# Dynamic Programming

**Aonghus Lawlor**

**Semester II**

**2016/17**

# 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

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

We typically apply dynamic programming to optimization problems.

Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.
When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1–3 form the basis of a dynamic-programming solution to a problem.

If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4. When we do perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

# 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)

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

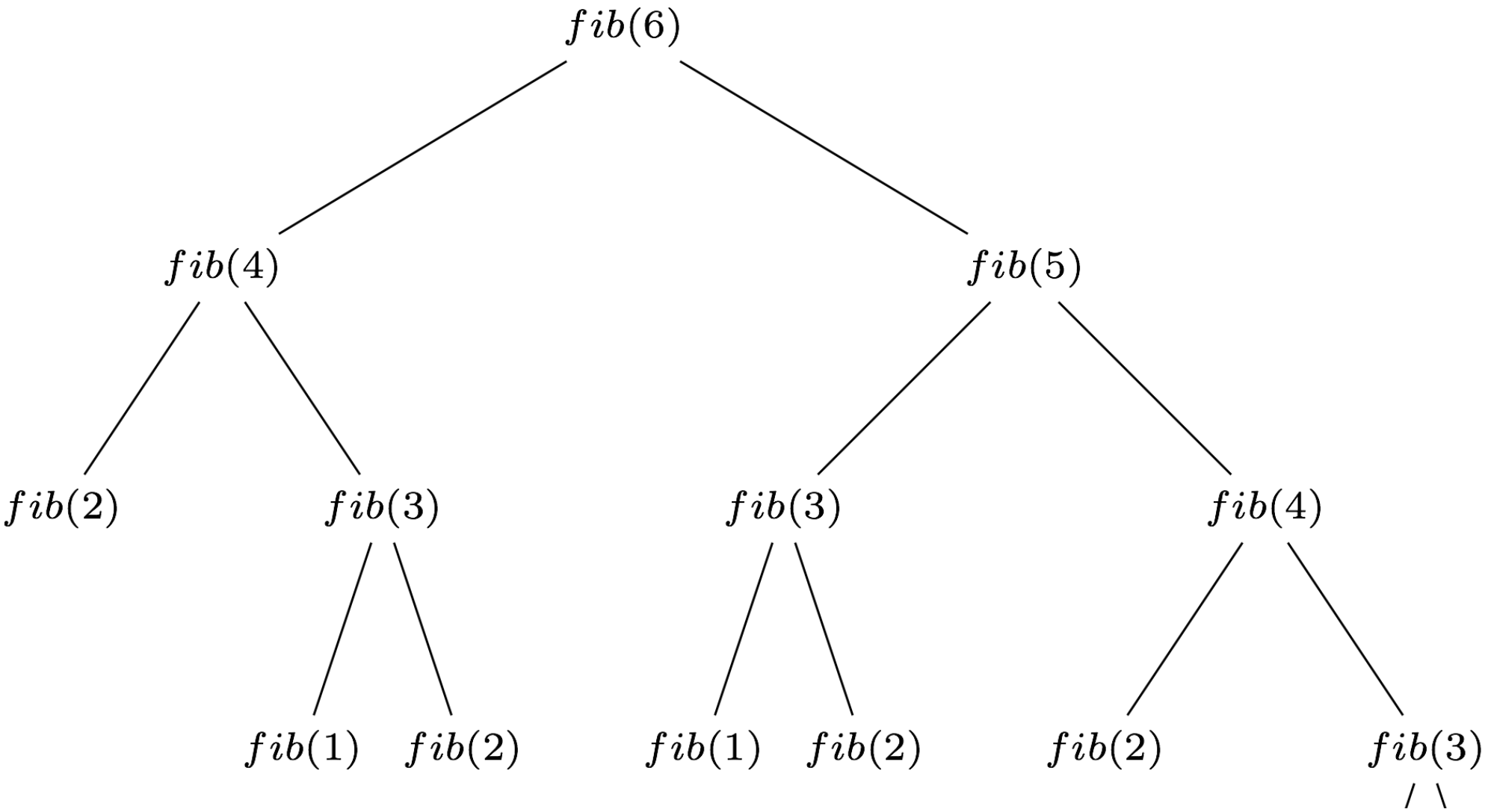
# The Fibonacci sequence

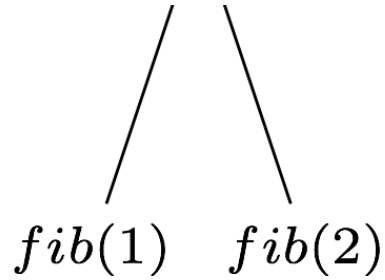
1. Turn this formulation into a recursive function

```
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?






Exponential complexity: $O(2^n)$

# The Fibonacci sequence

1. 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 $\text{arr}[0], \text{arr}[1], \ldots, \text{arr}[n-1]$ 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?
- First step is to formulate this recursively
- Let $\max\_\text{sum}(i)$ be the maximum sum interval in the range $0, \ldots, i$
- Base case: $\max\_\text{sum}(0) = \max(0, arr[0])$
- What about $\max\_\text{sum}(i)$?
- What does $\max\_\text{sum}(i-1)$ return?

- Is it possible to combine solutions to subproblems with smaller $i$ into a solution for $i$?
- At least it's not obvious…
- Let's try changing perspective
- Let $\max\_sum(i)$ be the maximum sum interval in the range $0, \ldots, i$, *that ends at* $i$
- Base case: $\max\_sum(0) = arr[0]$
- $\max\_sum(i) = \max(arr[i], arr[i] + \max\_sum(i-1))$
- Then the answer is just $\max_{0 \leq i < n} \{ \max\_sum(i) \}$
- 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;
}
```

- 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;
}
```

- 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);
```

- If you want to find the maximum sum interval in multiple arrays, remember to clear the memory in between
- 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 $\text{opt}(i, x)$ denote the minimum number of coins needed to represent the value $x$ if we're only allowed to use the coin denominations $d_0, \ldots, d_i$
- Base case: $\text{opt}(i, x) = \infty$ if $x < 0$
- Base case: $\text{opt}(i, 0) = 0$
- Base case: $\text{opt}(-1, x) = \infty$
- $\text{opt}(i, x) = \min \left\{ 1 + \text{opt}(i, x - d_i) \; \text{opt}(i - 1, x) \right.$

# 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;
}

```

```
def _get_change_making_matrix(set_of_coins, r):
    m = [[0 for _ in range(r + 1)] for _ in range(len(set_of_coins) + 1)]

    for i in range(r + 1):
        m[0][i] = i

```

```
    return m


def change_making(coins, n):

    """This function assumes that all coins are available infinitely.
    n is the number that we need to obtain with the fewest number of coins.
    coins is a list or tuple with the available denominations."""

    m = _get_change_making_matrix(coins, n)

    for c in range(1, len(coins) + 1):
        for r in range(1, n + 1):
            # Just use the coin coins[c - 1].
            if coins[c - 1] == r:
                m[c][r] = 1

            # coins[c - 1] cannot be included.
            # We use the previous solution for making r,
            # excluding coins[c - 1].
            elif coins[c - 1] > r:
                m[c][r] = m[c - 1][r]

            # We can use coins[c - 1].
            # We need to decide which one of the following solutions is the best:
            # 1. Using the previous solution for making r (without using coins[c - 1]).
            # 2. Using the previous solution for making r - coins[c - 1] (without using coins[c - 1]) plus this 1 extra coin.
            else:
                m[c][r] = min(m[c - 1][r], 1 + m[c][r - coins[c - 1]])

    return m[-1][-1]

```

# 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

# Dynamic Programming

- Canonical Problems
  - Shortest Paths (Floyd-Warshall)
  - Edit Distance
  - Coin Change
  - Longest Increasing Subsequence
  - Maximum Sum
  - Knapsack
  - Travelling salesman