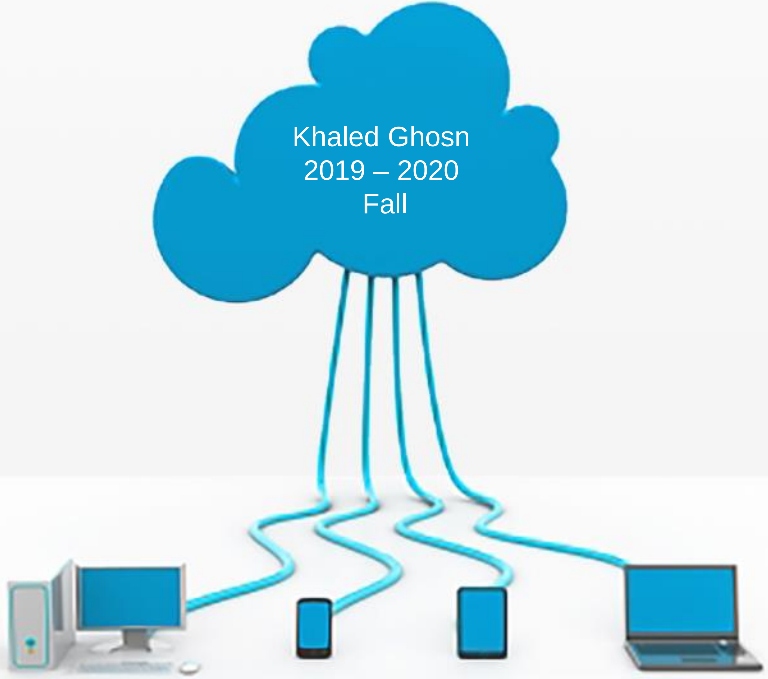
ARTS, SCIENCES & TECHNOLOGY UNIVERSITY IN LEBANON

Recursion

Analyzing, Designing, Examples



Khaled Ghosn
2019 – 2020
Fall

Recursion

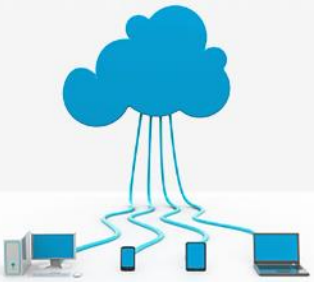
What is recursion?

- Recursion is a technique by which a method makes one or more calls to itself during execution

- A **recursive algorithm** is one expressed in terms of itself. In other words, at least one step of a recursive algorithm is a “call” to itself

- In computing, recursion provides an elegant and powerful alternative for performing repetitive tasks

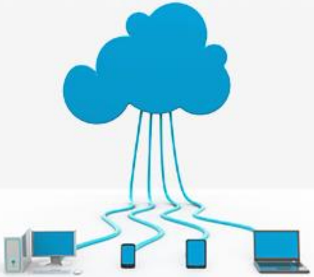
- In Java, a **recursive method** is one that calls itself



Recursion

When should recursion be used?

- Sometimes an algorithm can be expressed using either **iteration** or **recursion**. The recursive version tends to be:
  - \+ more elegant and easier to understand
  - – less efficient (extra calls consume time and space)
- Sometimes an algorithm can be expressed **only** using recursion



Recursion

Illustrative Examples

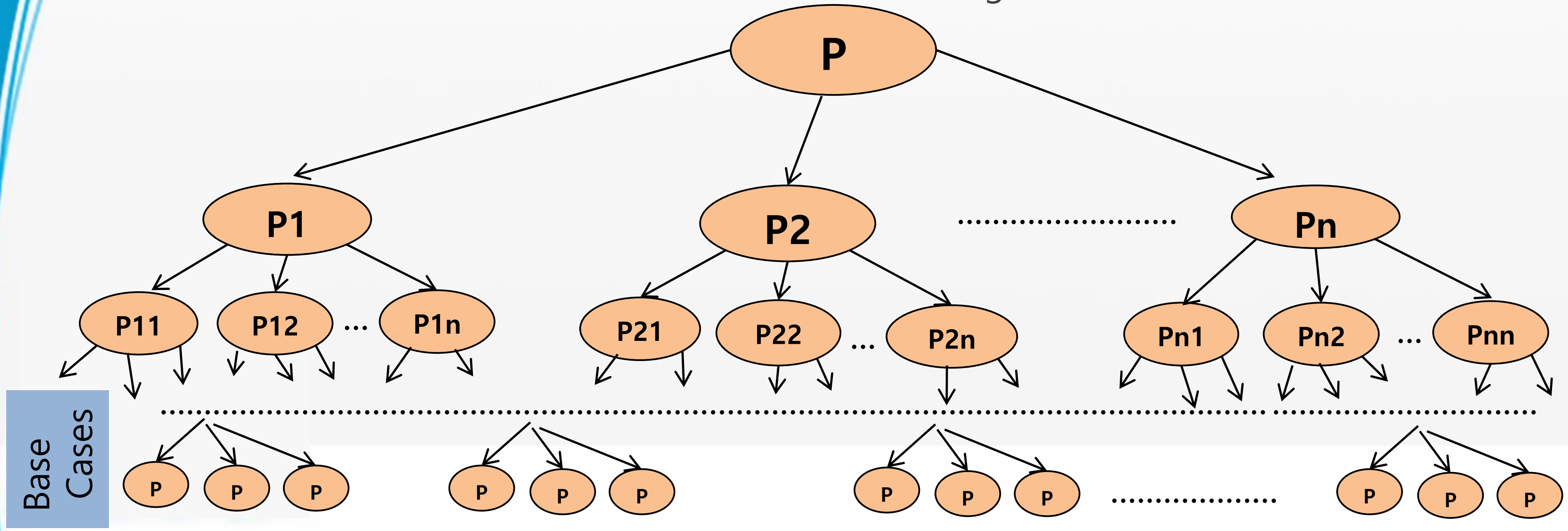
- Factorial Function
- English Ruler
- Binary Search
- File Systems
- …

Recursion

Divide & Conquer Strategy

Very important strategy in computer science:

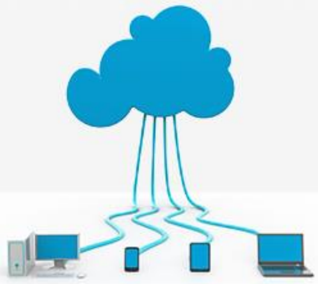
1. Divide problem into smaller parts
2. Independently solve the parts
3. Combine these solutions to get overall solution



Recursion

Divide & Conquer Strategy

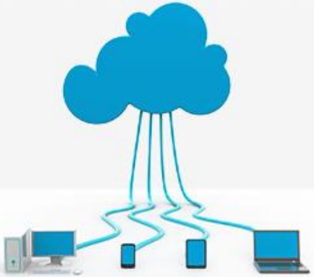
```
/* Solve a problem P */
Solve(P){
    /* Base case(s) */
    if P is a base case problem
        return the solution immediately

  /* Divide P into P1, P2, ..Pn each of smaller scale (n>=2) */
  /* Solve subproblems recursively */
    S1 = Solve(P1); /* Solve P1 recursively to obtain S1 */
    S2 = Solve(P2); /* Solve P2 recursively to obtain S2 */
    ........
    Sn = Solve(Pn); /* Solve Pn recursively to obtain Sn */

     /* Merge the solutions to subproblems */
     /* to get the solution to the original big problem */
    S = Merge(S1, S2, …, Sn);

     /* Return the solution */
     return S;
} //end-Solve
```



Recursion

Design Steps

1. Identify the (original) problem
2. Identify sub-problem(s) that are **simpler than** and **similar to** the original problem
3. Determine the parameter(s) that distinguishes the sub-problem(s) from the original problem
4. Identify the **simplest** problem instance (the base case) that will result at the end of **repeated** problem - sub-problem decomposition step
5. Identify the conditions (on the parameters) that will put you in a position to solve the **simplest** instance (the base case) of the problem.
6. Use recursion to solve the sub-problem(s)

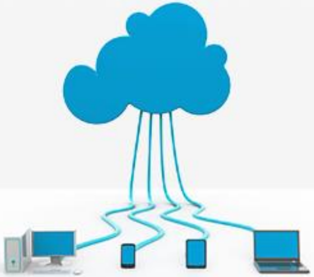


Recursion

Design Guidelines

A recursive method shall:

1. Take at least one parameter: the parameter(s) distinguishes an instance of the original problem from the sub-problem(s)
2. Include an if-(else) statement with one or more conditions that check whether the current problem instance to be solved is the simplest or a simpler instance.
3. The code in the if-block handles the case when the current problem instance is simpler, but not the simplest. The block shall include recursive call(s).
4. The code in the else-block handles the case when the current problem instance is the simplest (the base case), and no recursive call shall be made here.



Recursion

Design Guidelines

A recursive method shall:

5. The role of the if -and else- block can be switched, as long as you use them consistently.
6. Avoid access to / use of non-local variables. Use local variables whenever possible
7. Use non-void return properly. Result returned (via the return value) by a recursive call should be considered as the solution to a subtask. Typically this value is to be "combined" with other available data to incrementally build the final solution (to the original problem / task).

Recursion

When does recursion work?

Given a recursive algorithm, how can we sure that it terminates?

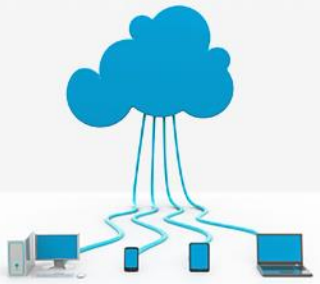
- The algorithm must have:
  - one or more “easy” cases
  - one or more “hard” cases
- In an “easy” case, the algorithm must give a direct answer without calling itself
- In a “hard” case, the algorithm may call itself, but only to deal with an “easier” case

Recursion

e.g. **Factorial Function**

As an example, we could implement the factorial function recursively:

```
int factorial( int n ) {
    if ( n <= 1 ) {
        return 1;

    return n * factorial( n – 1 );
    }
}
```



Recursion

Analysis (for Factorial Function)

Thus, we may analyze the run time of this function as follows:

$$\mathrm{T}_{!}(n) = \begin{cases} \boldsymbol{\Theta}(1) & n \le 1 \\ \mathrm{T}_{!}(n-1) + \boldsymbol{\Theta}(1) & n > 1 \end{cases}$$

We don't have to worry about the time of the conditional (***O*** (1)) nor is there a probability involved with the conditional statement

Recursion

Analysis

The analysis of the run time of this function yields a recurrence relation:

$$T_!(n) = T_!(n-1) + \mathbf{O}(1) \qquad T_!(1) = \mathbf{O}(1)$$

– Replace each symbol with a representative function:

$$T_!(n) = T_!(n-1) + 1 \qquad T_!(1) = 1$$

Recursion

Further examples of recursion

- Further examples of recursion
  - Linear recursion
    - Summing the Elements of an Array Recursively
    - Reversing a Sequence with Recursion
    - Recursive Algorithms for Computing Powers
  - Binary recursion
    - binarySum
  - Multiple recursion

Binary Search

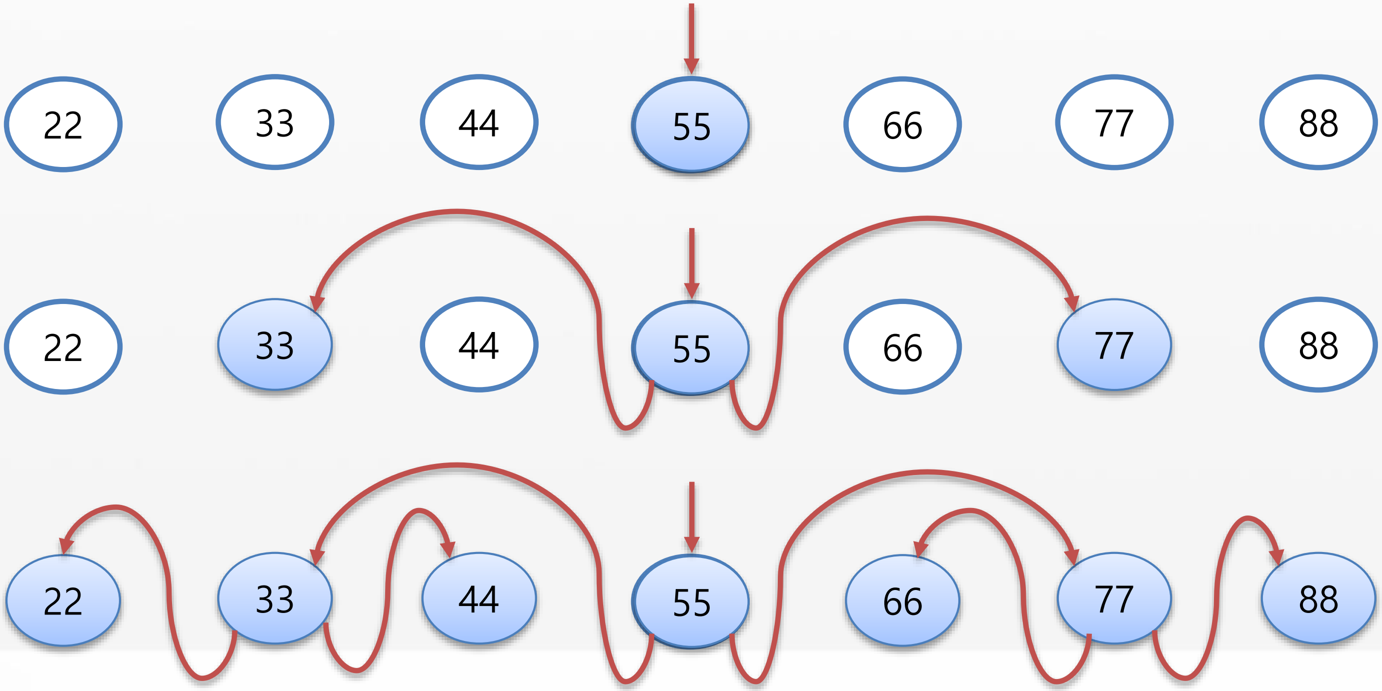
Introduction

- Problem: You are given a **sorted** array of integers, and you are searching for a key
  - Linear Search – T(n) = 3n+2 (Worst case)
  - Can we do better?
  - E.g. Search for 55 in the **sorted** array below

3	8	10	11	20	50	55	60	65	70	72	90	91	94	96	99
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Binary Search

Binary search



Binary Search

Algorithm analysis *O* (log n)

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size. Such as:

Binary search

Locate the element a in a sorted (in ascending order) array by first comparing a with the middle element and then (if they are not equal) dividing the array into two sub-arrays;

If a is less than the middle element you repeat the whole procedure in the left sub-array, otherwise - in the right sub-array.

The procedure repeats until a is found or sub-array is a zero dimension

.

Binary Search

Algorithm analysis: O (log n)

Unsuccessful search:

$$O(\log N + 1)$$

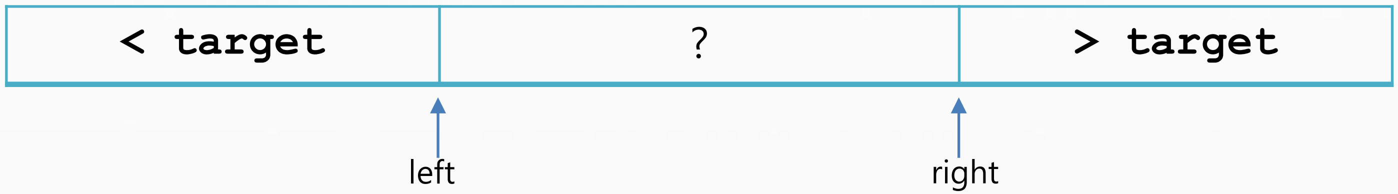
Successful search: worst-case;

$$O(\log N)$$

Successful search: average-case; is only one iteration better (because half of the elements require the worst case for their search, a quarter of the elements save one iteration, and only one in elements will save 2^i iterations from the worst case)

$$O(\log N - 1)$$

Binary Search



At any step during a search for “target”, we have restricted our search space to those keys between “left” and “right”.

Any key to the left of “left” is smaller than “target” and is thus eliminated from the search space

Any key to the right of “right” is greater than “target” and is thus eliminated from the search space

Binary Search

Since the array is sorted, we can reduce our search space in half by comparing the target key with the key contained in the middle of the array and continue this way

Example: Let's search for 55

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	8	10	11	20	50	55	60	65	70	72	90	91	94	96	99

left

$$middle = \frac{(left + right)}{2}$$

right

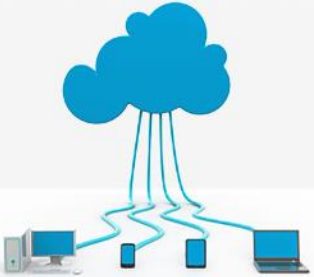
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	8	10	11	20	50	55	60	65	70	72	90	91	94	96	99

left

middle

right

eliminated



Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	8	10	11	20	50	55	60	65	70	72	90	91	94	96	99

eliminated left middle right

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	8	10	11	20	50	55	60	65	70	72	90	91	94	96	99

eliminated right left middle

Now we found 55→ Successful search

Had we searched for 57, we would have terminated at the next step unsuccessfully

Binary Search

Iterative binary search (non-recursive)

```
// Return the index of the array containing the key or –1 if key not found
int BinarySearch (int A[ ], int N, int key) {
    left = 0;
    right = N-1;

    while (left <= right) {
        int middle = ( left + right ) / 2;        // Index of the key to test against
        if (A[middle] == key) return middle;   // Key found. Return the index
        else if (key < A[middle]) right = middle – 1;    // Eliminate the right side
        else left = middle+1;                             // Eliminate the left side
    } //end-while

    return –1;    // Key not found
} //end-BinarySearch
```

Worst case running time: $T(n) = 3 + 5 * \log_2 N$. Why?

Binary Search

Recursive binary search

```
int BinarySearch ( ListItem [ ] L, int k, int low, int high ) {
        int mid= ( low + high ) / 2;
        if ( low > high )
                return -1;
        else if ( L [mid] == k )
                return mid;
        else if ( L [mid] < k )
                return BinarySearch ( L, k, mid+1, high );
        else
                return BinarySearch ( L, k, low, mid-1 );
}
```

Recursive Powers

Simple recursive power algorithm

Recursive definition of b^n:

$$b^n = 1 \quad \text{if } n = 0$$
$$b^n = b \times b^{n-1} \quad \text{if } n > 0$$

To compute b^n:

1. If $n = 0$
   1.1. Terminate with answer 1.
2. If $n > 0$
   2.1. Terminate with answer $b \times b^{n-1}$.

Easy case: solved directly

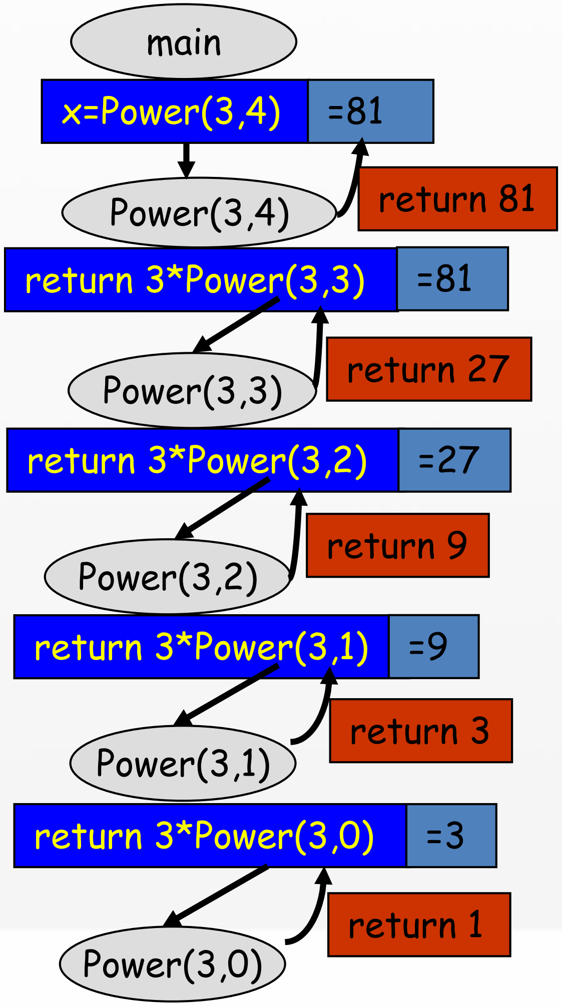
Hard case: solved by computing b^{n-1}, which is easier since $n-1$ is closer than n to 0

Recursive Powers

Simple recursive power algorithm

```
static int power (int b, int n) {
// Return bⁿ (where n is non-negative)
if (n == 0)
            return 1;
else
            return b * power (b, n-1);
}
```

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



Recursive Powers

Smart recursive power algorithm

To compute b^n:

1. If $n = 0$:
   1.1. Terminate with answer 1

Easy case: solved directly

2. If $n > 0$:
   2.1. Let p be $b^{n/2}$
   2.2. If n is even:
   2.2.1. Terminate with answer $p \times p$
   2.3. If n is odd:
   2.3.1. Terminate with answer $b \times p \times p$

Hard case: solved by computing $b^{n/2}$, which is easier since $n / 2$ is closer than n to 0

Recursive Powers

Smart recursive power algorithm

```
static int power (int b, int n) {
// Return b^n (where n is non-negative)
        if (n == 0)
                return 1;
        else {
                int p = power (b, n / 2);
                if (n % 2 == 0)  return p * p;
                else  return b * p * p;
        }
}
```

Recursive Powers

Smart recursive power algorithm - Analysis

Counting multiplications:

Each recursive power algorithm performs the same number of multiplications as the corresponding non-recursive algorithm.

So their time complexities are the same:

	non-recursive	recursive
Simple power algorithm	***O*** **(n)**	***O*** **(n)**
Smart power algorithm	***O*** **(log n)**	***O*** **(log n)**

Recursive Powers

Smart recursive power algorithm - Analysis

Analysis (space):

The non-recursive power algorithms use constant space, i.e., ***O*** **(1)**

A recursive algorithm uses extra space for each recursive call. The simple recursive power algorithm calls itself *n* times before returning, whereas the smart recursive power algorithm calls itself floor($\log_2 n$) times

	non-recursive	recursive
Simple power algorithm	***O*** **(1)**	***O*** **(n)**
Smart power algorithm	***O*** **(1)**	***O*** **(log n)**

Summation

Computing 1+2+..+N Recursively

Consider the problem of computing the sum of the number from 1 to n: 1+2+3+...+n

Here is how we can think recursively:

- In order to compute Sum(n) = 1+2+..+n
  - compute Sum(n-1) = 1+2+..+n-1 (a smaller problem of the same type)
  - Add n to Sum(n-1) to compute Sum(n)
  - i.e., Sum(n) = Sum(n-1) + n;
- We also need to identify base case(s)
  - A base case is a sub-problem that can easily be solved without further dividing the problem
  - If n = 1, then Sum(1) = 1;

Summation

Computing 1+2+..+N Recursively

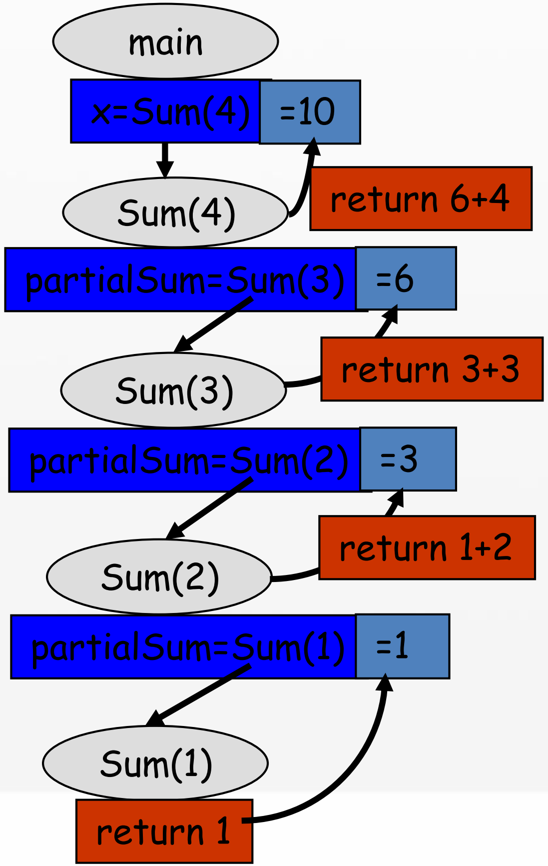
```
/* Computes 1+2+3+…+n */
int Sum (int n) {
  int partialSum = 0;

  /* Base case */
  if (n == 1) return 1;

  /* Divide and conquer */
  partialSum = Sum (n - 1);

  /* Merge */
  return partialSum + n;
} /* end-Sum */
```

```
main (String args [ ] ) {
  int x = 0;

  x = Sum (4);
  println("x: " + x);

  return 0;
} /* end-main */
```

Summation

Recursion Tree for Sum (4)

```
/* Computes 1+2+3+…+n */
int Sum (int n){
  int partialSum = 0;

  /* Base case */
  if (n == 1) return 1;

  /* Divide and conquer */
  partialSum = Sum (n-1);

  /* Merge */
  return partialSum + n;
} /* end-Sum */

main(String args[ ]) {
  int x = Sum (4);
  println("Sum: " + Sum (4));
} /* end-main */
```

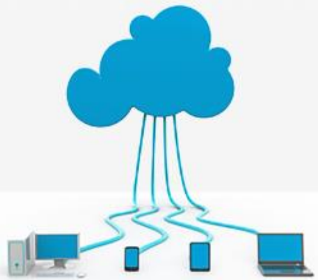


Summation

Running Time for Sum (n)

```
/* Computes 1+2+3+...+n */
int Sum (int n) {
  int partialSum = 0;

  /* Base case */
  if (n == 1) return 1;

  /* Divide and conquer */
  partialSum = Sum ( n-1 );

  /* Merge */
  return partialSum + n;
} /* end-Sum */
```

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



Summation

Summation I

Compute the sum of N numbers A[1..N]

Stopping rule (Base Case):

- If N == 1 then sum = A [1]

Key Step

- Divide:

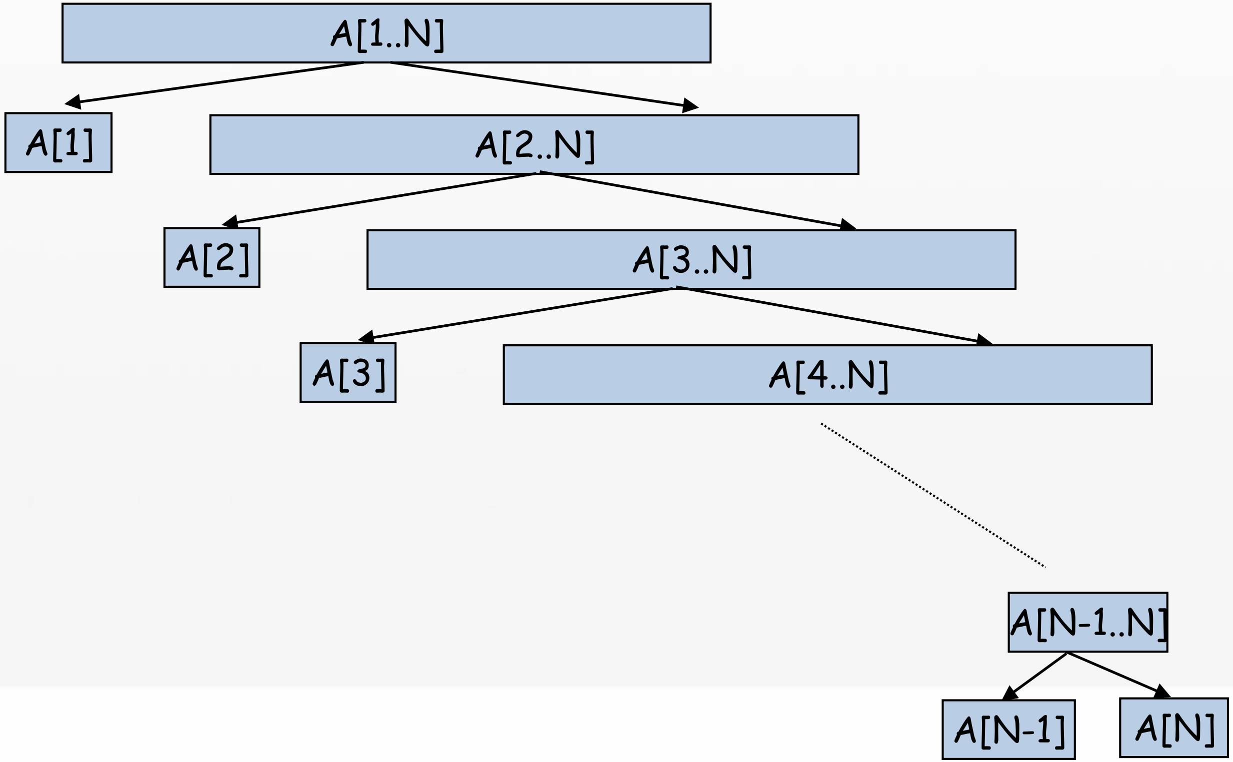
  Consider the smaller A[1] and A[2..N]
- Conquer:

  Compute Sum(A[2..N])
- Merge:

  Sum(A[1..N]) = A[1] + Sum(A[2..N])

Summation

Recursive Calls of Summation I



Summation

Summation I – Code

```
/* Computes the sum of an array of numbers A[0..N-1] */
int Sum (int A[ ], int index, int N ){
    /* Base case */
    if (N == 1) return A[index];

    /* Divide & Conquer */
    int localSum = Sum (A, index+1, N-1);

    /* Merge */
    return A[index] + localSum;
} //end-Sum
```

$$T(n) = \begin{cases} 1 \text{ if } N = 1 \text{ (Base case)} \\ T(n-1) + 1 \quad \text{if } N > 1 \end{cases}$$

Time to find the sum of n-1 numbers

Time to combine the results

Summation

Summation II

Compute the sum of N numbers A[1..N]

Stopping rule:

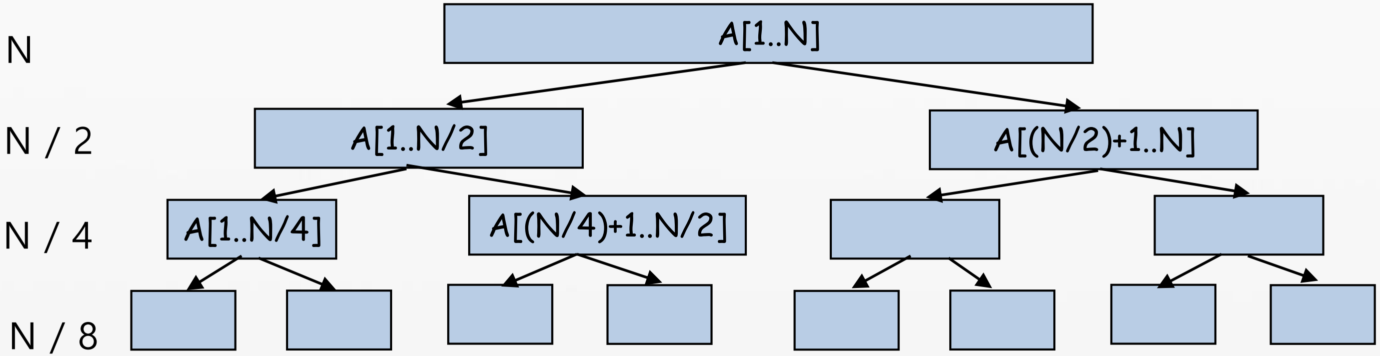
- If N == 1 then sum = A[1]

Key Step

- Divide:
  Consider the smaller A[1..N/2] and A[(N/2)+1..N]
- Conquer:
  Compute Sum(A[1..N/2]) and Sum(A[(N/2)+1..N])
- Merge:
  Sum(A[1..N]) = Sum(A[1..N/2]) + Sum(A[(N/2)+1..N])

Summation

Recursive Calls of Summation II



Summation

Summation II – Code

```
/* Computes the sum of an array of numbers A[0..N-1] */
int Sum (int A[ ], int index1, int index2) {
    /* Base case */
    if (index2-index1 == 1) return A [index1];

    /* Divide & Conquer */
    int middle = (index1+index2)/2;
    int localSum1 = Sum (A, index1, middle);
    int localSum2 = Sum (A, middle+1, index2);

    /* Merge */
    return localSum1 + localSum2;
} //end-Sum
```

$$T(n) = \begin{cases} 1 \text{ if } N = 1 \text{ (Base case)} \\ T(n/2) + T(n/2) + 1 \text{ if } N > 1 \end{cases}$$

Fibonacci Numbers

Fibonacci numbers are defined as follows

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2)

```
/* Computes nth Fibonacci number */
int Fibonacci (int n) {
  /* Base cases */
  if (n == 0) return 0;
  if (n == 1) return 1;

  return Fibonacci (n-1) + Fibonacci (n-2);
} /* end-Fibonacci */
```

Fibonacci Numbers

Recursion Tree for F(5)