Divide-and-Conquer
Divide-and-Conquer technique

- **Divide**: Divide (break) the problem (instance) into a number of **subproblems** that are smaller instances of the same problem.
- **Conquer**: Solve the subproblems recursively. If the subproblem sizes are small enough just solve them in a straightforward manner.
- **Combine**: Combine the subproblem solutions into the solution for the original problem.
Divide and Conquer technique: Proving correctness

- Prove base cases are correct
- Prove the correctness of the part combining the result
- Proof by induction the whole algorithm is correct
  - Assume that the subproblems are solved correctly and use that to prove the correctness of the problem
Divide and Conquer technique: Runtime

- Develop a recurrence relation representing the time complexity of the algorithm
- Solve the relation using one of the following method
  - guess a solution (using substitution method) and prove its correctness by induction
  - Use recurrence tree to find a solution and then prove its correctness using induction
  - Master theorem
Example: Mergesort

- Divide: Split the array into two half
- Conquer: Sort each half using mergesort
- Combine: Merge the two half

**Base case**: An empty or single-element array is sorted
<table>
<thead>
<tr>
<th>68</th>
<th>89</th>
<th>45</th>
<th>10</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>68</td>
<td>89</td>
<td>45</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>68</td>
<td>89</td>
<td>45</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>68</td>
<td>89</td>
<td>45</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>68</td>
<td>89</td>
<td>45</td>
<td>10</td>
<td>20</td>
</tr>
</tbody>
</table>

split

merge
def merge(A, B):
    result = []
    i = 0
    while i < len(A) and j < len(B):
        if A[i] <= A[j]:
            result.append(A[i])
            i += 1
        else:
            result.append(B[j])
            j += 1
    while i < len(A):
        result.append(A[i])
        i += 1
    while j < len(B):
        result.append(B[j])
        j += 1
    return result
Mergesort algorithm

def mergesort(A):
    if len(A) <= 1:
        return A
    # Divide the list into two halves L and R
    mid = (ceil(n/2))
    L = A[1 … mid]
    R = A[mid+1 … n]
    mergesort(L)
    mergesort(R)
    return merge(L, R)

Runtime:

$$T(1) = \Theta(1)$$

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + T\left(\left\lceil \frac{n}{2} \right\rceil \right) + \Theta(n)$$
Question

Why not split the array into fourth? Or eights?
Runtime of recursive algorithms

- **Recurrence relation:**
  a. describes the runtime of a problem of size n in terms of the runtime on smaller inputs.
  b. Used in algorithm that use recursion like divide-and-conquer technique

- **General approach for solving a recurrence**
  a. finding an explicit expression
  b. Finding an asymptotic bound on its growth rate

- **Techniques to solve recurrence relation:**
  a. Guessing method
     i. How to guess: Substitution method (educated guess)
  b. Recurrence tree method

- Both of the above techniques solve a recurrence intuitively. Then we can use induction to prove their correctness formally.
Solving Recurrence Relation for MergeSort

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

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

1) We assume \( n \) is a power of 2 → floor and ceiling are not needed
2) \( \Theta(1) \) is a constant and \( \Theta(n) \) is multiple of a constant.
   a) We use the same constant
Technique 1: Substitution method

- Guess a solution by plugging the recurrence into itself until you spot a pattern
- Prove its correctness using induction

Finding a guess assuming $n$ is a power of 2

$n = 2^k$
$n / 2^k = 1$
$log_2 n = k$

\[
T(n) \leq 2T \left( \frac{n}{2} \right) + cn
\]
\[
\leq 2 \left( 2T \left( \frac{n}{4} \right) + \left( \frac{cn}{2} \right) \right) + cn
\]
\[
= 4T \left( \frac{n}{4} \right) + cn + cn
\]
\[
= 4T \left( \frac{n}{4} \right) + 2cn
\]
\[
\leq 4 \left( 2T \left( \frac{n}{8} \right) + \left( \frac{cn}{4} \right) \right) + 2cn
\]
\[
= 8T \left( \frac{n}{8} \right) + cn + 2cn
\]
\[
= 8T \left( \frac{n}{8} \right) + 3cn
\]
\[
\vdots
\]
\[
\leq 2^k T \left( \frac{n}{2^k} \right) + kcn
\]
\[
= 2^{\log_2 n} T(1) + cn \log_2 n
\]
\[
= nT(1) + cn \log_2 n
\]
\[
\leq cn + cn \log_2 n
\]
\[
= O(n \log n)
\]
Technique 1: Substitution method: proving using induction

Proving the correctness of guess using induction. In other words, prove

\[ T(n) \leq cn + cn \log n \]

Proof: base case, if \( n = 1 \), then \( T(n) = T(1) \leq cn \log_2 n + cn = c \). 

Inductive step:

- **hypothesis**: assume the claim holds for all \( m < n \) that are powers of two
- **Induction**:
  \[
  T(n) \leq 2T\left(\frac{n}{2}\right) + cn \\
  \leq 2 \left( \left( \frac{cn}{2} \right) \lg \frac{n}{2} + \left( \frac{cn}{2} \right) \right) + cn \\
  = cn \lg \frac{n}{2} + cn + cn \\
  = cn (\lg n - 1) + cn + cn \\
  = cn \lg n - cn + cn + cn \\
  \leq cn \lg n + cn
  \]
What if $n$ is not a power of 2?

The previous inequality is true if $n$ is a power of two.

Since the function $T(n)$ is increasing, then $T(n) \leq T(n')$ where $n'$ is the first power of two larger than $n$: $n \leq n' < 2n$

So, if we prove a bound for $T(n')$ we can use that to get a bound for $T(n)$

\[
T(n) \leq T(n') \\
\leq cn' \log n' + cn' \\
\leq (2n)c \log(2n) + 2nc \\
\leq 2nc \log n + 2nc + 2nc \\
\leq 2nc \log n + 4nc \\
\leq 4nc \log n + 4nc \\
\leq nc' \log n + nc' \\
\in \Theta(n \log n)
\]
A different approach to deal with floor and ceiling

Once you have a guess, and you want to use induction to prove it, you can prove separately for odd and even number.
Methods for solving recurrence relation

Substitution method

Recursion tree
Recursion tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$h = \lg n$

$\Theta(1)$

$\#\text{leaves} = n$

Total $= \Theta(n \lg n)$
Master Theorem

Let $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where $a \geq 1, b \geq 2$ and $f(n) = \Theta(n^k)$

$$T(n) \in \begin{cases} 
\Theta(n^k) & \text{if } a < b^k \\
\Theta(n^k \log n) & \text{if } a = b^k \\
\Theta(n^{\log_b a}) & \text{if } a > b^k 
\end{cases}$$
Proving Master Theorem
The recursion tree for \( T(n) = aT\left(\frac{n}{b}\right) + f(n) \)

- **Level 0**: \( f(n) \)
- **Level 1**: \( af\left(\frac{n}{b}\right) \)
- **Level 2**: \( a^2 f\left(\frac{n}{b^2}\right) \)
- **Level h**: \( \theta(n^{\log_b a}) \)

**Leaf nodes**: \( a^h = a^{\log_b n} = n^{\log_b a} \)

\[ T(n) = \theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n-1} a^j f\left(\frac{n}{b^j}\right) \]
Proving Master Theorem \( T(n) = aT\left(\frac{n}{b}\right) + f(n) \)

Using Recursion Tree:

\[
T(n) = n^{\log_b a} T(1) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)
\]

\[
f(n) = cn^k
\]

\[
f(n/b^j) = c\left(\frac{n}{b^j}\right)^k
\]

\[
T(n) = n^{\log_b a} T(1) + \sum_{j=0}^{\log_b n - 1} a^j c\left(\frac{n}{b^j}\right)^k
\]
Case 1: \( T(n) = aT\left(\frac{n}{b}\right) + f(n), a < b^k \)

\[
T(n) = n^{\log_b a} T(1) + cn^k \sum_{j=0}^{\log_b n - 1} \left( \frac{a}{b^k} \right)^j
\]

\[
\sum \left( \frac{a}{b^k} \right)^j \text{ is a geometric series and } \frac{a}{b^k} < 1 \text{ So } \sum \text{ is constant:}
\]

\[
T(n) = n^{\log_b a} T(1) + \Theta(n^k) \in \Theta(n^k)
\]
Geometric series

$$1 + x + x^2 + \ldots + x^d = \frac{x^{d+1} - 1}{x - 1}$$

$$\sum_{i=0}^{t-1} x^i = \frac{x^t - 1}{x - 1} \in \Theta(x^t) \text{ if } x > 1$$
Case 2: \( T(n) = aT(n/b) + f(n), a = b^k \)

\[
T(n) = n^{\log_b a} T(1) + cn^k \sum_{j=0}^{\log_b n - 1} \left( \frac{a}{b^k} \right)^j
\]

If \( a = b^k \) then
\[
\sum_{j=0}^{\log_b n - 1} \left( \frac{a}{b^k} \right)^j = \sum_{j=0}^{\log_b n - 1} 1 = \Theta(\log_b n) = \Theta(\log n)
\]

So
\[
T(n) = n^{\log_b a} T(1) + cn^k (\Theta(\log n)) \in \Theta(n^k \log n)
\]
Case 3:

\[ T(n) = aT\left(\frac{n}{b}\right) + f(n), \quad a > b^k \]

\[
T(n) = n^{\log_b a} T(1) + cn^k \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^k}\right)^j
\]

If \( a > b^k \) then \( \sum_{j=0}^{\log_b n - 1} \left(\frac{a}{b^k}\right)^j \) is a geometric series with \( \frac{a}{b^k} > 1 \). Therefore

\[ b^{k \log_b n} = \left( b^{\log_b n} \right)^k = n^k \]

\[ a^{\log_b n} = \]

\[ (b^{\log_b a})^{\log_b n} = \]

\[ (b^{\log_b n})^{\log_b a} = n^{\log_b a} \]

\[ \leq c \frac{1}{b^k - 1} a^{\log_b n} \]

\[ \leq c' a^{\log_b n} = c' n^{\log_b a} \]

\[ \leq \Theta(n^{\log_b a}) \]
Master Theorem

Case 1: The parent node does more work than the child nodes

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

\[ T(1) = C \]
\[ T(n) = T(\frac{n}{2}) + Cn \]
\[ T(n) = \Theta(n) \]

\[ \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{n} = \sum_{n=1}^{\infty} \left(\frac{1}{2}\right)^n = 1 \]
Master Theorem

Case 2: The parent node and children nodes do an equal amount of work

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

\[ T(i) = C \]

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

\[ a=2 \]
\[ b=2 \]
\[ k=1 \]

\[ O(n \log n) \]
### Master Theorem

**Case 3:** The children nodes do more work than the parent node

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

- \(a=2\)
- \(b=2\)
- \(k=0\)

\[\log_b a = 1\]

\[T(n) = O(n)\]
Mergesort analysis using master theorem

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

\(a=2, \ b=2, \ k=1 \rightarrow \log_b a = 1 \rightarrow T(n) = O(n \log n)\)

\[
T(n) \in \begin{cases} 
  \Theta(n^k) & \text{if} \quad a < b^k \\
  \Theta(n^k \log n) & \text{if} \quad a = b^k \\
  \Theta(n^{\log_b a}) & \text{if} \quad a > b^k 
\end{cases}
\]
Proving correctness of Mergesort

- **Theorem:** for all $n \geq 1$, mergesort correctly sorts any array of size $n$.
- **Proof by induction:**
  - The base case is when $n=1$.
  - Induction step: mergesort correctly sorts any array of size $< n$.

```python
def mergesort(A):
    if len(A) <= 1:
        return A
    mid = len(A) // 2
    L = A[:mid]
    R = A[mid:]
    mergesort(L)
    mergesort(R)
    result = merge(L, R)
    return result
```
def merge(A, B):
    R = []
    i = 0
    j = 0
    while i < len(A) or j < len(B):
        if i==len(A):
            R.append(B[j])
            j += 1
        elif j==len(B):
            R.append(A[i])
            i += 1
        else:
            if A[i]<=B[j]:
                R.append(A[i])
                i += 1
            else:
                R.append(B[j])
                j += 1
    return result

Loop invariant: At the start of each iteration of the while loop:

1. R contains the first i-th elements of A and first j-th elements of B in sorted order.
2. Each element in A[i..len(A)-1] and B[j..len(B)-1] is greater than or equal to any elements in R

Proving correctness of Mergesort
Caution when proving correctness of recurrence relations

- Pay attention to the correctness proof

\[ T(n) = 2T(n/2) + 1 \quad T(1) = 0 \]

- What is wrong with the following?

**Guess:** \( T(n) = O(n) \)
**Proof:** There exists a constant \( c \) such that \( T(n) \leq cn \) for all \( n \geq 1 \).
  - **Base case:** \( n=1, \ T(1)=0 \leq c \) for any constant \( c \geq 0 \).
  - **Induction hypothesis:** Assume that \( T(n') \leq cn' \) for all \( n' < n \), some \( n \geq 2 \).
  - **Induction:** \( T(n) = 2T(n/2) + 1 \leq c \, n/2 + c \, n/2 + 1 = cn+1 \rightarrow T(n) \leq cn + 1 \)

  does not imply \( T(n) \leq cn \) for any choice of \( c \)

- Therefore we cannot conclude that \( T(n) \) is in \( O(n) \)
Caution when proving correctness of recurrence relations

- Pay attention to the correctness proof

\[ T(n) = 2T(n/2) + 1 \quad T(1) = 0 \]

- Does it mean \( T(n) \) is not \( O(n) \)? \textbf{NO}

Assume \( n = 2^k \) you can show that by substitution, \( T(n) = 2n - 1 \)

\[
\begin{align*}
T(n) &= 2T\left(\frac{n}{2}\right) + 1 \\
&= 4T\left(\frac{n}{2}\right) + 2 + 1 \\
&\vdots \\
&= 2^kT\left(\frac{n}{2^k}\right) + (2^{k-1} + \ldots + 2 + 1) \\
&= 2^k + 2^{k-1} + \ldots + 2 + 1 \\
&= 2^{k+1} - 1 \\
&= 2n - 1
\end{align*}
\]
Caution when proving correctness of recurrence relations

- Note: We are using mathematical induction here and it is not working unless we prove the exact form of the inductive hypothesis. **So we change our guess:**

- **Induction hypothesis:** $T(n') \leq cn' - d$ where $d \geq 0$ for all $n' < n$, some $n \geq 2$

- $T(n) = 2T(n/2) + 1 \leq c n/2 - d + c n/2 - d + 1 = cn -2d + 1 \leq cn - d$ when $d \geq 1$
An example of recurrence relations

Changing variables

\[ T(n) = 2 \ T([\sqrt{n}]) + \log n \quad \text{for} \quad m = \log n \]

\[ T(2^m) = 2 \ T(2^{m/2}) + m \]

\[ S(m) = T(2^m) \]

\[ S(m) = 2 \ S(m/2) + m \quad \rightarrow \quad S(m) = O(m \log m) \]

\[ T(n) = O(m \log m) = O(\log n \log \log n) \]
Divide-and-Conquer Algorithms

Counting Inversion
Counting Inversion

- **Input**: a sequence of numbers: \( a_1, \ldots, a_n \)
- **Output**: the number of inversions.
  - Two indices \( i < j \) form an inversion if \( a_i > a_j \)
Counting Inversion

- **Applications:**
  - A website tries to match your preferences with those of other people on the Internet
    - Measure similarity between your ranking and other’s ranking
  - Recommend new things to you which are the items in the list of people with ranking similar to you

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

- How many pairs are ranked differently in the two lists?
- How many pairs of lines cross: All pairs = \(\text{choose}(5,2)\)
- In other words, count number of inversions = number of pairs out of order in the second list
Counting Inversion: Brute-force solution

Check all pairs \(\binom{n}{2}\)

Runtime: O(n^2)
Counting Inversion: First Divide-and-Conquer

- **Divide**: Split the list into two parts A and B
- **Conquer**: Recursively count number of inversions in A ($r_A$) and B ($r_B$)
- **Combine**: Count inversion between A and B: $r_{AB}$
  - For each element $b$ in B count how many elements in A are greater than $b$
- Return $r_A + r_B + r_{AB}$

- Runtime: $T(n) = 2T(n/2) + O(n^2)$
Counting Inversion: First Divide-and-Conquer

- **Example:**
  - **Input:** \( L = [4, 6, 1, 5, 2, 3] \), \( n=6 \)
  - **Output:** Count the number of inversions

- **Algorithm:**
  - **Divide:** \( L = [4, 6, 1, 5, 2, 3] \)
  - Recursively count the number of inversions in \( A (r_A) \) and \( B (r_B) \)
    - \( r_A = 2 \) \( r_B = 2 \)
  - **Combine:** For each element in \( B \) count how many elements in \( A \) are greater
    - number of elements in \( A \) greater than 5: 1
    - number of elements in \( A \) greater than 2: 2
    - number of elements in \( A \) greater than 3: 2
    - \( r_{AB} = 5 \)
  - output is: \( r_A + r_B + r_{AB} = 2 + 2 + 5 = 9 \)
Counting Inversion: Second Divide-and-Conquer

- **Divide**: Split the list into two parts A and B
- **Conquer**: Recursively count number of inversions in A \(r_A\) and B \(r_B\)
- **Combine**: Count inversion between A and B: \(r_{AB}\)
  - Recursively sort the input array and replace the search with a linear search:
  - For each element \(b\) in B
    - Do a binary search in A to find how many elements in A are greater than \(b\)
- Return \(r_A + r_B + r_{AB}\)

- Runtime: \(T(n) = 2T(n/2) + O(n \log n)\)
Counting Inversion: Third Divide-and-Conquer

- **Divide**: Split the list into two parts A and B
- **Conquer**: Recursively sort and count number of inversions in A \( r_A \) and B \( r_B \)
- **Combine**: Count inversion between A and B: \( r_{AB} \)
  - \( r_{AB} = 0 \)
  - Perform merge operation between A and B
    - Scan A and B from left to right
    - Compare \( a_i \) and \( b_j \)
    - If \( a_i < b_j \), then \( a_i \) is not inverted with any element left in B
      - Append \( a_i \) to sorted output list
    - If \( a_i > b_j \), then \( b_j \) is inverted with every element left in A
      - Append \( b_j \) to sorted output list
      - \( r_{AB} = r_{AB} + \text{#elements remaining in A} \)
- Return \( r_A + r_B + r_{AB} \)

- Runtime: \( T(n) = 2T(n/2) + O(n) \)