Algorithm Analysis ~ 12

CSE/IT 122 ~ Algorithms & Data Structures

Divide and Conquer

- ➔ A lot of algorithms are recursive in nature
  - They call themselves one or more times recursively to deal with closely related problems
- ➔ Divide and Conquer
  - Break the problem into several subproblems that are similar to the original problem but smaller in size
  - Solve the subproblems recursively
  - Combine the solutions to come up with a solution for the original problem
  - Divide/Conquer/Combine

Solving Recurrences

- ➔ Solve f(n) = f(n/2) + 1; f(1) = 1
- ➔ Top Down Approach
  - f(n) = f(n/2) + 1
  - f(n/2) = f(n/4) + 1
  - Substituting in f(n) gives
    - f(n) = (f(n/4)+1)+1 = f(n/4) + 2
  - And f(n/4) = f(n/8) + 1
  - Substituting in f(n) gives
    - f(n) = (f(n/8)+1)+2 = f(n/8) + 3
  - Thus in general, we begin to see the following pattern
    - f(n) = f(n/2^k) + k

Solving Recurrences

- $f(n) = f(n/2^k) + k$
  - Let's assume that $n = 2^k$, then $n/2^k = 1$
  - Thus we can get an explicit formula when $n = 2^k$
  - Since $n = 2^k$, $k = \log(n)$
  - Substituting, you get $f(n) = 1 + \log(n)$
  - How do we know this is correct?
    - You guessed … INDUCTION
    - $f(1) = 1 + \log(1) = 1$
    - Assume: $f(k) = 1 + \log(k)$
    - Show: $f(2k) = 1 + \log(2k)$
    - $f(2k) = f(k)+1 = 1+(1+\log(k)) = 1+(\log(2)+\log(k)) = 1+\log(2k)$
  - Thus $T(n) = f(n) = O(\log(n))$

General Patterns

```
doodle(n,m)
    if (n > 0)
        draw_line(n, n, m, m); //cost of c
        draw_line(n, m, n, m); //cost of c
        doodle(n -1, m);
```

- When n > 0 does nothing, or draws two lines and calls doodle recursively
- Find the running time.
- Assume draw_lines is some constant time
- T(0) = 0

Doodle Example:

- ➔ $T(n) = T(n-1) + 2c$
- ➔ Unroll
  - $T(n-1) = T(n-2) + 2c$
- ➔ Substitute
  - $T(n) = (T(n-2)+2c)+2c = t(n-2) + 4c$
  - $T(n-2) = T(n-3)+2c$
  - $T(n) = (T(n-3)+2c)+4c = T(n-3)+6c$
- ➔ Generalize
  - $T(n) = T(n-k) + 2ck$
  - When $n - k = 0$, $T(n-k) = 0$
  - So $T(n) = T(0) + 2cn = 2cn$
  - $T(n) = O(n)$

Foo Example

```
foo(n)
    if(n == 0)
        return 1;
    else
        return foo(n - 1) + foo(n - 1)
```

- If n > 0, calls foo twice recursively; otherwise just a single return
- Find the running time.
- Assume T(0) = c

Foo Example

- ➔ $T(n) = 2T(n-1) + d$ to be the cost of addition
- ➔ Unroll
  - $T(n) = 2T(n-1) + d$
  - $T(n-1) = 2T(n-2)+d$
  - $T(n) = 2(2T(n-2)+d)+2 = 2^2T(n-2)+2d+d$
  - $T(n-2) = 2T(n-3)+d$
  - $T(n) = 2^2(2T(n-3)+d)+2d+d = 2^3(n-3)+2^2d+2d+d = 2^3T(n-3)+(2^2+2^1+2^0)d$
- ➔ In general
  - $T(n) = 2^kT(n-k)+(2^{k-1}+2^{k-2}+...2^1+2^0)d$
  - $T(n) = 2^kT(n-k)+\sum^{k-1}_{i=0}(2^i)$
  - $T(n) = 2^kT(n-k)+2^k-1$
  - And when $n-k=0$, $n=k$
  - So $T(n) = 2^nT(0)+2^n*c+2^n-1 = (c+1)2^n-1$
  - $T(n) = O(2^n)$