COE428 Lecture Notes Week 2 (Week of January 16, 2017)

Table of Contents

Announcements

- Course management distributed in class last week. (You can also get it from the course home page, from D2L or from my home page.)
- Some of the topics in Week 1 lecture notes were not covered. They have been moved to this week's lecture notes.

Topics (from course outline)

The following table shows the topics for this course week by week.

The topics in **bold** is for **this** week.

The topics in grey have been covered.

Other topics are for the future....

Week	Date	Topics
1	Jan 9	Introduction. Course overview. Intro to algorithms.
2	Jan 16	**Analyzing and designing algorithms. Recursion.**
3	Jan 23	Complexity analysis.
4	Jan 30	Recurrence equations. Data Structures.
5	Feb 6	Stacks and Queues.
6	Feb 13	Heapsort. Hashing.
	Feb 20	*Study week.*
7	Feb 27	Trees and Priority Queues.
8	March 6	Binary Search Trees (BST).
9	March13	Balanced BSTs (including Red-Black Trees)
10	March 20	Graphs.
11	March 27	Elementary graph algorithms.
12	April 3	Elementary graph algorithms. (continued)
13	April 20	Review

Topics (this week's lectures)

- Review
- Algorithm analysis examples
- Converting algorithms to C programs
- Discussion of lab 2
- Recursion
- Solving recurrences

Review

A problem of size n may be solved by an algorithm that could be:

- *Constant time*: the time to solve the problem is independent of the problem size. *Example:* Finding the largest (or smallest) item in a sorted list.
- *Logarithmic time:* the time to solve the problem is proportional to the logarithm of the size, i.e. $T(n) \propto \log n$. *Example:* "Binary search" to find an item in a sorted list.
- *Linear time:* $T(n) \propto n$. *Example:* Finding an item (or the maximum or minimum) in an unordered list.
- *Log-linear time*: $T(n) \propto n \log n$, *Example:* Optimal sort algorithms such as merge sort or heap sort.
- *Quadratic time:* $T(n) \propto n^2$ *Example*: Elementary sort algorithms such as bubble sort, insertion sort or selection sort.
- And an infinite other possibilities....

Euclid's algorithm for finding the Greatest Common Divisor (GCD)

Euclid's Algorithm: *Find greatest common divisor of* big *and* small *integers*

Non-recursive version

Step 1: Set *remainder* = *big* mod *small*
Step 2: If *remainder is 0,* answer is *small.* STOP
Step 3: Otherwise, Set *big* = *small* and Set *small* = *remainder*
Step 4: Go back to *Step 1.*

It was shown informally in class that this is a *logarithmic* algorithm.

NOTE:

One way to think of *logarithmic complexity* is that the algorithm running time is proportional to the number of digits in *n*. For example, a 12-digit number has twice as many digits as a 6-digit number. So a logarithmic algorithm that that solves a problem whose size is 6 digits in 1 second will be able to solve the problem whose size is 12 digits in 2 seconds.

And *log log n* complexity is even better. If *n* is expressed as a^b, then the time is proportional to the number of digits in the exponent, *b*. Suppose a log-log complexity algorithm that can solve a problem of size 10^{10} in 1 second. How big would the problem need to be to require 2 seconds? Answer: the unimaginably huge number 10^{100} (aka. A "googol"...)

Selection sort (a quadratic algorithm)

- Find the smallest value by examining all items. (i.e. look at *n* items.) Swap the smallest item with the first item.
- Now look at the next $n - 1$ items and swap the smallest with the second item.
- Continue until the items are sorted...
- You need to examine $n + (n-1) + (n-2) + \ldots 2 + 1 = (n^2 + n)/2$ items. The fastest growing term is *n-squared*, so it is a quadratic algorithm.

Merge Sort (an n log n algorithm)

- Split the items in 2.
- Continue splitting until each collection has only 1 item.
- Merge two items into a sorted pile of 2.
- Do this for all pairs of item collections.
- Merge the collections of 2 into collections of 4.

- Continue until the whole collection is sorted.
- The number of time you need to split is $\log_2 n$.
- Each *"merge"* requires examining *n* cards.
- Consequently, the complexity is *n* log *n*.

Converting an algorithm description into C

Euclid's algorithm (non-recursive solution)

Euclid's Algorithm: *Find greatest common divisor of* big *and* small *integers*
Non-recursive version

Step 1: Set *remainder* = *big* mod *small*
Step 2: If *remainder is 0,* answer is *small.* STOP
Step 3: Otherwise, Set *big* = *small* and Set *small* = *remainder*
Step 4: Go back to *Step 1.*

```
//Non-recursive version of gcd (in C)
unsigned int gcd(unsigned int big, unsigned int small) {
    unsigned int remainder = big % small;
    while(remainder!= 0) {
         big = small;
         small = remainder;
         remainder = small % remainder;
    }
    return small;
}
```

Euclid’s Algorithm (recursive solution)

> **Euclid's Algorithm**: *Find greatest common divisor of* big *and* small *integers*
> ***Recursive version***
>
> *Step 1:* Set *remainder* = *big* mod *small*
> *Step 2:* If *remainder is 0,* answer is *small.* STOP
> *Step 3:* Otherwise, find GCD of *small* and *remainder* using this algorithm

```
//Recursive version of gcd (in C)
unsigned int gcd(unsigned int big, unsigned int small) {
    unsigned int remainder = big % small;
    return remainder == 0 ? small : gcd(small, remainder);
}
```

- The recursive version is shorter and, arguably, more "elegant".
- Both versions implement the same algorithm and have the same computational complexity (they are both logarithmic: i.e. time $\propto \log small$).
- Many (most) algorithms in this course are best expressed recursively.
- This is especially true for "divide and conquer" algorithms.
- A partial proof that the complexity is logarithmic:
  - It can be shown that the worst possible case is when *small* and *big* are two sequential Fibonacci numbers.
  - The gcd is 1 in this case and all the smaller Fibonacci numbers are generated as remainders.
  - But $F_i = \lfloor \phi^n / \sqrt{5} + .5 \rfloor$ where $\phi = (1 + \sqrt{5})/2$ where F_i is the *i*'th Fibonacci number.
  - Consequently, in the worst case, the number of steps is $\log_\phi n$
  - In other words, Euclid's algorithm has *logarithmic* complexity. (Recall, the logarithm base is irrelevant.)

Analysis of algorithm complexity

MergeSort

```
        MergeSort Algorithm (deck of n cards, time = T(n))
Step 1: If there is only one card (or none), STOP. (time = a)
Step 2: Divide the deck of cards in 2.            (time = b)
```

```
Step 3: Sort the left deck using this algorithm.   (time = T(n/2))
Step 4: Sort the right deck using this algorithm.  (time = T(n/2))
Step 5: Merge the two decks.                       (time = cn + d)
```

- By definition, $T(n)$ is the time to sort *n* cards.
- (Note: we saw last week that merging two sorted decks is a linear algorithm which is why the merge step 5 is a linear function of n.)
- Adding up the times for each step, we get:
  $T(n) = a + b + 2T(n/2) + cn + d = 2T(n/2) + k_1 n + k_2$
- This kind of equation where the function value depends on its value for smaller arguments) is called a *recurrence*.

Solving recurrences

- There is no "universal" algorithm for solving recurrences. (This is similar to integration: sometimes you use integration by parts, sometimes trigonometric substitutions, sometimes other methods...)
- Consider the simplified recurrence: $T(n) = 2T(n/2) + n$
- We also need a base case. T(1) will be some constant. For simplicity, let's assume T(1) = 0.
- Working "bottom up", we have:
  - $T(2) = 2T(1) + 2 = 2$
  - $T(4) = 2T(2) + 4 = 8$
  - $T(8) = 2T(4) + 8 = 24$
  - $T(16) = 2T(8) + 16 = 64$
  - $T(32) = 2T(16) + 32 = 160$
  - $T(64) = 2T(32) + 64 = 384$
- Examining the numbers allows us to guess: $T(n) = n \lg n$
- If this is guess is correct, it can be proven using **mathematical induction**.

*What is **mathematical induction?***

- (Note: more explanation is available in a wikipedia article.)
- Is used to prove a formula with a single integer is true for all integers.
- You have to show that the formula is true for at least one base case. (Show it is trur for n = 1.)
- Example: Show that the sum of *n* integers is $S(n) = \frac{n(n+1)}{2}$
  - It is true for base cases for n = 1 and n = 2.
  - We assume that it is true for n and prove that it must also be true for *n* + 1:

- By definition: $S(n+1) = n+1+S(n) = n+1+\frac{n(n+1)}{2}$
- This can be re-written as: $S(n) = \frac{2(n+1)+n(n+1)}{2} = \frac{(n+1)(n+2)}{2}$ Q.E.D.

- Example: Show that the sum of squares ($S(n) = \sum_{i=0}^{n} i = n(n+1)(2n+1)/6$)[i]
  - True for base cases (*n* = 0 or 1 or 2...)
  - Inductive hypothesis: $S(n) = \sum_{i=0}^{n} i = n(n+1)(2n+1)/6$
  - Then, by definition:
    $S(n+1) = (n+1)^2 + S(n) = (n+1)^2 + \sum_{i=0}^{n} i = (n+1)^2 + n(n+1)(2n+1)/6$
  - Then: $S(n+1) = (n+1)^2 + n(n+1)(2n+1)/6 = (6(n+1)((n+1)+n(2n+1)))/6$
  - So: $S(n+1) = (6(n+1)((n+1)+n(2n+1)))/6 = ((n+1)(6n+6+2n^2+n))/6$
  - Factoring, get: $S(n+1) = ((n+1)(6n+6+2n^2+n))/6 = (n+1)(n+2)(2n+3)/6$
  - QED

Proving that T(n) = 2T(n/2) + n = n lg n by Mathematical Induction

- We modify induction. Instead of proving that if T(n) implies T(n+1), we will show that if T(n) is true then so is T(2n).
- In other words, we will assume that *n* is a power of 2 (i.e. $n = 2^i$ and do mathematical induction on *i*.)
- The "guess", $T(n) = 2T(n/2) + n = n \lg n$ is clearly true for several base cases (*n* = 1, 2, 4, 8...64) as demonstrated earlier.
- Our inductive hypothesis is $T(n) = n \lg n$
- We need to prove that $T(2n) = 2n \lg 2n$.
- By definition, $T(2n) = 2T(n) + 2n$.
- Using the inductive hypothesis, we obtain: $T(2n) = 2n \lg n + 2n$
- Factoring out *2n*, we obtain: $T(2n) = 2n(\lg n + 1)$
- Noting that $1 = \lg 2$, we can write: $T(2n) = 2n(\lg n + \lg 2)$
- Using the identity that $\log a + \log b = \log ab$, we can say $T(2n) = 2n \lg 2n$
- Q.E.D.

Finding a guess by "unfolding" (aka "substitution")

- Previously we calculated *T(n)* from the bottom up.
- We can "unfold" it from the "top down" as follows:

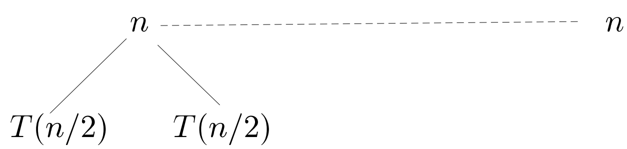
$$
\begin{aligned}
T(n) &= 2T(n/2) + n = 2(2T(n/4) + n/2) + n \\
&= 4T(n/4) + 2n \\
&= 8T(n/8) + 3n \\
&= 16T(n/16) + 4n
\end{aligned}
$$

etc...

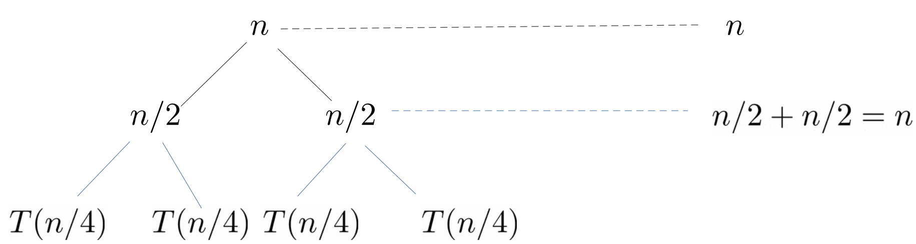
- If we assume that *n* is a power of 2, we would eventually obtain: $T(n) = nT(n/n) + n\lg n$
- Since we have assumed T(1) = 0, this implies $T(n) = nT(n/n) + n\lg n = nT(1) + n\lg n = n \times 0 + n\lg n = n\lg n$

Finding a guess by drawing a recursion-tree

- We start by representing $T(n) = 2T(n/2) + n = T(n/2) + T(n/2) + n$ as a graph where we put the non-recursive part (*n* in this case) on the top row and put each recursive part on a row below.



- We now expand the tree diagram downwards:



Asymptotic notation

Big-O

- We say that $f(n) = O(g(n))$ if there exist constants *c* and n_0 such that:

  $f(n) \leq cg(n)$ for all $n > n_0$

Big-Omega (Ω)

- We say that $f(n) = \Omega(g(n))$ if there exist constants *c* and n_0 such that:

  $f(n) \geq cg(n)$ for all $n > n_0$

Big Theta (Θ)

- We say that $f(n) = \Theta(g(n))$ if there exist constants c_1, c_2 and n_0 such that

  $c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n > n_0$
- Equivalently, $f(n) = \Theta(g(n))$ iff $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Lab 2: Towers of Hanoi and using Recursion

- See lab 2 and Chapter 2.6 of my book.
- The Wikipedia article gives a good overviewof this classic problem.

Towers of Hanoi algorithm

`Towers of Hanoi Algorithm`

Step 1: If there are no (zero) disks to move, do nothing. (Time: a)
Step 2: Otherwise, use *this* algorithm to move *n-1* disks to the *spare* tower. (Time: T(n-1))
Step 3: Move 1 disk from the source tower to the destination tower. (Time: b)
Step 4: Move *n-1* disks from the spare tower to the destination.

Analysis of Towers algorithm

- $T(n) = a + T(n-1) + b + T(n-1) = 2T(n-1) + a + b$
- Assume $a + b = 1$; so $T(n) = 2T(n-1) + 1$
- Assume $T(1) = 1$
- Then $T(2) = 2T(1) + 1 = 3$;
- Similarly, $T(3) = 2T(2) + 1 = 7$
- And, $T(4) = 2T(3) + 1 = 15$
- It looks like $T(n) = 2^n - 1$
- Proof by mathematical induction:

Show $T(n) = 2^n - 1$ implies $T(n+1) = 2^{n+1} - 1$

By definition: $T(n+1) = 2T(n) + 1 = 2(2^n - 1) + 1 = 2^{n+1} - 1$ (QED)

Questions

1. An algorithm with complexity $\Theta(\sqrt{n})$ takes 6 ms to solve a problem of size 1600. Estimate the time to solve a problem of size 10,000.
2. Draw a recursion tree for $T(n) = 2T(n/2) + k_1 n + k_2$. Guess the exact solution and prove it by mathematical induction.
3. Draw a recursion tree for $T(n) = 2T(n/4) + T(n/2) + n$. Guess the solution. Try to prove it.
4. A proposed simpler implementation for Euclid's algorithm is:

```
unsigned long gcd(unsigned long big, unsigned long small) {
      return small == 0 ? big : gcd(small, big % small);
}
```

Will this work? Explain. (You can try it!)

References (text book and online)

- *CLRS:* Chapter 1, 2, 3.1
- kclowes book: Chapter 1, 2.1. 2.2, 2.6, 4.1, 4.2, 4.3