CS 2009
Design and Analysis of Algorithms

Waheed Ahmed
Email : waheedahmed@nu.edu.pk

Methods for Solving Recurrences

Revision

- **Recursion tree method**
- **Iteration method** or (**Iterative Substitution Method**)
- **Master method**
- **Substitution method or (Substitution Guess and Test method)**

SUBSTITUTION METHOD

Algorithm, Proof of Correctness, Runtime

SUBSTITUTION METHOD

1. Guess the form of the solution or Guess what the answer is (iterative substitution: iteratively apply the recurrence equation to itself to find a possible pattern)

2. Prove your guess is correct (using mathematical induction)

    - If proven ok
    - else retry different solution

Solving Recurrences by Substitution: Guess-and-Test

$T(n) = 2T(n/2) + n$

Guess (#1) — $T(n) = O(n)$

Inductive Hypothesis — $T(n) \le cn$ for some constant $c>0$

Inductive Step — $T(n/2) \le cn/2$

$T(n) = 2T(n/2) + n$

$T(n) \le 2 \cdot c(n/2) + n$

$T(n) \le cn + n$

$T(n) \le (c+1)\,n$

Our guess was wrong!!

no choice of c could ever make $(c+1)\,n \le cn$!

Solving Recurrences by Substitution: G #2

	$T(n) = 2T(n/2) + n$
Guess (#2)	**$T(n) = O(n^2)$**
IH	**$T(n) <= cn^2$** for some constant $c>0$
Inductive Step	$T(n/2) <= c.\frac{n^2}{4}$

$$T(n) = 2T(n/2) + n$$

$$T(n) \le 2 \cdot (\frac{cn^2}{4}) + n$$

$$T(n) \le \frac{cn^2}{2} + n$$

$$\mathbf{T(n) \le \frac{cn2}{2} + n \le cn^2}$$

Works for all n as long as c>=2 !!

Solving Recurrences by Substitution: G #3

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

Guess (#3) **T(n) = O(nlogn)**

IH **T(n) <= cnlogn** for some constant c>0

Inductive Step $T\left(\frac{n}{2}\right) \leq c\,\frac{n}{2}\,\log(\frac{n}{2})$

$$T(n) = 2T(n/2) + n$$

$$T(n) \leq 2 \cdot c\,\frac{n}{2}\,\log(\frac{n}{2}) + n$$

$$T(n) \leq cn\,(logn - \log 2) + n$$

$$T(n) \leq cn\log n - cn + n$$

Thus $T(n) \leq cn\log n - cn + n$ <= cnlogn

Works for all n as long as c>=1 !!

Guess and Test Method by Substitution: Ex #2, G # 1

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

Guess (# 1) $\quad T(n) = O(n \log n)$

(Inductive Hypothesis): $\quad \mathrm{T(n)} \leq \mathrm{c\ n\ log\ n} \quad \text{for c} > 0$

Inductive step, Assume $\quad \mathrm{T}\left(\frac{\mathrm{n}}{2}\right) \leq \mathrm{c}\ \frac{\mathrm{n}}{2}\ \log(\frac{\mathrm{n}}{2})$

$$\mathrm{T(n)} = 2\mathrm{T}\left(\frac{\mathrm{n}}{2}\right) + \mathrm{bnlogn}$$

$$\mathrm{T(n)} \leq 2 \cdot \mathrm{c}\ \frac{\mathrm{n}}{2}\ \log(\frac{\mathrm{n}}{2}) + \mathrm{bnlogn}$$

$$\mathrm{T(n)} \leq cn\ (logn - \log 2) + \mathrm{bnlogn}$$

$$\mathrm{T(n)} \leq cn\log n - cn + \mathrm{bnlogn}$$

$$\mathrm{T(n)} \leq (c+b)n\log n - cn$$

Wrong: we cannot make this last line be less than cn log n

Guess and Test Method by Substitution: Ex #2, G # 2

$$T(n) = \begin{cases} b & \text{if } n < 2 \\ 2T(n/2) + bn\log n & \text{if } n \geq 2 \end{cases}$$

Guess (# 1) $\quad$ $T(n) = O(n \log^2 n)$

(Inductive Hypothesis): $\quad$ $T(n) \leq c\, n \log^2 n \quad \text{for } c > 0$

Inductive step, Assume $\quad$ $T\left(\frac{n}{2}\right) \leq c\,\frac{n}{2} \log^2\left(\frac{n}{2}\right)$

if c > b.
So, T(n) is O(n $\log^2$ n).
In general, to use this method, you need to have a good guess and you need to be good at induction proofs.

$$T(n) = 2T\left(\frac{n}{2}\right) + \text{bnlogn}$$

$$T(n) \leq 2 \cdot c\,\frac{n}{2} \log^2\left(\frac{n}{2}\right) + \text{bnlogn}$$

$$T(n) \leq cn\,(logn - \log 2)^2 + \text{bnlogn}$$

$$T(n) \leq cn\log^2 n - 2cnlogn + cn + \text{bnlogn}$$

$$T(n) \leq cn\log^2 n + (b - 2c)nlogn + cn$$

Home Work: Apply **Substitution Guess and Test method** by assuming given guesses to find correct one.

$T(n) = T(n/2) + n^2$

Guess 1 : $\boldsymbol{T(n)=O(n)}$, Guess 2 : $\boldsymbol{T(n)=O(n^2)}$

$T(n) = 4\,T(n/4) + n$

Guess 1 : $\boldsymbol{T(n) = O(n)}$, Guess 2 : $\boldsymbol{T(n)=O(n}$ log n)

$T(n) = T(n/2) + n$

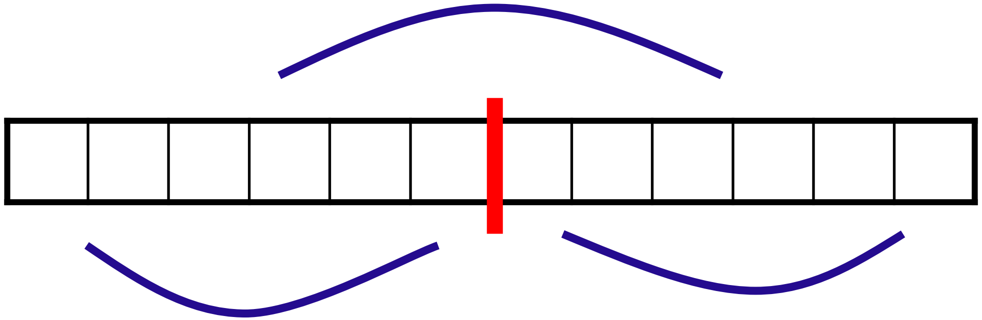
Guess 1 : $\boldsymbol{T(n)=O(n)}$, Guess 2 : $\boldsymbol{T(n)=O(n^2)}$

The Maximum Subarray Sum Problem

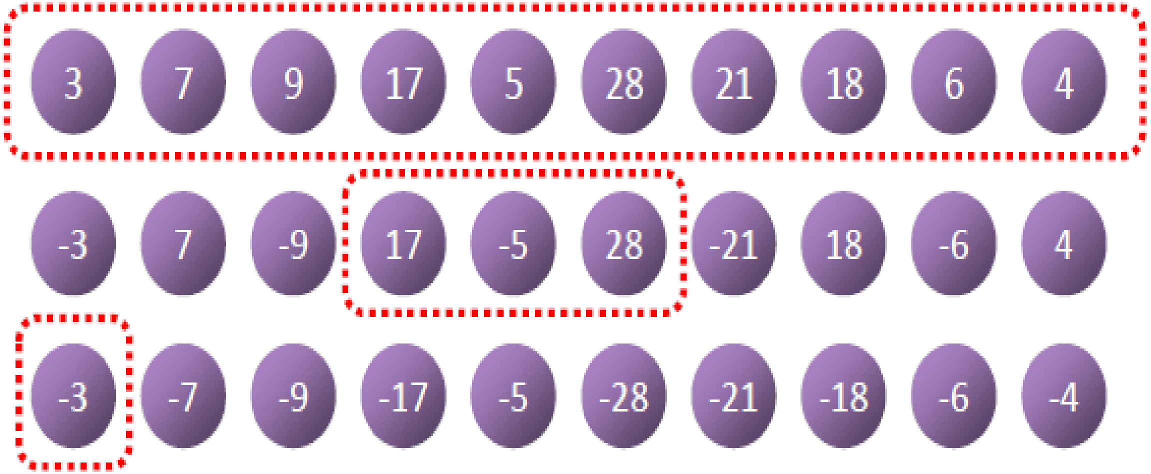
The Maximum Subarray Problem

- *Def:* The maximum subarray problem is the task of finding the largest possible sum of a contiguous subarray, within a given one-dimensional array A[1…n] of numbers.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
13	*-3*	*-25*	*20*	*-3*	*-16*	*-23*	*18*	*20*	*-7*	*12*	*-5*	*-22*	*15*	*-4*	*7*



The Maximum Subarray Problem

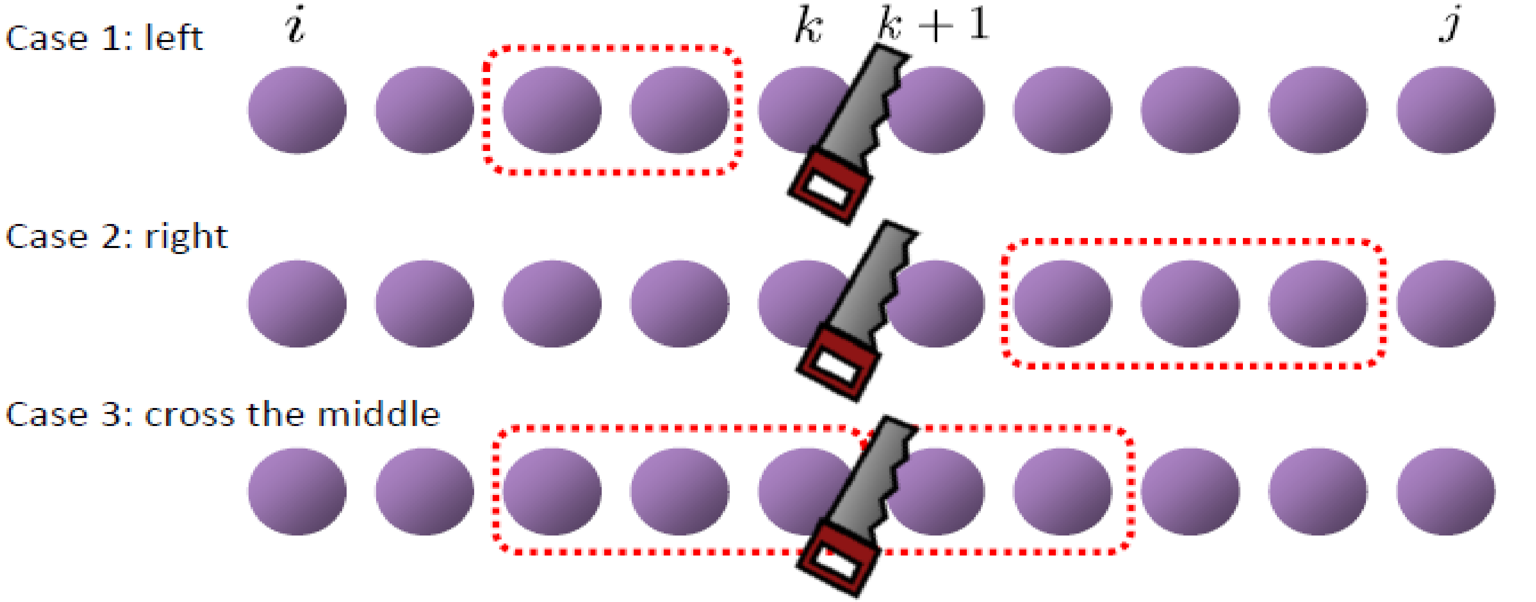


Divide-and-Conquer

- **Base Case (n = 1)**
  - Return itself (maximum subarray)
- **Recursive Case** (n > 1)
  - Divide array into two subarrays.
  - Find maximum sub array recursively
  - **Merge** the results.

Where is Result?

- The maximum subarray for any input must be in one of following cases:



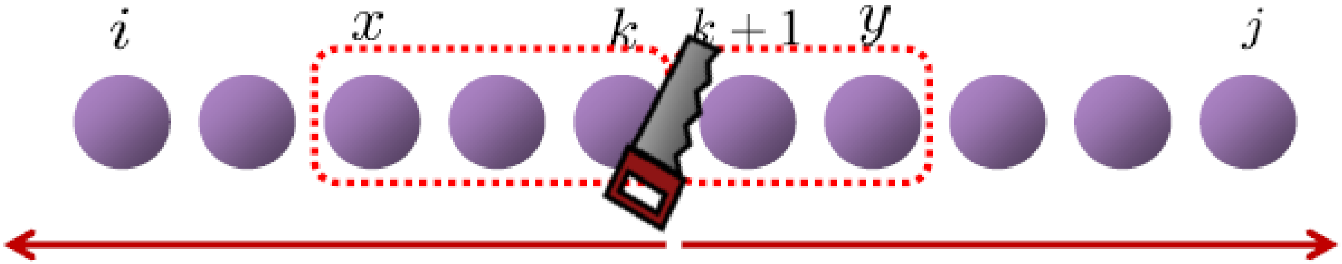
Case 1: `MaxSub(A, i, j) = MaxSub(A, i, k)`

Case 2: `MaxSub(A, i, j) = MaxSub(A, k+1, j)`

Case 3: `MaxSub(A, i, j)` cannot be expressed using `MaxSub`!

Case 3: Cross the Middle

- Goal: find the maximum subarray that crosses the middle



(1) Start from the middle to find the left maximum subarray

(2) Start from the middle to find the right maximum subarray

The solution of Case 3 is the combination of (1) and (2)

- Observation
  - The sum of $A[x \ldots k]$ must be the maximum among $A[i \ldots k]$ (left: $i \leq k$)
  - The sum of $A[k+1 \ldots y]$ must be the maximum among $A[k+1 \ldots j]$ (right: $j > k$)
  - Solvable in linear time → $\Theta(n)$

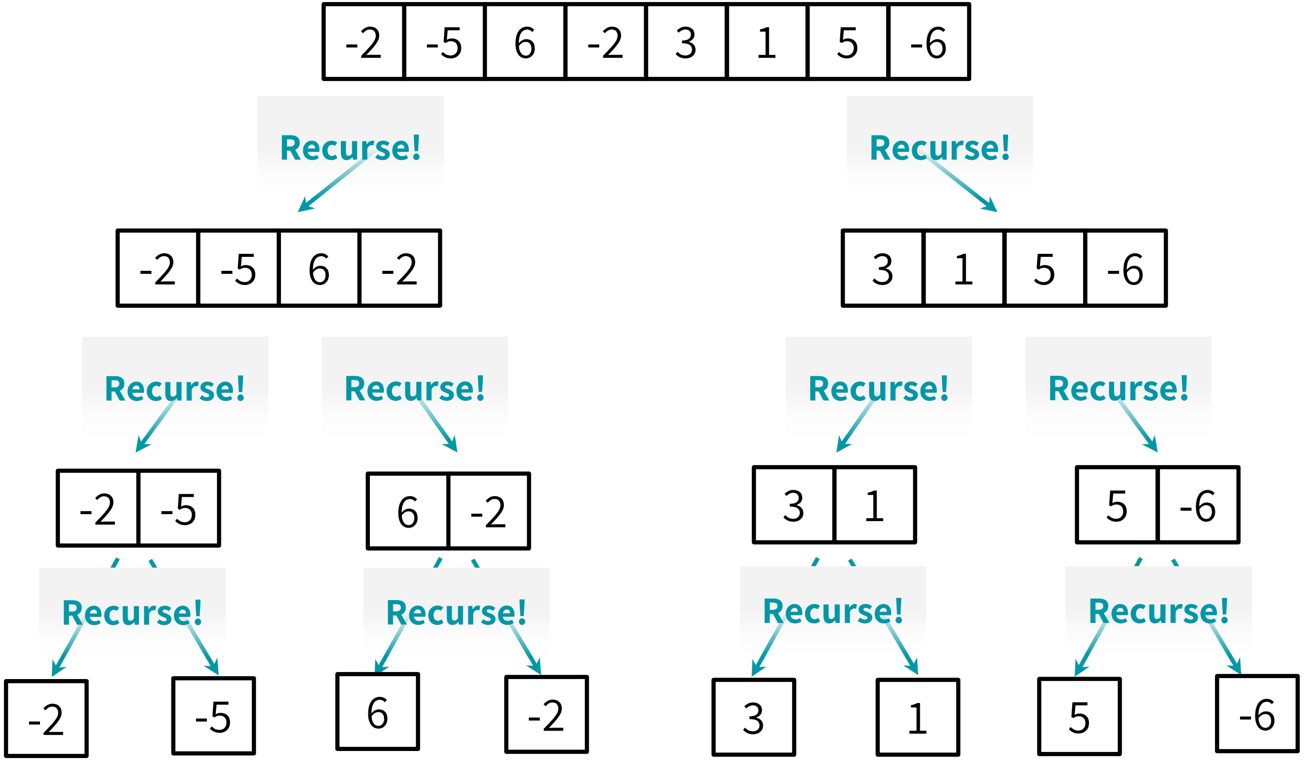
The Maximum Subarray Problem - Example

-2	-5	6	-2	3	1	5	-6

What is maximum subarray sum of this array

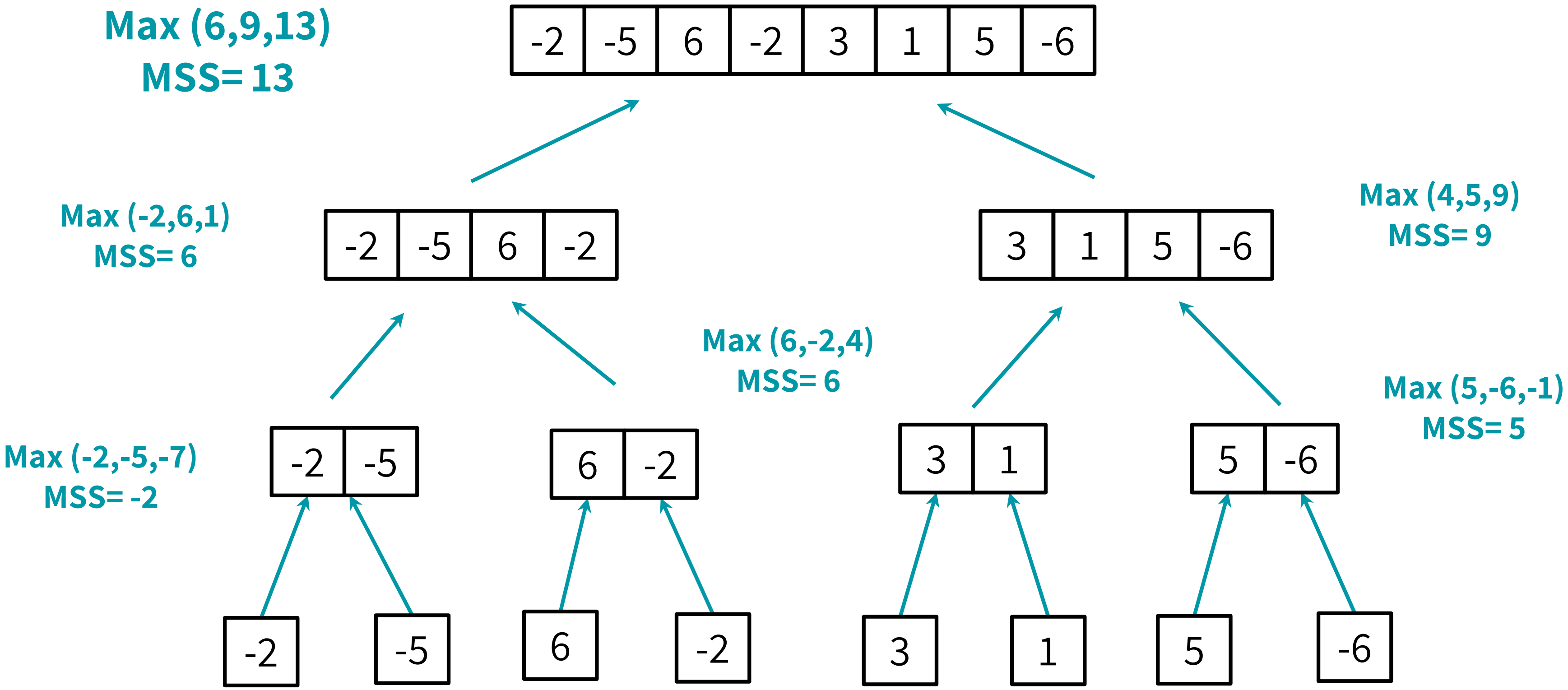
Maximum subarray sum is
6-2+3+1+5 = 13

MaxSubArray: RECURSIVE CALLS



This is where we hit our base case!

MaxSubArray: RECURSIVE CALLS



Divide and Conquer Solution

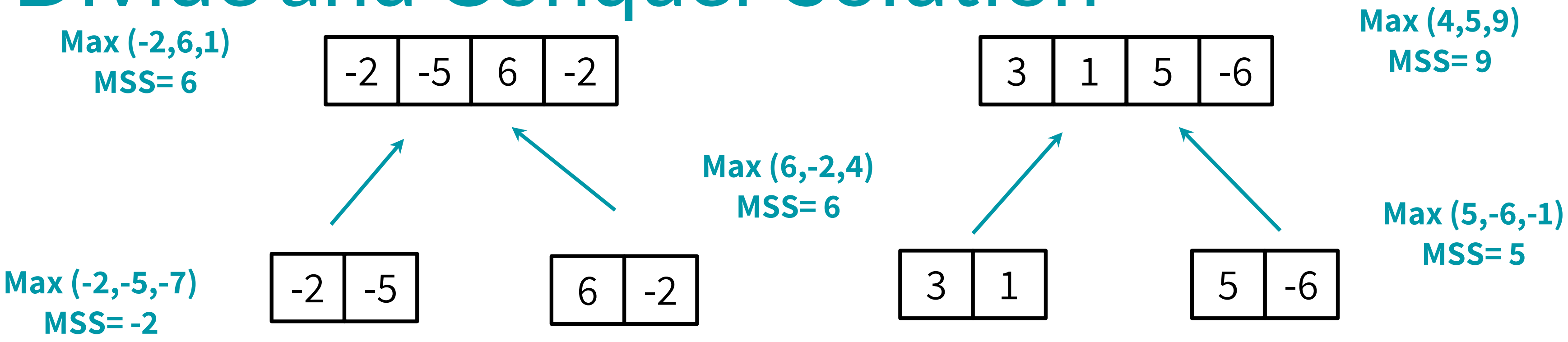
```
MaxSubarray(A, i, j)
  if i == j // base case
    return (i, j, A[i])
  else // recursive case
    k = floor((i + j) / 2)
    (l_low, l_high, l_sum) = MaxSubarray(A, i, k)
    (r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)
    (c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)

  if l_sum >= r_sum and l_sum >= c_sum // case 1
    return (l_low, l_high, l_sum)
  else if r_sum >= l_sum and r_sum >= c_sum // case 2
    return (r_low, r_high, r_sum)
  else // case 3
    return (c_low, c_high, c_sum)
```

Divide

Conquer

Combine

Divide and Conquer Solution



Divide

```
(l_low, l_high, l_sum) = MaxSubarray(A, i, k)
(r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)
(c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)
```

Conquer

```
if l_sum >= r_sum and l_sum >= c_sum // case 1
  return (l_low, l_high, l_sum)
else if r_sum >= l_sum and r_sum >= c_sum // case 2
  return (r_low, r_high, r_sum)
else // case 3
  return (c_low, c_high, c_sum)
```

Combine

Divide and Conquer Solution

Max (6,9,13)
MSS= 13

-2	-5	6	-2	3	1	5	-6

Max (-2,6,1)
MSS= 6

-2	-5	6	-2

3	1	5	-6

Max (4,5,9)
MSS= 9

Divide

```
(l_low, l_high, l_sum) = MaxSubarray(A, i, k)
(r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)
(c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)
```

Conquer

```
if l_sum >= r_sum and l_sum >= c_sum // case 1
  return (l_low, l_high, l_sum)
else if r_sum >= l_sum and r_sum >= c_sum // case 2
  return (r_low, r_high, r_sum)
else // case 3
  return (c_low, c_high, c_sum)
```

Combine

Divide and Conquer Solution

```
MaxCrossSubarray(A, i, k, j)
  left_sum = -∞
  sum=0
  for p = k downto i
    sum = sum + A[p]
    if sum > left_sum
      left_sum = sum
      max_left = p

  right_sum = -∞
  sum=0
  for q = k+1 to j
    sum = sum + A[q]
    if sum > right_sum
      right_sum = sum
      max_right = q
  return (max_left, max_right, left_sum + right_sum)
```

$O(k-i+1)$

$O(j-k)$

$= O(j-i+1)$

Divide and Conquer Solution

```
MaxSubarray(A, i, j)
  if i == j // base case                                          O(1)
    return (i, j, A[i])
  else // recursive case
    k = floor((i + j) / 2)                                        T(k - i + 1)
    (l_low, l_high, l_sum) = MaxSubarray(A, i, k)                 T(j - k)
    (r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)
    (c_low, c_high, c_sum) = MaxCrossSubarray(A, i, k, j)
                                                                  O(j - i + 1)
  if l_sum >= r_sum and l_sum >= c_sum // case 1                  O(1)
    return (l_low, l_high, l_sum)
  else if r_sum >= l_sum and r_sum >= c_sum // case 2             O(1)
    return (r_low, r_high, r_sum)
  else // case 3                                                  O(1)
    return (c_low, c_high, c_sum)
```

Divide and Conquer Solution

1. Divide

- Divide a list of size *n* into 2 subarrays of size *n/2* $\boxed{\Theta(1)}$

2. Conquer

- Recursive case $(n > 1)$ $\boxed{T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)}$
  - find **MaxSub** for each subarrays
- Base case $(n = 1)$ $\boxed{\Theta(1)}$
  - Return itself
- Find **MaxCrossSub** for the original list $\boxed{\Theta(n)}$

3. Combine

- Pick the subarray with the maximum sum among 3 subarrays $\boxed{\Theta(1)}$

- $T(n)$ = time for running `MaxSubarray(A, i, j)` with $j - i + 1 = n$

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) & \text{if } n \geq 2 \end{cases}$$

Binary Search

Binary Search Problem

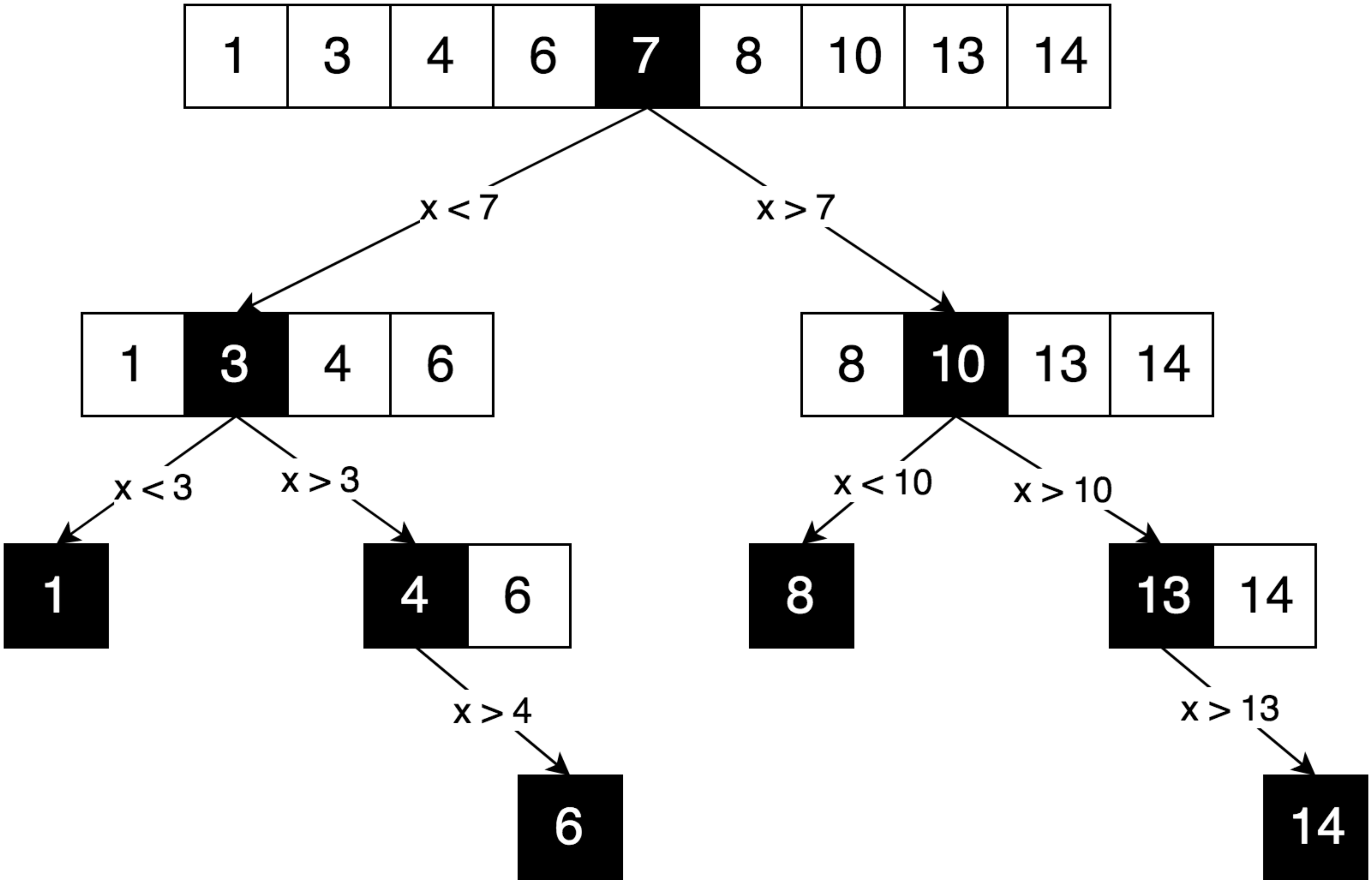
- Given a sorted array of integers and a target value, find out if target exists in the array or not
- Input: arr[] = {3,4,6,7}, target = 4
- Output: Target is in index 2
- Trivial Solution: Linear Search O(n) Complexity

Binary Search Problem

- Design O(logn) complexity algorithm
- Divide & Conquer

```
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? cout << "Element is not present in array"
                   : cout << "Element is present at index " << result;
    return 0;
}
```

```
// C program to implement recursive Binary Search
#include <stdio.h>

// A recursive binary search function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}
```


1 3 4 6 7 8 10 13 14
x < 7
x > 7
1 3 4 6
8 10 13 14
x < 3
x > 3
x < 10
x > 10
1
4 6
8
13 14
x > 4
x > 13
6
14

Binary Search Loop Invariant

- Three conditions
  - Array arr is sorted in ascending order
  - l <= r
  - x belong to arr [l…..r]

Binary Search Loop Invariant

- Use loop invariant that the code is correct

- Initialization: The loop invariant has three parts

1. Array is sorted due to precondition of the method
2. Since arr.length is at least 1, thus l<=r
3. x is in arr b/c it is whole array and precondition guarantees that x is in array

Binary Search Loop Invariant

- Maintenance: The loop invariant has three parts

1. Array arr is never changed so Case 1 is always true i.e. arr is sorted
2. Let l’ and r’ are the values of l and r at the end of 1^{st} iteration, then we need l’<r’ and x belongs to arr[l’…..r’]
3. Let m be the average of l and r, thus x belongs to arr[l…m] or arr [m+1….j]
4. Case k belongs to arr[1….m]

   must have x<=a[m] and thus if condition is true, then r’=m, l =1, this l’<r’ and since x belongs to arr[1…m], by assumption its belong to arr[l’…..j’]

Binary Search Loop Invariant

- Maintenance: The loop invariant has three parts

5. Case k does not belong to arr[1….m]

   must have x>a[m] and thus if condition is true, then r'=r, l =m+1, this l'<r' and since x belongs to arr[m+1…..r], by assumption its belong to arr[l'…..j']

   For the algorithm to be correct, arr[1] = x and happens only when l = r

- Termination: The value r−l is guaranteed to be non-negative. Because integer division rounds down, it gets smaller on every loop iteration. Therefore the loop eventually terminates

- More Detail: https://www.cs.cornell.edu/courses/cs2112/2015fa/lectures/lec_loopinv/

Time Complexity

- $T(n) = 1 + T(n/2)$

- $O(n)$