CS 341: ALGORITHMS
Lecture 4: divide & conquer I
Readings: see website
Trevor Brown (co-taught with Anna Lubiw)
https://www.student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

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., \( S = \text{Combine}(S_1, ..., S_a) \).

D&C PROTO-ALGORITHM

```c
D&C_template(I) {
    if BaseCase(I) return Result(I)
    subproblems = \{I_1, I_2, ..., I_a\}
    subsolutions = \{\}
    for j = 1 to a
        subsolutions[j] = D&C_template(I_j)
    return Combine(subsolutions)
}
```

CORRECTNESS

1. **D&C_template(I)** returns the solution to \( I \).
2. **Prove base cases are correct**
3. **Inductively assume subproblems are solved correctly**
4. **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 \( n \) integers, which we want to sort in increasing order. The size of the problem instance is \( n \).

**divide**: Split \( A \) into two subarrays: \( A_L \) consists of the first \( \frac{n}{2} \) elements in \( A \) and \( A_R \) consists of the last \( \frac{n}{2} \) elements in \( A \).

**conquer**: Run MERGESORT on \( A_L \) and \( A_R \).

**combine**: After \( A_L \) and \( A_R \) have been sorted, use a function Merge to merge \( A_L \) and \( A_R \) into a single sorted array. Recall that this can be done in time \( \Theta(n) \) with a single pass through \( A_L \) and \( A_R \). We simply keep track of the “current” element of \( A_L \) and \( A_R \), always copying the smaller one into the sorted array.

**DIVIDE**

**MERGE: CONQUER AND COMBINE**

**MERGE SIMULATION**

**PSEUDOCODE FOR MERGESORT**

```
1 Mergesort(A[1..n])
2   if n == 1 then return A
3   mL = A[1..(n/2)]
4   mR = A[(n/2)+1..n]
5   mL = Mergesort(mL)
6   mR = Mergesort(mR)
7   return Merge(mL, mR)
```

**PSEUDOCODE FOR MERGE**

```
Merge(A[1..l], B[1..r], out[1..l+r])
   out[1..l+r] = empty array
   L = 1, R = 1
   while L <= l and R <= r
      if mL < mR \( \Rightarrow \) out[L] = mL, mL += 1
         else out[L] = mR, mR += 1
      if mL < l \( \Rightarrow \) out[L] = mL, mL += 1
         else out[L] = mR, mR += 1
      while L < l \( \Rightarrow \) out[L] = mL, mL += 1
         else out[L] = mR, mR += 1
   return out
```
So, MergeSort(A) takes $O(n)$ time plus the time for its two recursive calls.

How can we analyze this recursive program structure?

**Recurrence Relations**

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 $a_1, \ldots, a_{n-1}$.

A recurrence relation will also specify one or more initial values starting at $a_1$.

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

There are many methods of solving recurrence relations. Two important methods are **guess and check** and the **recursion tree method**.

**Mathematically Expressing the Complexity of MergeSort**

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

- *divide* takes time $\Theta(n)$.
- *conquer* takes time $T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T\left(\left\lceil \frac{n}{2} \right\rceil \right)$.
- *combine* takes time $O(n)$.

Recurrence relation:

$$T(n) = \begin{cases} T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T\left(\left\lceil \frac{n}{2} \right\rceil \right) + O(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

$T(n)$ is a function of $T(\_\_\_)$ so $T$ is a recurrence relation.

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 $T(n/v)$ terms.

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

Total = $cn + \text{# levels}$

Total = $n \log(n)$

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

Can also compute using a table.
RECURSION TREE METHOD FORMALIZED

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

\[ T(n) = \frac{1}{2} T(\frac{n}{2}) + c, \quad \text{if } n > 1 \text{ is a power of 2} \]

\[ T(n) = d, \quad \text{if } n = 1. \]

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 root node, say $N$, having the value $T(n)$.

**Step 2.** Draw two children of $N$. These children, say $N_1$ and $N_2$, have the value $T(n/2)$, and the value of $N$ is replaced by $c$.

**Step 3.** Repeat this process recursively, terminating when a node reaches the value $T(1) = d$.

**Step 4.** Sum the values on each level of the tree, and then compute the sum of all these sums; the result is $T(n)$.

GUESS-AND-CHECK METHOD

In Math, I use the 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: \[ T(0) = 4; \quad T(n) = T(n-1) + 6n - 5 \]

• \[ T(n) = T(n-2) + 6(n-1) - 5 + 6n - 5 \]
  \[ \text{ (substitute) } \]

  \[ = T(n-2) + 2(6n-5) - 6 \]

• \[ = T(n-3) + 6n - 5 + 2(6n-5) - 6 \]
  \[ \text{ (substitute) } \]

• \[ = T(n-3) + 6n - 5 + 2(6n-5) - 6 \]

• \[ = T(n-3) + 3(6n-5) - 6(1+2) \]

• \[ = T(0) + n(6n-5) - 6(1+2 + \cdots + (n-1)) = \text{guess}(n) \]

WANT TO KNOW?

• \[ \text{guess}(n) = T(0) + n(6n-5) - 6(1+2+3+ \cdots + (n-1)) \]

• \[ = 4 + 6n^2 - 5n - 6n(n-1)/2 \]
  \[ \text{ (simplify) } \]

• \[ = 3n^2 - 2n + 4 \]

Are we done?

• The form of \[ \text{guess}(n) \] was an educated guess.

To be formal, we must prove it correct using induction.

ANOTHER APPROACH

• Suppose you look for a while at the previous recurrence:
  \[ T(0) = 4; \quad T(n) = T(n-1) + 6n - 5 \]

• With some experience, you might just guess it’s quadratic.

• If you’re right, it should have the form:
  \[ an^2 + bn + c \]

• 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:

\[ T(0) = 4; \quad T(n) = T(n-1) + 6n - 5 \]

\[ \text{guess}(n) = 3n^2 - 2n + 4 \]

WANT TO PROVE:

\[ \text{guess}(n) = T(n) \text{ for all n} \]

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 \[ \text{guess}(n+1) = T(n+1) \].

• \[ T(n+1) = T(n) + 6(n+1) - 5 \] \[ \text{ (by definition) } \]

• \[ = \text{guess}(n) + 6(n+1) - 5 \] \[ \text{ (by inductive hypothesis) } \]

• \[ = 3n^2 + 4n + 5 \] \[ \text{ (substitute & simplify) } \]

\[ \text{guess}(n+1) = 3(n+1)^2 - 2(n+1) + 4 \] \[ \text{ (by definition) } \]

\[ = 3n^2 + 4n + 5 = T(n+1) \] \[ \text{ (simplify) } \]
• \( T(0) = 4; T(n) = T(n-1) + 6n - 5 \), \( \text{guess}(n) = an^2 + bn + c \)

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

• Base case: \( \text{guess}(0) = a(0)^2 + b(0) + c = T(0) = 4 \)
  
  \( c = 4 \) 
  
  \((a, b) \) are not constrained \)

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

• \( T(n+1) = T(n) + 6(n+1) - 5 \)

• \( = \text{guess}(n) + 6(n+1) - 5 \) 

• \( = an^2 + bn + 4 + 6n + 1 - 5 \) 

• \( = an^2 + (b + 6)n + 5 \) 

• \( \text{Recall:} \) \( \text{guess}(n) = an^2 + bn + c \) 

• \( c = 4 \) 

• \( T(n) = an^2 + (b + 6)n + 5 \) 

• \( \text{Inductive hypothesis is correct for } a = 3, b = -2, c = 4 \)

Example corresponding algorithm

```
Lvl 0 = 1
Lvl 1 = a
Lvl 2 = a^2
...
```

Let's rearrange this into a geometric sequence and solve.

```
\[ T(n) = da^0 + \sum_{i=0}^{n} c a^i \]
```

Let's solve for \( T(n) \) as \( n \rightarrow \infty \).
The document contains several mathematical formulas and explanations related to solving geometric sequences and recurrence relations. Here is a transcription of the key points:

**Solving the Geometric Seq**

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

- **Case 2:** \( r = b^{y-x} = 1 \Rightarrow x - y = 0 \Rightarrow x = y 
  \)
  - \( T(n) = dn^2 + cn \sum_{i=0}^{r-1} r^i \in \Theta(n^2) \)
  - \( T(n) \in \Theta(n^2 + n^2) = \Theta(n^2) \) since \( x = y \)
  - Recall \( b^l = n \), so \( \log_{b} n \).
  - So \( T(n) = \Theta(n^2 + n^2) = \Theta(n^2 \log n) \)

**Solving the Geometric Seq**

- **Case 3:** \( 0 < r = b^{y-x} < 1 \Rightarrow x - y < 0 \Rightarrow x < y 
  \)
  - \( T(n) = dn^2 + cn \sum_{i=0}^{r-1} r^i \in \Theta(n^2) \)
  - \( T(n) \in \Theta(n^2 + n^2r^r) \)
  - Since \( x < y \), we simply have \( T(n) \in \Theta(n^2) \)

**Some Bonus Intuition for \( r \) Cases**

Recall: \( T(n) = dn^2 + cn \sum_{i=0}^{r-1} r^i \) where \( r = b^{y-x} \)

- **Heavy leaves:** \( r > 1 \) and \( y < r \), \( T(n) \in \Theta(n^2) \)
- **Balanced:** \( r = 1 \) and \( y = r \), \( T(n) \in \Theta(n^2 \log n) \)
- **Heavy top:** \( r < 1 \) and \( y > r \), \( T(n) \in \Theta(n^2) \)

**Master Theorem for Recurrences**

- **Simplified version:**
  Consider recurrence:
  \( T(n) = aT(\frac{n}{b}) + \Theta(n^k) \) where \( a \geq 1 \), \( b \geq 2 \) and \( n = b^l 
  \)
  And let \( x = \log_b a \).

  \[ T(n) = \begin{cases} \Theta(n^k) & \text{if } y < x \\ \Theta(n^{k(\log n)}) & \text{if } y = x \\ \Theta(n^x) & \text{if } y > x. \end{cases} \]

**Worked Examples**

- **Recall:** simplified master theorem
- **Suppose that \( a \geq 1 \) and \( b > 1 \). Consider the recurrence
  \( T(n) = aT(\frac{n}{b}) + \Theta(n^k) \), where \( n \) is a power of \( b \).

  Denote \( x = \log_b a \). Then
  \( T(n) \in \begin{cases} \Theta(n^k) & \text{if } y < x \\ \Theta(n^{k(\log n)}) & \text{if } y = x \\ \Theta(n^x) & \text{if } y > x. \end{cases} \)

  **Questions:**
  - \( a=2 \), \( b=2 \), \( y=1 \), \( x=1 \)

  \( T(n) = 2T(\frac{n}{2}) + n \)

  \( \Theta(n^2 \log n) = \Theta(n \log n) \)

  \( T(n) = 2T(n/2) + \Theta(n) \)

  **Questions:**
  - \( a=2 \), \( b=2 \), \( y=1 \), \( x=1 \)

  \( T(n) = 2T(n/2) + \Theta(n) \)
**GENERAL MASTER THEOREM**

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

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

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

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

**REVISITING THE RECURSION TREE METHOD**

### Example recurrence:

\[
T(n) = 3T\left(\frac{n}{3}\right) + 7
\]

<table>
<thead>
<tr>
<th>level</th>
<th>nodes</th>
<th>value at each node</th>
<th>value of the level</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j )</td>
<td>( 1 )</td>
<td>( 2^j )</td>
<td>( 2^j )</td>
</tr>
<tr>
<td>( j-1 )</td>
<td>( 2 )</td>
<td>( j)2^{j-1} )</td>
<td>( j)2^{j-1} )</td>
</tr>
<tr>
<td>( j-2 )</td>
<td>( 2^2 )</td>
<td>( j-2)2^{j-2} )</td>
<td>( j-2)2^{j-2} )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( 1 )</td>
<td>( 2^1 )</td>
<td>( 2^1 )</td>
<td>( 2^1 )</td>
</tr>
<tr>
<td>( 0 )</td>
<td>( 2^0 )</td>
<td>( 1 )</td>
<td>( 2^0 )</td>
</tr>
</tbody>
</table>

**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 \( a = 3 \); \( T(1) = 1 \); \( T(n) = 3T\left(\frac{n}{3}\right) + n \log n \)

**MASTER THEOREM WHEN \( b^{1/2} < n < b^j \)**

- \( n \) is not always an integer
- Floor/ceilings are hard
- Not a geometric sequence
- Suppose we get a \( \Theta \) bound for \( b^{1/2} < n < b^j \)

\[
\begin{align*}
\Theta((b^j)k^x) & \text{ if } y < x \\
\Theta((b^j)k^x \log b^j) & \text{ if } y = x \\
\Theta((b^j)k^x) & \text{ if } y > x
\end{align*}
\]

**MASTER THEOREM WHEN \( b^{1/2} < n < b^j \)**

- \( T(n) \leq \Theta((b^j)k^x \log b^j) \)

**Case 1** (\( y < x \)): \( (b^j)^x = b^{jx} \) and \( b^j \) is a constant

- So \( T(n) \in \Theta((b^j)k^x) \)

**Case 2** (\( y = x \)): \( (b^j)^x \log b^j = b^{jx} \log b^j + \log b^j \)

- So \( T(n) \in \Theta((b^j)k^x \log b^j) \)

**Case 3** (\( y > x \)): \( (b^j)^x \log b^j + (b^j)^x \log b^j \)

- So \( T(n) \in \Theta((b^j)k^x \log b^j) \)

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