19th Oct 2020

Studio 10 Searching & Sorting, Memoization

CS1101S AY20/21 SEM 1

Studio 03A

Chen Xihao
Year 2 Computer Science

chenxihao@u.nus.edu
@BooleanValue

Studio 10

Agenda

- Admin
- Recap:
  - Searching
  - Sorting
  - Memoization
  - Orders of growth
  - Environment Model
- Studio sheets

Studio 10

Admin

- Reading Assessment 2 this Friday, 23rd Oct
- Details on LumiNUS, do past year papers
- Practice drawing env diagrams quickly, until it becomes second nature
- Questions: ask in the group! My workload is getting heavy so I might not have time to reply to everyone.
    - Chances are your friends have probably encountered the same problems before

Recap

Recap: Searching

Recap

Searching - Linear

- Most intuitive searching algorithm and simplest to implement
- Just go through every single element in the array and check if it matches

Recap

Searching - Binary

- Maintain two pointers
  - Range of the array we want to check
- Halves the array at each iteration
- Requirements:
  - Array must be sorted

Recap: Sorting

Recap
Sorting - Insertion

- Go through the unsorted array
- Insert each element into the new array at the correct position

Recap

Sorting - Selection

- Select the smallest element
- Place at the front
- Select the second smallest element
- Place after the smallest
- Select the third smallest element
- Place after the second smallest
- … rinse and repeat

Recap

Sorting - Merge

- Divide the array into two parts
- Sort and merge both halves
- Base condition:
  - Array of size 1: just return this element

Recap: Memoization

Recap
Memoization

- No it’s not a typo… even if your phone doesn’t contain this word
    - It’s not “memorisation”
    - Memoization is something like “taking a memo”
    - Google if you are interested

Recap

Memoization

- What is it?
  - “In computing, **memoization** or **memoisation** is an optimization technique used primarily to speed up computer programs by storing the results of expensive function calls and returning the cached result when the same inputs occur again.” — wikipedia
  - Used in dynamic programming (youdontneedtoknowthis)

Recap
Memoization

- Why do we use this?
  - Can drastically reduce the runtime of programmes
  - Who wants to carry out the same computation many times…

Recap

Memoization

- Implementation:
  - Store values in a table of values
    - Key: function call parameter(s)
    - Value: corresponding calculated value

Recap

Memoization

- Recall: arrays can be accessed in constant time!
  - Best choice for the table! (at least for now)
  - You can use a list too (but it’s going to be slower)
- Note: local tables are not the only way of memoizing values

Recap
Memoization

- How it works:
  - When the function is called, check if the function has been called with the same parameters previously (by looking at the table)
    - Yes: just return that value
    - No: calculated the value, then write it into the table

Recap

Memoization

- How to memoize?
  - Choose only the parameters that are useful for calculation!
  - Why?
    - Try to minimise space consumption!
    - If we only need to memoize 1 parameter: O(n) space
    - If we need to memoize 2 parameters: O(n^2) space
    - If we need to memoize k parameters: O(n^k) space! (not gud)

Recap

Memoization

- Example:

```
function f(x, y, z) {
    return y === 0
        ? x
        : y + z + f(x, y-1, z) + f(x, y-1, z+1);
}
```

- What should we memoize?
    - y and z, since we don’t need x to do the heavy calculations!

Recap

Memoization

- Another example:

```
function f(x) {
    return x === 0 ? 1 : x * f(x-1);
}
```

- What should we memoize?
    - Nothing!
    - Since we will never call `f` with the same arguments more than once!

Recap: Orders of Growth

Recap

Orders of Growth

- Quick revision, what's the time complexity of these:
  - Selection sort
  - Insertion sort
  - Merge sort
  - Quick sort
  - Linear search
  - Binary search

Recap

Orders of Growth

- Interesting fact:
  - Insertion sort takes O(n) time if the original array is sorted
  - Quick sort takes O(n^2) time if original array is sorted
    - Not so quick is it…
- In general, just stick to:
  - Insertion sort is O(n^2), quick sort is O(n log n) on average!

Recap

Orders of Growth

- Quiz time!

```
function f(n) {
    return n * 2;
}
```

  - This function runs in O(n log n) time. True or false?
  - Answer: true
    - Big-O notation is used. Although this is constant time, saying it’s upper bounded by nlogn is correct!

Recap

Orders of Growth

- Quiz time!

```
function f(n) {
    for (let i = 0; i < n; i = i + 1) {
        display(n);
    }
}
```

  - What is the order of growth for this function?
  - Answer: O(n)

Recap

Orders of Growth

- Discussion time!

```
function f(n) {
    for (let i = 0; i < n; i = i + 1) {
        for (let j = 0; j < n / 2; j = j + 1) {
            display(n);
        }
    }
}
```

- What is the order of growth for this function?
- Answer: O(n^2). Inner loops runs n/2 times, which is just O(n) but not O(log n)!

Recap

Orders of Growth

- Discussion time!

```
function f(n) {
    for (let i = 0; i < n; i = i + 1) {
        const p = pair("i love cs", null);
        display(n + 1);
        set_head(p, "kiddingz");
    }
}
```

  - What is the order of growth for this function?
  - Answer: O(n). The garbage in the loop body are all constant time operations!

Recap

Orders of Growth

- Discussion time!

```
function f(n) {
    for (let i = 1; i < n; i = i * 2) {
        for (let j = 1; j < i; j = j * 2) {
            display(j);
        }
    }
}
```

  - What is the order of growth for this function?
  - Answer: O((log n) ^ 2)

Recap

Orders of Growth

- Discussion time!

```
function f(n) {
    function helper(x) {
        return x < 1 ? “oof” : helper(x / 2);
    }
    helper(99999);
}
```

  - What is the order of growth for this function?
  - Answer: O(1) BUT WHY?

Recap

Orders of Growth

- This runs in O(1)!

```
function f(n) {
    function helper(x) {
        return x < 1 ? “oof” : helper(x / 2);
    }
    helper(99999);
}
```

- Notice that `helper` is always called with the argument `99999` no matter the value of `n`! So this function is independent of `n`!

Recap: Environment Model

Recap

Environment Model

- A note on pre-declared functions:
  - Consists of:
    - Primitive functions
    - Compound functions

Recap

Environment Model

- Predeclared primitive functions:
  - Implemented using underlying JavaScript
  - No frames created when applying function
  - Values appear automagically~

```
array_length;
```

```
function array_length(arr) {
        [implementation hidden]
}
```

Recap

Environment Model

- Predeclared compound functions:
  - Implemented directly in Source and using Source
  - Frames are created when applying function !!!

```
length;
```

```
function length(xs) {
  return is_null(xs) ? 0 : 1 + length(tail(xs));
}
```

Recap

Environment Model

- A note on pre-declared functions:
  - Consists of:
    - Primitive functions
    - Compound functions
  - Evaluated in the <u>global</u> environment
    - E.g. map(f, xs):
      - `map` is evaluated in global
      - BUT `f` may not be evaluated in global!

Any questions?

End of Recap

End of File