COE428 Lecture Notes Week 1 (Week of January 9, 2017)

Table of Contents

Announcements

- Course management distributed in class last week.
- No labs this week. Labs start next Monday, January 16.
- Counselling hours (ENG-449): Mon 1 pm – 2 pm, Fri 2 pm – 3 pm, kclowes@ryerson.ca

Topics

- An informal look at some basic algorithms

- Algorithm analysis examples
- Big-O ("Big Oh"), Big-Θ ("Big Theta"), Big-Ω ("Big Omega") notations
- Discussion of lab 1

Informal analysis of selected algorithms

Searching an unordered list (a linear algorithm)

- How many items do you have to look at?
- If very lucky, the first one you look at is the one you're trying to find! This is the *best case*.
- But if you're unlucky, it may be the last one you look at.
- If there are *n* items, you have to look at all of them in the *worst case*.
- If the item you are looking for is not in the list, you have to look at all *n* of them.
- In this course, we are almost always interested in the *worst case.*
- Searching a list with *n* items requires examining *n* elements in the worst case.
- If *n* is bigger, it takes more time.
- For example, if it takes you 1 minute to search a deck of 100 cards, then it will take 3 minutes to search a deck of 300 cards.
- Since the time for searching is proportional to the size of the list (*n*), we call this a linear *algorithm*.

Searching an unordered list for minimum/maximum (a linear algorithm)

- All items need to be examined.
- Consequently, this is also a linear algorithm.

Searching an ordered list for minimum (or maximum) (a constant time algorithm)

- A very simple algorithm:
  - If looking for minimum, just look at the first item.
  - I looking for the maximum, just look at the last item.
- No matter how big *n* is, you only have to look at one item.
- This is a *constant time* algorithm.

Searching an ordered list for a specific value (a logarithmic algorithm)

- The algorithm:

- Look at the middle element.
- If it is what you are searching for, STOP.
- Otherwise, if the item you want is smaller than the middle element, repeast the search in the first half of the list; other wise repeat the search in the second half.

- In either case, the problem size is cut in half.
- For example, if *n* were 63, the problem would reduce to searching 31 elements. Then 15, then 7, then 3 and then 1 element.
- This is a *logarithmic algorithm*.

Remarks on algorithmic complexity

- 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.
- Knowing only the time for a particular size and the nature of the algorithm, it is possible to estimate the time for other sizes (assuming the size is large). Some examples:
  - If a linear algorithm solves a problem of size 3000 in 5 ms, how long will it take to solve a problem of size 9000?
    - *Answer*: 15 ms, because if the size is is 3 times bigger, the time will also increase by a factor of 3.
  - If a quadratic algorithm solves a problem of size 3000 in 5 ms, how long will it take to solve a problem of size 9000?
    - Answer: 45 ms, because if the size is is 3 times bigger, the time will increase by a factor of $3^2 = 9$.
  - If a logarithmic algorithm solves a problem of size 1000 in 5 ms, how long will it take to solve a problem of size 1,000,000?
    - Answer: 10 ms, because the logarithm of 1,000,000 is twice as big as the logarithm of 1000. i.e. $\frac{\log 1{,}000{,}000}{\log 1000} = \frac{\log 10^6}{\log 10^3} = \frac{6 \log 10}{3 \log 10} = 6/3 = 2$

What is an algorithm?

An algorithm is a precise description of how to solve a problem and must have the following features:

- It must terminate in finite time.
- Each step must be precisely defined. (No ambiguity allowed.)
- There must be zero or more inputs.
- There are one or more outputs.

Analysis of selected algorithms

Selection sort

SelectionSort Algorithm: *Sort n cards*

Step 1: If there are no cards to sort, then STOP. (*Time per step: a; number steps: n + 1*)

Step 2: Otherwise, find the smallest card, remove it and place it on top of the sorted card pile. (*Time per step: X -- see below; number of steps: n*)

Step 3: Go back to step 1. (*Time per step: c; number of steps: n*)

- Hence total time to sort = $a(n+1) + Xn + cn$
- **Problem**: Step 2 is **not** elementary; the time it takes depends on how many cards we have to examine to determine the minimum. Thus X is not a constant. To determine the minimum we have to examine *n* cards the first time, then *n-1* the second time, *n-2* the third time, etc. If the time to examine a card is *2b*, then:
- So, total time to sort = $a(n+1) + bn(n+1) + cn = bn^2 + (a+b+c)n + a$
- SelectionSort is a **quadratic** algorithm.

Euclid's algorithm (and a first look at recursion)

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.*

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

```
//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;
}
```

```
//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.)

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: merging two sorted decks is a linear algorithm which is why the merge step 5 is a linear function of n. It is linear because we simply choose the smaller of the top card in the two decks and add it to the merged deck.)
- 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: $S(n+1) = \frac{2(n+1)+n(n+1)}{2} = \frac{(n+1)(n+2)}{2}$ Q.E.D.

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:

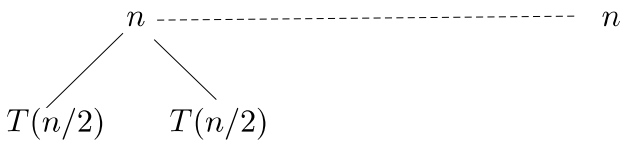
$$\begin{aligned} T(n) &= 2T(n/2) + n = 2(2T(n/4) + n/2) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n \\ &= 16T(n/16) + 4n \end{aligned}$$

etc...

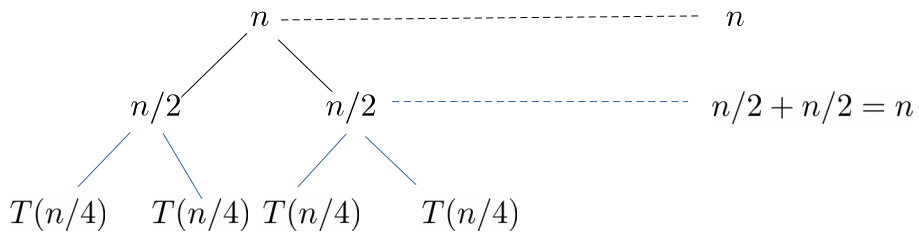
- If we assume that *n* is a power of 2, we would eventually obtain: $T(n) = nT(n/n) + n \lg n$
- Since we have assumed T(1) = 0, this implies
  $T(n) = nT(n/n) + n \lg n = nT(1) + n \lg n = n \times 0 + n \lg n = n \lg n$

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:

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

Questions

Questions from these notes

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!)

Questions from CLRS textbook

1-1, 2-1

Questions from my book

1.1, 1.2, 1.3,.1.4, 1.5, 1.9,1.13,1.15, 1.20, 1.22

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 (Click here)
- My C programming notes. (Click here)
- Wikipedia article on algorithms.