DIVIDE-AND-CONQUER DESIGN STRATEGY

divide: Given a problem instance $I$, construct one or more smaller problem instances $I_1, \ldots, I_n$
- These are called subproblems
- Usually, want subproblems to be small compared to the size of $I$ (e.g., half the size)

conquer: For $1 \leq j \leq n$, solve instance $I_j$ recursively, obtaining solutions $S_1, \ldots, S_n$

combine: Given solutions $S_1, \ldots, S_n$, use an appropriate combining function to find the solution $S$ to the problem instance $I$
- i.e., $S = \text{Combine}(S_1, \ldots, S_n)$.

D&C PROTO-ALGORITHM

```
D&C_template(I)
1 if BaseCase(I) return Result(I)
2 subproblems = [I_1, I_2, \ldots, I_n]
3 subsolutions = []
4 for j = 1 to n
5     subsolutions[j] = D&C_template(I_j)
6 return Combine(subsolutions)
```

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

Here, a problem instance consists of an array $A$ of 15 integers, which we want to sort in increasing order. The size of the problem instance is $n$.

**Divide**: Split $A$ into two subarrays, $A_L$ and $A_R$, consisting of the first $\frac{n}{2}$ and last $\frac{n}{2}$ elements in $A$.

**Pseudocode for Merge**

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

**Merge Simulation**

There are still elements left in both arrays.

**Merge: Conquer and Combine**

- For $i_L < n_L$ and $i_R < n_R$:
  - $aOut[iOut] = a[i_L]$
  - $i_L++$
  - $iOut++$

- While $i_L < n_L$: $aOut[iOut] = a[i_L]$
- $i_L++$

- While $i_R < n_R$: $aOut[iOut] = a[i_R]$
- $i_R++$

- Return $aOut$

**Pseudocode for Merge**

```plaintext
Merger(aL[1..nL], aR[1..nR])
1 out[1..nL+nR] = empty array
2 iL = 1; iR = 1; iOut = 1
3 while iL < nL and iR < nR:
4 if aL[iL] < aR[iR]:
5 out[iOut] = aL[iL];
6 iL++;
7 iOut++;
8 else:
9 out[iOut] = aR[iR];
10 iR++;
11 iOut++;
12 while iL < nL:
13 out[iOut] = aL[iL];
14 iL++;
15 iOut++;
16 while iR < nR:
17 out[iOut] = aR[iR];
18 iR++;
19 iOut++;
20 return out
```
ANALYSIS OF MERGESORT

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

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

How can we analyze this recursive program structure?

RECURSION RELATIONS

A crucial analysis tool for recursive algorithms

```
Hulk(n) = Face - Chin + Hulk(n-1)
```

MATHEMATICALLY EXPRESSING THE COMPLEXITY OF MERGESORT

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

- Hulk takes time H(1)
- Compare takes time T((n/2)) - T((n/2))
- Combine takes time O(n)

Recurrence relation:

```
T(n) = \left\{ 
\begin{array}{ll}
T(\lfloor n/2 \rfloor) & \text{if } n > 1 \\
0 & \text{if } n = 1
\end{array}
\right.
```

How can we compute/solve for T(n)?

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

RECURSION TREE METHOD

Evaluating recurrences with \(T(n/2)\) terms

<table>
<thead>
<tr>
<th>Level</th>
<th># of modes</th>
<th>runtime per mode</th>
<th>total runtime for level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>cn</td>
<td>cn</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>cn/2</td>
<td>2(cn/2) = cn</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>cn/4</td>
<td>4(cn/4) = 2cn</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>log n</td>
<td>n</td>
<td>cn</td>
<td>cn</td>
</tr>
</tbody>
</table>

Can also compute using a table...

So, mergesort has runtime \(O(n \log n)\)
RECURSION TREE METHOD FORMALIZED

Sample recurrence for two recursive calls on problem size \( n/2 \):

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

where \( a \) and \( b \) are constants.

We can solve this recurrence relation when \( n \) is a power of two by constructing a recursion tree as follows:

**Step 1:** Start with a one-node tree. Say \( V \), having the value \( T(n) \).

**Step 2:** Draw two children. Say \( V_1 \) and \( V_2 \), having the value \( T(n/2) \) and the value \( T(n/2) \), and the value of \( T \) is replaced by \( n \).

**Step 3:** Repeat this process recursively, terminating when a node is reached that is over the value \( T(1) = 0 \).

**Step 4:** Sum the values on each level of the tree, and then compute the sum of the leaves over the entire tree.

GUESS-AND-CHECK METHOD

- Suppose we have the following recurrence
  \[
  T(0) = 4; \quad T(n) = T(n-1) + 6n - 5
  \]
- **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: \( T(0) = 4; \ T(n) = T(n-1) + 6n - 5 \)

- \( T(n-1) = T(n-2) + 6(n-1) - 5 \)
- \( T(n) = T(n-2) + 6n - 5 \) \( \Rightarrow \) new terms?
- \( = T(n-2) + 2(6n - 5) - 6 \) \( \Rightarrow \) try to preserve structure
- \( = T(n-3) + 6(n-2) - 5 + 2(6n - 5) - 6 \) \( \Rightarrow \) new terms?
- \( = T(n-3) + 3(6n - 5) - 6(1+2) \) \( \Rightarrow \) new terms?
- \( ... \) identify patterns and **guess** what happens in the limit

\[
guess(n) = T(0) + n(6n - 5) - 6(1 + 2 + \ldots + (n - 1)) = 3n^2 - 2n + 4
\]

**PROOF**

- **Recall:** \( T(0) = 4; \ T(n) = T(n-1) + 6n - 5; \ guess(n) = 3n^2 - 2n + 4 \)
- Want to prove: \( guess(n) = T(n) \) for all \( n \)
  - Base case: \( guess(0) = 3(0)^2 - 2(0) + 4 = T(0) \)
  - **PROOF**

**In Math, I use the GUESS & CHECK Method**

- **Suppose** we have the following recurrence
  \[
  T(0) = 4; \quad T(n) = T(n-1) + 6n - 5
  \]
- **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

- **Guess** \( T(n) = T(0) + n(6n - 5) - 6(1 + 2 + \ldots + (n - 1)) \)
- Use \( 1 + 2 + \ldots + (n - 1) = \frac{n(n-1)}{2} \)
- \( guess(n) = 4 + 6n^2 - 5n - 6n(n-1)/2 \) \( \Rightarrow \) simplify
  - \( = 3n^2 - 2n + 4 \)
- Are we done?
  - The form of \( guess(n) \) was an **educated guess**.
  - To be sure, we must **prove** it correct using induction

**PROOF**

- **Recall:** \( T(0) = 4; \ T(n) = T(n-1) + 6n - 5; \ guess(n) = 3n^2 - 2n + 4 \)
- Want to prove: \( guess(n) = T(n) \) for all \( n \)
  - **Inductive case:** suppose \( guess(n) = T(n) \) for \( n \geq 0 \)
  - Show \( guess(n+1) = T(n+1) \)
  - \( T(n+1) = T(n) + 6(n+1) - 5 \) \( \Rightarrow \) by definition
    - \( = guess(n) + 6(n+1) - 5 \) \( \Rightarrow \) by inductive hypothesis
    - \( = 3n^2 - 2n + 4 + 6(n+1) - 5 \) \( \Rightarrow \) substitute
    - \( = 3n^2 + 4n + 5 \) \( \Rightarrow \) simplify
    - \( guess(n+1) = 3(n+1)^2 - 2(n+1) + 4 \) \( \Rightarrow \) by definition
    - \( = 3n^2 + 4n + 5 = T(n+1) \) \( \Rightarrow \) simplify
Suppose you look for a while at the previous recurrence:
\[ T(n) = 4T(n-1) + 6n - 5 \]
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.

Recall: \( \text{guess}(n) = an^2 + bn + c \) where \( c = 4 \)
Inductive case: suppose \( \text{guess}(n) = T(n) \) for \( n \geq 0 \).
Show \( \text{guess}(n+1) = T(n+1) \).
\[ T(n+1) = an^2 + (b+6)n + 5 \]
\[ \text{guess}(n+1) = a(n+1)^2 + (b+6)(n+1) + 5 \]
\[ = a(n^2 + 2n + 1) + b(n+1) + 5 \]
\[ = an^2 + 2an + a + bn + b + 5 \]
\[ = an^2 + bn + c \] (rearrange polynomial)
We want this to be equal to \( T(n+1) \).
\[ an^2 + (2a+b)n + (a+b+4) = an^2 + bn + c \]
equivalent to \( (2a+b) = (b+6) \) and \( (a+b+4) = 5 \)
first implies \( a = 3 \) plug a into second to get \( b = 5 - 4 - 3 = -2 \)

\[ T(0) = 4; T(n) = T(n-1) + 6n - 5 ; \text{guess}(n) = an^2 + bn + c \]
Want to prove: \( \text{guess}(n) = T(n) \) for all \( n \)
Base case: \( \text{guess}(0) = a(0)^2 + b(0) + c = T(0) = 4 \)
This holds iff \( c = 4 \)
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)
\[ = \text{guess}(n) + 6(n+1) - 5 \] (by inductive hypothesis)
\[ = an^2 + bn + 4 + 6(n+1) - 5 \] (substitute)
\[ = an^2 + (b+6)n + 5 \] (simplify)

Another Approach

Provides a formula for solving many recurrence relations.
We start with a simplified version.
Consider recurrence: \( T(k) = d ; T(n) = aT(\frac{n}{b}) + \Theta(n^c) \)
where \( a \geq 1, b \geq 2 \) and \( n \) is a power of \( b \) (i.e., \( n = b^x \) for integer \( x \))

**Master Theorem for Recurrences**

- Provides a formula for solving many recurrence relations.
- We start with a simplified version.

Consider recurrence: \( T(k) = d ; T(n) = aT(\frac{n}{b}) + \Theta(n^c) \)
where \( a \geq 1, b \geq 2 \) and \( n \) is a power of \( b \) (i.e., \( n = b^x \) for integer \( x \)).

**Formula corresponding algorithm**

- **Base Case**: return \( T(1) \)
- **Induction**: \( T(n) = aT(\frac{n}{b}) + \Theta(n^c) \)
- \( T(n) = \Theta(n^c) \) if \( c < \log_b a \)
- \( T(n) = \Theta(n^c \log n) \) if \( c = \log_b a \)
- \( T(n) = \Theta(n^c) \) if \( c > \log_b a \)

**Simplified Master Theorem**

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } c < \log_b a \\
\Theta(n^c \log n) & \text{if } c = \log_b a \\
\Theta(n^c) & \text{if } c > \log_b a 
\end{cases} \]

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } c < \log_b a \\
\Theta(n^c \log n) & \text{if } c = \log_b a \\
\Theta(n^c) & \text{if } c > \log_b a 
\end{cases} \]

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } c < \log_b a \\
\Theta(n^c \log n) & \text{if } c = \log_b a \\
\Theta(n^c) & \text{if } c > \log_b a 
\end{cases} \]

**Rearranging**

- \( T(n) = d + \sum_{i=0}^{\log_b n} \Theta(n^i) \)
- Let \( x = \log_b a \)
- \( x \) relates \# of subproblems to their size.
- Rearranging we have \( b^x = a \)
- So \( T(n) = d + \sum_{i=0}^{x} \Theta(n^i) \)
- \( = d + n^0 \sum_{i=0}^{x} \Theta(n^i) \)
- Also \( T(n) = d(b^x)^i = d(b^x)^x \)
- Since \( n = b^x \) this is just \( dx^x \)
- So \( T(n) = dx^x + \sum_{i=0}^{x} \Theta(n^i) \)
- And we can simplify: let \( r = b^{x-1} \)
SOLVING THE GEOMETRIC SEQ

\[ T(n) = dn^r + cn^r \sum_{i=0}^{r-1} r^i \] where \( r = b^r \)

- Geo. Seq. formula: \( \sum_{i=0}^{r-1} ar^i = \begin{cases} \frac{a(r^{r+1} - 1)}{r-1} & \text{if } r > 1 \\ ar^0 & \text{if } r = 1 \\ \frac{a}{1-r} & \text{if } 0 < r < 1 \end{cases} \)
- So different solutions depending on \( r \)
  - Case 1: \( r = b^r > 1 \) \( \Rightarrow x - y > 0 \) \( \Rightarrow x > y \)
  - Case 2: \( r = b^r = 1 \) \( \Rightarrow x - y = 0 \) \( \Rightarrow x = y \)
  - Case 3: \( 0 < r = b^r < 1 \) \( \Rightarrow x - y < 0 \) \( \Rightarrow x < y \)

\[
\begin{align*}
T(n) &= dn^r + cn^r \sum_{i=0}^{r-1} r^i \\
&= \frac{dn^r}{r-1} + \frac{cn^r}{1-r}
\end{align*}
\]

SOLVING THE GEOMETRIC SEQ

- Formula: \( \sum_{i=0}^{r-1} ar^i = \begin{cases} \frac{a(r^{r+1} - 1)}{r-1} & \text{if } r > 1 \\ ar^0 & \text{if } r = 1 \\ \frac{a}{1-r} & \text{if } 0 < r < 1 \end{cases} \)
- Case 1: \( r = b^r > 1 \) \( \Rightarrow x - y > 0 \) \( \Rightarrow x > y \)
- \( T(n) = dn^r + cn^r \sum_{i=0}^{r-1} r^i \) \( \in dn^r + cn^r \Theta(r^d) \)
- Recall \( b^r = n \), so \( T(n) = \Theta(n^r + n^r r^d) \) \( = \Theta(n^r + n^{r+1}) \)

\[ T(n) = \Theta(n^r) \]

SOLVING THE GEOMETRIC SEQ

- Formula: \( \sum_{i=0}^{r-1} ar^i = \begin{cases} \frac{a(r^{r+1} - 1)}{r-1} & \text{if } r > 1 \\ ar^0 & \text{if } r = 1 \\ \frac{a}{1-r} & \text{if } 0 < r < 1 \end{cases} \)
- Case 1: \( r = b^r > 1 \) \( \Rightarrow x - y > 0 \) \( \Rightarrow x > y \)
- \( T(n) = dn^r + cn^r \sum_{i=0}^{r-1} r^i \) \( \in dn^r + cn^r \Theta(r^d) \)
- Recall \( b^r = n \), so \( T(n) = \Theta(n^r + n^r r^d) \) \( = \Theta(n^r + n^{r+1}) \)

\[ T(n) = \Theta(n^r) \]

M ASTER THEOREM FOR RECURRENCES

Simplified version

Consider recurrence:
\[ T(n) = aT(\frac{n}{b}) + \Theta(n^d) \] where \( a \geq 1, b \geq 2 \) and \( n = b^i \)
And let \( x = \log_b n \)

| \( T(n) \) | \( \Theta(n^d) \) if \( y < x \) | \( \Theta(n^{d+1} \log n) \) if \( y = x \) | \( \Theta(n^y) \) if \( y > x \) |

SOME BONUS INTUITION FOR R CASES

Recall: \( T(n) = dn^r + cn^r \sum_{i=0}^{r-1} i^j \)
\( x = \log_b n \) \( i.e. \) \( \log \text{leaf problem size} \) \( \log \) \( \text{subproblems} \)

\[
\begin{array}{|c|c|c|}
\hline
\text{Case} & r & g \cap f \\
\hline
\text{Heavy hails} & r > 1 & g < r \\
\text{balanced} & r = 1 & g = r \\
\text{heavy top} & r < 1 & g > r \\
\hline
\end{array}
\]

- Heavy hails means that the value of the recursion tree is dominated by the values of the leaf nodes.
- Balanced means that the values of the levels of the recursion tree are constant (except for the last level).
- Heavy top means that the value of the recursion tree is dominated by the value of the root node.
WORKED EXAMPLES
Recall: simplified master theorem
Suppose that $a \geq 1$ and $b > 1$. Consider the recurrence $T(n) = aT\left(\frac{n}{b}\right) + O(n^y)$, where $n$ is a power of $b$. Denote $x = \log_b n$. Then

$T(n) \in \begin{cases} 
\Theta(n^x) & \text{if } y < x \\
\Theta(n^x \log n) & \text{if } y = x \\
\Theta(n^x) & \text{if } y > x 
\end{cases}$

Questions: $a=?$ $b=?$ $y=?$ $x=?$

which $\Theta$ function?

$\Theta(n^x \log n) = \Theta(n^x)$

$a=3$; $b=2$; $y=1$; $x=\log_2 3$

$\Theta(n^x) = \Theta(n^x \log n)$

$a=4$; $b=2$; $y=1$; $x=\log_2 4$

$\Theta(n^x) = \Theta(n^x \log n)$

$a=2$; $b=2$; $y=3/2$; $x=1$

$\Theta(n^{3/2}) = \Theta(n^{3/2} \log n)$

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 big-O bound for $b^{j-1} < n < b^j$ by instead considering the larger problem size $b^j$

- $T(n) \leq T\left(b^j\right) \in \Theta(b^{j} \log b)$ if $y = x$
- $T\left(b^j\right) \leq T\left(b^{j+1}\right) \in \Theta\left(b^{j+1}\right)$ if $y > x$

 Bonus slide, for you at home

CASE 1 ($y < x$):

- $T(n) \in \Theta(b^y \log b)$ if $y < x$
- $T(n) \in \Theta(b^y)$ if $y = x$
- $T(n) \in \Theta(b^y \log b)$ if $y > x$

CASE 2 ($y = x$):

- $T(n) \in \Theta(b^n \log b)$ if $y = x$
- $T(n) \in \Theta(b^n)$ if $y > x$

CASE 3 ($y > x$):

- $T(n) \in \Theta(b^n \log b)$ if $y > x$
- $T(n) \in \Theta(b^n)$ if $y > x$

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

$T(n) \in \begin{cases} 
\Theta(n^x) & \text{if } f(n) = O(n^{x-\epsilon}) \text{ for some } \epsilon > 0 \\
\Theta(n^x \log n) & \text{if } f(n) = O(n^{x-1}) \\
\Theta(f(n)) & \text{if } f(n) = O(n^{x+\epsilon}) \text{ for some } \epsilon > 0
\end{cases}$

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

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

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 $n = 2^j$; $T(1) = 1$; $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$

Level | Nodes | Value at Each Node | Value of the Value of the
------|-------|-------------------|-------------------
0     | 1     | $0$               | $0$               
1     | 3     | $1$               | $2$               
2     | 7     | $2 + 2 + 2^2$     | $2 + 2 + 2^2$     
3     | 15    | $4 + 4 + 4 + 4$   | $4 + 4 + 4 + 4$   
4     | 31    | $8 + 8 + 8 + 8 + 8$ | $8 + 8 + 8 + 8 + 8$ 
5     | 63    | $16 + 16 + 16 + 16 + 16 + 16 + 16$ | $16 + 16 + 16 + 16 + 16 + 16 + 16$ 

Note

$\log n = j$; $j^2 = n \log n$ and

$U = 2^j - \frac{3}{2}$

Bonus slide, for you at home
REVISITING THE RECURSION TREE METHOD

Recall: \( n = 2^j \), \( T(1) = 1 \), \( T(n) = 2T \left( \frac{n}{2} \right) + n \log n \)

Summing the values at all levels of the recursion tree, we have

\[
T(n) = 2 \left( \sum_{j=0}^{j=\log n} j \right) = 2 \left( \log n \cdot \frac{\log n - 1}{2} \right)
\]

Since \( n = 2^j \), we have \( j = \log n \) and \( T(n) = O(n \log n) \).