Mid-Quarter Review/Summary

Friday, July 17, 2020 3:27 PM

CPP Fundamentals

- Parameter passing
  - ↳ Pass-by-value
    - parameter & var ch main() are independent copy vars
    - foo wouldn't change bar w/o '&'
  - ↳ pass by reference (W)
    - just one var (one place in memory) & the param var is a pointer

```
void foo(& num) {
    num = 0
}
```

```
main() {
    int bar = 5
    foo(bar)
    // now bar == 0
```

- String
  - ↳ 'string' converts a C-string to C++ string

```
string str = "example"
(converter) (C++ container) (C++ string)
```

  - ↳ there are a crap ton of useful methods for strings (strlib ≠ <string>)
    - some char methods return ASCII values (integers), so careful w/ concatenation w/ strings
    - "npos" (string::npos) is the constant for no-position

```
if (str.find != string::npos) {
}
```

※ example problems: Trip written up, I'll use pseudo code

```
bool isPalindrome(string word) {
    CleanString(word);
    for(i < word.length/2) {
        if (word[i] != word[word.length() - 1 - i]) {
            return false;
        }
    }
}
```

※ easier to negate requirement than have multiple returns

※ efficiency

```
string cleanString(string word) {
    string newStr = "";
    for (char ch: word)
        if (isalpha(ch)) {
            newStr += ch
        }
    }
    word = toLowerCase(newStr);
}
```

※ don't hesitate to use helper functions - decomposition

※ string processing questions are easier building up from empty string

ADTs

- Abstract because user doesn't need to know implementation

Ordered

Vectors

- popular, versatile, can kinda do whatever unordered ADTs can too
- ton of helper methods
- everything has an index

※ but don't use all of them in exam if O(n) limited

↳ **Grids**

- 2-D vector
- remember to use the struct GridLocation

Unordered

Map

- store key-value pairs
- map.put(key, value)
- auto-insertion: when using [], C++ will add the K-V pair if it doesn't exist
  - if you try & access -can create bunch of empty pairs

Stacks

push() pop() peek()

LIFO

Queue

enqueue() dequeue() epeek

FIFO

※ very efficient, methods are O(1), but can't iterate through non-destructively

Sets

- .contains() is very efficient
- cannot contain duplicates
- need a for-each loop to iterate through

```
Queue<int> reverseQueue(Queue<int> q) {
    stack<int> storage;
    while(!q.empty()) {
        storage.push(q.dequeue());
    }
    while(!storage.empty()) {
        q.enqueue(storage.pop());
    }
}
```

O(n) performance
- exactly is "O(2n)" b/c each loop is O(n)
- push(), ... are all O(1)

- this can be done w/ vector instead

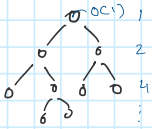
Algorithmic Analysis

※ not so much of a summary ch

Algorithmic Analysis

⁂ not so much of a worry for this exam

- Big-O represents rate of change of execution time, as per the one (con) limiting / bottleneck var
- For recursive methods, it helps to draw out the tree of calls



→ $O(2^n)$

generally ish
$\sim(n-1)+(n-2)$
↳ 2^n

$\sim(n/2)$
↳ $\log n$

1. count the (max) # calls per level
2. find pattern in $C^{\wedge}$)
3. Find O(?) for each call
4. multiply (# calls) by (work per call)

Recursion

- try drawing out the problem to identify base case & recursive case
  - base case = what would you do if input was the most basic input (0, 1, "", etc.)

```
bool isPalindrome(string word) {
    if (word == "") { // base case (default)
        return true;
    word = toLowerCase(cleanString(word));
    if (word == "") error
    return isPalindromeHelper(word);
}

bool isPalindromeHelper(string word) {
    if (word.size == 0 or 1) { return true; }
    if (word[0] == word[word.length() - 1])
        return isPalindromeHelper(word.substr(1, length-2));
}

string cleanString(string word) {
    if (word == "") { // base case
        return "";
    } else { // recursive case
        char first = word[0]
        if (isAlpha(first)) {
            return first + cleanString(word.substr(1));
        } else {
            return cleanString(word.substr(1));
        }
    }
}
```

for bool recursive functions (non-boolean) base case is usually true

key to recursion: Break problem into smaller steps!

isPalindrome(racecar)
= isPalindrome(aceca)
= (eec a)

it is helpful to use helper functions (also) when you don't need to clean inputs or check for errors at every recursive call

- **Backtracking** = when recursive case has multiple branches, multiple solutions being checked
  - ↳ when we find/try all solutions; choose - explore - unchoose (choose)
  - ↳ all the options (divisions at every branching point) are explored recursively (explore). To set an option as explored / to reset the state after an explore path (= unchoose)

⁂ technically EC on this exam

Backtracking Problems
= generating all possible solutions !!
(whether to find best possible solution or to find if a solution exists)

Generating Permutations (order matters)
- at every branch, every available elem from set is added
- sequences of Heads / Tails
- unscramble word (sequence of letters)
- DFS maze search?

Generating Subsets (order doesn't matter) - choose at every step whether or not to include elem from set
- create teams of teaching team
- jury selection
  - ↳ combinations (size limited)
    - combination of Justices
  - ↳ optimization (only return the best solution of for at every branch)
    - knapsack