Divide & Conquer

- Break up the problem into (multiple, smaller) parts
- Solve each of the parts recursively
- Combine the solution of each of the parts into a solution of the original problem
First example: Merge sort

- Divide the array into two halves
- Recursively sort each half
- Merge the two sorted halves

Analysis

Divide $O(1)$

Merge $O(n)$

What about the recursive calls?

$2T\left(\frac{n}{2}\right)$

Complexity of merge

- Time: $O(n)$
- Space: $O(n)$
  - Often with two arrays of length $n$
  - Can you do (a constant factor) better?
**Recurrence relations**

- A recurrence relation for a sequence, \( \{a_n\} \) is an equation that expresses \( a_n \) in terms of one or more of the previous elements of the sequence, \( a_1, a_2, \ldots a_{n-1} \)
- There may be base cases, and the equation hold for \( n \geq n_0 \) for some constant \( n_0 \)
- Example: \( a_n = 2a_{n-1} + 1 \) and \( a_1 = 1 \)
- After setting up the recurrence relation, we solve it

**Recurrence relation for Merge-sort**

- Define the number of comparisons to sort an input of length \( n \) as: \( T(n) \)
- Use the structure of the D&C algorithm to define an equation/relation for \( T(n) \)

\[
T(n) \leq \begin{cases} 
  0 & \text{if } n = 1 \\
  c + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T\left(\left\lceil \frac{n}{2} \right\rceil \right) + cn & \text{otherwise}
\end{cases}
\]
Solving the Recurrence

\[ T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T\left(\frac{n}{2}\right) + cn & \text{otherwise}
\end{cases} \]

- **Solution:**
  \[ T(n) = \Theta(n \log n) \]

- **Number of techniques**
  - Unrolling the recurrence
  - Repeated substitution
  - See a pattern, guess and then prove by induction

Unrolling \( T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T\left(\frac{n}{2}\right) + cn & \text{otherwise}
\end{cases} \)
What is the "label" of each node?
When does the label become "small enough" (base case)?
How many levels in the tree? [Hint: use the above two]
How many nodes at each level?
What is the "contribution" of each node?
What is the contribution of each level?
How many leaves?
Contribution of the leaves (different from contribution of other levels)

Recurrent substitution for \( T(n) = \begin{cases} \frac{c}{2} \log_2(n) + cn & \text{if } n = 1 \\ \frac{c}{2} \log_2(n) + cn & \text{otherwise} \end{cases} \)

Claim: \( T(n) = cn \log_2 n \)

\[
T(n) = 2T(n/2) + cn \\
= 4T(n/4) + cn + 2cn/2 \\
= 8T(n/8) + cn + cn + 4cn/4 \\
\vdots \\
= 2^{\log_2 n} T(1) + \underbrace{cn + \cdots + cn}_{\log_2 n} \\
= O(n \log_2 n)
\]

This reaches \( T(1) \) when \( n = 2^{\log_2 n} \) by definition of \( \log_2 n \)
Towers of Hanoi

- Move all disks to third peg, without ever placing a larger disk on a smaller one.
- What’s the recurrence relation? \( a_n = 2a_{n-1} + 1 \) with the base case that \( a_1 = 1 \)
- Let’s solve by repeated substitution
  - Plug in the definition
  - Do the algebra to collect all the non-recursive expressions together
  - Identify a pattern
  - Determine how many times the pattern occurs until we hit the base case

Hanoi by repeated substitution

\[
T(n) = 2T(n-1) + 1 \\
= 2(2T(n-2) + 1) + 1 \\
= 4T(n-2) + 2 + 1 \\
= 4(2T(n-3) + 1) + 2 + 1 \\
= 8T(n-3) + 4 + 2 + 1
\]

- What is the label and how is it changing?
- What about the other terms?
- When do we hit the base case?
Hanoi by repeated substitution

\[ T(n) = 2T(n - 1) + 1 \]
\[ = 2(2T(n - 2) + 1) + 1 \]
\[ = 4T(n - 2) + 2 + 1 \]
\[ = 4(2T(n - 3) + 1) + 2 + 1 \]
\[ = 8T(n - 3) + 4 + 2 + 1 \]
\[ \vdots \]
\[ = 2^i T(n - i) + \sum_{j=0}^{i-1} 2^j \]

- When does the label become 1?
- When \( i = n - 1 \) So our solution is

\[ \sum_{j=0}^{n-1} 2^j = 2^n - 1 = \Theta(2^n) \]

This is a geometric series.
Binary search

function BS(x, start, end)
    if (end <= start)
        return A[start]
    mid = (end + start)/2
    if A[mid] < x
        return BS(x, mid, end)
    return BS(x, start, mid-1)

- What is the recurrence?
- Apply repeated substitution (on doc cam or exercise)

Find max in an unsorted array

Algorithm:
- Base case n=1
- Otherwise: find the max of the two halves, and return the max of that

function FM(start, end)
    if (end = start)
        return A[start]
    mid = (end + start)/2
    return max( FM(start, mid-1), FM(mid, end) )
Find max in an unsorted array

Recurrence: base case: \( T(1) = 0 \)
Otherwise: \( T(n) = 2T\left(\frac{n}{2}\right) + 1 \)
\[
= 4T\left(\frac{n}{4}\right) + 2 + 1 \\
= 8T\left(\frac{n}{8}\right) + 4 + 2 + 1 \\
\vdots \\
= 2^k T\left(\frac{n}{2^k}\right) + 2^{k-1} + 2^{k-2} + \ldots + 2^0 \\
= 2^k T\left(\frac{n}{2^k}\right) + 2^{k-1} - 1 \\
= 2^k T\left(\frac{n}{2^k}\right) + 2^k - 1 \\
\]
Bae case is reached when \( 2^k = n \), i.e., \( k = \log_2 n \), So
\( T(n) = 0 + 2^{\log n} - 1 = n - 1 \)

Another example

function foo(A, B) // the size of A is n
    if (n == 1):
        return fuzz(A, B) // base case, fuzz is constant time

    // Process A to build two parts, A_0 and A_1 of size n/2 each
    C_0 = foo(A_0, B)
    C_1 = foo(A_1, B)
    return buzz(C_0, C_1) // buzz is \( O(n^2) \)
**General D&C**

```plaintext
function foo(parameters) // the size of A is n
    if (n <= b): // base case
        return fuzz(A, B) // base case
    // Divide input into a parts, each of size n/b
    divide() // Make a calls to
    foo(new parameters) // size is n/b
    return combine(r1, ra) // Complexity of divide and combine is O(n^d)
```

**Master Theorem**

- Let \( a \geq 1, b > 1, n = b^k \) and \( T(n) \) be given by
  \[
  T(n) = aT\left(\frac{n}{b}\right) + cn^d
  \]
- The solution of the recurrence is
  \[
  T(n) = \begin{cases} 
  O(n^d) & \text{if } a < b^d \\
  O(n^d \log n) & \text{if } a = b^d \\
  O(n^{\log_b a}) & \text{if } a > b^d 
  \end{cases}
  \]
Merge-sort by master theorem

- $a = 2, b = 2, d = 1$
- So, $b^d = 2 = a$

... and the solution is

$$T(n) = O(n^d \log n) = O(n \log n)$$