DIVIDE-AND-CONQUER DESIGN STRATEGY

divide: Given a problem instance $I$, construct one or more smaller problem instances $I_1, ..., I_a$
• These are called subproblems
• Usually, want subproblems to be small compared to the size of $I$ (e.g., half the size)
conquer: For $1 \leq j \leq a$, solve instance $I_j$ recursively, obtaining solutions $S_1, ..., S_a$
combine: Given solutions $S_1, ..., S_a$
• use an appropriate combining function to find the solution $S$ to the problem instance $I$
• i.e., $S = \text{Combine}(S_1, ..., S_a)$.

CORRECTNESS

Prove base cases are correct
Inductively assume subproblems are solved correctly
Show they are correctly assembled into a solution

RUNTIME/SPACE COMPLEXITY?

Techniques covered in this lecture
• Model complexities using recurrence relations
• Solve with substitution, master theorem, etc.
WORKED EXAMPLE: DESIGN OF MERGESORT

We're given a problem instance consisting of an array \( A \) of integers, which we want to sort in increasing order. The size of the problem instance is \( n \).

**Divide:** Split \( A \) into two subarrays, \( A \) consists of the first \( \lceil n / 2 \rceil \) elements in \( i \) and \( A \) consists of the last \( \lfloor n / 2 \rfloor \) elements in \( i \).

**Merge:** Run MergeSort on \( i \) and \( j \).

**Combine:** After \( i \) and \( j \) have been sorted, use a function Merge to merge \( i \) and \( j \) into a single sorted array. Recall that this can be done by traversing \( i \) and \( j \) with a single pass through \( i \) and \( j \). We simply keep track of the current element of \( i \) and \( j \), always copying the smaller element into the sorted array.

MERGE: CONQUER AND COMBINE

```
\[
\begin{align*}
1 & \ 4 & \ 5 & \ 10 & \ 12 & \ 21 & \ 31 & \ 96 & \ 98 \\
7 & \ 13 & \ 14 & \ 105 & \ 1 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 \\
& \ 105 & \ 8 & \ 13 & \ 14 & \ 11 & \ 19 & \ 10 & \ 12 & \ 21 & \ 31 \\
& \ 105 & \ 105 & \ 105 & \ 8 & \ 13 & \ 14 & \ 11 & \ 19 & \ 10 & \ 12 & \ 21 & \ 31 \\
& & & & & & & & & & & &
\end{align*}
\]
```

PSEUDOCODE FOR MERGESORT

```
1. Mergesort(A[1..n])
2. if n == 1 then return A
3. aL = cell(n/2)
4. aR = A[cell(n/2)+1..n]
5. sL = Mergesort(aL)
6. sR = Mergesort(aR)
7. return Merge(sL, sR)
```

PSEUDOCODE FOR MERGE

```
Merge(sL, sR)  WHEN BOTH:
    out[1..nL+nR] = empty array
    IL = 1; IR = 1; IOUT = 0;
    while IL < nL and IR < nR
        if AL(IL) < AR(IR)
            out[IOUT] = AL(IL);
            IL++; IOUT++
        else
            out[IOUT] = AR(IR);
            IR++; IOUT++
        while IL < nL
            out[IOUT] = AL(IL);
            IL++; IOUT++
        while IR < nR
            out[IOUT] = AR(IR);
            IR++; IOUT++
    return out
```

MERGE SIMULATION

\[
\begin{align*}
L & \ 4 & \ 10 & \ 96 & \ 98 & \ 5 & \ 12 & \ 21 & \ 31 & \ 96 & \ 98 \\
R & \ 4 & \ 10 & 10 & \ 12 & \ 21 & \ 31 & \ 96 & \ 98 & \ & \\
O & & & & & & & & & &
\end{align*}
\]

DIVIDE

\[
\begin{align*}
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
105 & \ 7 & \ 13 & \ 14 & \ 11 & \ 19 & \ 5 & \ 21 & \ 31 & \ 96 & \ 98 \\
\end{align*}
\]
ANALYSIS OF MERGESORT

```
1 Mergesort(A[1..n]) = O(1)
2 if n == 1 then return A
3 nl = ceil(n/2) = O(1)
4 sl = Mergesort(A[1..nl]) = O(1)
5 ar = A[(nl+1)..n] = O(n)
6 sr = Mergesort(sl, ar) = O(n)
7 return Merge(sl, sr) = O(n)
```

So, MergeSort(A) takes O(n) time, plus the time for its two recursive calls.

How can we analyze this recursive program structure?

RECURSION TREE METHOD

Evaluating recurrences with \( T(n) \) terms

\[
\begin{array}{c|c|c|c|c}
\text{Level} & \# \text{of modes} & \text{Runtime per mode} & \text{Total runtime for level} \\
\hline
0 & 1 & cn & cn \\
1 & 2 & c(n/2) & 2c(n/2) = cn \\
2 & 4 & c(n/4) & 4c(n/4) = cn \\
& & & \\
\vdots & \vdots & \vdots & \vdots \\
\log n & n & c(n) = cn & n(c) = cn \\
\hline
\end{array}
\]

Total = \( cn + \# \text{levels} \) = \( cn \log(n) \)

So, mergesort has runtime \( \Theta(n \log n) \)

Can also compute using a table...

MATHEMATICALLY EXPRESSING THE COMPLEXITY OF MERGESORT

Let \( T(n) \) denote the time to run Mergesort on an array of length \( n \).

\( \text{Split} \) takes time \( T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor) \)

\( \text{combine} \) takes time \( O(n) \).

Recurrence relation:

\[
T(n) = \begin{cases} 
\frac{n}{2} & \text{if } n > 1 \\
0 & \text{if } n = 1 
\end{cases}
\]

To make this easier, assume \( n = 2^k \), which lets us ignore floors/ceilings

RECURSION RELATIONS

A crucial analysis tool for recursive algorithms

Suppose \( a_1, a_2, \ldots \) is an infinite sequence of real numbers.

A recurrence relation is a formula that expresses a general term \( a_n \) in terms of one or more previous terms \( a_{n-1}, a_{n-2}, \ldots \).

A recurrence relation will also specify one or more initial values starting at \( a_0 \).

Solving a recurrence relation means finding a formula for \( a_n \) that does not involve any previous terms \( a_1, a_2, \ldots, a_{n-1} \).

There are many methods of solving recurrence relations. Two important methods are plugging and chugging and the recursion tree method.
RECURSION TREE METHOD FORMALIZED

Sample recurrence for two recursive calls on problem size $n/2$

where $c$ and $d$ are constants.

We can solve this recurrence relation when $n$ is a power of two by constructing a 2D table as follows:

| Step 1 | Start with a one-node tree. Say $V_1$ having the value $T(1)$. 
| Step 2 | There are children $V_2$ and $V_3$, having the value $T(2)$, and the value of $V$ is replaced by $c + d$. 
| Step 3 | Repeat this process recursively. Terminating while a node $V$ has two children. 
| Step 4 | Look for the values on each head of the tree, and then compute the sum of all those over the root $V$.

GUESS-AND-CHECK METHOD

- Suppose we have the following recurrence
  
  $T(0) = 4$; 
  $T(n) = T(n-1) + 6n - 5$

- **Guess** the form of the solution **any** way you like

- **My approach:** the substitution method

  - Recursively substitute the formula into itself
  - Try to identify patterns to **guess** the final closed form

- **Prove** that the guess was correct

SUBSTITUTION METHOD: WORKED EXAMPLE

Recurrence: $T(0) = 4; T(n) = T(n-1) + 6n - 5$

- $T(n) = (T(n-2) + 6(n-1) - 5) + 6n - 5$  
  (substitute)$T(n) = T(n-2) + 6n - 6 - 5 + 6n - 5$

Compare: new terms? $+ (6n - 5) - 6$

- $T(n) = (T(n-3) + 6(n-2) - 5) + 2(6n - 5) - 6$  
  (substitute) $T(n) = T(n-3) + 6n - 2(6) - 5 + 2(6n - 5) - 6$

New terms? $+ (6n - 5) - 2(6)$

- $T(n) = T(n-3) + 3(6n - 5) - 6(1+2)$  
  \(\ldots\) identify patterns and **guess** what happens in the limit

- $T(n) = T(0) + m(6n - 5) - 6(1 + 2 + 3 + \ldots + (n-1)) = guess(n)$

\(\text{Recall: } T(0) = 4 ; T(n) = T(n-1) + 6n - 5 ; \text{guess}(n) = 3n^2 - 2n + 4 \)

Want to prove: $\text{guess}(n) = T(n)$ for all $n$

Base case: $\text{guess}(0) = 3(0^2) - 2(0) + 4 = T(0)$

Inductive case: suppose $\text{guess}(n) = T(n)$ for $n \geq 0$, show $\text{guess}(n+1) = T(n+1)$.

$T(n+1) = T(n) + 6(n+1) - 5$  
(by definition)$\text{guess}(n+1) = T(n+1) - 5$

$= 3(n+1)^2 - 2(n+1 + 4)$  
(by definition)$= 3n^2 + 4n + 5$  
(simplify)$\text{guess}(n+1) = T(n+1)$

ANOTHER APPROACH

- Suppose you look for a while at the previous recurrence:

  $T(0) = 4; T(n) = T(n-1) + 6n - 5$

- With some experience, you might just **guess** it's quadratic

- If you're right, it should have the form:

  $an^2 + bn + c$ for some **unknown** constants $a$, $b$, $c$

- So, just carry the unknown constants **into the proof**:

  - You can then determine what the constants **must be** for the proof to work out
Consider recurrence: \( T(1) = d; \ T(n) = aT\left(\frac{n}{2}\right) + \Theta(n^2) \) where \( a \geq 1, b \geq 2 \) and \( n = b^i \) for integer \( i \).

**Example corresponding algorithm**

1. If BaseCase then return Result
2. n subproblems = 1
3. for \( j = \log_b n \) do
   a. Let \( a = \text{subproblem size of size n/b} \)
   b. \( \text{substitution}[j] = \text{Base algo}(a) \)
4. Solution = combine in \( n \log_b n \) time
5. return solution

**REARRANGING**

- \( T(n) = da^j + \sum_{i=0}^{j-1} ca^i \) where \( c \neq 0 \)
- \( T(n) = da^j + \sum_{i=0}^{j-1} ca^i \)
- Rearranging we have \( b^j = a \)
- \( T(n) = da^j + \sum_{i=0}^{j-1} ca^i \)
- \( T(n) = da^j + \sum_{i=0}^{j-1} ca^i \)
- \( T(n) = da^j + \sum_{i=0}^{j-1} ca^i \)
- \( T(n) = da^j + \sum_{i=0}^{j-1} ca^i \)
- \( T(n) = da^j + \sum_{i=0}^{j-1} ca^i \)

**SOLVING THE GEOMETRIC SEQ**

- \( T(n) = dn^r + cn^r \sum_{i=0}^{r-1} a^i \) where \( r = b^i \)
- Recall formula: \( T(n) = \sum_{i=0}^{r-1} a^i \) if \( r > 1 \)
- \( a^i = 0 \) if \( r = 1 \)
- \( a^i = 0 \) if \( 0 < r < 1 \)

So different solutions depending on \( r \):

- Case 1: \( r = b^{i-1} > 1 \) \( \Rightarrow x > y > 0 \) \( \Rightarrow x > y \)
- Case 2: \( r = b^{i-1} = 1 \) \( \Rightarrow x = y \)
- Case 3: \( 0 < r = b^{i-1} < 1 \) \( \Rightarrow x > y < 0 \) \( \Rightarrow x > y \)
SOLVING THE GEOMETRIC SEQ
- Formula: \( \sum_{i=0}^{n-1} ar^i = \begin{cases} \frac{a r^n - r}{r-1} & \text{if } r > 1 \\ na + \Theta(n) & \text{if } r = 1 \\ \frac{a}{1-r} \Theta(1) & \text{if } 0 < r < 1 \end{cases} \)

Case 1: \( r = b^{-y} > 1 \quad \Rightarrow \quad x - y > 0 \quad \Rightarrow \quad x > y \)
- \( T(n) = dn^y + cn^y \sum_{i=0}^{n-1} r^i \in \Theta(n^y + n^y(r)) = \Theta(n^y + n^y(b^{-y})) \)
- \( T(n) \in \Theta(n^y + n^y(b^{-y})) \)
- Recall \( b^y = n \), so \( T(n) \in \Theta(n^y(n^y)) = \Theta(n^{2y}) \)
- So \( T(n) \in \Theta(n^y) \)

Case 2: \( 0 < r = b^{-y} < 1 \quad \Rightarrow \quad x - y < 0 \quad \Rightarrow \quad x < y \)
- \( T(n) = dn^y + cn^y \sum_{i=0}^{n-1} r^i \in \Theta(n^y + n^y(r)) \)
- \( T(n) \in \Theta(n^y + n^y(r)) \)
- Since \( x < y \), we simply have \( T(n) \in \Theta(n^y) \)

SOLVING THE GEOMETRIC SEQ
- Formula: \( \sum_{i=0}^{n-1} ar^i = \begin{cases} \frac{a r^n - r}{r-1} & \text{if } r > 1 \\ na + \Theta(n) & \text{if } r = 1 \\ \frac{a}{1-r} \Theta(1) & \text{if } 0 < r < 1 \end{cases} \)

Case 1: \( r = b^{-y} > 1 \quad \Rightarrow \quad x - y > 0 \quad \Rightarrow \quad x > y \)
- \( T(n) = dn^y + cn^y \sum_{i=0}^{n-1} r^i \in \Theta(n^y + n^y(r)) = \Theta(n^y + n^y(b^{-y})) \)
- \( T(n) \in \Theta(n^y + n^y(b^{-y})) \)
- Recall \( x = y \), so \( T(n) \in \Theta(n^y) \)
- So \( T(n) \in \Theta(n^y) \)

SOLVING THE GEOMETRIC SEQ
- Formula: \( \sum_{i=0}^{n-1} ar^i = \begin{cases} \frac{a r^n - r}{r-1} & \text{if } r > 1 \\ na + \Theta(n) & \text{if } r = 1 \\ \frac{a}{1-r} \Theta(1) & \text{if } 0 < r < 1 \end{cases} \)

Case 2: \( 0 < r = b^{-y} < 1 \quad \Rightarrow \quad x - y < 0 \quad \Rightarrow \quad x < y \)
- \( T(n) = dn^y + cn^y \sum_{i=0}^{n-1} r^i \in \Theta(n^y + n^y(r)) \)
- \( T(n) \in \Theta(n^y + n^y(r)) \)
- Since \( x < y \), we simply have \( T(n) \in \Theta(n^y) \)

SOME BONUS INTUITION FOR R CASES

Recall: simplified master theorem
- Suppose that \( n \geq 1 \) and \( b > 1 \). Consider the recurrence
- \( T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d) \), where \( d \) is a power of \( b \).
- Define \( x = \log_b a \). Then
- \( T(n) \in \Theta(n^d) \)
- \( T(n) \in \Theta(n^d \log n) \)
- \( T(n) \in \Theta(n^d \log^2 n) \)

Questions:
- For \( a = 2 \), \( b = 2 \), \( y = 1 \), \( x = 2 \), which \( \Theta \) function? \( \Theta(n) \)

WORKED EXAMPLES

Recall: simplified master theorem
- Suppose that \( n \geq 1 \) and \( b > 1 \). Consider the recurrence
- \( T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d) \), where \( d \) is a power of \( b \).
- Define \( x = \log_b a \). Then
- \( T(n) \in \Theta(n^d) \)
- \( T(n) \in \Theta(n^d \log n) \)
- \( T(n) \in \Theta(n^d \log^2 n) \)

Suppose that \( 0 < b^{-y} < 1 \), \( n > 1 \), and \( y > 1 \).
- Then \( x = \log_b a \) and \( T(n) \in \Theta(n^d \log n) \).
**General Master Theorem**

Suppose that \( a \geq 1 \) and \( b > 1 \). Consider the recurrence

\[
T(n) = aT\left(\frac{n}{b}\right) + f(n),
\]

where \( n \) is a power of \( b \). Denote \( x = \log_b n \). Then

\[
T(n) \in \begin{cases} 
\Theta(n^x) & \text{if } f(n) \in O(n^{x+\epsilon}) \text{ for some } \epsilon > 0 \\
\Theta(n^x \log n) & \text{if } f(n) \in \Theta(n^x) \\
\Theta(f(n)) & \text{if } f(n)/n^{x+\epsilon} \text{ is an increasing function of } n \text{ for some } \epsilon > 0.
\end{cases}
\]

*Example recurrence:* \( T(n) = 2T\left(\frac{n}{4}\right) + n \log n \)

- Must reason about relationship between \( f(n) \) and \( n^x \)

**Revisiting the Recursion Tree Method**

- Some recurrences with complex \( f(n) \) functions (such as \( f(n) = \log n \)) can still be solved “by hand”

*Example:* Let \( n = 2^j \); \( T(1) = 1 \); \( T(n) = 2T\left(\frac{n}{2}\right) + n \log n \)

- Note \( \log_2 n = j \)
- So \( j \leq n \leq 2^j \)
- And \( j - \frac{1}{2} \leq \log_2 n \leq j \)

**Master Theorem When \( b^{j-1} < n < b^j \)**

- \( n/b \) is not always an integer!
- Floors/ceilings are hard
- Not a geometric sequence

Suppose we get a \( \Theta \) bound for \( b^{j-1} < n < b^j \) by instead considering the larger problem size \( b^j \)

- \( T(n) \leq T\left(b^j\right) \in \begin{cases} 
\Theta\left((b^j)^x\right) & \text{if } y < x \\
\Theta\left((b^j)^x \log b^j\right) & \text{if } y = x \\
\Theta\left((b^j)^y\right) & \text{if } y > x
\end{cases} \)

- So \( T(n) \in \Theta\left((b^j)^x \log b^j\right) \) if \( y = x \)

- Observation: \( b^j < bn \) since \( n \) is between \( b^{j-1} \) and \( b^j \)

- So \( T(n) \leq T\left(b^j\right) \in \begin{cases} 
\Theta\left((bn)^x\right) & \text{if } y < x \\
\Theta\left((bn)^x \log bn\right) & \text{if } y = x \\
\Theta\left((bn)^y\right) & \text{if } y > x
\end{cases} \)

**Master Theorem When \( b^{j-1} < n < b^j \)**

- \( T(n) \leq T\left(b^j\right) \in \begin{cases} 
\Theta\left((bn)^x\right) & \text{if } y < x \\
\Theta\left((bn)^x \log bn\right) & \text{if } y = x \\
\Theta\left((bn)^y\right) & \text{if } y > x
\end{cases} \)

- Case 1 \( (y < x): \) \( (bn)^x = b^n n^x \) and \( b^n \) is a constant
- So \( T(n) \in O(n^x) \)

- Case 2 \( (y = x): \) \( (bn)^x \log bn = b^n n^x \log b + \log n \)
- So \( T(n) \in \Theta(b^n n^x \log b + b^n n^x \log n) = \Theta(n^x + n^x \log n) \)

- Case 3 \( (y > x): \) \( (bn)^y = b^n n^y \)
- So \( T(n) \in O(n^y) \)

Can tackle \( \Omega \) similarly to get \( \Theta \)