CS1101S Studio Session Week 10: *Iteration & Memoization*

Niu Yunpeng

niuyunpeng@u.nus.edu

October 23, 2018

Overview

Iteration

Array is better than list ...

- Accessing an element in array `arr[n]` needs $O(1)$ time.
- Accessing an element in list `list_ref(lst, n)` needs $O(n)$ time.
  - But, `list_ref` can be $O(1)$ as well, if you are accessing the first element.

Then ...

- We have been using recrusion & list so far to solve problems because
  - We access the elements in a list in the incremental order. We often only need $O(1)$ time.
- However, in the more general case, array is more flexible.

Iterative Search

Linear search

```
// Returns true if the target if found in the array.
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;
}
```

Iterative Search

Binary search

```
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);
}
```

Iterative Sort

Selection sort

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

    for (let i = 0; i < len - 1; i = i + 1) {
        let j_min = i;
        for (let j = i + 1; j < len; j = j + 1) {
            if (A[j] < A[j_min]) {
                j_min = j;
            } else {}
        }

        if (j_min !== i) {
            swap(A, i, j_min);
        } else {}
    }
}
```

Iterative Sort

Insertion sort

```
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;
        }
    }
}
```

Iterative Sort

Insertion sort 2

```
function insertion_sort2(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;
        }
    }
}
```

Iterative Sort

Merge

```
function merge(A, low, mid, high) {
    const B = [];
    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;
    }
    ...
}
```

Iterative Sort

Merge *(continued)*

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

Iterative Sort

Merge sort

```
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);
    } else {}
}
```

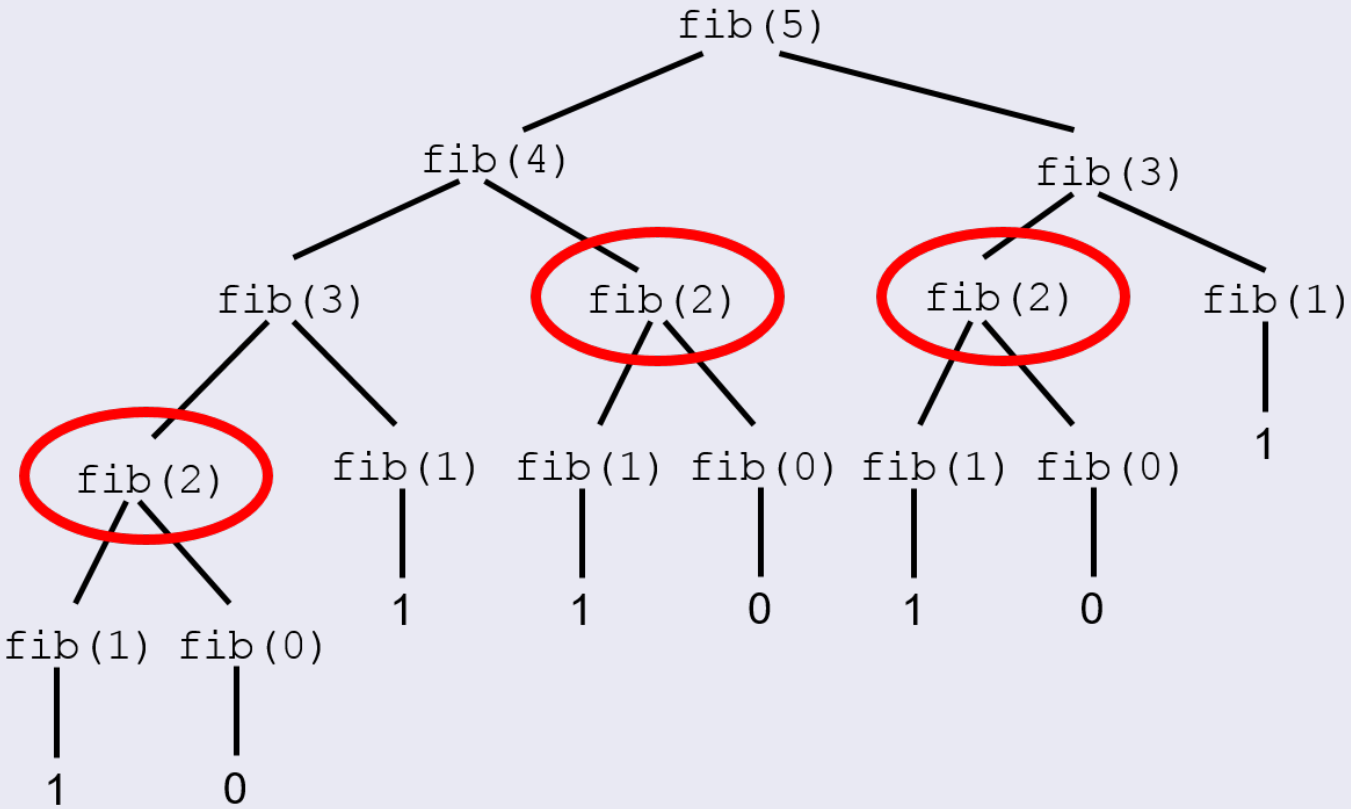
Overview

1. Iteration
   - Why using iteration?
   - Search with iteration
   - Sort with iteration

2. Memoization
   - Inspiration
   - To use memoization
   - Memoization & tabulation

Memoization

Inspiration from Fibonacci



Memoization

Why is this version of Fibonacci **bad**?

- Because it **repeat**s solving the same sub-programs.
- A waste of resources both in time and space.

Suggestion

- Solve each sub-problem only once, and use the result repeatedly.

Memoization

A straightforward example

```
function slow_example(x) {
    if (x > 100) {
        return 1;
    } else {
        return slow_example(x + 3) + slow_example(x + 3);
    }
}

slow_example(2);
```

Memoization

A straightforward example

```
function fast_example(x) {
    if (x > 100) {
        return 1;
    } else {
        return fast_example(x + 3) * 2;
    }
}

fast_example(2);
```

Memoization

A straightforward principle

- **DRY** (don't repeat youself)

Significance

The ***DRY*** principle is the underlying reason for:

- abstraction/wishful thinking
- modular design
- memoization/dynamic programming
- ...

Memoization

Memoization

- How can we repeatedly use the results previously been computed?
- Store them and access the data whenever in need.

Problem...

- We need to store a lot of data.
- We need a proper data structure.

Memoization

To choose a proper data structure

- What to store: the results for every value of the function parameter, like `fibo(1)`, `fibo(2)`, `fibo(3)`, etc.
- How to store: store in a linear data structure, like array or table.
- When the function has 1 parameter, use 1D list/array.
- When the function has 2 parameters, use 2D list/array.
- ...

Memoization

List or array?

- List is better if we can store data incrementally, like 1, 2, 3, ...
- If we cannot store them one by one in the incremental order, then it will become meaningless when we access the data using `list_ref(lst, n)`.

Thus...

- We should choose to use array.
- After we solve a new problem, add `arr[n + 1]`.

Memoization

memoize

```
function memoize(func) {
    let arr = [];

    return function (x) {
        if (arr[x] !== undefined) {
            return arr[x];
        } else {
            const result = func(x);
            arr[x] = result;

            return result;
        }
    };
}
```

Memoization

Problem here!

- For each element in `arr`, its index is the parameter `n`, the value is the return value `func(n)`.
- What if the value of the parameter is not a "**non-negative integer**"?
  - Although JavaScript allows everything to be used as index, that is bad programming practice. It will make your program not intuitive anymore as well. We need a layer of abstraction.

Memoization

Solution

- Create an abstract data structure, called *table* or *dictionary*.
- It has a lot of entries, just like array.
  - Each entry has a key and a value, just like array.
  - In fact, it should even be implemented using array!
- The only difference: keys do not have to be non-negative integers!

Caution

- Later you will see literal objects, which is like a built-in dictionary in JavaScript.

Memoization

Example

- The possible values of the parameter are -2, -1, 0, 1, 2, ...
  - Table will just use `arr[n + 3]` rather than `arr[n]`
- The possible values are 0.5, 1, 1.5 ...
  - Table will just use `arr[n * 2]` rather than `arr[n]`
- The possible values are ..., -3, -2, -1, 0, 1, 2, ...
  - How?

Memoization

Understanding

- *Table* or *dictionary* is simply an improvement to array.
  - By using `map` to transform keys into non-negative integers.

What if the range of possible values do not have a pattern?

- Hash function!

Memoization

To use table or dictionary

- Use `make_table()` rather than `let arr = []`
- Use `contains()` rather than `XXX !== undefined`
- Use `put()` rather than `arr[?] = XXX`
- Use `lookup()` rather than `return arr[?]`

Memoization

memoize

```
function memoize(func) {
    const table = make_table();

    return function (x) {
        if (contains(x, table)) {
            return lookup(x, table);
        } else {
            const result = func(x);
            put(x, result, table);

            return result;
        }
    };
}
```

Memoization

memoize_2d

```
function memoize_2d(func) {
    const table = make_2d_table();

    return function (x, y) {
        if (contains(x, y, table)) {
            return lookup(x, y, table);
        } else {
            const result = func(x, y);
            put(x, y, result, table);

            return result;
        }
    };
}
```

Memoization

A few examples using memoization

- Fibonacci
- k-combination
- coin_change
- ...

Memoization

Fibonacci

```
function fibo(n) {
    if (n <= 1) {
        return n;
    } else {
        return fibo(n - 1) + fibo(n - 2);
    }
}
```

Think about it...

- Time/space complexity

Memoization

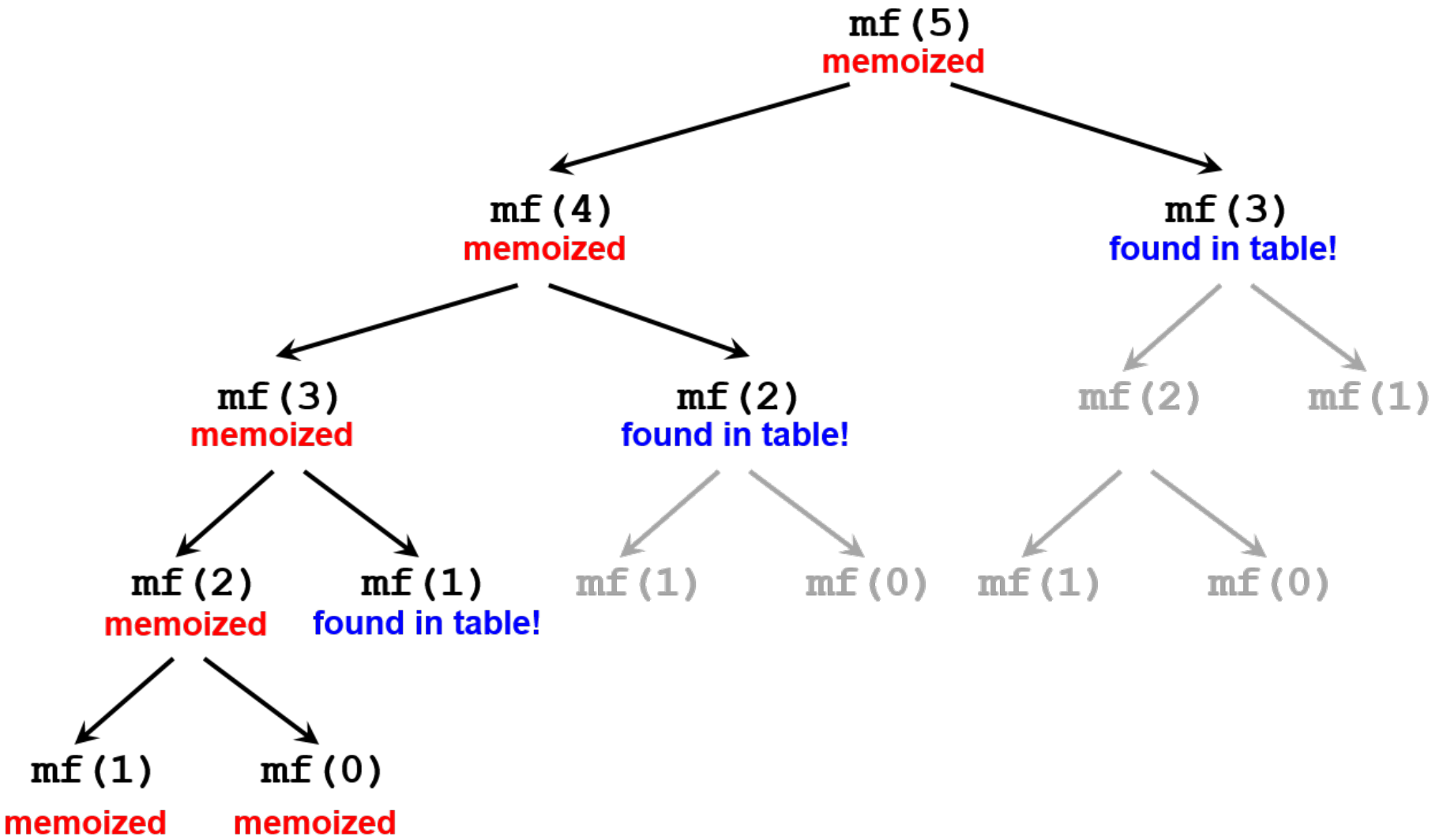
Use `memoize` to improve Fibonacci

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

Reason

- Never solve the same sub-problem again.
- **DRY**!

Memoization



Memoization

Another k-combination

- No need to list all possible k-combinations.
- We only want to count the number of k-combinations.
- After that, we try to use `memoize` to improve it.

Thus...

- We do not care about the actual values for n items in the list.
- We use their indexes $1, 2, ..., n$ to represent them.

Memoization

k-combination

```
function k_combination(n, k) {
    if (k > n) {
        return 0;
    } else if (k === 0) {
        return 1;
    } else {
        return k_combination(n - 1, k - 1) +
               k_combination(n - 1, k);
    }
}
```

Memoization

Use memoize_2d to improve k-combination

```
const memo_k_combination = memoize_2d(function (n, k) {
    if (k > n) {
        return 0;
    } else if (k === 0) {
        return 1;
    } else {
        return memo_k_combination(n - 1, k - 1) +
               memo_k_combination(n - 1, k);
    }
});
```

Memoization

`coin_change` problem

- Find the number of ways to make changes.
- Still remember?

Memoization

coin_change problem

```
function coin_change(amount, kind) {
    if (amount === 0) {
        return 1;
    } else if (amount < 0 || kind === 0) {
        return 0;
    } else {
        return coin_change(amount, kind - 1) +
               coin_change(amount - value(kind), kind);
    }
}
```

Memoization

Use memoize_2d to improve coin_change

```
const memo_coin_change = memoize_2d(function (amount, kind)
    {
     if (amount === 0) {
         return 1;
     } else if (amound < 0 || kind === 0) {
         return 0;
     } else {
         return memo_coin_change(amount, kind - 1) +
                memo_coin_change(amount - value(kind), kind);
     }
});
```

Memoization

An interesting fact

- "memoization" is a domain-specific word.
- If you look it up in the dictionary, you cannot find it.
- A similar word is "memoris(z)ation". But we didn't misspell it.
- "memoization" is only used in Computer Science.

Domain-specific language (DSL)

- In CS, DSL is actually a family of programming languages.
- *Google* this term and you will find some interesting things.

Memoization

Review: two approaches

- *Iteration*: the buttom-up approach;
- *Recursion*: the top-down approach.

Recall: why do we use array/table rather than list?

- We may not traverse in the incremental order $1, 2, ..., n$.
- Using `list_ref(lst, n)` is meaningless.

Memoization

Think about memoization again

- Is it the buttom-up approach or top-down approach?

Look at it...

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

Memoization

Memoization & tabulation

- Memoization: top-down approach;
- Tabulation: buttom-up approach.

Data structure

- Memoization: table;
- Tabulation: table or list (array).

Memoization

To use tabulation

- To use tabulation, we will start from the smallest sub-problems.
- Then, we will solve larger and larger sub-problems until the whole problem has been solved.

Example

- If we use tabulation for Fibonacci, we will solve sub-problems in the incremental order, like `fibo(1)`, `fibo(2)`, `fibo(3)`, ...
- Due to the incremental order, we can also use list.

Memoization

Practical usage of memoization/tabulation

- Essentially, they are just "cache".
  - CPU cache
  - SQL execution plan caching
  - Redis LRU/LFU (least recently/frequently used) cache

Memoization

Dynamic programming

- **Dynamic programming** (DP) is a technique for solving problems recursively and is applicable when the computations of the subproblems overlap.
- **Memoization** and **tabulation** are two approaches for DP.

Studio Group Problems

Let's discuss them now.

The End

Copyright