Academic Integrity

• Collaboration Encouraged!
  – Groups of up to 5 per assignment (you + 4)
  – List your collaborators (by UVA computing ID)
• Write-ups/code written independently
  – DO NOT share written notes / pictures / code
  – DO NOT share documents (ex: Overleaf)
• Be able to explain any solution you submit!
• DO NOT seek published solutions online
Late Policy

• By default, late submissions not accepted
• If something comes up that prevents you from submitting quality work on time, let me know what’s going on
Exams

• No exams
Regrades

• Conducted using the submission system:
  – Submit within 5 days of receiving your grade
  – Request a regrade if the rubric was misapplied
Course webpage

• www.cs.virginia.edu/~njb2b/cs4102/su20
Extra “credit”

• Given for extraordinary acts of engagement
  – Good questions/comments
  – Quality discussions
  – Analysis of current events
  – References to arts and music
  – Extra credit projects
  – Slide corrections
  – Etc. Just ask!

• Submit to me via email
• Will be used for qualitative grade adjustments
Can you cover an $8 \times 8$ grid with 1 square missing using “trominoes?”

With these?
Where does it end?

• I have a pile of string
• I have one end of the string in my hand
• I need to find the other end
• How can I do this efficiently?
Rope End Finding

1. Set aside the already obtained end

2. Separate the pile of rope into 2 piles, note which connects to the known end (call it pile A, the other pile B)

3. Count the number of strands crossing the piles

4. If the count is even, pile A contains the end, else pile B does
How efficient is it?

- \( T(n) = \text{count}(n) + T\left(\frac{n}{2}\right) \)
- \( T(n) = 5 + T\left(\frac{n}{2}\right) \)
- Base case: \( T(1) = 1 \)
Let’s solve the recurrence!

\[ T(n) = 5 + T\left(\frac{n}{2}\right) \]

\[ T(1) = 1 \]

\[ T(n) = 5 + T\left(\frac{n}{4}\right) \]

\[ T(n) = 5 + T\left(\frac{n}{8}\right) \]

\[ \vdots \]

\[ T(n) = \sum_{i=0}^{\log_2 n} 5 + 1 = 5 \log_2 n + 1 \]
Algorithm Running Times

- We can’t just measure running time with a number
- Why shouldn’t we say: “The running time of this algorithm is 8.”?
- Units: 8 what? Seconds?
- What if the input is large?
- Algorithm running times are functions
  - Domain: “sizes” of the algorithm’s input
  - Co-Domain: “counts” of operations
- We want to be able to say “algorithm 1 is faster than algorithm 2”
- How can we compare functions?
Asymptotic Notation*

• \( O(g(n)) \)
  – At most within constant of \( g \) for large \( n \)
  – \( \{ \text{functions } f | \exists \text{ constants } c, n_0 > 0 \text{ s.t. } \forall n > n_0, f(n) \leq c \cdot g(n) \} \)

• \( \Omega(g(n)) \)
  – At least within constant of \( g \) for large \( n \)
  – \( \{ \text{functions } f | \exists \text{ constants } c, n_0 > 0 \text{ s.t. } \forall n > n_0, f(n) \geq c \cdot g(n) \} \)

• \( \Theta(g(n)) \)
  – “Tightly” within constant of \( g \) for large \( n \)
  – \( \Omega(g(n)) \cap O(g(n)) \)

*CLRS Chapter 3
\[ f(n) = O(g(n)) \]
\[ f(n) = \Theta(g(n)) \]
\[ f(n) = \Omega(g(n)) \]
Asymptotic Notation

- $o(g(n))$
  - Below any constant of $g$ for large $n$
  - $\{f \mid \forall$ constants $c$, $\exists n_0$ s.t. $\forall n > n_0, f(n) < c \cdot g(n)\}$

- $\omega(g(n))$
  - Above any constant of $g$ for large $n$
  - $\{f \mid \forall$ constants $c$, $\exists n_0$ s.t. $\forall n > n_0, f(n) > c \cdot g(n)\}$

- $\Theta(g(n))$
  - $o(g(n)) \cap \omega(g(n)) = \emptyset$
Asymptotic Notation Example

• Show: $n \log n \in O(n^2)$
Asymptotic Notation Example

• To Show: \( n \log n \in O(n^2) \)
  
  – **Technique:** Find \( c, n_0 > 0 \) s.t. \( \forall n > n_0, n \log n \leq c \cdot n^2 \)

  – **Proof:** Let \( c = 1, n_0 = 1 \). Then,
    
    \[
    n_0 \log n_0 = (1) \log (1) = 0, \\
    c \cdot n_0^2 = 1 \cdot 1^2 = 1, \\
    0 \leq 1.
    \]

    \( \forall n \geq 1, \log(n) < n \Rightarrow n \log n \leq n^2 \quad \square \)
Asymptotic Notation Example

• Show: $n^2 \not\in O(n)$
Asymptotic Notation Example

• To Show: $n^2 \notin O(n)$
  – **Technique: Contradiction**
  – **Proof:** Assume $n^2 \in O(n)$. Then $\exists c, n_0 > 0$ s.t. $\forall n > n_0, n^2 \leq cn$
    Let us derive constant $c$. For all $n > n_0 > 0$, we know:
    $cn \geq n^2$,
    $c \geq n$.

    Since $c$ is lower bounded by $n$, it is not a constant.
    Contradiction. Therefore $n^2 \notin O(n)$. □
Asymptotic Notation Example

• \( o(g(n)) = \{ \text{functions } f : \forall \text{ constants } c > 0, \exists n_0 \text{ s.t. } \forall n > n_0, f(n) < c \cdot g(n) \} \)

• Show: \( n \log n \in o(n^2) \)
  
  – given any \( c \) find a \( n_0 > 0 \) s.t. \( \forall n > n_0, n \log n < c \cdot n^2 \)
  
  – Find a value of \( n \) in terms of \( c \):
    
    • \( n \log n < c \cdot n^2 \)
    • \( \log n < c \cdot n \)
    • \( \frac{\log n}{n} < c \)

  – For a given \( c \), select any value of \( n_0 \) such that \( \frac{\log n}{n} < c \)
Trominoes Puzzle Solution

What about larger boards?
Trominoes Puzzle Solution

Divide the board into quadrants
Trominoes Puzzle Solution

Place a tromino to occupy the three quadrants without the missing piece
Trominoes Puzzle Solution

Each quadrant is now a smaller subproblem
Trominoes Puzzle Solution

Solve Recursively
Divide and Conquer

Our first algorithmic technique!
Trominoes Puzzle Solution
Divide and Conquer*

• **Divide:**
  – Break the problem into multiple subproblems, each smaller instances of the original

• **Conquer:**
  – If the suproblems are “large”:
    • Solve each subproblem recursively
  – If the subproblems are “small”:
    • Solve them directly (base case)

• **Combine:**
  – Merge together solutions to subproblems

*CLRS Chapter 4
Analyzing Divide and Conquer

1. Break into smaller **subproblems**
2. Use **recurrence** relation to express recursive running time
3. Use **asymptotic** notation to simplify

- **Divide**: $D(n)$ time,
- **Conquer**: recurse on small problems, size $s$
- **Combine**: $C(n)$ time
- **Recurrence**: 
  
  
  $T(n) = D(n) + \sum T(s) + C(n)$
Recurrence Solving Techniques

Tree  get a picture of recursion

Guess/Check  guess and use induction to prove

“Cookbook”  MAGIC!
Merge Sort

- **Divide:**
  - Break $n$-element list into two lists of $n/2$ elements

- **Conquer:**
  - If $n > 1$:
    - Sort each sublist *recursively*
  - If $n = 1$:
    - List is already sorted (base case)

- **Combine:**
  - Merge together sorted sublists into one sorted list
**Merge**

- **Combine**: Merge sorted sublists into one sorted list
- We have:
  - 2 sorted lists \((L_1, L_2)\)
  - 1 output list \((L_{out})\)

While \((L_1 \text{ and } L_2\) not empty):

  If \(L_1[0] \leq L_2[0]\):

    \[
    L_{out}.append(L_1.pop())
    \]

  Else:

    \[
    L_{out}.append(L_2.pop())
    \]

\[
L_{out}.append(L_1) \\
L_{out}.append(L_2)
\]

\(O(n)\)
Analyzing Merge Sort

1. Break into smaller **subproblems**
2. Use **recurrence** relation to express recursive running time
3. Use **asymptotic** notation to simplify

- **Divide**: 0 comparisons
- **Conquer**: recurse on 2 small subproblems, size $\frac{n}{2}$
- **Combine**: $n$ comparisons
- **Recurrence**:

  $$ T(n) = 2 \ T\left(\frac{n}{2}\right) + n $$
Recurrence Solving Techniques

Tree

get a picture of recursion

Guess/Check

guess and use induction to prove

“Cookbook” MAGIC!
Tree method

\[ T(n) = 2T\left(\frac{n}{2}\right) + n \]

\[ \Rightarrow n \text{ total / level} \]

\[ \log_2 n \text{ levels of recursion} \]

\[ T(n) = \sum_{i=1}^{\log_2 n} n = n \log_2 n \]