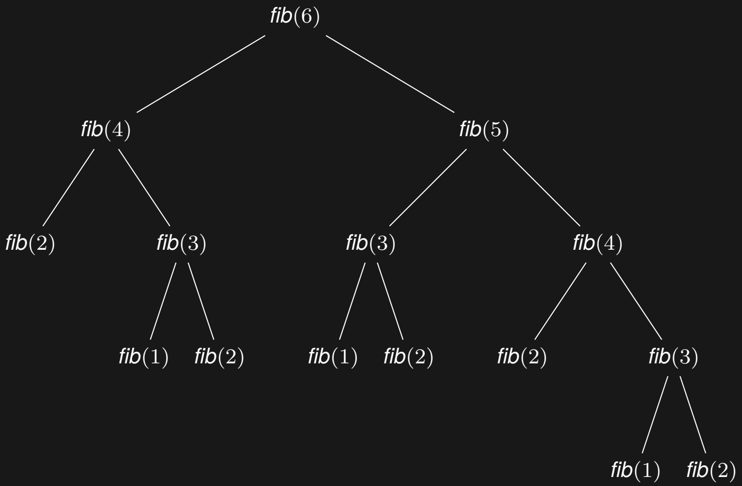
The Fibonacci sequence

- What is the time complexity of this?



The Fibonacci sequence

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



The Fibonacci sequence

3. Memoize the function (remember results that have been computed)

```
map<int, int> mem;

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

    if (mem.find(n) != mem.end()) {
        return mem[n];
    }

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

    mem[n] = res;
    return res;
}
```

The Fibonacci sequence

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

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

    if (mem[n] != -1) {
        return mem[n];
    }

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

    mem[n] = res;
    return res;
}
```

The Fibonacci sequence

- What is the time complexity now?
- We have n possible inputs to the function: $1, 2, \ldots, n$.
- Each input will either:
  - be computed, and the result saved
  - be returned from the memory
- Each input will be computed at most once
- Time complexity is $O(n \times f)$, where f is the time complexity of computing an input if we assume that the recursive calls are returned directly from memory ($O(1)$)
- Since we're only doing constant amount of work to compute the answer to an input, $f = O(1)$
- Total time complexity is $O(n)$

Maximum sum

- Given an array $\mathrm{arr}[0]$, $\mathrm{arr}[1]$, …, $\mathrm{arr}[n-1]$ of integers, find the interval with the highest sum

-15	8	-2	1	0	6	-3

Maximum sum

- Given an array $\mathrm{arr}[0], \mathrm{arr}[1], \ldots, \mathrm{arr}[n-1]$ of integers, find the interval with the highest sum

-15	8	-2	1	0	6	-3

- The maximum sum of an interval in this array is 13

Maximum sum

- Given an array $\mathrm{arr}[0], \mathrm{arr}[1], \ldots, \mathrm{arr}[n-1]$ of integers, find the interval with the highest sum

-15	8	-2	1	0	6	-3

- The maximum sum of an interval in this array is 13
- But how do we solve this in general?
  - Easy to loop through all $\approx n^2$ intervals, and calculate their sums, but that is $O(n^3)$
  - We could use our static range sum trick to get this down to $O(n^2)$
  - Can we do better with dynamic programming?

Maximum sum

- First step is to formulate this recursively
- Let $\mathrm{max\_sum}(i)$ be the maximum sum interval in the range $0, \ldots, i$
- Base case: $\mathrm{max\_sum}(0) = \max(0, arr[0])$
- What about $\mathrm{max\_sum}(i)$?
- What does $\mathrm{max\_sum}(i-1)$ return?
- Is it possible to combine solutions to subproblems with smaller i into a solution for i?
- At least it's not obvious...

Maximum sum

- Let's try changing perspective
- Let $\text{max\_sum}(i)$ be the maximum sum interval in the range $0, \ldots, i$, *that ends at i*
- Base case: $\text{max\_sum}(0) = arr[0]$
- $\text{max\_sum}(i) = \max(arr[i], arr[i] + \text{max\_sum}(i-1))$
- Then the answer is just $\max_{0 \leq i < n} \{ \text{max\_sum}(i) \}$

Maximum sum

- Next step is to turn this into a function

```
int arr[1000];

int max_sum(int i) {
    if (i == 0) {
        return arr[i];
    }

    int res = max(arr[i], arr[i] + max_sum(i - 1));

    return res;
}
```

Maximum sum

- Final step is to memoize the function

```
int arr[1000];
int mem[1000];
bool comp[1000];
memset(comp, 0, sizeof(comp));

int max_sum(int i) {
    if (i == 0) {
        return arr[i];
    }
    if (comp[i]) {
        return mem[i];
    }

    int res = max(arr[i], arr[i] + max_sum(i - 1));

    mem[i] = res;
    comp[i] = true;
    return res;
}
```

Maximum sum

- Then the answer is just the maximum over all interval ends

```
int maximum = 0;
for (int i = 0; i < n; i++) {
    maximum = max(maximum, best_sum(i));
}

printf("%d\n", maximum);
```

Maximum sum

- If you want to find the maximum sum interval in multiple arrays, remember to clear the memory in between

Maximum sum

- What about time complexity?
- There are n possible inputs to the function
- Each input is processed in $O(1)$ time, assuming recursive calls are $O(1)$
- Time complexity is $O(n)$

Coin change

- Given an array of coin denominations $d_0, d_2, \ldots, d_{n-1}$, and some amount x: What is minimum number of coins needed to represent the value x?
- Remember the greedy algorithm for Coin change?
- It didn’t always give the optimal solution, and sometimes it didn’t even give a solution at all...
- What about dynamic programming?

Coin change

- First step: formulate the problem recursively

- Let $\mathrm{opt}(i, x)$ denote the minimum number of coins needed to represent the value x if we're only allowed to use the coin denominations d_0, …, d_i

- Base case: $\mathrm{opt}(i, x) = \infty$ if $x < 0$
- Base case: $\mathrm{opt}(i, 0) = 0$
- Base case: $\mathrm{opt}(-1, x) = \infty$

- $\mathrm{opt}(i, x) = \min \begin{cases} 1 + \mathrm{opt}(i, x - d_i) \\ \mathrm{opt}(i - 1, x) \end{cases}$

Coin change

```
int INF = 100000;
int d[10];

int opt(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == -1) return INF;

    int res = INF;
    res = min(res, 1 + opt(i, x - d[i]));
    res = min(res, opt(i - 1, x));

    return res;
}
```

Coin change

```
int INF = 100000;
int d[10];
int mem[10][10000];
memset(mem, -1, sizeof(mem));

int opt(int i, int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (i == -1) return INF;

    if (mem[i][x] != -1) return mem[i][x];

    int res = INF;
    res = min(res, 1 + opt(i, x - d[i]));
    res = min(res, opt(i - 1, x));

    mem[i][x] = res;
    return res;
}
```

Coin change

- Time complexity?
- Number of possible inputs are $n \times x$
- Each input will be processed in $O(1)$ time, assuming recursive calls are constant
- Total time complexity is $O(n \times x)$

Coin change

- How do we know which coins the optimal solution used?
- We can store backpointers, or some extra information, to trace backwards through the states
- See example...

Longest increasing subsequence

- Given an array $a[0]$, $a[1]$, ..., $a[n-1]$ of integers, what is the length of the longest increasing subsequence?

- First, what is a subsequence?
- If we delete zero or more elements from a, then we have a subsequence of a
- Example: $a = [5, 1, 8, 1, 9, 2]$
- $[5, 8, 9]$ 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

- Given an array $a[0]$, $a[1]$, ..., $a[n-1]$ of integers, what is the length of the longest increasing subsequence?
- An increasing subsequence of a is a subsequence of a such that the elements are in (strictly) increasing order
- $[5, 8, 9]$ and $[1, 8, 9]$ are the longest increasing subsequences of $a = [5, 1, 8, 1, 9, 2]$
- How do we compute the length of the longest increasing subsequence?
- There are 2^n subsequences, so we can go through all of them
- That would result in an $O(n2^n)$ algorithm, which can only handle $n \leq 23$
- What about dynamic programming?

Longest increasing subsequence

- Let $\mathrm{lis}(i)$ denote the length of the longest increasing subsequence of the array $a[0], \ldots, a[i]$
- Base case: $\mathrm{lis}(0) = 1$
- What about $\mathrm{lis}(i)$?
- We have the same issue as in the maximum sum problem, so let's try changing perspective

Longest increasing subsequence

- Let $\mathrm{lis}(i)$ denote the length of the longest increasing subsequence of the array $a[0], \ldots, a[i]$, *that ends at i*
- Base case: we don't need one
- $\mathrm{lis}(i) = \max(1, \max_{j \text{ s.t. } a[j]<a[i]}\{1 + \mathrm{lis}(j)\})$

Longest increasing subsequence

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

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

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

    mem[i] = res;
    return res;
}
```

Longest increasing subsequence

- And then the longest increasing subsequence can be found by checking all endpoints:

```
int mx = 0;
for (int i = 0; i < n; i++) {
    mx = max(mx, lis(i));
}

printf("%d\n", mx);
```

Longest increasing subsequence

- Time complexity?

- There are n possible inputs
- Each input is computed in $O(n)$ time, assuming recursive calls are $O(1)$
- Total time complexity is $O(n^2)$

- This will be fast enough for $n \leq 10\ 000$, much better than the brute force method!

Longest common subsequence

- Given two strings (or arrays of integers) $a[0]$, …, $a[n-1]$ and $b[0]$, …, $b[m-1]$, find the length of the longest subsequence that they have in common.

- $a =$"b$\underline{\texttt{an}}$an$\underline{\texttt{inn}}$"
- $b =$"k$\underline{\texttt{anin}}a\underline{\texttt{n}}$"

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

Longest common subsequence

- Let $\mathrm{lcs}(i, j)$ be the length of the longest common subsequence of the strings $a[0]$, ..., $a[i]$ and $b[0]$, ..., $b[j]$
- Base case: $\mathrm{lcs}(-1, j) = 0$
- Base case: $\mathrm{lcs}(i, -1) = 0$
- $\mathrm{lcs}(i, j) = \max \begin{cases} \mathrm{lcs}(i, j-1) \\ \mathrm{lcs}(i-1, j) \\ 1 + \mathrm{lcs}(i-1, j-1) & \text{if } a[i] = b[j] \end{cases}$

Longest common subsequence

```
string a = "bananinn",
       b = "kaninan";
int mem[1000][1000];
memset(mem, -1, sizeof(mem));

int lcs(int i, int j) {
    if (i == -1 || j == -1) {
        return 0;
    }
    if (mem[i][j] != -1) {
        return mem[i][j];
    }

    int res = 0;
    res = max(res, lcs(i, j - 1));
    res = max(res, lcs(i - 1, j));

    if (a[i] == b[j]) {
        res = max(res, 1 + lcs(i - 1, j - 1));
    }

    mem[i][j] = res;
    return res;
}
```

Longest common subsequence

- Time complexity?

- There are $n \times m$ possible inputs
- Each input is processed in $O(1)$, assuming recursive calls are $O(1)$
- Total time complexity is $O(n \times m)$

DP over bitmasks

- Remember the bitmask representation of subsets?
- Each subset of n elements are mapped to an integer in the range $0, \ldots, 2^n - 1$
- This makes it easy to do dynamic programming over subsets

Traveling salesman problem

- We have a graph of n vertices, and a cost $c_{i,j}$ between each pair of vertices i, j. We want to find a cycle through all vertices in the graph so that the sum of the edge costs in the cycle is minimal.
- This problem is NP-Hard, so there is no known deterministic polynomial time algorithm that solves it
- Simple to do in $O(n!)$ by going through all permutations of the vertices, but that's too slow if $n > 11$
- Can we go higher if we use dynamic programming?

Traveling salesman problem

- Without loss of generality, assume we start and end the cycle at vertex 0
- Let $\mathrm{tsp}(i, S)$ represent the cheapest way to go through all vertices in the graph and back to vertex 0, if we're currently at vertex i and we've already visited the vertices in the set S
- Base case: $\mathrm{tsp}(i, \text{all vertices}) = c_{i,0}$
- Otherwise $\mathrm{tsp}(i, S) = \min_{j \notin S} \{ \, c_{i,j} + \mathrm{tsp}(j, S \cup \{j\}) \, \}$

Traveling salesman problem

```
const int N = 20;
const int INF = 100000000;
int c[N][N];
int mem[N][1<<N];
memset(mem, -1, sizeof(mem));

int tsp(int i, int S) {
    if (S == ((1 << N) - 1)) {
        return c[i][0];
    }
    if (mem[i][S] != -1) {
        return mem[i][S];
    }

    int res = INF;
    for (int j = 0; j < N; j++) {
        if (S & (1 << j))
            continue;

        res = min(res, c[i][j] + tsp(j, S | (1 << j)));
    }

    mem[i][S] = res;
    return res;
}
```

Traveling salesman problem

- Then the optimal solution can be found as follows:

```
printf("%d\n", tsp(0, 1<<0));
```

Traveling salesman problem

- Time complexity?
- There are $n \times 2^n$ possible inputs
- Each input is computed in $O(n)$ assuming recursive calls are $O(1)$
- Total time complexity is $O(n^2 2^n)$
- Now n can go up to about 20

Traveling salesman problem