L9: Searching and Sorting II; Memoization

CS1101S: Programming Methodology

Low Kok Lim

October 13, 2021

Outline

- Searching
- Sorting
- Memoization

Outline

- Searching
  - Linear search
  - Binary search
- Sorting
- Memoization

Linear / Sequential Search

- To find a number in a **list** — inspect the list from the front, element by element
- The equivalent approach for **arrays**:

```
function linear_search(A, v) {
    const len = array_length(A);
    let i = 0;
    while (i < len && A[i] !== v) {
        i = i + 1;
    }
    return (i < len);
}
linear_search([1,2,3,4,5,6,7,8,9], 5);
```

Show in Playground

- Arrays are **random access**
  - Any value can be retrieved in $O(1)$ time

Binary Search

- Make sure input **array** of length *n* is **sorted** in ascending order

- **Idea:** Checking the **middle element** in a given **range** allows us to cut the search space in half

2	5	6	8	10	13	14	19	20	26	29
2	5	6	8	10	**13**	14	19	20	26	29
2	5	6	8	10	**13**	14	19	**20**	26	29
2	5	6	8	10	**13**	14	19	**20**	**26**	29

searching
for **26**

- Same idea as **binary search trees** (BST)

Binary Search

- Same idea as **binary search trees**

- The result is a runtime of ***O*(log *n*)**

2	5	6	8	10	13	14	19	20	26	29

- Can we do **binary search** on a **list**?

Binary Search (using Recursion)

```
function binary_search(A, v) {
    function search(low, high) {
        if (low > high) {
            return false;
        } else {
            const mid = math_floor((low + high) / 2);
            return (v === A[mid]) ||
                   (v < A[mid]
                    ? search(low, mid - 1)
                    : search(mid + 1, high));
        }
    }
    return search(0, array_length(A) - 1);
}
binary_search([1,2,3,4,5,6,7,8,9], 8);
```

Show in Playground

Binary Search (using Loop)

```
function binary_search(A, v) {
    let low = 0;
    let high = array_length(A) - 1;

    while (low <= high) {
        const mid = math_floor((low + high) / 2 );
        if (v === A[mid]) {
            break;
        } else if (v < A[mid]) {
            high = mid - 1;
        } else {
            low = mid + 1;
        }
    }
    return (low <= high);
}

binary_search([1,2,3,4,5,6,7,8,9], 8);
```

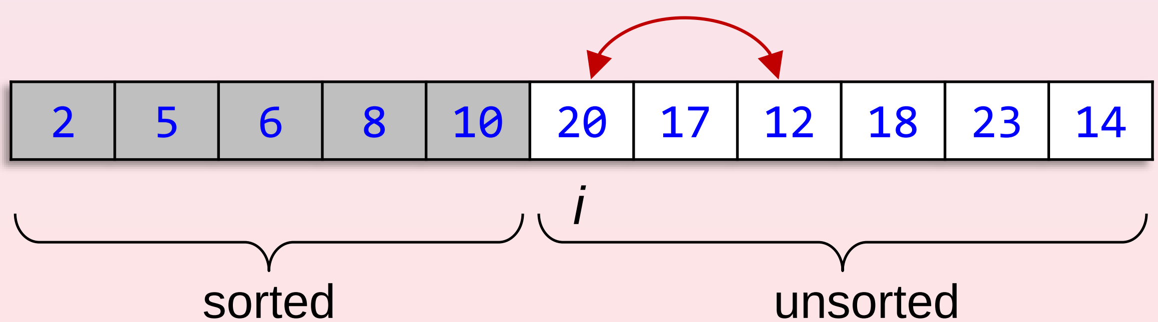
[Show in Playground]

Outline

- Searching
- Sorting
    - Selection Sort
    - Insertion Sort
    - Merge Sort
- Memoization

Selection Sort

- **Idea for *lists*:**
  - Find the smallest element *x* and remove it from the list
  - Sort the remaining list, and put *x* in front
- **Similar idea for *arrays*:**
  - Build the sorted array from left to right
  - For each remaining unsorted portion to the right of position *i*, find the smallest element and swap it into position *i*

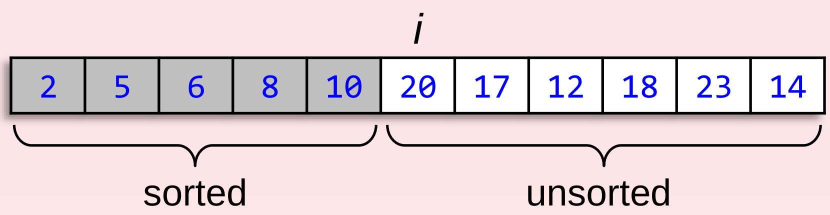


Selection Sort for Arrays

Show in Playground

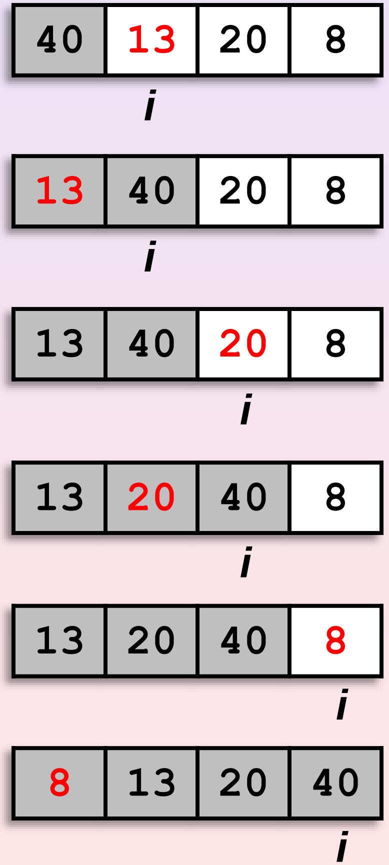
```
function selection_sort(A) {
    const len = array_length(A);

    for (let i = 0; i < len - 1; i = i + 1) {
        let min_pos = find_min_pos(A, i, len - 1);
        swap(A, i, min_pos);
    }
}

function find_min_pos(A, low, high) {
    let min_pos = low;
    for (let j = low + 1; j <= high; j = j + 1) {
        if (A[j] < A[min_pos]) {
            min_pos = j;
        }
    }
    return min_pos;
}
```

```
function swap(A, x, y) {
    const temp = A[x];
    A[x] = A[y];
    A[y] = temp;
}
```



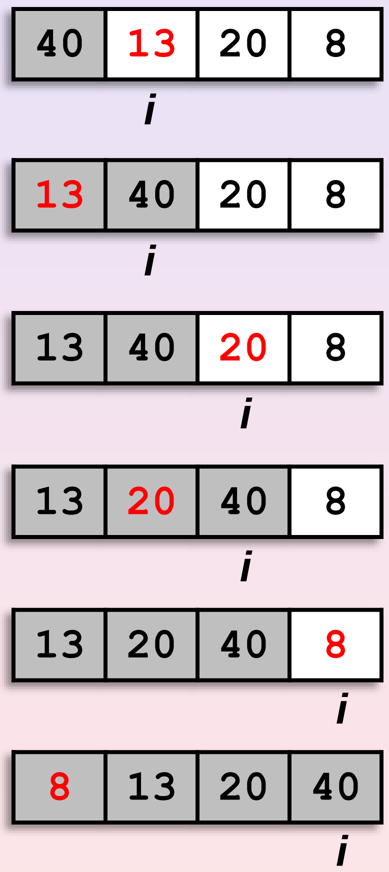
Insertion Sort

- **Idea for *lists*:**
  - Sort the tail of the given list using wishful thinking
  - Insert the head in the right place
- **Idea for *arrays*:**
  - Move a pointer *i* from left to right
  - The array to the left of *i* is sorted already
  - Swap the value at *i* with its neighbor to the left, until the neighbor is smaller



Insertion Sort for Arrays

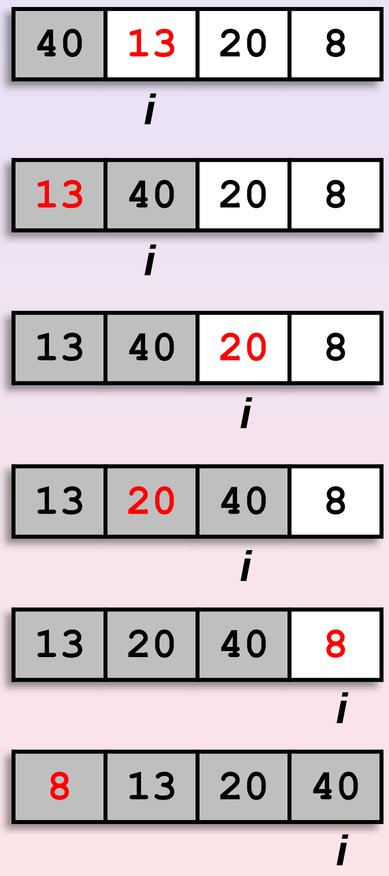
```
function insertion_sort(A) {
    const len = array_length(A);

    for (let i = 1; i < len; i = i + 1) {
        let j = i - 1;
        while (j >= 0 && A[j] > A[j + 1]) {
            swap(A, j, j + 1);
            j = j - 1;
        }
    }
}
```



[Show in Playground]

Insertion Sort for Arrays (Alternative Version)

```
// This alternative method replaces
// the swaps by shifting elements right.

function insertion_sort2(A) {
    const len = array_length(A);

    for (let i = 1; i < len; i = i + 1) {
        const x = A[i];
        let j = i - 1;
        while (j >= 0 && A[j] > x) {
            A[j + 1] = A[j]; // shift right
            j = j - 1;
        }
        A[j + 1] = x;
    }
}
```



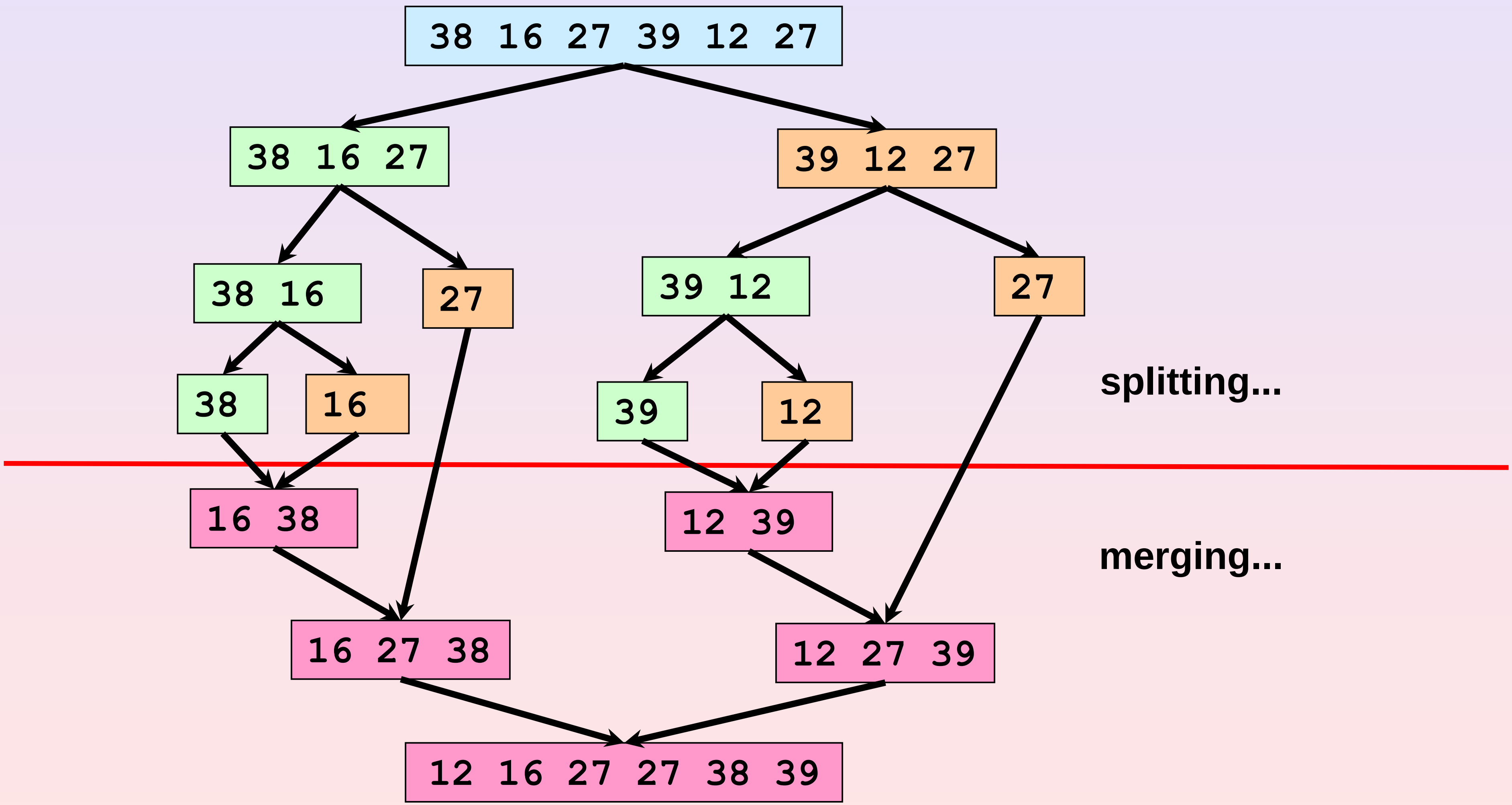
[Show in Playground]

Merge Sort

- **Idea for *lists*:**
  - Split the list **in half**, sort each half using wishful thinking
  - Merge the **sorted** lists together

- **Similar idea for *arrays*:**
  - Sort the halves
  - Merge the halves (using temporary arrays)

Merge Sort: Example



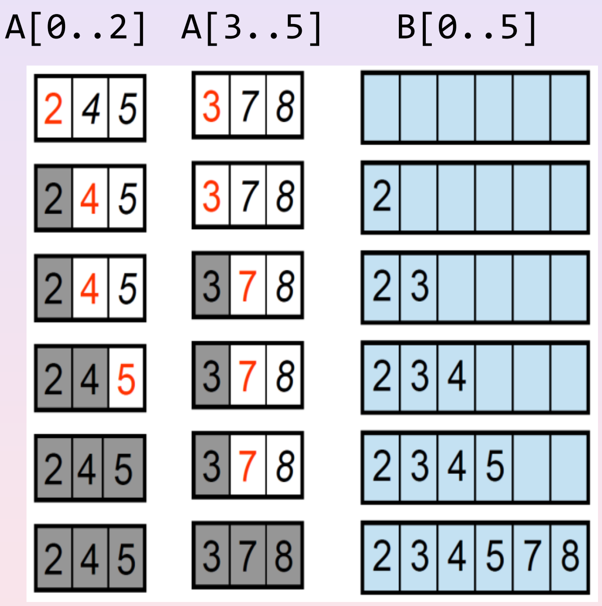
Merge Sort for Arrays

```
function merge_sort(A) {
    merge_sort_helper(A, 0, array_length(A) - 1);
}

function merge_sort_helper(A, low, high) {
    if (low < high) {
        const mid = math_floor((low + high) / 2);
        merge_sort_helper(A, low, mid);
        merge_sort_helper(A, mid + 1, high);
        merge(A, low, mid, high);
    }
}

function merge(A, low, mid, high) {
    ...
}
```

[Show in Playground]

Merge Sort for Arrays

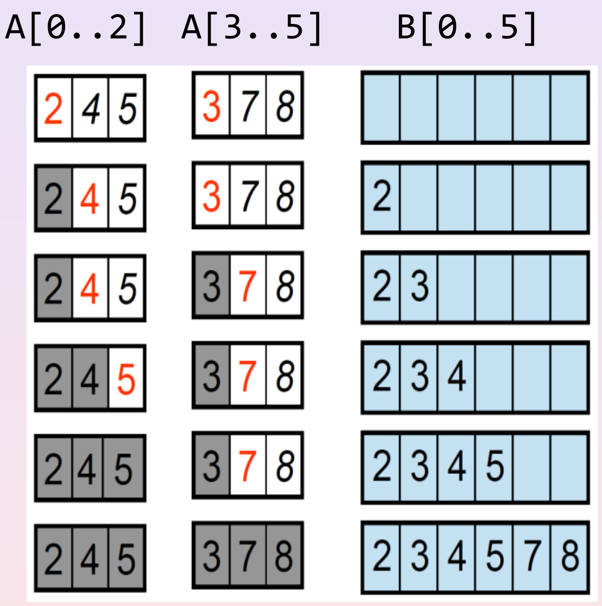
```
function merge(A, low, mid, high) {
    const B = []; // temporary array
    let left = low;
    let right = mid + 1;
    let Bidx = 0;

    while (left <= mid && right <= high) {
        if (A[left] <= A[right]) {
            B[Bidx] = A[left];
            left = left + 1;
        } else {
            B[Bidx] = A[right];
            right = right + 1;
        }
        Bidx = Bidx + 1;
    }
    // continue on next page
```



Show in Playground

Merge Sort for Arrays

```
// continue from previous page

    while (left <= mid) {
        B[Bidx] = A[left];
        Bidx = Bidx + 1;
        left = left + 1;
    }

    while (right <= high) {
        B[Bidx] = A[right];
        Bidx = Bidx + 1;
        right = right + 1;
    }

    for (let k = 0; k < high - low + 1; k = k + 1) {
        A[low + k] = B[k];
    }
}
```



[Show in Playground]

Outline

- Searching
- Sorting
- Memoization
  - Motivation
  - A memoization abstraction
  - More examples

Naïve Implementation of Fibonacci

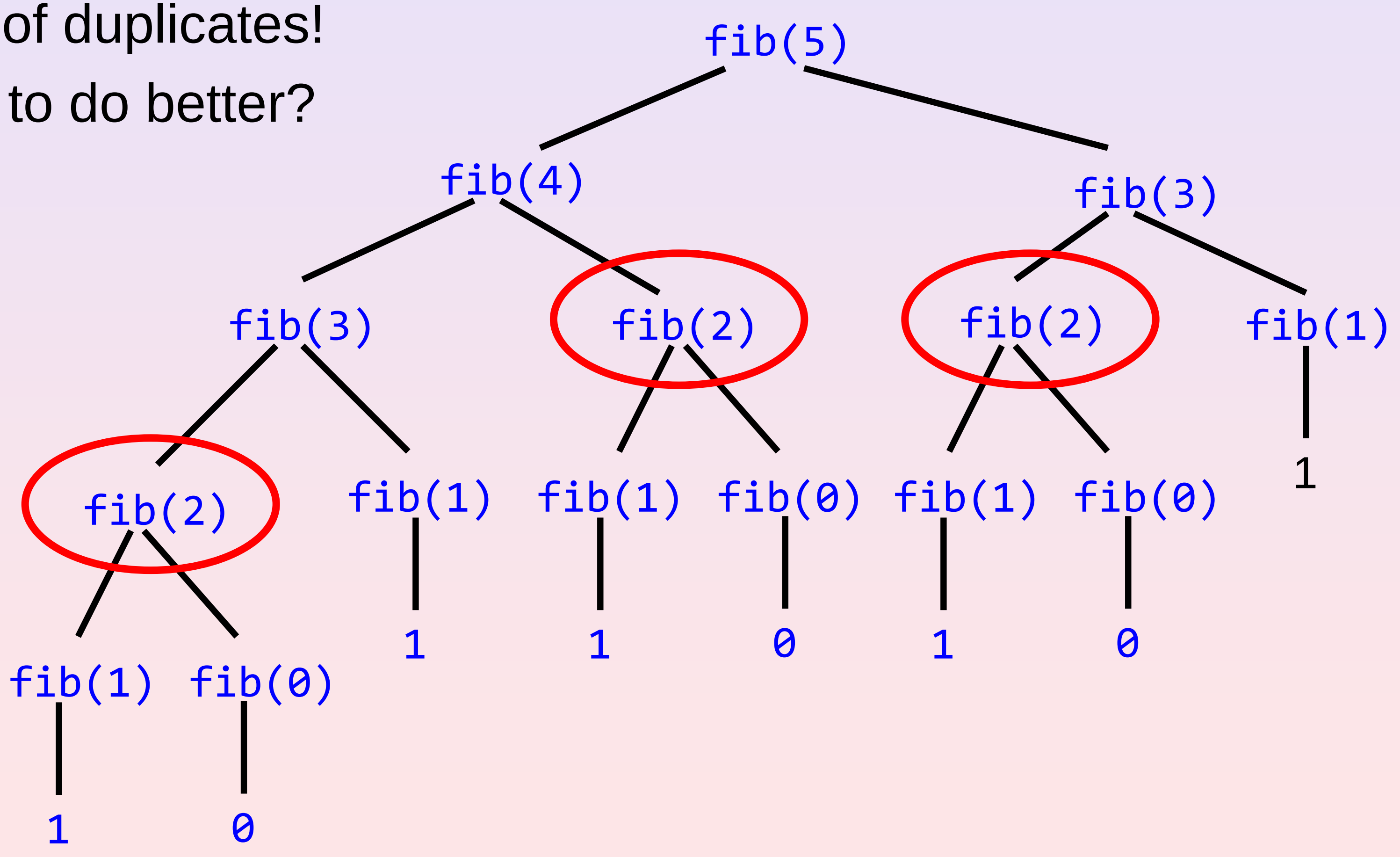
- Recall the not-so-smart **Fibonacci** function

```
function fib(n) {
    return n <= 1
           ? n
           : fib(n - 1) + fib(n - 2);
}
```

- Order of growth in time: $\Theta(\Phi^n)$
  - **Exponential**
  - **Why?**

Why Exponential?

- Lots of duplicates!
- How to do better?



How to do Better?

- **Idea**: Remember what you had computed earlier!
- ***Memoization***
  - Function records, in a **“*local table*”**, values that have previously been computed
  - For example, we can use an **array** as the **local table**

Memoized Fibonacci Function

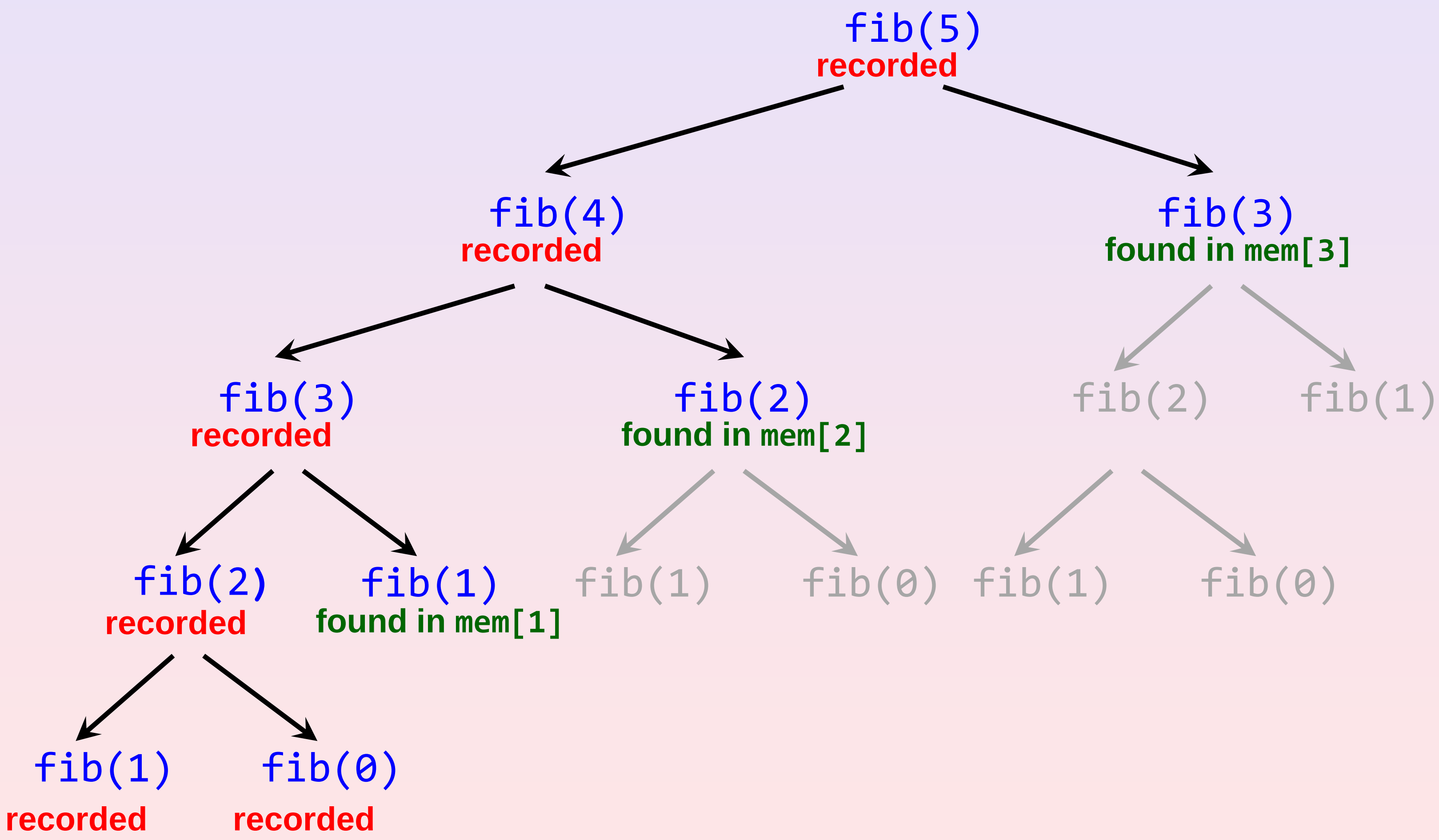
```
function mfib(n) {
    // array mem serves as memory for
    // already computed results of fib
    const mem = [];

    function fib(k) {
        // test if fib(k) has been computed already
        if (mem[k] !== undefined) {
            return mem[k]; // just access memory
        } else { // compute fib and add result to mem
            const result =
                k <= 1 ? k : fib(k - 1) + fib(k - 2);
            mem[k] = result;
            return result;
        }
    }
    return fib(n);
}
```

Show in Playground

Memoized Fibonacci Function

- What is the order of growth in time?
  - $\Theta(n)$
  - **How come?**

Evaluating `mfib(5)`



Using "Global Memory"

[Show in Playground]

```
// array mem serves as memory for
// already computed results of mfib
const mem = [];

function mfib(n) {
    // test if mfib(k) has been computed already
    if (mem[n] !== undefined) {
        return mem[n]; // just access memory
    } else { // compute fib and add result to mem
        const result =
            n <= 1 ? n : mfib(n - 1) + mfib(n - 2);
        mem[n] = result;
        return result;
    }
}
```

Why use "global memory" instead of "local memory"?

Tribonacci Numbers

- Naïve implementation:

```
function trib(n) {
    return n === 0
           ? 0
           : n === 1
           ? 1
           : n === 2
           ? 1
           : trib(n - 1) + trib(n - 2) + trib(n - 3);
}
```

Tribonacci with “Global Memory”

```
const mem = [];

function mtrib(n) {
    if (mem[n] !== undefined) {
        return mem[n];
    } else {
        const result =
            n === 0 ? 0
            : n === 1 ? 1
            : n === 2 ? 1
            : mtrib(n-1) + mtrib(n-2) + mtrib(n-3);
        mem[n] = result;
        return result;
    }
}
```

Show in Playground

An Abstraction for Memoization

```
function memoize(f) {
    const mem = [];

    function mf(x) {
        if (mem[x] !== undefined) {
            return mem[x];
        } else {
            const result = f(x);
            mem[x] = result;
            return result;
        }
    }

    return mf;
}
```

Show in Playground

Tribonacci using `memoize`: First Attempt

```
function trib(n) {
    return n === 0 ? 0
           : n === 1 ? 1
           : n === 2 ? 1
           : trib(n - 1) + trib(n - 2) + trib(n - 3);
}
const memo_trib = memoize(trib);
memo_trib(23);
```

Show in Playground

- What is the order of growth in time?
  - **Still exponential**
  - **How come?**

Why Still Exponential?

```
function memoize(f) {
    const mem = [];

    function mf(x) {
        if (mem[x] !== undefined) {
            return mem[x];
        } else {
            const result = f(x);
            mem[x] = result;
            return result;
        }
    }

    return mf;
}
```

Tribonacci using `memoize`: Second Attempt

```
const mtrib =
    memoize(n => n === 0 ? 0
                 : n === 1 ? 1
                 : n === 2 ? 1
                 : mtrib(n - 1) +
                   mtrib(n - 2) +
                   mtrib(n - 3));

mtrib(23);
```

Show in Playground

- What is the order of growth in time?
  - $\Theta(n)$

The *n*-Choose-*k* Problem

- How many ways to choose ***k*** items out of ***n*** possible choices?

- **Strategy:**
  - Consider one of the items ***x*** — ***x*** is either chosen or it is not
  - Then, numbers of ways is sum of
    - **Not chosen:** Ways to choose ***k*** items out of remaining ***n* – 1** items; and
    - **Chosen:** Ways to choose ***k* – 1** items out of remaining ***n* – 1** items

A Straightforward Implementation

```
function choose(n, k) {
    return k > n
           ? 0
           : k === 0 || k === n
           ? 1
           : choose(n - 1, k) + choose(n - 1, k - 1);
}
```

Show in Playground

- What is the order of growth in time?
- How can we speed up the computation?
  - **Memoization!**

Memoization Considerations for *n*-Choose-*k*

- The result depends on two inputs: ***n*** and ***k***

- Need a **two-dimensional table** to record intermediate results
  - Use a **2-D array**
  - Record result of `choose(n, k)` in `mem[n][k]`

Read and Write from/to Global 2-D Array

```
const mem = [];

function read(n, k) {
    return mem[n] === undefined
           ? undefined
           : mem[n][k];
}

function write(n, k, value) {
    if (mem[n] === undefined) {
        mem[n] = [];
    }
    mem[n][k] = value;
}
```

n-Choose-*k* with Memoization

```
function mchoose(n, k) {
    if (read(n, k) !== undefined) {
        return read(n, k);
    } else {
        const result = k > n ? 0
                     : k === 0 || k === n ? 1
                     : mchoose(n - 1, k) +
                       mchoose(n - 1, k - 1);
        write(n, k, result);
        return result;
    }
}
```

Show in Playground

- What is the order of growth in **time**?
- What is the order of growth in **space**?

Find out in Reflection!

Summary

- The idea of **binary search** transfers well from **binary search trees** to **sorted arrays**

- The core ideas of **sorting algorithms for lists** can be used for **sorting arrays**
    - We take advantage of **random-access memory** and use **loops**, **swapping** and **array copying**

- **Memoization** turns **naïve** solutions into **smart** ones