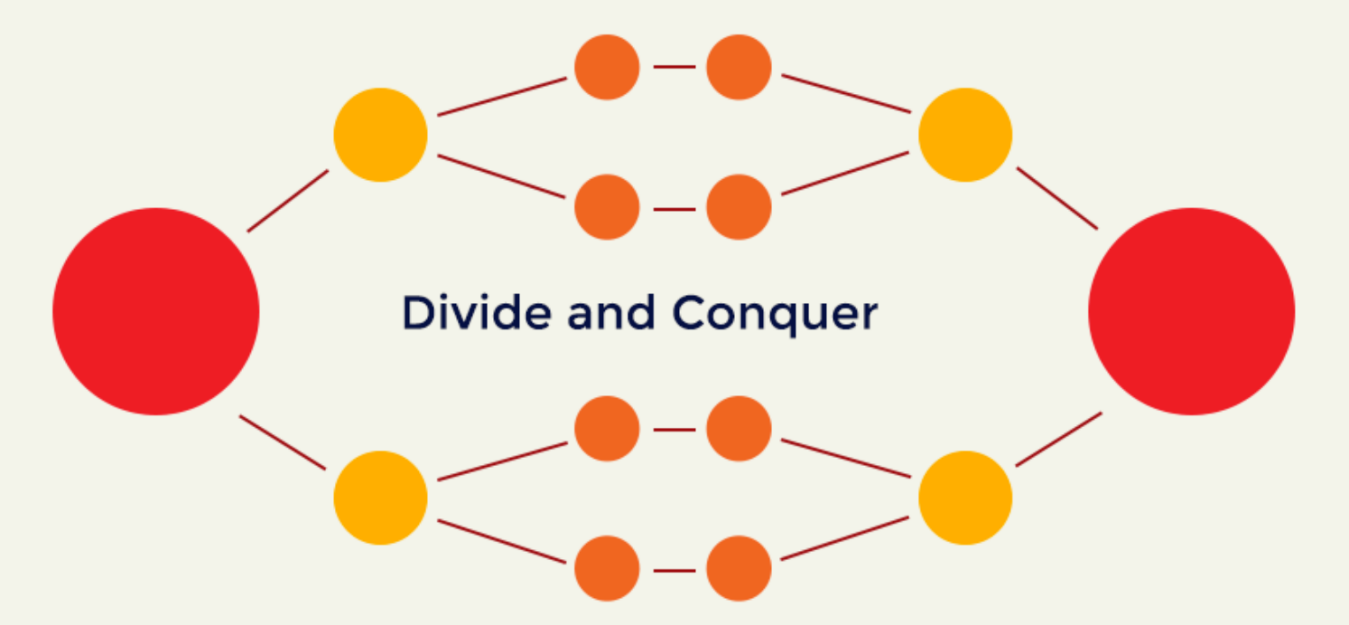
CSCI 3104: Algorithms

Lecture 5: More Asymptotic Analysis, Divide and Conquer

Rachel Cox

Department of Computer Science



What will we learn today?

- ❑ Asymptotic Analysis
- ❑ Proving Order of Growth
- ❑ Divide and Conquer Algorithms
- ❑ Computation Trees
- ❑ Recursion
- ❑ Unrolling Method

Intro to Algorithms, CLRS: Section 2.3, Section 3.1

Asymptotic Notation

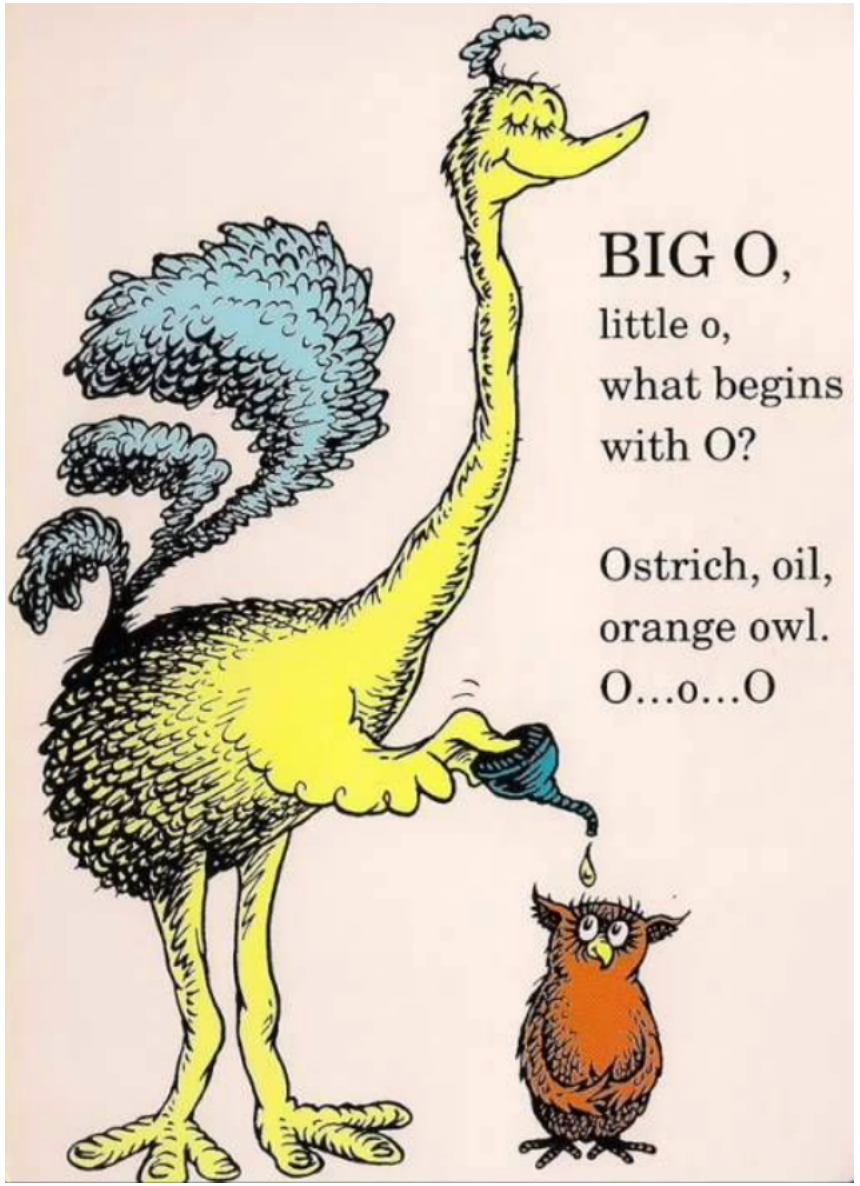
”Big O”: $\mathcal{O}$ used to describe asymptotic $\leq$

“Little o”: o used to describe the asymptotic $<$

“Big Omega”: Ω used to describe the asymptotic $\geq$

“Little omega”: ω used to describe the asymptotic $>$

“Theta”: Θ used to describe the asymptotic $=$



Limits at Infinity – Review!

$$\log^q(n) < n^p < a^n < n! < n^n$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \implies f(n) < o\big(g(n)\big)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \implies f(n) > \omega\big(g(n)\big)$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \implies f(n) = \Theta\big(g(n)\big),$$

L'hospital's Rule: $\lim_{x\to c} \frac{f(x)}{g(x)} = \lim_{x\to c} \frac{f'(x)}{g'(x)}$

$\frac{\infty}{\infty}$, $\frac{0}{0}$

Common Functions

function	description
1	constant, independent of n
$\log n$	sublinear, specifically logarithmic
n^c for $c < 1$	sublinear
n	linear
$n \log n$	super-linear, but much less than quadratic
n^2	quadratic
n^3	cubic
n^c for $c > 1$	polynomial, super-linear
c^n for $c > 1$	exponential
$n!$ or n^n	extremely fast-growing functions

polynomial

exponentials

factorials

Order of Growth

Example: Construct specific functions $f(n)$ and $g(n)$ such that $f(n) = \Theta\big(g(n)\big)$ but $3^{f(n)} \neq \Theta\big(3^{g(n)}\big)$.

Suppose $f(n) = n$ and $g(n) = 2n$

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n}{2n} = \frac{1}{2} \Rightarrow f(n)$ is $\Theta(g(n))$ *

- Next let's consider

$$\lim_{n\to\infty} \frac{3^{f(n)}}{3^{g(n)}} = \lim_{n\to\infty} \frac{3^n}{3^{2n}} = \lim_{n\to\infty} \frac{3^n}{(3^n)^2} = \lim_{n\to\infty} \frac{1}{3^n} = 0$$

$$\Rightarrow 3^{f(n)} \text{ is } o(3^{g(n)})$$

- Hence we can conclude that $3^{f(n)} \neq \Theta(3^{g(n)})$

Order of Growth

Example: Which function would you prefer to represent the time complexity of your algorithm?

$T_1(n) = n^{100}$ }

$T_2(n) = n^{1+0.02\ln(n)}$ }

Compare 100 with $1 + 0.02\ln(n)$

Note that $1 + 0.02\ln(n)$ will go to infinity. So we may decide that we prefer $T_1(n)$

$100 = 1 + 0.02\ln(n)$

$99 = 0.02\ln(n)$

$4950 = \ln(n)$

$e^{4950} = n$

This is a huge number.

Not a cut and dried answer!

e.g. Consider $n = 1{,}000{,}000$ $T_2(n) = n^{1.12}$

→ So for practical purposes, we might pick $T_2(n)$.

Asymptotic Analysis

Example: Prove the claim that $2^{n+10} = O(2^n)$

Proof: Need to choose constants c, n_0 such that

$$2^{n+10} \leq c \cdot 2^n \quad \text{for} \quad n \geq n_0$$

$$2^{n+10} = 2^{10} \cdot 2^n$$

$$= 1024 \cdot 2^n$$

Let $c = 1024$ and $n_0 = 1$. Then we've shown that 2^{n+10} is $O(2^n)$ as desired. ∎

Asymptotic Analysis

Example: Prove the claim that $2^{10n} \neq \mathcal{O}(2^n)$

Proof: By contradiction. Suppose that 2^{10n} is $\mathcal{O}(2^n)$.

Then, there exist constants $c, n_0 > 0$ such that

$$2^{10n} \leq c \cdot 2^n \quad \text{for all } n \geq n_0$$

$$\frac{2^{10n}}{2^n} \leq c$$

$$2^{10n-n} \leq c$$

$$2^{9n} \leq c$$

This cannot be true for all $n \geq n_0$. We've found a contradiction so it must be the case that 2^{10n} is not $\mathcal{O}(2^n)$.

Asymptotic Analysis

$x = \log_2 2^x$

$x = 2^{\log_2 x}$

Inverse Property of logs + exponents

Example: Prove the claim that $n^{1/\log_2(n)} = \Theta(1)$.

Proof: Notice that $n^{\frac{1}{\log_2(n)}} = 2^{\log_2\left(n^{\frac{1}{\log_2(n)}}\right)}$

Power Rule for logs: $\log b^a = a \log b$

$= 2^{\frac{1}{\log_2(n)} \cdot \log_2(n)}$

$= 2^1$

So the claim we are being asked to show is really that $2 = \Theta(1)$

$$\lim_{n\to\infty} \frac{n^{1/\log_2(n)}}{1} = \lim_{n\to\infty} \frac{2}{1} = 2$$

Thus we can conclude that $n^{1/\log_2(n)}$ is $\Theta(1)$, as desired.

∎

Asymptotic Analysis

Example: Let $f(n) = (n-3)!$ and $g(n) = 3^{5n}$. Determine which of the following relations best applies: $f(n) \in \mathcal{O}(g(n))$, $f(n) \in \Omega(g(n))$, or $f(n) \in \Theta(g(n))$.

Factorials grow faster at infinity than exponentials.

Therefore we expect $\lim_{n\to\infty} \frac{(n-3)!}{3^{5n}} = \infty$

Let's compute $\lim_{n\to\infty} \frac{3^{5n}}{(n-3)!}$.

Recall the Squeeze theorem.
if $g(x) \le f(x) \le h(x)$
and $\lim_{x\to\infty} g(x) = \lim_{x\to\infty} h(x) = L$
$\Rightarrow$ thus $\lim_{x\to\infty} f(x) = L$

$\frac{3^{5n}}{(n-3)!} \ge 0$

$\frac{3^{5n}}{(n-3)!} = \frac{(3^5)^n}{(n-3)!} = \frac{(3^5)^3 (3^5)^{n-3}}{(n-3)!} = (3^5)^3 \cdot \frac{(3^5)(3^5)(3^5)\cdots(3^5)(3^5)(3^5)\cdots(3^5)(3^5)(3^5)}{(n-3)(n-4)(n-5)\cdots(3^5+1)(3^5)(3^5-1)\cdots 3\cdot 2\cdot 1}$

$= C$, with the factor $(3^5)(3^5)(3^5) = 1$

Asymptotic Analysis

$o(g(n)) \Rightarrow O(g(n))$
But not other way.

Example: Let $f(n) = (n-3)!$ and $g(n) = 3^{5n}$. Determine which of the following relations best applies: $f(n) \in \mathcal{O}(g(n))$, $f(n) \in \Omega(g(n))$, or $f(n) \in \Theta(g(n))$.

$$\frac{3^{5n}}{(n-3)!} = (3^5)^3 \cdot C^* \cdot \frac{3^5}{n-3} \cdot \frac{3^5}{n-4} \cdot \frac{3^5}{n-5} \cdots \frac{3^5}{3^5+1}$$

$$\leq (3^5)^3 \cdot C \left(\frac{3^5}{3^5+1}\right)^{(n-3)-3^5}$$

$$\lim_{n\to\infty} 0 = 0 \quad \text{and} \quad \lim_{n\to\infty} (3^5)^3 C \left(\frac{3^5}{3^5+1}\right)^{n-3-3^5} = 0$$

$\left(\frac{3^5}{3^5+1}\right) < 1$

Thus by the Squeeze Theorem

$$\lim_{n\to\infty} \frac{3^{5n}}{(n-3)!} = 0$$

$\Rightarrow$ this implies that 3^{5n} is $o((n-3)!)$

$\Rightarrow 3^{5n}$ is $O((n-3)!)$

$\Rightarrow (n-3)!$ is $\Omega(3^{5n})$

Divide and Conquer

Idea behind this method:

1. **Divide**: Take an instance and divide it into smaller instances of the same problem.

2. **Conquer**: if: smaller instance is trivial, then solve it directly
   else: divide again

3. **Combine**: Take the solutions for the smaller instances and combine them to give the solution to the larger instance.

Divide and Conquer

Connection to recursion:

```
def solve(A):
    if A is trivial:
        return the trivial solution

    else:
        B, C = split(A)
        S_1 = solve(B)
        S_2 = solve(C)

        return combine(S_1, S_2)
```

} Base Case of Conquer part

Divide into smaller problems

} Conquer

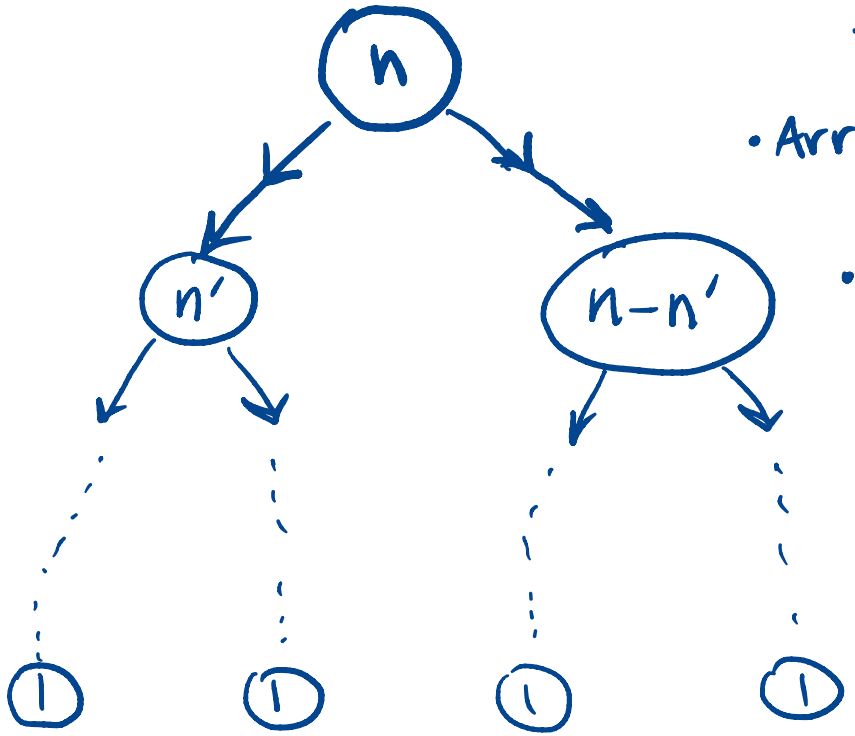
} combine

- We can visualize what's happening in a divide and conquer algorithm with a computation tree.

Divide and Conquer

There is a computation tree associated with divide and conquer algorithms.

Example 1: Generic split, n', and $n - n'$



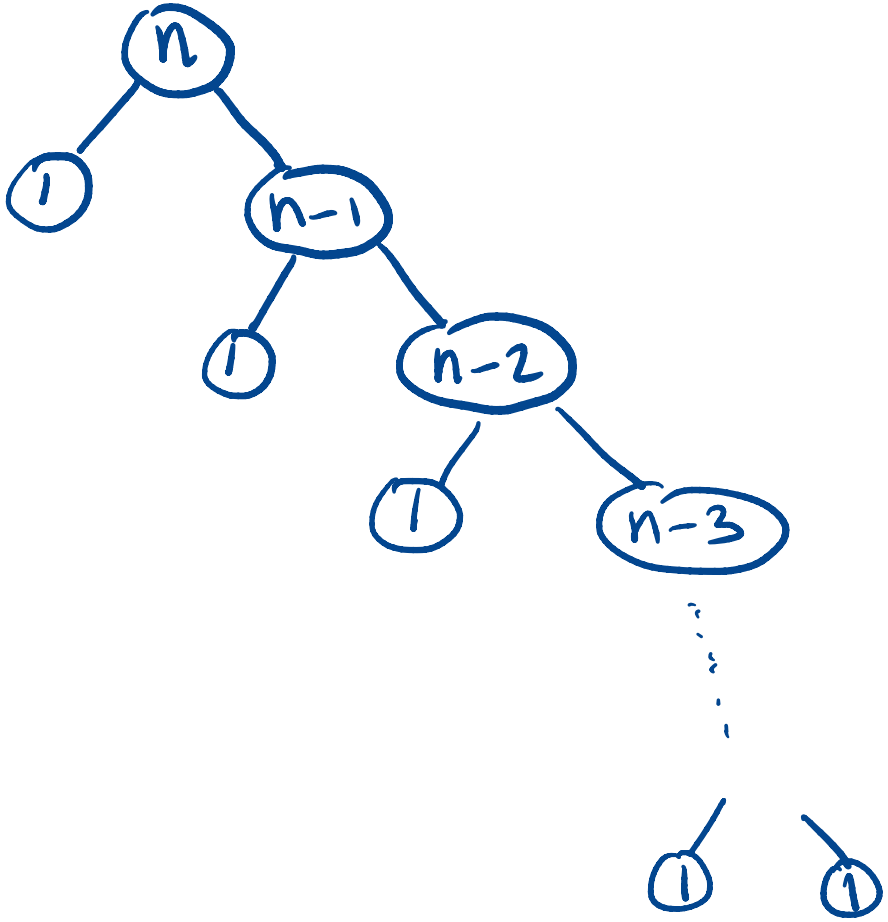
- instance of a problem with input size n
- Arrows indicate a recursive call.
- Each level indicates a split
  - smaller instances ↓

- Eventually we get to the "base cases"
  - these are the instances with a trivial solution.

Divide and Conquer

There is a computation tree associated with divide and conquer algorithms.

Example 2: Always split into $n-1$ and 1



} number of divisions into subproblems is n

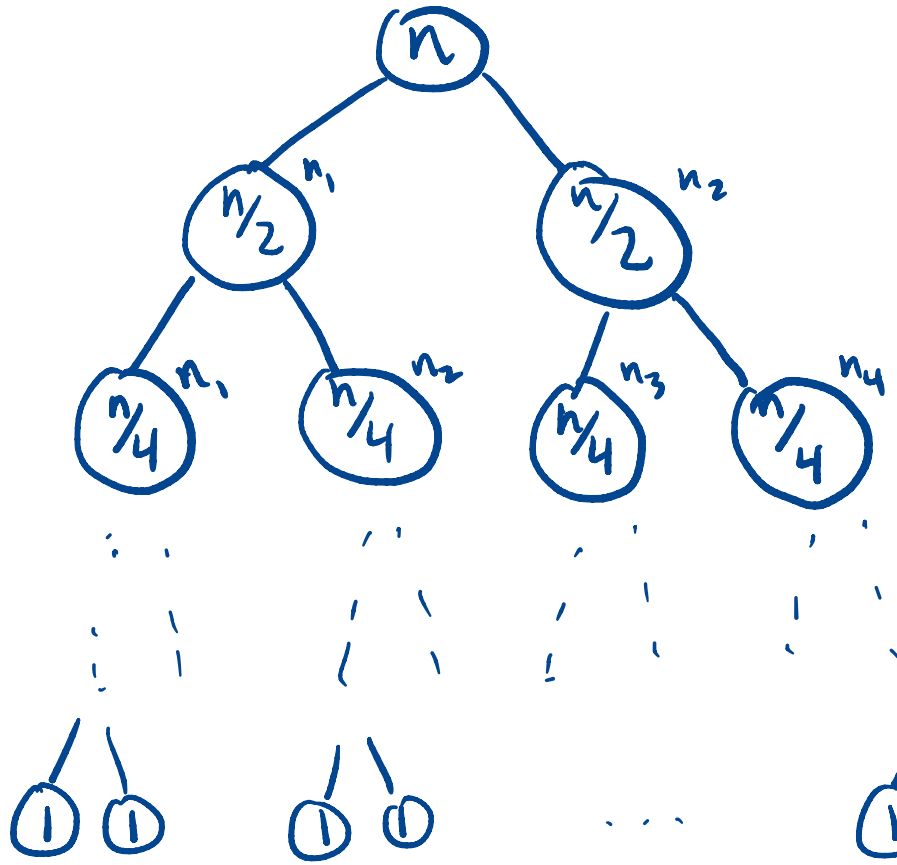
$\Rightarrow$ runtime is $\Theta(n)$

(doesn't matter if we a split $n-3, 3$ or $n-4, 4$ or $n-k, k$ ... still $\Theta(n)$)

Divide and Conquer

There is a computation tree associated with divide and conquer algorithms.

Example 3: Always split in half



Runtime?

Suppose $n = 2^k$

1st split: $n_1 = \frac{2^k}{2} = 2^{k-1}$

$n_2 = \frac{2^k}{2} = 2^{k-1}$

2nd split: $n_1 = \frac{2^{k-1}}{2} = 2^{k-2}$, $n_2 = \frac{2^{k-1}}{2} = 2^{k-2}$

$n_3 = 2^{k-2}$, $n_4 = 2^{k-2}$ } 4 arrays of size 2^{k-2}

:

kth split: 2^k arrays of size 1

How many levels does the computation tree have? k levels?

$n = 2^k \Rightarrow k = \log_2(n) \Rightarrow \boxed{\Theta(\log n)}$

Divide and Conquer

- Binary Search is an example of a Divide and Conquer algorithm.

```
Binary Search (A, key)
c1   . n = len(A)
c2   . if n == 0
c3   .     return not found
c4   . m = ⌊(n+1)/2⌋        find the middle index
c5   . if key > A[m]                ← upper half of list
     .     Binary Search (A[m+1, ..., n], key)
c6   . else if key < A[m]           ← lower half of list
           Binary Search (A, [1, ..., m-1], key)
c7   . else
           return found
```

- **Best Case**
  when $A[m] = key$
  $\Theta(1)$

- **Worst Case**
  technically the most work is done when the key is not found.

→ Next, we will write a recursion to help us find the complexity

Divide and Conquer

Example: Use **recursion** and the "**unrolling method**" to find the best-case and worst-case time complexity of Binary Search.

- Let $T(n)$ be an upper bound on the number of operations on an input of size n.

- After one recursive call, the input size $\leq n/2$

$$T(n) \leq c_1 + T(n/2)$$

c_1: constants outside the recursive call.

$T(n/2)$: recursive call on the input that is half the original size

} This is our Recursion.

technically: $T(n) \leq O(1) + T(n/2)$

$T(0) = O(1)$

$T(1) = O(1)$

may either go straight to "not found" or we may recurse exactly once and we consider that constant time.

Divide and Conquer

Example: (continued) Use **recursion** and the "**unrolling method**" to find the best-case and worst-case time complexity of Binary Search.

We now solve this recursion with the "unrolling method"

$$T(n) \le c_1 + T(n/2)$$
$$\le c_1 + (c_1 + T(n/4))$$
$$= 2c_1 + T(n/4)$$
$$\le 2c_1 + (c_1 + T(n/8))$$
$$= 3c_1 + T(n/8)$$
$$\le 3c_1 + (c_1 + T(n/16))$$
$$= 4c_1 + T(n/16)$$
$$\vdots$$
$$T(n) \le k c_1 + T(n/2^k)$$

- Keep applying the recursive relationship and look for a pattern.
- continue until we reach a base case.

Divide and Conquer

Example: (continued) Use **recursion** and the "**unrolling method**" to find the best-case and worst-case time complexity of Binary Search.

When can we stop.

$\left\lfloor \frac{n}{2^k} \right\rfloor \leq 1$ equivalent to $\frac{n}{2^k} \leq 1$

$n \leq 2^k$

$\log_2 n \leq k$

$\Rightarrow$ At $k = \lceil \log_2 n \rceil$ we must hit a base case.

$\Rightarrow T(n) \leq kc_1 + T(\text{either } 0 \text{ or } 1)$

$\leq kc_1 + c_2$

$= c_1 \lceil \log_2 n \rceil + c_2$

$\leq c_1 (\log_2 n + 1) + c_2$

$\leq c_3 \log_2 n$

$\Rightarrow T(n)$ is $\Theta(\log n)$

$\log_2 n = \frac{\log n}{\log 2}$

Next Time

- Recurrence relations - unrolling method, tree method, master method