week3

Tommy MacWilliam

tmacwilliam@cs50.net

September 26, 2011

Announcements

- pset{0,1}: returned!
- pset2: Friday
- `https://cs50.net/lectures`
- `http://cs50.net/ohs`

Today

- Design
- GDB <333333333
- Running Time
- Searching
- Sorting
- Recursion
- Practice Problems

Hacker Tip of the Week

- `ssh jharvard@192.168.56.50`
  - equivalent to Terminal in your Appliance!
  - Windows: `http://www.chiark.greenend.org.uk/~sgtatham/putty/`
  - Mac: `http://www.iterm2.com/`
- `smb://192.168.56.50`
  - open files in your Appliance on your desktop!
  - more detail: `https://manual.cs50.net/Appliance#How_to_Transfer_Files_between_Appliance_and_Your_Computer`

Correctness

- don't forget to check for corner cases!
  - 0
  - negative numbers
  - word instead of number
  - number instead of word
  - nothing
  - not enough arguments
  - too many arguments
  - etc

Design

- don't do too much work if you don't have to!
- don't check conditions you know are true

```
if (x == 5) {}
else if (x != 5) {}
```

- don't create unnecessary variables

```
int y = x + 5;
int z = y * 2;
```

Design

- always ask yourself “is there any way I can solve this problem better?”
  - problems have many correct answers, but few elegant ones
  - often elegant != straightforward
- shorter is not always better

Greedy Revisited

- calculating change can be done in **one line**
  - no one did this, and you were not expected to do this
  - I promise.

Greedy Revisited

- example time!
  - `greedy.c`

Greedy Revisited

- don't do too much work if you don't have to
  - need loops? not really
  - need more variables? not really
- however, meaning must still be clear!
  - long, complicated expressions may look fancy, but can be really confusing to read

GDB

- GDB: The **G**NU **Deb**ugger
  - the awesome debugger
- allows you to walk through your program step-by-step
- as nice as a million `printf` statements are

GDB

- `gdb program`
- `run`: run the loaded program
- `break`: create a breakpoint
  - `break 10`: create a breakpoint at line 10
  - `break function`: create a breakpoint at a function

GDB

- `info break`: list all breakpoints
- `disable 1`: disable breakpoint 1
- `enable 1`: enable breakpoint 1

GDB

- `continue`: continue until next breakpoint
- `list`: show source code around breakpoint

GDB

- `next`: execute the next line of code
  - function call == one line
- `step`: execute next line of code
  - if function, go into function

GDB

- `print variable`: print the value of a variable
  - `printf`: just like C's `printf`, so good for arrays/strings
- `backtrace`: see list of functions that got you here
- `set var variable=value`: set the value of a variable
- `call function()`: execute a function

GDB

- example time!
  - `gdb.c`

Definition

- describes how long an algorithm takes to run
- not in terms of milliseconds, but steps
  - since seconds will vary by machine
  - e.g. one step == look at one array element

Big O

- *O*: worst-case running time
  - given the worst possible scenario, how fast can we solve a problem?
    - e.g. array is in descending order, we want it in ascending order
  - most important when describing an algorithm

Omega

- Ω: best-case running time
  - given the best possible scenario, how fast can we solve a problem?
    - e.g. array is already sorted

Theta

- Θ: both best-case and worst-case
  - e.g. best and worst are the same running time

Common Running Times

- in ascending order:
  - $O(1)$: constant
  - $O(\log n)$: logarithmic
  - $O(n)$: linear
  - $O(n^c)$: polynomial
  - $O(c^n)$: exponential
  - $O(n!)$: factorial

Comparing Running Times

- $O(n)$, $O(2n)$, and $O(5n + 3)$ are all the same thing: $O(n)$
  - constants drop out, because n dominates
- similarly, $O(n^3 + 2n^2) = O(n^3)$
  - n^3 dominates n^2
- however, $O(n^3) > O(n^2)$
  - 2 and 3 are not constants here

Determining Running Times

- example time!
  - `n1.c, n2.c, n3.c`

Real-World Efficiency

- runtimes are nice, but actual efficiency is about more than *O*'s and *n*'s
- problem?

```
for (int i = 0; i < strlen(word); i++)
    printf("%c", word[i]);
```

Real-World Efficiency

- length of `word` isn't going to change, so why are we recomputing it?!

```
int length = strlen(word);
for (int i = 0; i < length; i++)
```

Linear Search

- implementation: iterate through each element of the list, looking for it
- runtime: $O(n)$, $\Omega(1)$
- does not require list to be sorted

Binary Search

- implementation: keep looking at middle elements
  - start at middle of list
  - if too high, forget right half and look at middle of left half
  - if too low, forget left half and look at middle of right half
- runtime: $O(\log n)$, $\Omega(1)$
- requires list to be sorted

Bubble Sort

- implementation: if adjacent elements are out of place, switch them
  - repeat until no swaps are made
- runtime: $O(n^2)$, $\Omega(n)$

Selection Sort

- implementation: start at beginning of list, find smallest element
  - swap first element with smallest element
  - go to second element, treat that as the new first element, continue
    - because everything to the left is already sorted
- runtime: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$

Insertion Sort

- implementation: start at beginning of list, find smallest element
  - insert smallest element to right of first element
    - everything to the right shifts over one spot
  - go to second element, treat as the new first element, continue
- runtime: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$

Sorting Demos

- `http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html`
- `http://cg.scs.carleton.ca/~morin/misc/sortalg/`

Recursion

- basic idea: function calls itself
- sounds simple, but can lead to very elegant solutions to problems

Recursion

- base case: when function should stop calling itself
  - without a base case, function would call itself forever!
  - result: segfault
- recursive case: function calls itself, probably using different arguments
  - same arguments every time means function would call itself forever!
    - (okay maybe not with global variables, but those are a bad idea)

Recursion

- don't forget to return something!
  - base case usually returns some constant value
  - recursive case usually returns a call to itself

Recursion

- your turn!
  - `factorial.c`, `fibonacci.c`

Practice Problems

- reverse a string
  - think about design and runtime!
- implement the GCD formula recursively
  - `http://en.wikipedia.org/wiki/Greatest_common_divisor`

Feedback

- how was section today?
  - `http://tommymacwilliam.com/cs50/feedback`