

Dynamic Programming

"Those who do not remember the past are condemned to repeat it."

Miloš Trujić

First things first... **Problem of the Week: Deck of Cards** (simplified)

Input: n , k , and n non-negative integers v_0, v_1, \dots, v_{n-1}

Output: a pair (i, j) such that $0 \leq i \leq j \leq n - 1$ and $\sum_{\ell=i}^j v_{\ell} = k$

Example: $n = 6, k = 7$

3	1	4	1	1	8
---	---	---	---	---	---

Solution: $i = 1, j = 4$

Problem of the Week: Deck of Cards (simplified)

Input: n , k , and n non-negative integers v_0, v_1, \dots, v_{n-1}

Output: a pair (i, j) such that $0 \leq i \leq j \leq n - 1$ and $\sum_{\ell=i}^j v_{\ell} = k$

- ▶ Test set 1: $n \leq 200 \rightarrow$ time complexity $O(n^3)$ (e.g. just do it!)
- ▶ Test set 2: $n \leq 3000 \rightarrow$ time complexity $O(n^2)$ (e.g. partial sums)
- ▶ Test set 3: $n \leq 100000 \rightarrow$ time complexity $O(n \log n)$ (e.g. binary search)
- ▶ Test set 3: $n \leq 100000 \rightarrow$ time complexity $O(n)$

How to solve Deck of Cards in $O(n)$?

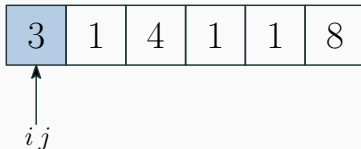
Sliding Window

How to solve Deck of Cards in $O(n)$? Sliding Window!

Idea:

- ▶ Keep two pointers that keep track of the current interval (window)
- ▶ If the value of the window is too large: increase the left pointer
- ▶ If the value of the window is too small: increase the right pointer

Example: $n = 6, k = 7$



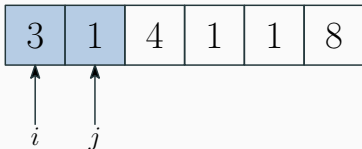
$v_0 = 3 < k \rightarrow$ increase j

How to solve Deck of Cards in $O(n)$? Sliding Window!

Idea:

- ▶ Keep two pointers that keep track of the current interval (window)
- ▶ If the value of the window is too large: increase the left pointer
- ▶ If the value of the window is too small: increase the right pointer

Example: $n = 6, k = 7$



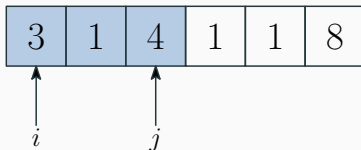
$$v_0 + v_1 = 4 < k \rightarrow \text{increase } j$$

How to solve Deck of Cards in $O(n)$? Sliding Window!

Idea:

- ▶ Keep two pointers that keep track of the current interval (window)
- ▶ If the value of the window is too large: increase the left pointer
- ▶ If the value of the window is too small: increase the right pointer

Example: $n = 6, k = 7$



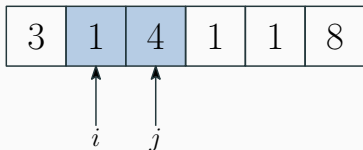
$$v_0 + v_1 + v_2 = 8 > k \longrightarrow \text{increase } i$$

How to solve Deck of Cards in $O(n)$? Sliding Window!

Idea:

- ▶ Keep two pointers that keep track of the current interval (window)
- ▶ If the value of the window is too large: increase the left pointer
- ▶ If the value of the window is too small: increase the right pointer

Example: $n = 6, k = 7$



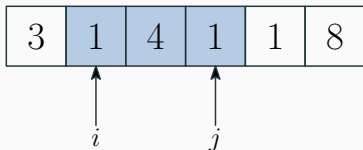
$$v_1 + v_2 = 5 < k \longrightarrow \text{increase } j$$

How to solve Deck of Cards in $O(n)$? Sliding Window!

Idea:

- ▶ Keep two pointers that keep track of the current interval (window)
- ▶ If the value of the window is too large: increase the left pointer
- ▶ If the value of the window is too small: increase the right pointer

Example: $n = 6, k = 7$



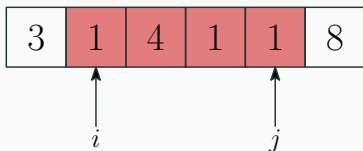
$$v_1 + v_2 + v_3 = 6 < k \longrightarrow \text{increase } j$$

How to solve Deck of Cards in $O(n)$? Sliding Window!

Idea:

- ▶ Keep two pointers that keep track of the current interval (window)
- ▶ If the value of the window is too large: increase the left pointer
- ▶ If the value of the window is too small: increase the right pointer

Example: $n = 6, k = 7$



$v_1 + v_2 + v_3 + v_4 = 7 = k \rightarrow \text{YAY!}$ The solution is $i = 1$ and $j = 4$

How to solve Deck of Cards in $O(n)$? Sliding Window!

```
int i = 0, j = 0;
int val = v[0];

while (j < n) {
    if (val == k) break;

    if (val < k) {
        j++;
        val += v[j];
    } else {
        val -= v[i];
        i++;
        if (i > j) {
            j = i;
            val = v[i];
        }
    }
}
```

Sketch of the proof:

- ▶ at every point we **increase** either i or j by **one** (so we eventually terminate)
- ▶ assume j first reaches the end of the '**target window**'; then i keeps increasing until it hits the start of it
- ▶ assume i first reaches the start of the '**target window**'; then j keeps increasing until it hits the end of it

Exercise: Extend it to solve the 'real' problem.

Trick/technique (Sliding window)

Some problems in which you need to find some **optimal interval** can be solved in linear time using a similar **sliding window** approach.

Let's get to the point!

- ▶ Most of you **know** Dynamic Programming (DP) :)
- ▶ Many struggle to **apply** it :(
 - ▶ How to identify a DP problem?
 - ▶ How to tackle it?
 - ▶ How to implement it?
- ▶ Today we **start from scratch**

Outline for today:

- ▶ Three examples (Fibonacci, Rod Cutting, LSI)
- ▶ Elements of Dynamic Programming on examples
- ▶ Common pitfalls
- ▶ Tips & Tricks

First Example: Fibonacci Numbers

Definition: $F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 3$.

Problem: compute F_n

Solution: transform the definition into a recursive algorithm

```
int F(int n) {  
    if (n == 1 || n == 2) return 1;  
    return F(n - 1) + F(n - 2);  
}
```

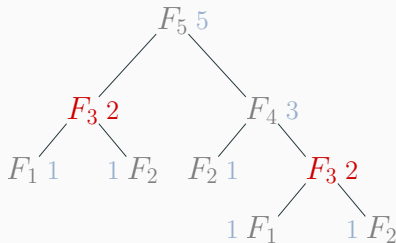
Time complexity: $\Theta(\varphi^n)$

Source of inefficiency?

???


```
int F(int n) {
    if (n == 1 || n == 2) return 1;
    return F(n - 1) + F(n - 2);
}
```

Example: F_5



Source of inefficiency?

Overlapping subproblems

$F_1 = 1$, $F_2 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 3$

Idea: do not recompute, recall from memory

"Those who do not remember the past are condemned to repeat it."

```
vector<int> memo(n + 1, -1); // Memory

int F(int n) {
    if (n == 1 || n == 2) return 1;

    return F(n - 1) + F(n - 2);
}

int F(int n) {
    if (n == 1 || n == 2) return 1;
    if (memo[n] == -1) // I do not remember it
        memo[n] = F(n - 1) + F(n - 2); // compute and remember it
    return memo[n]; // I remember it, recall
}
```

Time complexity: $\Theta(n)$

Memoization (or top-down DP) is simple and powerful :)

(not a typo, comes from *memo*)

Second Example: Rod Cutting

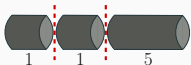
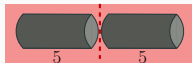
Input:

- ▶ a metal rod of length n
- ▶ values p_1, \dots, p_n s.t. p_i denotes the price for a rod of length i

Output: r_n , maximal possible revenue for a rod of length n (i.e. maximal sum of prices of pieces over all possible partitions)

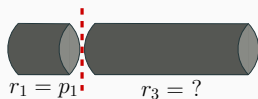
Example: $n = 4$

length i	1	2	3	4
price p_i	1	5	8	9



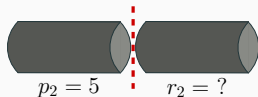
Recursive maximisation:

$$r_n = \max \{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1\}$$



Reformulation: piece containing the left end + a partition of the rest

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \quad \text{with } r_0 = 0$$



$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \quad \text{with } r_0 = 0$$

Recursive algorithm:

```
int r(vector<int> &p, int n) {  
    if (n == 0) return 0;  
    int res = -1;  
    for (int i = 1; i <= n; i++) {  
        res = max(res, p[i] + r(p, n - i));  
    }  
    return res;  
}
```

Time complexity: $\Theta(2^n)$

Why? **Overlapping subproblems**. (Can you see them?)

How to be more efficient?

Dynamic Programming

Dynamic Programming

- ▶ Top-Down (recursion + memo)
- ▶ Bottom-Up (fill up a table)

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \quad \text{with } r_0 = 0$$

Top-Down DP

```
vector<int> memo(n + 1, -1);

int r(vector<int> &p, int n) {
    if (n == 0) return 0;
    if (memo[n] != -1) return memo[n];
    int res = -1;
    for (int i = 1; i <= n; i++) {
        res = max(res, p[i] + r(p, n - i));
    }
    memo[n] = res;
    return res;
}
```

Time complexity: $\Theta(n^2)$

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \quad \text{with } r_0 = 0$$

Bottom-Up DP

```
vector<int> r(n + 1, -1);  
r[0] = 0;  
for (int i = 1; i <= n; i++) {  
    for (int j = 1; j <= i; j++) {  
        r[i] = max(r[i], p[j] + r[i - j]); // IMPORTANT: Current subproblem  
                                           // relies only on the solution of  
                                           // the smaller subproblems  
    }  
}  
return r[n];
```

Time complexity: $\Theta(n^2)$

Reconstructing a Solution: What if we also want to know *where* to cut?

No problem!

```
vector<int> cut(n + 1, 0); // cut[i] stores where to optimally cut a rod of
                           // length i
vector<int> memo(n + 1, -1);

int r(vector<int> &p, int n) {
    if (n == 0) return 0;
    if (memo[n] != -1) {
        return memo[n];
    }
    int res = -1;
    for (int i = 1; i <= n; i++) {
        if (p[i] + r(p, n - i) > res) {
            res = p[i] + r(p, n - i);
            cut[n] = i; // We should cut at position i
        }
    }
    memo[n] = res;
    return res;
}
```

That was **easy**! Why is it so **difficult** in general?

Remembering is easy (right?)—apply **memoization**.

Deriving a **recursive algorithm** is the **difficult part**.

Usually problems are not given in a way that can straightforwardly be translated into a recursive definition. :(

Essential elements of a DP problem:

- ▶ Usually **optimisation problems**, i.e. maximise or minimise some quantity
- ▶ Exhibits the **optimal subproblem** structure
- ▶ **Overlapping subproblems**

Essence of DP (with examples):

- Usually **optimisation problems**, i.e. maximise or minimise some quantity

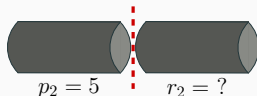
Example: Rod Cutting Problem

r_n , **maximal** possible revenue for a rod of length n

Essence of DP (with examples):

- ▶ Exhibits the **optimal subproblem** structure

Example: Rod Cutting Problem

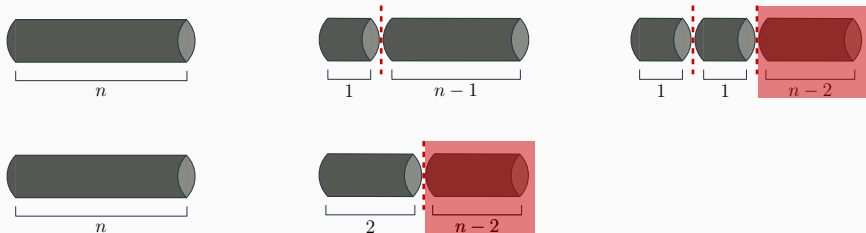


- ▶ Suppose the DP gods show you what is the ‘last’ choice to make.
- ▶ Look at which subproblems arise once making this choice.
- ▶ Show that the subproblems used in an optimal solution must themselves be optimal.

Essence of DP (with examples):

- ▶ Overlapping subproblems

Example: Rod Cutting Problem



- ▶ Remember, do not recompute!
- ▶ *"Those who do not remember the past are condemned to repeat it."*

Common Pitfalls

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \quad \text{with } r_0 = 0$$

```
vector<int> memo(n + 1, -1);

int r(vector<int> &p, int n) {
    if (n == 0) return 0;
    if (memo[n] != -1) return memo[n];
    int res = -1;
    for (int i = 1; i <= n + 1; i++) {
        res = max(res, p[i] + r(p, n - i));
    }
    memo[n] = res;
    return res;
}
```

This results in a **SEG FAULT/RUN ERROR**, can you see why?

Make sure that you stay 'within' the memo boundaries! Similarly, sometimes the memo table does not/cannot contain the base cases.

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \quad \text{with } r_0 = 0$$

```
vector<int> memo(n + 1, 0);
```

```
int r(vector<int> &p, int n) {
    if (n == 0) return 0;
    if (memo[n] != 0) return memo[n];
    int res = -1;
    for (int i = 1; i <= n; i++) {
        res = max(res, p[i] + r(p, n - i));
    }
    memo[n] = res;
    return res;
}
```

This results in a **TIMELIMIT**, can you see why?

Make sure that the **default** memo value is **not** a possible output!

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \quad \text{with } r_0 = 0$$

```
map<int, int> memo;
```

```
int r(vector<int> &p, int n) {
    if (n == 0) return 0;
    if (memo.find(n) != memo.end()) return memo[n];
    int res = -1;
    for (int i = 1; i <= n; i++) {
        res = max(res, p[i] + r(p, n - i));
    }
    memo[n] = res;
    return res;
}
```

This (possibly) results in a **TIMELIMIT**, can you see why?

`std::map` adds an $O(\log n)$ insert/find/access overhead.

Third Example: Longest Increasing Subsequence

Input: a sequence of n integers a_1, \dots, a_n

Output: the **length** of a **longest increasing subsequence** (LIS)

Example:

2 4 **3** 7 4 **5**

Result: LIS = **4**

Input: a sequence of n integers a_1, \dots, a_n

Output: the **length** of a **longest increasing subsequence** (LIS)

First attempt: $f(i) :=$ 'length of the LIS in a_1, \dots, a_i '.

▶ Base cases: $f(1) = 1$

▶ $f(i) = ???$

Second attempt: $f(i) :=$ 'length of the LIS in a_1, \dots, a_i **ending at a_i** '.

▶ Base cases: $f(1) = 1$

▶ $f(i) = \max_{j < i: a_j \leq a_i} (1 + f(j))$

We had to **reformulate** the problem s.t. it admits a **recursive formulation**, this is **difficult**!

Time complexity: $O(n^2)$

Explanation: n function calls (with memo), i -th call takes $O(i)$ time.

(**Exercise:** Can you do it in $O(n \log n)$?)

Tips & Tricks

Top-Down (memoization) vs Bottom-Up (iterative)

Usually both work and it boils down to a personal preference :)

- ▶ Simple to implement
- ▶ Easy to describe subproblems (by using a `std::map`)
- ▶ Computes only necessary subproblems
- ▶ Time complexity sometimes not so obvious
- ▶ Overhead of function calls
- ▶ More effort to code
- ▶ Subproblems must be described by integers
- ▶ Always computes all subproblems
- ▶ Time complexity obvious
- ▶ Saves some constant factors

How to determine the runtime?

Informally, a product of two factors: the overall **number of subproblems** and **how many choices** for each subproblem

Bottom-Up: Easy!

Top-Down: Sometimes harder to see immediately.

`std::map` or `std::vector`? **Always `std::vector`!**

Unless... the subproblem (state space) cannot be described by integers.
Then use maps.

Remember! `std::map` has a insert/find/access overhead of $O(\log n)$.

DP – Summary

- ▶ Idea of DP: solve subproblems **only once** by storing solutions of subproblems
- ▶ Start by defining **recurrence relation** (on paper)
- ▶ Implement it. It will be **correct** but slow...
- ▶ Are there **overlapping subproblems**?
- ▶ Add **memo** (usually does the trick) or construct a DP table
- ▶ **Practice deriving recurrent relations on paper** for standard DP problems (e.g. Knapsack, SubsetSum, Coin Change, LCS, Edit Distance, LIS, etc.)

That's all for today!