# CSE 326
# Lecture 7

Spring Quarter 1995



## Order Notation

- $T(n) = O(f(n))$ if there are constants c and $n_0$ such that $T(n) \le cf(n)$ for all $n \ge n_0$.
- $T(n) = \Omega(f(n))$ if there are constants c and $n_0$ such that $T(n) \ge cf(n)$ for all $n \ge n_0$.
- $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.
- Examples:

  $10{,}000\ n^2 + 25\ n = \Theta(n^2)$

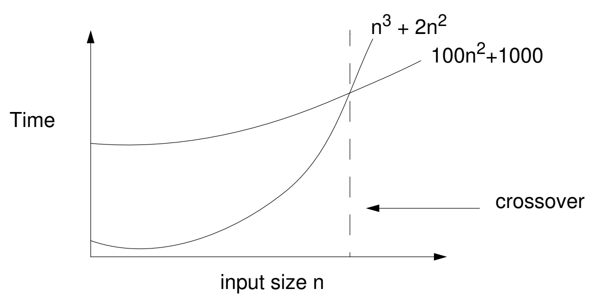
  $10^{-10}\ n^2 = \Theta(n^2)$

  $n \log n = O(n^2)$

  $n \log n = \Omega(n)$



## Understanding Orders of Magnitude






## Some Basic Relationships

- if for all $c > 0$ there is an $n_0$ such that $f(n) \le cg(n)$ for all $n \ge n_0$ then f(n) will crossover g(n).

  Example: $10{,}000\ n^2$ and $10\text{-}9\ n^3$
- Polynomial time is time $O(n^k)$ for some k.
- Exponential time is time $O(c^n)$ for some $c > 1$.
- Logarithmic time is time $O(\log n)$

  Note that in order notation the base of the logarithm is ignored. Why? Because $\log_k n = \log_k 2 \log_2 n$ for any k.
- $c^n = \Omega(n^k)$ for any $c > 1$ and any k.
- $\log^k n = O(n)$ for any k.

## Kinds of Analysis

- Time or Space
- Upper bounds or lower bounds
  - T(n) = O(f(n)) implies f(n) is an upper bound
  - T(n) = Ω(f(n)) implies f(n) is a lower bound
- Worst case - worst over all possibilites
- Average case - assuming some distribution of inputs.
- Complexity averaged over many operations vs. complexity of each operation.
- "Common case analysis"



## Example 1

- For loops - assume k = 0 initially.

```
for i = 1 to n do
  for j = 1 to n do
    k := k + 1
```

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{n} 1\right) = \sum_{i=1}^{n} n = n^2$$



## Example 2

- ```
  for i = 1 to n do
    for j = i to n  do
      k := k + 1
  ```

$$\sum_{i=1}^{n}\left(\sum_{j=i}^{n} 1\right) = \sum_{i=1}^{n} (n-i+1) = \sum_{i=1}^{n} (n+1) - \sum_{i=1}^{n} i = n(n+1) - \frac{n(n+1)}{2}$$

$$\frac{n(n+1)}{2} = O(n^2)$$



## Example 3

- Conditional

  if C then S1 else S2

  time = time of C + max(time of S1, time of S2)
- While loop

  while C do S

  time = sum of times of each iteration of the loop
  ```

## Example 4

- Recursion
- Analysis amost always yields a recurrence of some form.
- Recursive max.

  $T(0) \leq b$

  $T(n) \leq c + T(n-1)$ if $n > 0$
- Analysis

  $T(n) \leq c + c + T(n-2)$ by substitution

  $\leq c + c + c + T(n-3)$ by substitution again

  $\leq kc + T(n - k)$ for all k from 1 to n.

  $\leq nc + T(0)$

  $\leq cn + b$
- $T(n) = O(n)$



## Example 5

- Binary Search -

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 12 | 30 | 35 | 75 | 83 | 87 | 90 | 97 | 99 |

- $T(1) \leq b$

  $T(n) \leq T(n/2) + c$ if $n > 1$
- Analysis

  $T(n) \leq T(n/2) + c$
  $\leq T(n/4) + c + c$
  $\leq T(n/8) + c + c + c$
  $\leq T(n/2^k) + kc$
  $\leq T(1) + kc$ where $n = 2^k$
  $\leq b + c \log_2 n$
  $= O(\log n)$



## Example 6

- Mergesort

  Basic idea: mergesort the first half, mergesort the second half then merge the two halves together.
- $T(1) \leq b$

  $T(n) \leq 2T(n/2) + cn$ if $n > 1$
- Analysis

  $T(n) \leq 2T(n/2) + cn$
  $\leq 2(2T(n/4) + cn/2) + cn$
  $= 4T(n/4) + cn + cn$
  $\leq 4(2T(n/8) + c(n/4)) + cn + cn$
  $= 8T(n/8) + cn + cn + cn$
  $\leq 2^k T(n/2^k) + kcn$
  $\leq 2^k T(1) + kcn$ where $n = 2^k$
  $\leq bn + cn \log_2 n$
  $= O(n \log n)$



## Example 7

- Recursive Fibonacci
- Fib(n) {

```
case {
  n = 0:  return(0);
  n = 1:  return(1);
  n > 1:  return(Fib(n-1) + Fib(n-2))}}
```

- Lower bound analysis

  $T(0), T(1) \geq b$

  $T(n) \geq T(n-1) + T(n-2) + c$ if $n > 1$
- Analysis

  let $\phi = \frac{1 + \sqrt{5}}{2}$ which satisfies the property $\phi^2 = \phi + 1$

  We show by induction on n that $T(n) \geq b\phi^{n-1}$

## Example 7 (Cont')

- Basis: $T(0) \geq b > b\phi^{-1}$ and $T(1) \geq b = b\phi^{0}$
- Inductive Step:

  Assume that $T(m) \geq b\phi^{m-1}$ for all $m < n$.

  $T(n) \geq T(n-1) + T(n-2) + c$

  $\geq b\phi^{n-2} + b\phi^{n-3} + c$

  $\geq b\phi^{n-3}(\phi + 1) + c$

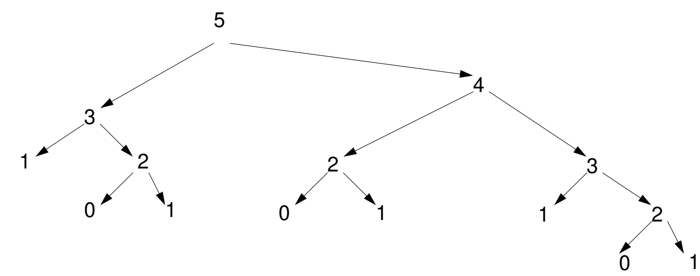
  $= b\phi^{n-3}\phi^{2} + c$

  $\geq b\phi^{n-1}$
- The bottom line is that this version of recursive Fibonacci runs in exponential time.

  This is an example where the same recursive call is made numerous times.



## Fibonacci Call Tree






## Strategy to Avoid Repeated Recursive Calls

- Create an array or other storage structure called the *memoization structure* and initialize it to some value that cannot be returned by any recursive call.

1. All the basis values are stored in the memoizaiton structure.
2. The return value of a recursive call is remembered in the memoization structure.
3. Before making any recursive call check the memoization structure to see if the value has already been computed. If not compute it, otherwise use the already computed value.



## Efficient Fibonacci using Memoization

- Array F[0..N] where N is larger than any Fibonacci number to be computed, initialize F to -1.
- Fib(n) {

```
F[0] := 0;
F[1] := 1:
case {
  n = 0:  return(F[0]);
  n = 1:  return(F[1]);
  n > 1: { if F[n-1] = -1 then F[n-1] := Fib(n-1);
           if F[n-2] = -1 then F[n-2] := Fib(n-2);
           return(F[n-1] + F[n-2])}}}
```

- This version of recursive Fibonacci runs in O(n) time instead of exponential time!
- Naturally, there is a fairly simple iterative version of Fibonacci which only remembers the last two values computed.