UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Advanced algorithm I Recurrence Relation

Mai Dahshan

October 25, 2024

Learning Objectives

- Understand recursion and it’s different types
- Understand a recurrence relation and explain its role in analyzing the time complexity of recursive algorithms
- Understand the divide and conquer strategy and its three main steps: divide, conquer, and combine

Recursion

- Recursion is a programming concept where a function calls itself to solve a problem.
- A recursive function breaks a problem down into smaller subproblems that are easier to solve.
- Each call to the function operates on a smaller portion of the original problem, working toward a "base case" where the function can stop calling itself and return a result.

Recursion

- Every recursive function/algorithm involves at least 2 cases:
  - **base case**: A simple occurrence that can be answered directly
  - **recursive case**: A more complex occurrence of the problem that cannot be directly answered, but can instead be described in terms of smaller occurrences of the same problem
- Some recursive algorithms have more than one base or recursive case, but all have at least one of each
- A crucial part of recursive programming is identifying base cases

Example

```python
def countDown(number):
    # This function prints numbers in a countdown to 0 and then back up.
    # It demonstrates direct recursion by printing the number on the way down
    # and implicitly printing back up as the call stack unwinds.

    print(number)  # Print the current number. This happens both on the way down and as the stack unwinds.

    if number == 0:
        # BASE CASE: When 'number' reaches 0, we have reached the stopping condition.
        # At this point, the recursion stops and starts unwinding.
        print('Reached the base case.')
        return  # Exit the function, stopping further recursive calls.

    else:
        # RECURSIVE CASE: If 'number' is not 0, make a recursive call with 'number - 1'.
        # This call continues to decrease 'number' by 1 until the base case is reached.
        countDown(number - 1)
        # After reaching the base case, the call stack starts to unwind, printing 'number'
        # again as it returns to each previous call in reverse order.
#Example
countDown(5)
```

Types of Recursion

- **Direct Recursion**: A function directly calls itself within its body

```python
def power(base, exponent):
    if exponent == 0:
        return 1
    elif exponent < 0:
        return 1 / power(base, -exponent)
    else:
        return base * power(base, exponent - 1)
```

- **Indirect Recursion**: A function calls another function, which eventually calls the original function.
  - Mutual recursion (two functions calling each other)

```python
def is_even(n):
    if n == 0:
        return True
    else:
        return is_odd(n-1)

def is_odd(n):
    if n == 0:
        return False
    else:
        return is_even(n-1)
```

Types of Recursion

- **Tree Recursion**: A function makes multiple recursive calls within its body.

```
def traverse(node):
    if node is not None:
        traverse(node.left)
        print(node.value)
        traverse(node.right)
```

- **Tail Recursion**: The recursive call is the last operation in the function. Can be optimized by some compilers to avoid stack overflow

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
```

- **Nested Recursion**: A recursive call passes the result of another recursive call as an argument

```
def ackermann(m, n):
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))
```

Recursion Problem Types

- Divide and Conquer Algorithms: Divide the problem into sub-problems, solve them recursively (e.g., Merge Sort, Quick Sort).
- Backtracking Algorithms**:** Find all possible solutions by exploring each possibility (e.g., Maze Solving, N-Queens).
- Tree Traversal: Used in data structures like binary trees (e.g., In-order, Pre-order, Post-order traversal).

Drawbacks of Recursion

- High Memory Usage: Each recursive call uses additional memory on the call stack.
- Risk of Stack Overflow: Too many recursive calls can exhaust memory.
- Repeated Calculations: Without optimization, recursion can become inefficient.
- Can result in a runtime error if there
  - No base case
  - No progress towards the base case
  - Using up too many resources (e.g., variable declarations) for each function call

Recurrence Relation

- What is the time complexity of the countDown function?

```python
def countDown(number):
    # This function prints numbers in a countdown to 0 and then back up.
    # It demonstrates direct recursion by printing the number on the way down
    # and implicitly printing back up as the call stack unwinds.

    print(number)  # Print the current number. This happens both on the way down and as the stack unwinds.

    if number == 0:
        # BASE CASE: When 'number' reaches 0, we have reached the stopping condition.
        # At this point, the recursion stops and starts unwinding.
        print('Reached the base case.')
        return  # Exit the function, stopping further recursive calls.

    else:
        # RECURSIVE CASE: If 'number' is not 0, make a recursive call with 'number - 1'.
        # This call continues to decrease 'number' by 1 until the base case is reached.
        countDown(number - 1)
        # After reaching the base case, the call stack starts to unwind, printing 'number'
        # again as it returns to each previous call in reverse order.
#Example
countDown(5)
```

Recurrence Relation

- A **recursive algorithm** provides the solution of a problem of size n in terms of the solutions of one or more instances of the same problem of smaller size
- When we **analyze the complexity of a recursive algorithm**, we obtain a **recurrence relation** that expresses the number of operations required to solve a problem of size n in terms of the number of operations required to solve the problem for one or more instance of smaller size

Recurrence Relation

- **Recurrence relation** is a mathematical equation that defines a sequence of numbers in terms of previous terms in that sequence. It expresses each term as a function of one or more of its predecessors

Recurrence Relation

The recurrence relation of countDown function is given by:

$$T(n) = \begin{cases} O(1) & \text{if } n = 0 \\ T(n-1) + O(1) & \text{if } n > 0 \end{cases}$$

```python
def countDown(number):
    # This function prints numbers in a countdown to 0 and then back up.
    # It demonstrates direct recursion by printing the number on the way down
    # and implicitly printing back up as the call stack unwinds.

    print(number)  # Print the current number. This happens both on the way down and as the stack unwinds.

    if number == 0:
        # BASE CASE: When 'number' reaches 0, we have reached the stopping condition.
        # At this point, the recursion stops and starts unwinding.
        print('Reached the base case.')
        return  # Exit the function, stopping further recursive calls.

    else:
        # RECURSIVE CASE: If 'number' is not 0, make a recursive call with 'number - 1'.
        # This call continues to decrease 'number' by 1 until the base case is reached.
        countDown(number - 1)
        # After reaching the base case, the call stack starts to unwind, printing 'number'
        # again as it returns to each previous call in reverse order.
#Example
countDown(5)
```

Recurrence Relation

- What is the actual running time of the recursive algorithm?
- Need to solve the recurrence relation
  - Find an explicit formula of the expression
  - Bound the recurrence by an expression that involves n

Methods for Solving Recurrences

- Iteration method
- Substitution method
- Recursion tree method
- Master method

Iteration Method

- The **iterative method** solves recurrence relations by unfolding them step by step until a pattern emerges.
- **Steps in the Iterative Method**
  - Start with the recurrence you want to solve.
  - Expand the recurrence by substituting it repeatedly until you reach the base case.
    - This means you will express the recurrence in terms of previous terms until you get to a point where the base case can be directly applied
  - As you unfold the recurrence, look for a pattern in the terms. This pattern will help you generalize the solution.
  - If applicable, sum the contributions of all the terms to derive a closed-form expression.
  - Conclude with the final expression in Big O notation.

Iteration Method - Example

- The recurrence relation for the **countDown** function is expressed as:

$$T(n) = T(n-1) + (1)$$

- T(n) is the time complexity for the **countDown** function when it is called with a parameter n.
- The 1 represents the constant time taken for printing the number and checking the base case. For simplicity we are using 1 instead of O(1)

Iteration Method - Example

- Expand the recurrence by substituting it repeatedly until you reach the base case
- For n: $T(n)=T(n-1)+(1)$
- For n−1: $T(n-1)=T(n-2)+(1)$

  Therefore, $T(n)= (T(n-2)+(1))+(1) =T(n-2)+2 \cdot (1)$
- For n−2: $T(n-2)=T(n-3)+(1)$

  Therefore, $T(n)=T(n-3)+3 \cdot (1)$

  …..

Iteration Method - Example

- Identify the Pattern: we can generalize the pattern to:

$$T(n)=T(n-k)+k \cdot (1)$$

- When we reach the base case where $n-k=0$, which implies $k=n$:

$$T(n)=T(0)+n \cdot (1)$$

- The base case $T(0)= 1$
- Substituting this back into our earlier equation: $T(n)=(1)+n \cdot (1)$
- Thus, we can express the final time complexity as:

$$T(n)=O(n)$$

Substitution Method

- The substitution method involves guessing a solution to the recurrence and then using mathematical induction to prove that the guess is correct.
- Steps in the Substitution Method:
  - Guess the form of the solution based on the recurrence
  - Use induction to prove that your guess is correct
  - Solve the base case to finalize the solution

Substitution Method - Example

- The recurrence relation for the **countDown** function is expressed as:

$$T(n)= T(n−1)+ O (1)$$

- **Make an Educated Guess:** Based on the structure of the recurrence relation, we will guess that:

$$T(n)=O(n)$$

Substitution Method

- **Prove the Guess by Induction:** We will use mathematical induction to prove our guess.
- **Base Case:** For $n=0$ $T(0)= O(1)$
- **Inductive Step**
  - Assume that our guess is true for $n=k$, i.e., $T(k) \leq c \cdot k$ for some constant $c>0$
  - Now, we need to prove that it holds for $n=k+1$: $T(k+1)=T(k)+O(1)$
  - Using the inductive hypothesis: $T(k+1) \leq c \cdot k+O(1)$
  - Since $O(1)$ is a constant, we can denote it as d, where d is a constant: $T(k+1) \leq c \cdot k+d$
  - Now we can write: $T(k+1) \leq c \cdot k+d \leq c \cdot (k+1)$

    for sufficiently large k, assuming c is large enough to accommodate the constant d

Substitution Method

- For instance, if d is relatively small compared to c, we can find a suitable constant c such that:

$$c \cdot k + d \leq c \cdot (k+1)$$

- Thus: $T(k+1) \leq c \cdot (k+1)$
- By the principle of mathematical induction, we have shown that:

$$T(n) \leq c \cdot n \text{ for all } n \geq 0$$

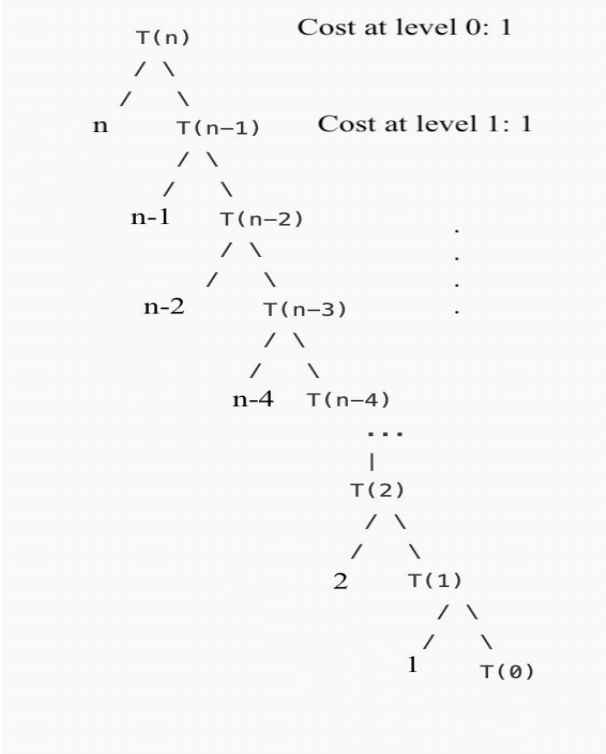
- Since we have shown that T(n) is bounded above by a linear function, we conclude that:

$$T(n) = O(n)$$

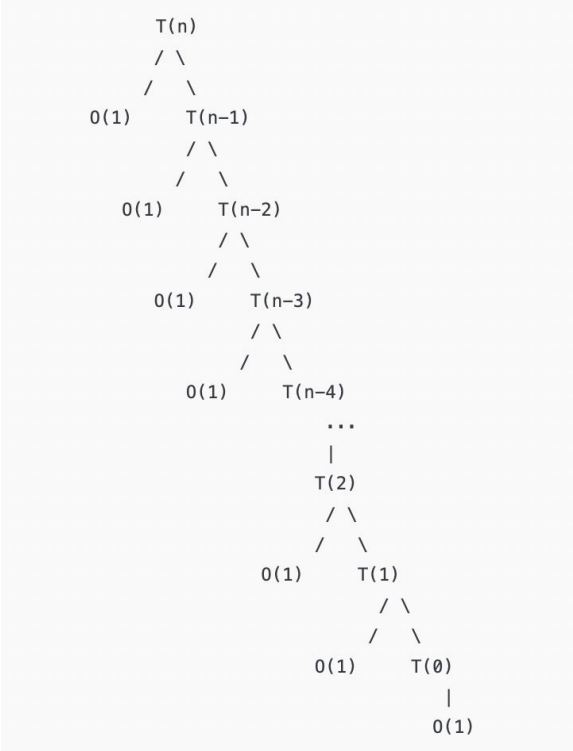
Recursion Tree Method

- The recursion tree method visualizes the recursion as a tree, where each node represents a function call. The method involves calculating the cost at each level and summing them.
- **Steps in Recursion Tree Method**:
  - Draw the recursion tree for the recurrence
  - Calculate the cost at each level of the tree
  - Sum the costs across all levels to get the total cost

Recursion Tree Method



Recursion Tree Method



Master Method

- "Cookbook" for solving recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where, $a \geq 1$, $b > 1$, and $f(n) > 0$

- T(n) is the total time complexity for an input size n.
- a≥1 is the number of subproblems.
- b>1 is the factor by which the problem size is reduced in each subproblem.
- f(n)is a function that describes the cost of the work done outside the recursive calls

Master Method

- **Idea:** compare f(n) with $n^{\log_b a}$
- f(n) is asymptotically smaller or larger than $n^{\log_b a}$ by a polynomial factor n^{ε}
- **Case 1:** If f(n) is dominated by $n^{\log_b a}$: ($f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$), then $T(n) = \Theta(n^{\log_b n})$
- **Case 2:** If $f(n) = \Theta(n^{\log_b a})$: $T(n) = \Theta(n^{\log_b a} \log n)$
- **Case 3:** If f(n) dominates $n^{\log_b a}$: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$, and if $af(n/b) \leq cf(n)$ for some $c < 1$ and all sufficiently large n, then:$T(n) = \Theta(f(n))$

Master Method

Why $n^{\log_b a}$?

$$T(n) = \underbrace{aT\left(\frac{n}{b}\right)}$$

$$\underbrace{a^2T\left(\frac{n}{b^2}\right)}$$

$$a^3T\left(\frac{n}{b^3}\right)$$

$$\vdots$$

$$T(n) = a^iT\left(\frac{n}{b^i}\right) \quad \forall i$$

Assume $n = b^k \Rightarrow k = \log_b n$

At the end of iteration i= k:

$$T(n) = a^{\log_b n}T\left(\frac{b^i}{b^i}\right) = a^{\log_b n}T(1) = \Theta\left(a^{\log_b n}\right) = \Theta\left(n^{\log_b a}\right)$$

Master Method

- The recurrence relation for **countDown function** is not in the standard form suitable for the Master Theorem, but we can still analyze it
- Since T(n) does not divide the problem size, it is more appropriate to evaluate it through iterative or substitution methods rather than the Master Theorem

UVA DATA SCIENCE

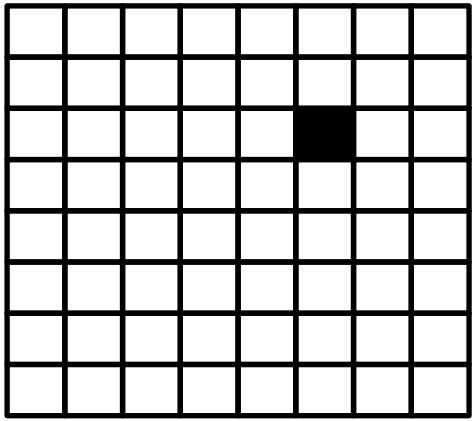
Foundation of Computer Science for Data Science

Advanced algorithm I: Divide and conquer

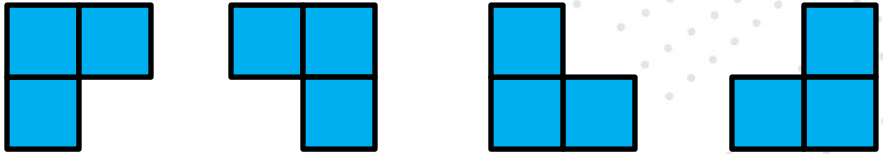
UVA DATA SCIENCE

Tiling Problem with One Defective Square

- The **Tiling Problem** is a combinatorial problem that involves determining the number of ways to cover a grid with specified tiles without overlaps and without leaving any gaps

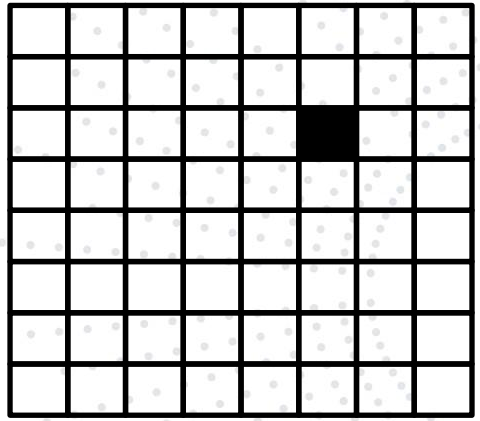


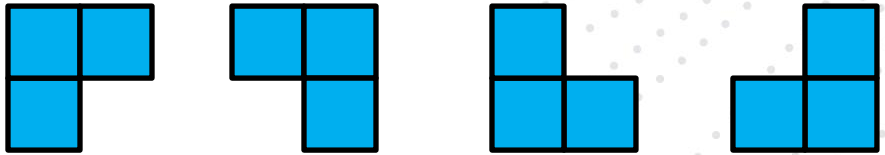
Can you cover an 8 × 8 grid with 1 square missing using "trominoes?"



Tiling Problem with One Defective Square

- Given a grid with size n*n, where $n = 2^k$ with one defective square
- How to cover the grid with L-shape tiles(trominoes)?
- Total number of squares in grid = $2^k * 2^k = 2^{2k}$
- Total number of non-defective squares = $2^{2k} - 1$





Trominoes

Tiling Problem with One Defective Square

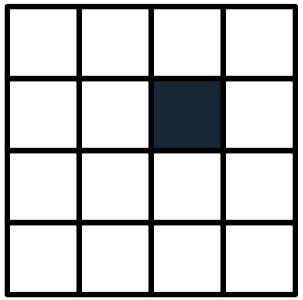
1*1

2*2

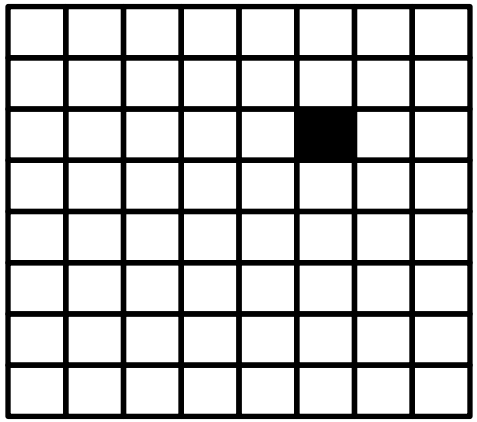
Tiling Problem with One Defective Square

How to solve larger grids?

4*4



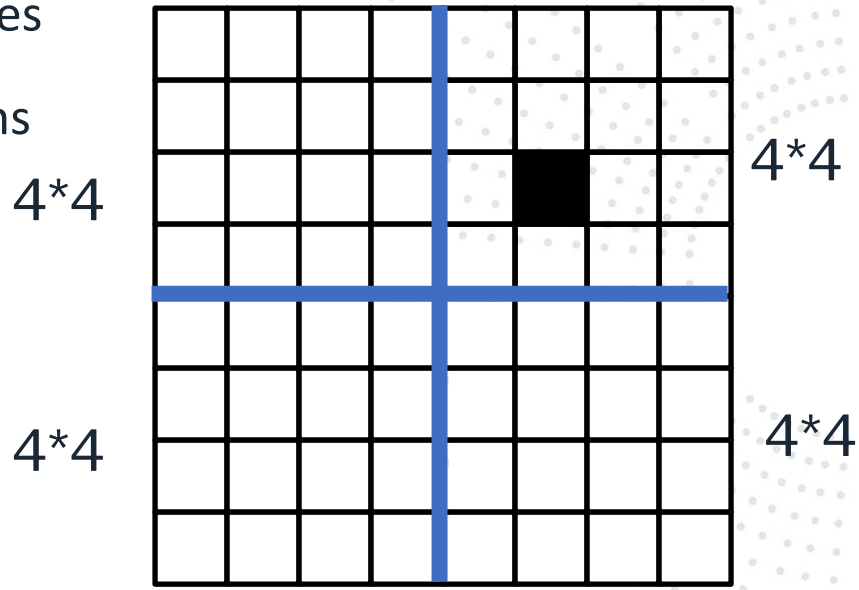
8*8



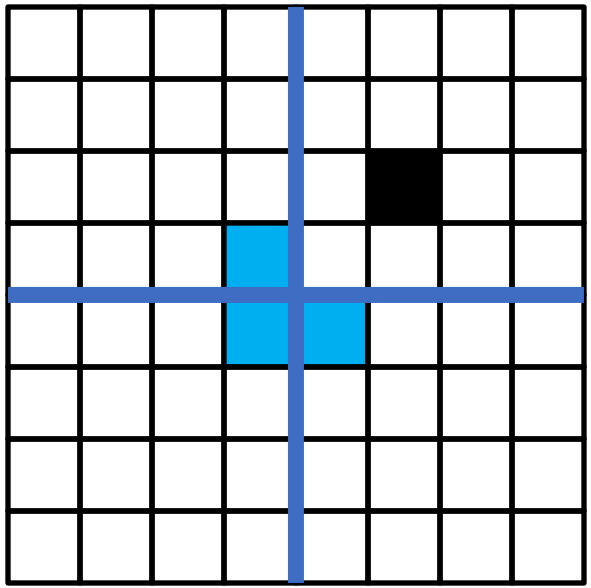
n*n gird

Tiling Problem with One Defective Square

- Divide the larger grids into smaller ones
- Combine solutions to smaller problems

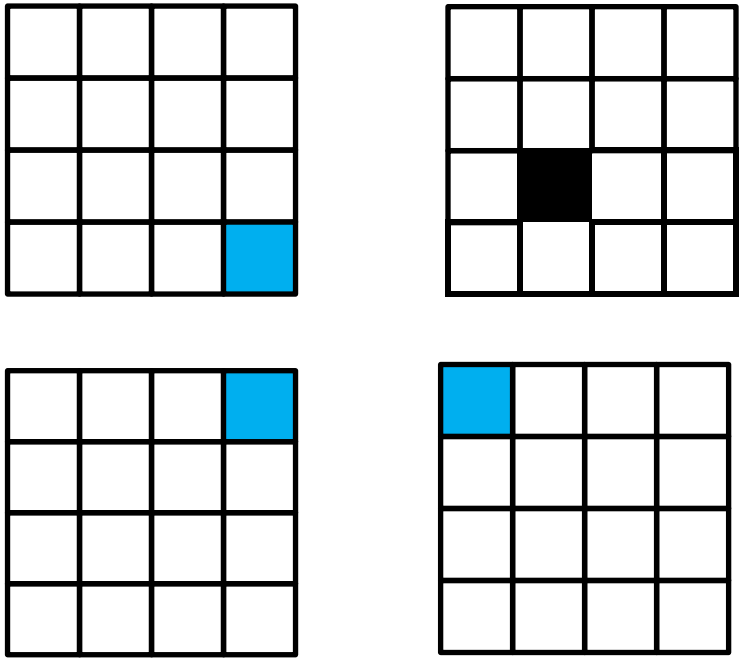


Tiling Problem with One Defective Square



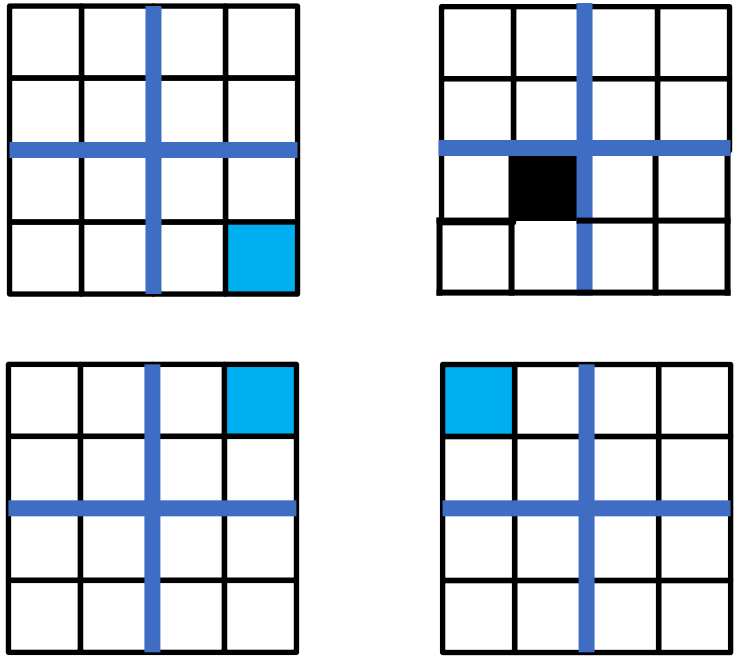
Place a tromino to occupy the three quadrants without the missing piece

Tiling Problem with One Defective Square



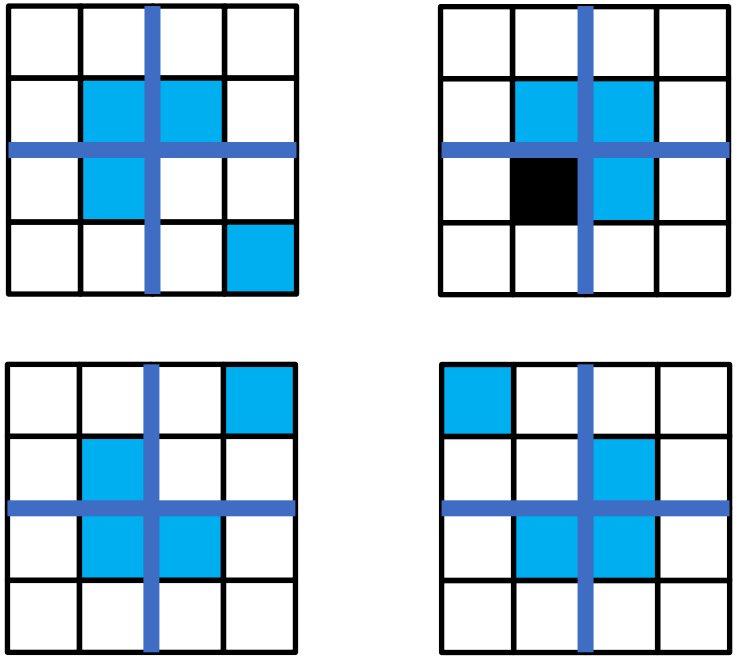
Observe: Each quadrant is now a smaller and similar subproblem!

Tiling Problem with One Defective Square



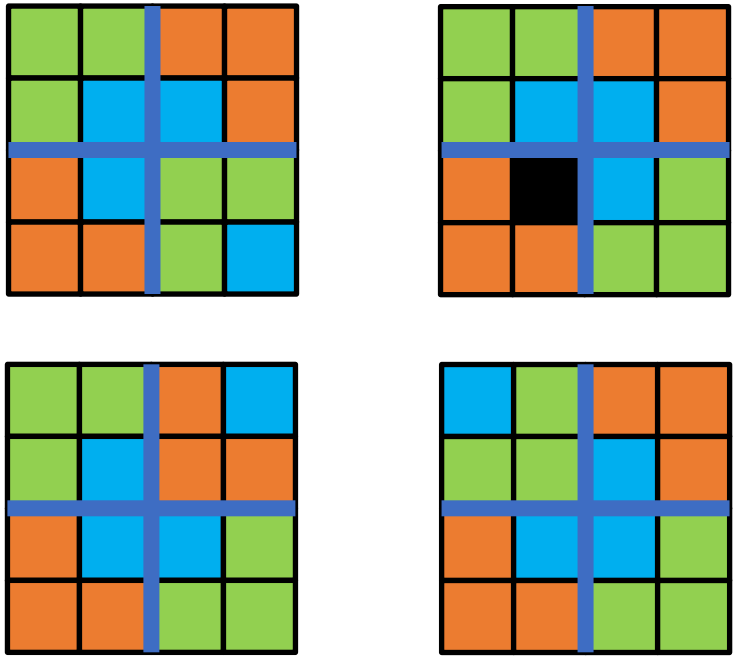
Solve **Recursively**

Tiling Problem with One Defective Square



Solve **Recursively** until we reach a base case, eg, 2x2

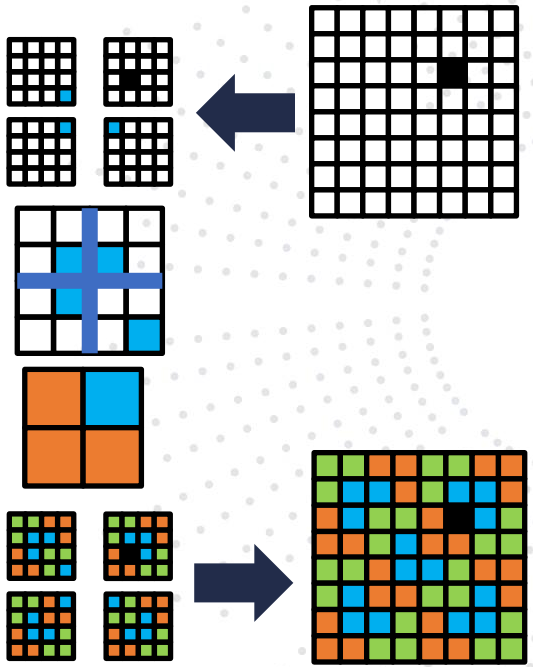
Tiling Problem with One Defective Square



A good Divide and Conquer Algorithmic Strategy!

Divide and Conquer

- **Divide:**
  - Break the problem into multiple subproblems, each smaller instances of the original
- **Conquer:**
  - If the sub-problems are “large”:
    - Solve each subproblem **recursively**
  - If the subproblems are “small”:
    - Solve directly: based problem
- **Combine:**
  - Merge solutions to subproblems to obtain solution for original problem



Analyze Divide and Conquer Algorithms

- Break into smaller **subproblems**
- Use **recurrence** relation to express recursive running time

$$T(n) = D(n) + \sum_{i=1}^{k} T(s_i) + C(n)$$

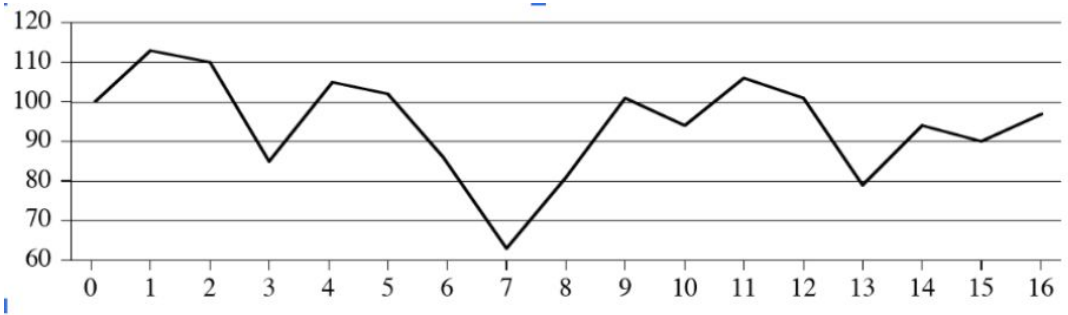
  - **T(n)**: The total time complexity for solving the problem of size n
  - **D(n)**: The time complexity to divide the problem into smaller subproblems.
  - **k**: The number of subproblems the original problem is divided into.
  - **s_1**: The size of the i-th subproblem.
  - **C(n)**: The time complexity to combine the results of the subproblems
  - **T(s_i):** represents the time complexity of solving a subproblem of size s_i
- Use **asymptotic** notation to simplify

Analyze Divide and Conquer Algorithms

- **Goal:** Reduce recurrence to ***closed form: T(n) = f(n)***
- We have T(n). But you what order class does that belong to?
  - $T(n) \in O(???)$

Example - The Maximum SubArray Problem

- Analyze best stock buying strategy



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

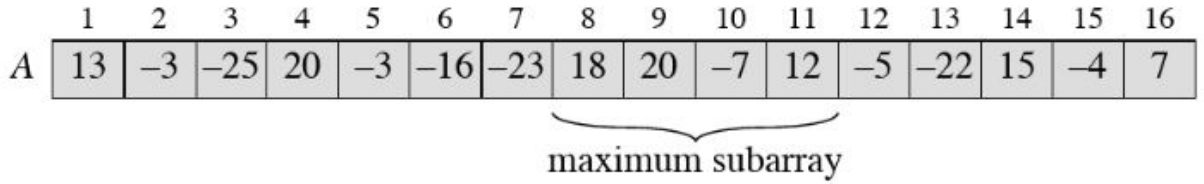
Objective: Find the best **buy** and **sell time** to maximize the profit

Finding the sub-array with maximum sum

sub array

Example - The Maximum SubArray Problem

- The goal of Maximum Sub-array Problem is to find the contiguous subarray within a one-dimensional array of numbers that has the largest sum.
- **Brute Force Approach**
  - Generate All Subarrays: Use two nested loops to generate all possible subarrays
  - Calculate the Sum: For each subarray, calculate the sum
  - Track the Maximum Sum: Keep track of the maximum sum encountered during the iteration



Example - The Maximum SubArray Problem

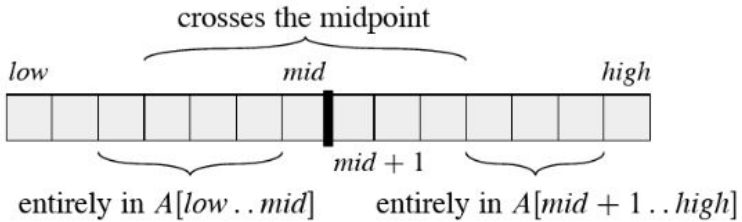
- Average-Case Time Complexity of original brute force approach is $O(N^3)$. This is because:
  - The outer loop runs N times.
  - The middle loop runs up to N times.
  - The innermost loop calculates the sum and runs up to N times.
- Average-Case Time Complexity of optimized the brute force approach is $O(N^2)$. This is because:
  - The outer loop runs N times
  - The inner loop, on average, runs about N/2 times, leading to an overall complexity of $O(N^2)$.

```
function maxSubarrayBruteForce(arr):
    max_sum = negative infinity
    N = length(arr)

    for i from 0 to N - 1:  // Outer Loop
        for j from i to N - 1:  // Inner Loop
            current_sum = 0
            for k from i to j:  // Innermost Loop
                current_sum += arr[k]
            max_sum = max(max_sum, current_sum)

    return max_sum
```

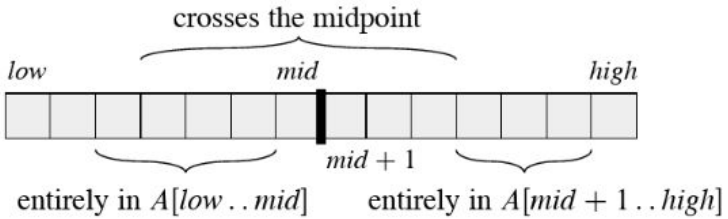
```
function maxSubarrayBruteForceOptimized(arr):
    max_sum = negative infinity
    N = length(arr)

    for i from 0 to N - 1:  // Outer Loop: starting index
        current_sum = 0  // Reset current_sum for each starting index
        for j from i to N - 1:  // Inner Loop: ending index
            current_sum += arr[j]  // Incrementally calculate the sum
            max_sum = max(max_sum, current_sum)  // Update max_sum if needed

    return max_sum
```

Example - The Maximum SubArray Problem

- The divide and conquer approach:
  - **Divide**: Split the array into two halves.
  - **Conquer**:
    - Recursively find the maximum subarray in the left half
    - Recursively find the maximum subarray in the right half
    - Find the maximum subarray that crosses the midpoint
- **Combine**: Return the maximum of the three values obtained from the above steps

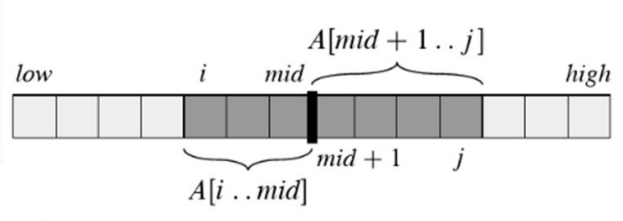


Example - The Maximum SubArray Problem




```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2      return (low, high, A[low])              // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4      (left-low, left-high, left-sum) =
            FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5      (right-low, right-high, right-sum) =
            FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6      (cross-low, cross-high, cross-sum) =
            FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7      if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8          return (left-low, left-high, left-sum)
 9      elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10          return (right-low, right-high, right-sum)
11      else return (cross-low, cross-high, cross-sum)
```

Example - The Maximum SubArray Problem




```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

Example - The Maximum SubArray Problem

- **What is the time complexity of max subarray problem?**

$$T(N) = 2T(N/2) + O(N)$$

Example - The Maximum SubArray Problem

$$T(N) = \begin{cases} O(1) & \text{if } N = 1 \\ T(N/2) + T(N/2) + O(N) & \text{if } N > 1 \end{cases}$$

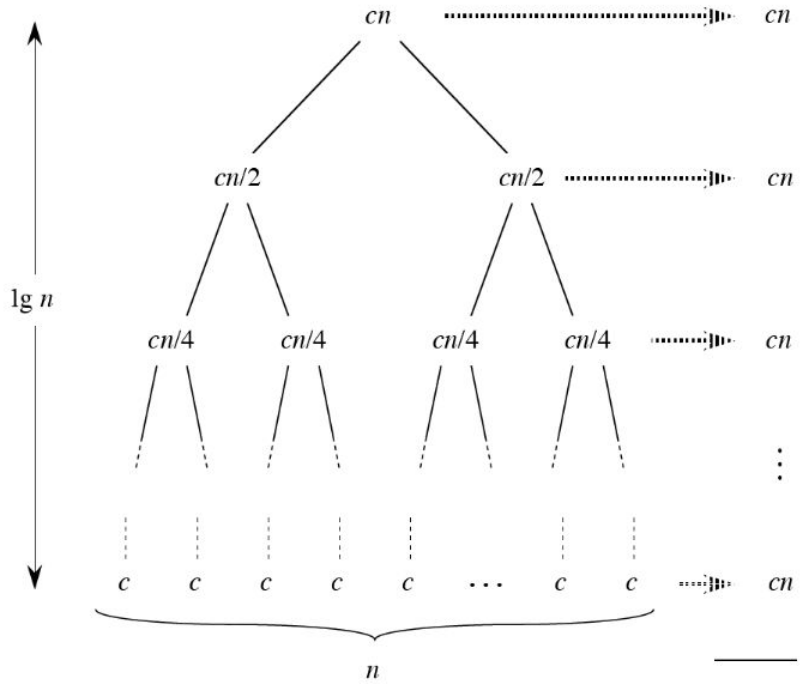
- **Base Case**
  - When N=1: The time complexity is O(1) because you simply return the single element, and no further calculations are needed.
- **Recursive Case**:
  - For N>1:
    - **Divide**: The array is split into two halves, each of size N/2 This leads to two recursive calls, T(N/2) for the left half and T(N/2) for the right half.
    - **Combine**: The O(N) term represents the time taken to find the maximum subarray that crosses the midpoint. This involves summing elements from the left and right halves, which takes linear time.

Example - The Maximum SubArray Problem

- **What is the time complexity**

$$T(N) = 2T(N/2) + O(N)$$

- Using recursion-tree method
  - Time complexity O(nlgn) -> (Why?)



Example - The Maximum SubArray Problem

- Using the Master Theorem, we can analyze the recurrence relation:
- **Recurrence Relation**: $T(n)=2T(n/2)+O(n)$
- **Parameters**:
  - $a=2$ (the number of subproblems),
  - $b=2$ (the factor by which the problem size is reduced),
  - $f(n)=O(n)$
- According to the Master Theorem:
- Compare $f(n)$ with $n^{\log_b a}$
  - Here, $\log_b^a = \log_2^2 = 1$
  - $n^{\log_2 2} = n \Rightarrow O(n)$
  - Since $f(n)$ is polynomially equal to $n^{\log_b a}$, we can apply Case 2 of the Master Theorem: $T(N)=O(n \log n)$

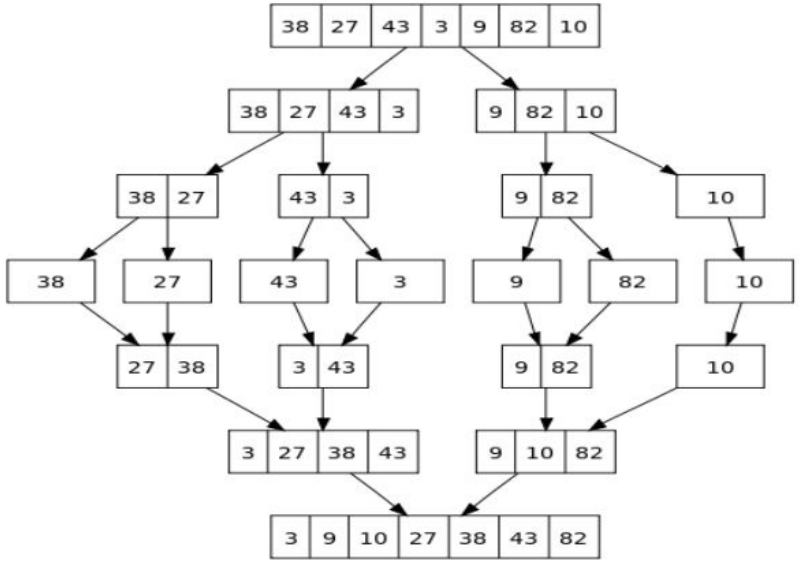
Recall Merge Sort

- **Divide:**
  - Break n-element list into two lists of n/2 elements
- **Conquer:**
  - If $n > 1$:
    - Sort each sub-list **recursively**
  - If $n = 1$:
    - List is already sorted (**base case**)
- **Combine:**
  - Merge together sorted sub-lists into one sorted list

Merge Sort

Pseudocode

```
def mergesort(list, first, last):
    if first < last:
        mid = (first+last)/2
        mergesort(list, first, mid)
        mergesort(list, mid+1, last)
        merge(list, first, mid, last)    // merge 2 sorted halves
    return
```



Time Complexity Analysis

- **ICA:** Analyze the complexity in average case for merge sort using the recursion tree and master theorem?
- **ICA:** Find the recurrence relation for binary search and solve it using master theorem?