DSC 40B

Lecture 9 : Recurrences

Recurrence Relations

Recurrence Relations

- Last Lecture: We found

$$T(n) = \begin{cases} T(n/2) + \Theta(1), & n \geq 2 \\ \Theta(1), & n = 1 \end{cases}$$

- This is a recurrence relation.

Solving Recurrences

- We want simple, non-recursive formula for $T(n)$ so we can see how fast $T(n)$ grows.
    - Is it $\Theta(n)$? $\Theta(n^2)$? Something else?
- Obtaining a simple formula is called solving the recurrence.

Example: Getting Rich

- Suppose on day 1 of job, you are paid \$3.
- Each day thereafter, your pay is doubled.
- Let $S(n)$ be your pay on day n:

$$S(n) = \begin{cases} 2 \cdot S(n-1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

Example: Unrolling

$$S(n) = \begin{cases} 2 \cdot S(n-1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

- How much are you paid on day 4?
- $S(4) = 2 \cdot S(3)$

$= 2 \cdot 2 \cdot S(2)$

$= 2 \cdot 2 \cdot 2 \cdot S(1)$

$= 2 \cdot 2 \cdot 2 \cdot 3$

$= 24$

$$S(n) = 2^{n-1} \cdot 3$$

Solving Recurrences in general

We'll use a **four-step** process to solve recurrences:

1. "Unroll" several times to find a **pattern**.

2. Write general **formula** for kth unroll.

3. Solve for # of unrolls needed to reach base case.

4. Plug this number into general formula.

Step 1: Unroll several times

$$S(n) = \begin{cases} 2 \cdot S(n-1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

$S(n) = 2 \cdot S(n-1)$

Step 1: Unroll several times

$$S(n) = \begin{cases} 2 \cdot S(n-1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

$$\begin{aligned} S(n) &= 2 \cdot S(n-1) \\ &= 2 \cdot 2 \cdot S(n-2) \end{aligned}$$

Step 1: Unroll several times

$$S(n) = \begin{cases} 2 \cdot S(n-1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

$$\begin{aligned} S(n) &= 2 \cdot S(n-1) \\ &= 2 \cdot 2 \cdot S(n-2) \\ &= 2 \cdot 2 \cdot 2 \cdot S(n-3) \end{aligned}$$

Step 1: Unroll several times

$$S(n) = \begin{cases} 2 \cdot S(n-1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

$$\begin{aligned} S(n) &= 2 \cdot S(n-1) \\ &= 2 \cdot 2 \cdot S(n-2) \\ &= 2 \cdot 2 \cdot 2 \cdot S(n-3) \\ &= 2 \cdot 2 \cdot 2 \cdot 2 \cdot S(n-4) \end{aligned}$$

……

Step 2: Find general formula

$S(n) = 2 \cdot S(n-1)$ **#k=1**

$= 2 \cdot 2 \cdot S(n-2)$ **#k=2**

$= 2 \cdot 2 \cdot 2 \cdot S(n-3)$ **#k=3**

$= 2 \cdot 2 \cdot 2 \cdot 2 \cdot S(n-4)$

On step k:

Step 2: Find general formula

$S(n) = 2 \cdot S(n-1)$	**#k=1**
$= 2 \cdot 2 \cdot S(n-2)$	**#k=2**
$= 2 \cdot 2 \cdot 2 \cdot S(n-3)$	**#k=3**
$= 2 \cdot 2 \cdot 2 \cdot 2 \cdot S(n-4)$	**#k=4**

On step k:

$S(n) = 2^k \cdot S(n-k)$

Step 3: Find step # of base case

- On step k, $S(n) = 2^k \cdot S(n-k)$.
- When do we see $S(1)$?

Step 3: Find step # of base case

- On step k, $S(n) = 2^k \cdot S(\boldsymbol{n - k})$.
- When do we see $S(1)$?
- When **n - k** =1

Step 3: Find step # of base case

- On step k, $S(n) = 2^k \cdot S(\boldsymbol{n} - \boldsymbol{k})$.
- When do we see $S(1)$?
- When **n - k** =1 => **k = n - 1**

Step 4: Plug into general formula

- From **step 2**: $S(n) = 2^k \cdot S(n-k)$.
- From **step 3**: Base case of $S(1)$ reached when $k = n - 1$.
- So

Step 4: Plug into general formula

- From step 2: $S(n) = 2^k \cdot S(n - k)$.
- From step 3: Base case of $S(1)$ reached when $k = n - 1$.
- So $S(n) = 2^{n-1} \cdot S(n - (n-1))$

Step 4: Plug into general formula

- From step 2: $S(n) = 2^k \cdot S(n-k)$.
- From step 3: Base case of $S(1)$ reached when $k = n - 1$.
- So $S(n) = 2^{n-1} \cdot S(n-(n-1))$

$$= 2^{n-1} \cdot S(1)$$

Step 4: Plug into general formula

- From step 2: $S(n) = 2^k \cdot S(n - k)$.
- From step 3: Base case of $S(1)$ reached when $k = n - 1$.
- So $S(n) = 2^{n-1} \cdot S(n - (n-1))$

$$= 2^{n-1} \cdot S(1)$$

$$= 2^{n-1} \cdot 3$$

$$S(n) = \begin{cases} 2 \cdot S(n-1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

Solving the Recurrence

- We have solved the recurrence: $S(n) = 3 \cdot 2^{n-1}$
- This is the **exact** solution. The **asymptotic** solution is

$$S(n) = \Theta(2^n).$$

- We'll call this method **"solving by unrolling".**
- **Take the job? Yes! On day 20 you will get ~ 1.5 mill.$**

Binary Search Recurrence

Binary Search

- What is the time complexity of `binary_search`?
- **Best case**: $\Theta(1)$.
- **Worst case**:

$$T(n) = \begin{cases} T(n/2) + \Theta(1), & n \geq 2 \\ \Theta(1), & n = 1 \end{cases}$$

Simplification

- When solving, we can **replace** $\Theta(f(n))$ with $f(n)$:

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

- As long as we state **final answer** using Θ notation!

Another Simplification

- When solving, we can assume n is a power of 2.

Step 1: Unroll several times

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

$T(n) = T(n/2) + 1$

Step 1: Unroll several times

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

T(n) = **T(n/2)** + 1

Step 1: Unroll several times

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

$T(n) = \mathbf{T(n/2)} + 1$

$= \mathbf{[T(n/4) + 1]} + 1 = T(n/4) + 2$

Step 1: Unroll several times

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

$T(n) = T(n/2) + 1$

$= [T(n/4) + 1] + 1 = \mathbf{T(n/4)} + 2$

$= \mathbf{[T(n/8) + 1\ ]} + 2 = T(n/8) + 3$

Step 2: Find general formula

$T(n) = T(n/2) + 1$ **#k=1**

$= T(n/4) + 2$ **#k=2**

$= T(n/8) + 3$ **#k=3**

On step k:

Step 2: Find general formula

$T(n) = T(n/2) + 1$ **#k=1**

$= T(n/4) + 2$ **#k=2**

$= T(n/8) + 3$ **#k=3**

On step k:

$T(n) =$ **?** $+ k$

Step 2: Find general formula

$T(n) = T(n/2) + 1$ **#k=1**

$= T(n/4) + 2$ **#k=2**

$= T(n/8) + 3$ **#k=3**

On step k:

$$T(n) = T\left(\frac{n}{2^k}\right) + k$$

Step 3: Find step # of base case

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

- On step k, $T(n) = T(\frac{n}{2^k}) + k$
- When do we see $T(1)$?

Step 3: Find step # of base case

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

- On step k, $T(n) = T(\frac{n}{2^k}) + k$
- When do we see $T(1)$?

Step 3: Find step # of base case

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

- On step k, $T(n) = T(\frac{n}{2^k}) + k$
- When do we see $T(1)$?

$$\frac{n}{2^k} = 1$$

Step 3: Find step # of base case

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

- On step k, $T(n) = T(\frac{n}{2^k}) + k$
- When do we see $T(1)$?

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

Step 3: Find step # of base case

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

- On step k, $T(n) = T(\frac{n}{2^k}) + k$
- When do we see $T(1)$?

$$\frac{n}{2^k} = 1$$

$$n = 2^k \longrightarrow k = \log_2 n$$

Step 4: Plug into general formula

- $T(n) = T(\frac{n}{2^k}) + k$
- Base case of $T(1)$ reached when $\boldsymbol{k}$ **= log$_2$** $\boldsymbol{n}$**.**

Step 4: Plug into general formula

- $T(n) = T(\frac{n}{2^k}) + k$
- Base case of $T(1)$ reached when $\boldsymbol{k}$ **= log$_2$ $\boldsymbol{n}$.**

$$T(n) = T(\frac{n}{2^{\log_2 n}}) + \log_2 n$$

Step 4: Plug into general formula

- $T(n) = T(\frac{n}{2^k}) + k$
- Base case of $T(1)$ reached when $\boldsymbol{k}$ **= log$_2$** $\boldsymbol{n}$**.**

$$T(n) = T(\frac{n}{2^{\log_2 n}}) + \log_2 n$$

$$T(n) = T(1) + \log_2 n$$

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

Step 4: Plug into general formula

- $T(n) = T(\frac{n}{2^k}) + k$
- Base case of $T(1)$ reached when $\boldsymbol{k} = \mathbf{log_2}\ \boldsymbol{n}$.

$$T(n) = T(\frac{n}{2^{log_2 n}}) + log_2 n$$

$$T(n) = T(1) + log_2 n$$

$$T(n) = 1 + log_2 n$$

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

Step 4: Plug into general formula

- $T(n) = T(\frac{n}{2^k}) + k$
- Base case of $T(1)$ reached when $\boldsymbol{k}$ **=** $\mathbf{log_2}\ \boldsymbol{n}$**.**

$$T(n) = T(\frac{n}{2^{log_2 n}}) + log_2 n$$

$$T(n) = T(1) + log_2 n$$

$$T(n) = 1 + log_2 n = \Theta(log\, n)$$

Reminder

$$\log_a n = \frac{\log_c n}{\log_c a} = \frac{1}{\log_c a} \log_c n$$

Reminder

$$\log_a n = \frac{\log_c n}{\log_c a} = \frac{1}{\log_c a} \log_c n$$

Constant

Reminder

$$\log_a n = \frac{\log_c n}{\log_c a} = \frac{1}{\log_c a} \log_c n$$

Constant

- So we **don't** write $\Theta(\log_2 n)$
- Instead, just: $\Theta(\log n)$

Time Complexity of Binary Search

- Best case: $\Theta(1)$
- Worst case: $\Theta(\log n)$

Is binary search fast?

- Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- **Goal**: find a particular grain.
- *Assume* one basic operation takes 1 nanosecond.

Is binary search fast?

- Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- **Goal**: find a particular grain.
- *Assume* one basic operation takes 1 nanosecond.
- Linear search: ?

Is binary search fast?

- Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- **Goal**: find a particular grain.
- *Assume* one basic operation takes 1 nanosecond.
- Linear search: 317 years.
- Binary search: ≈ 60 nanoseconds.

Think!

Binary search seems so much faster than linear search.

What’s the caveat?

Caveat

- The array must be sorted.
- This takes $\Omega(n)$ time.

Why use binary search?

- If data is **not sorted**, sorting + binary search **takes longer** than linear search.
- But if doing **multiple** queries, looking for nearby elements, sort once and use binary search after.

Theoretical Lower Bounds

- A tight lower bound for searching a sorted list is $\Omega(\log n)$.
- This means that binary search has **optimal** worst case time complexity.

Thank you!

Do you have any questions?

CampusWire!