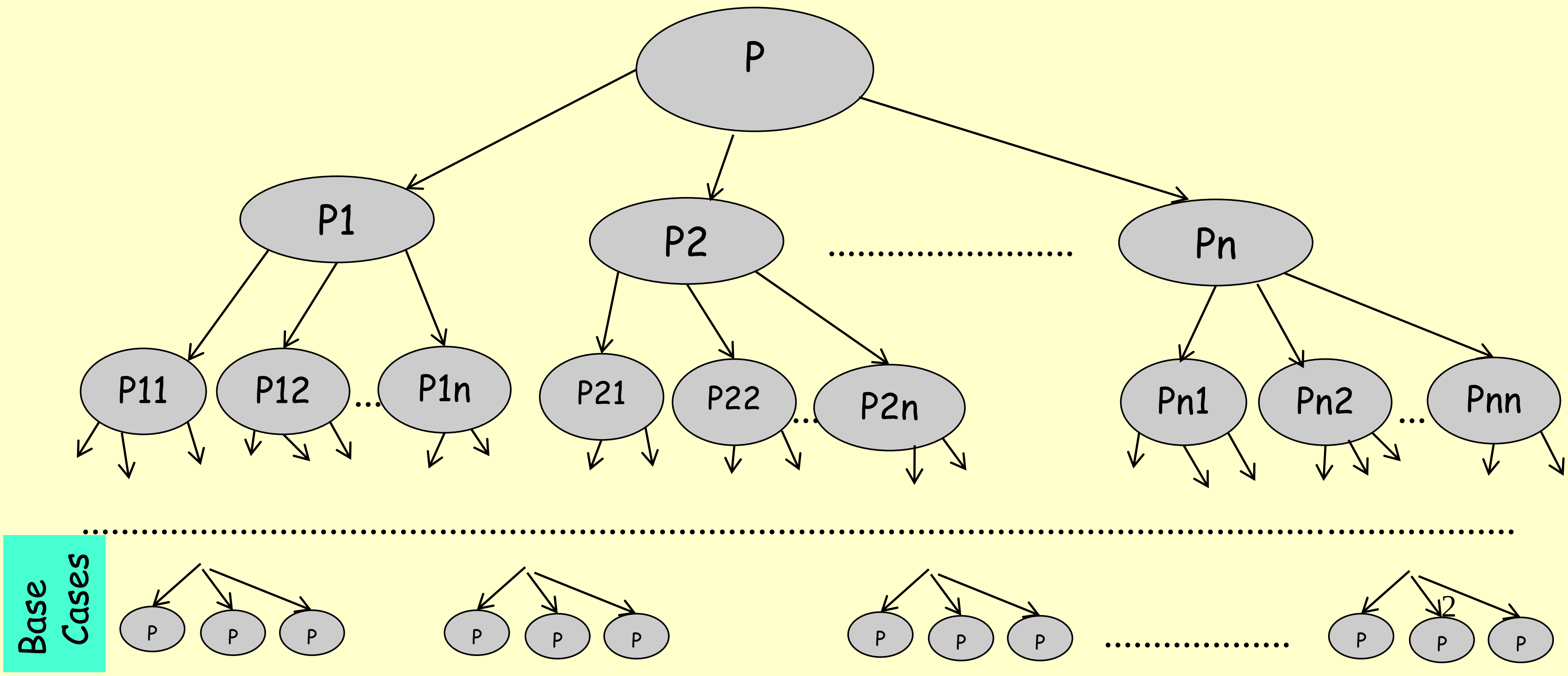
Today's Material

- Divide & Conquer (Recursive) Algorithms
  - Design
  - Analysis
  - Solving Recurrences
    - Master theorem
    - Repeated expansion (Backward substitution)

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



Divide & Conquer Strategy (cont)

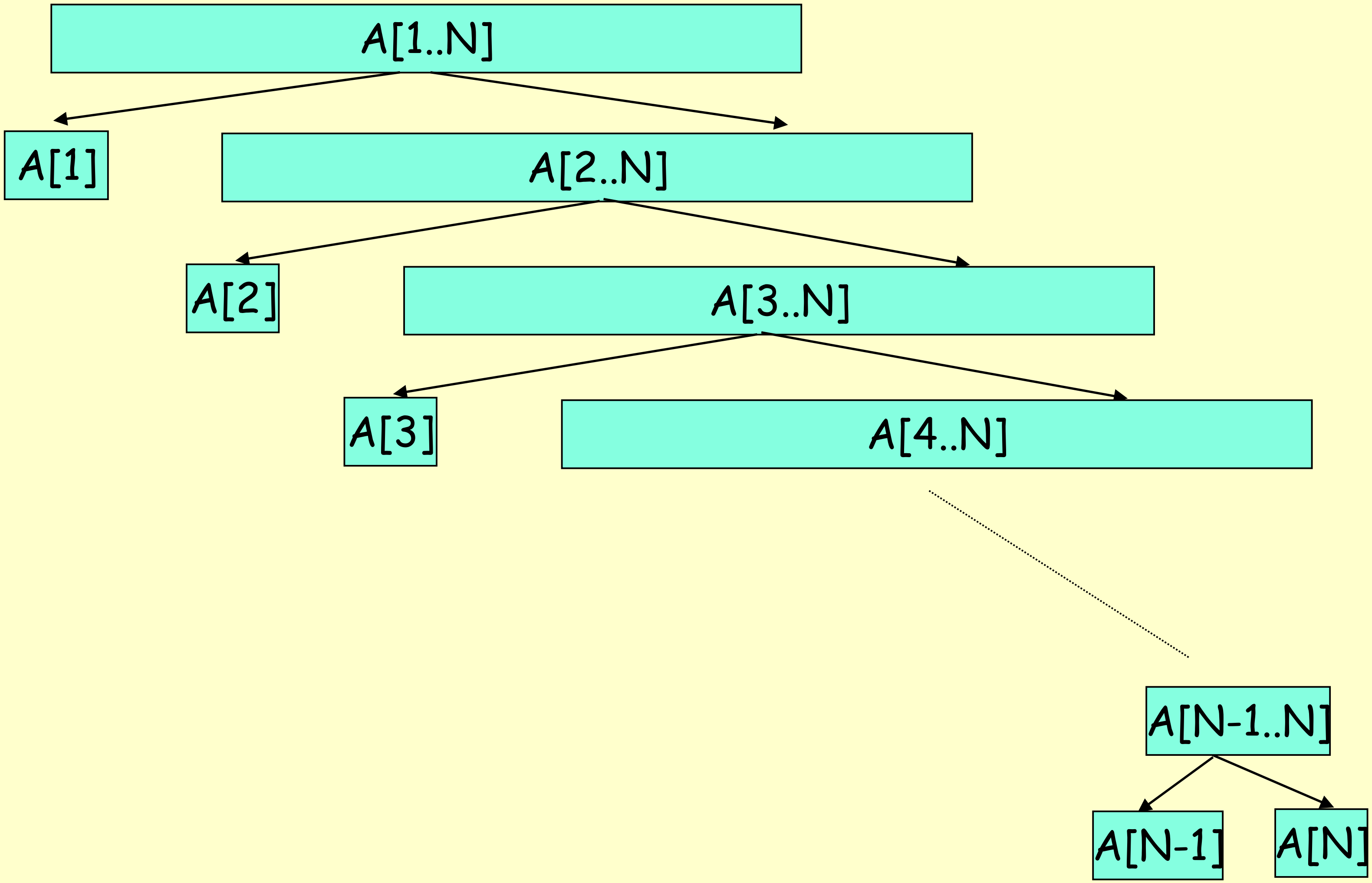
```
/* Solve a problem P */
Solve(P){
    /* Base case(s) */
    if P is a base case problem
        return the solution immediately

 /* 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
```

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])

Recursive Calls of Summation I



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
```

$$T(n) = \begin{cases} 1 \text{ if } N = 1 \text{ (Base case)} \\ \underbrace{T(n-1)}_{\text{Time to find the sum of n-1 numbers}} + \underbrace{1}_{\text{Time to combine the results}} \;\; \text{if } N>1 \end{cases}$$

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 subproblem that can easily be solved without further dividing the problem
    - If n = 1, then Sum(1) = 1;

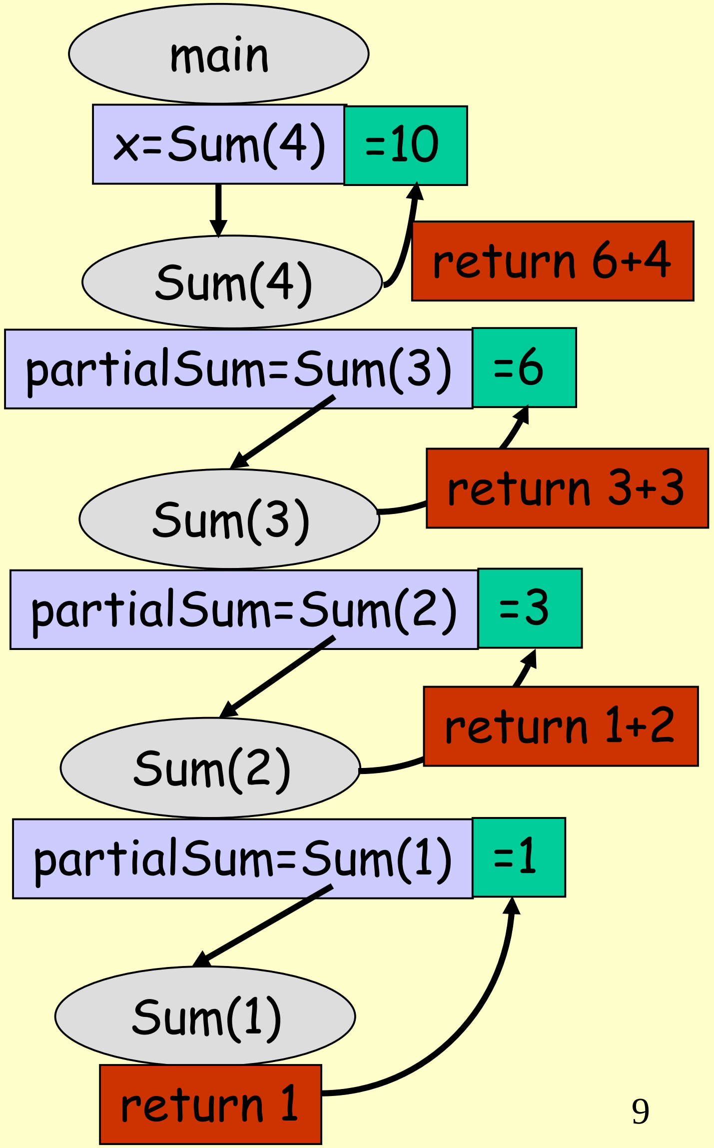
Computing 1+2+..+N Recursively

```
/* 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 */
```

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 */
```

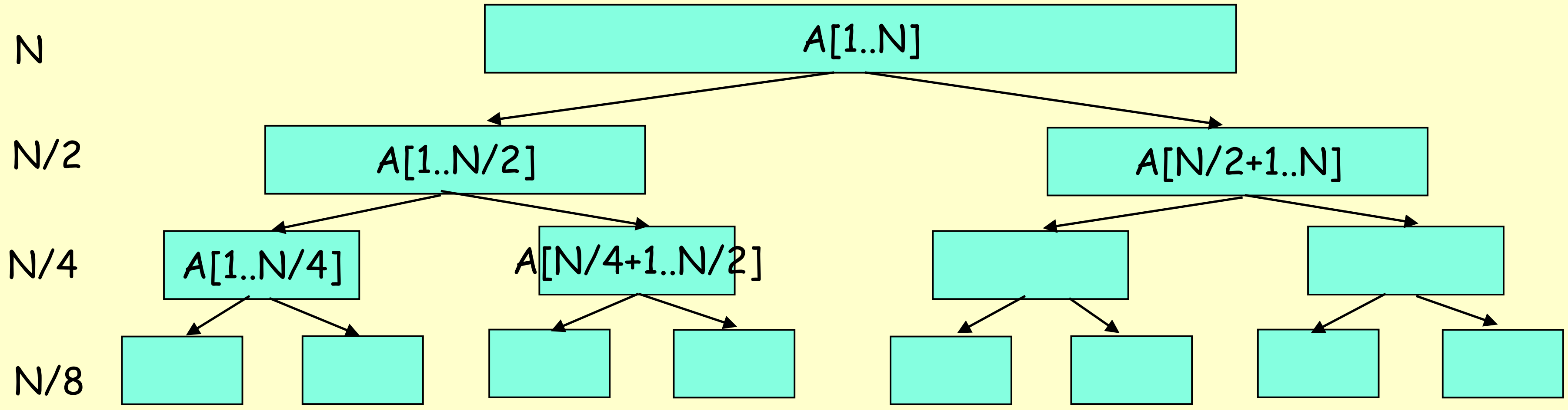


Summation II

- 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])

Recursive Calls of Summation II



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, index2);

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

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

Computing a^n Recursively

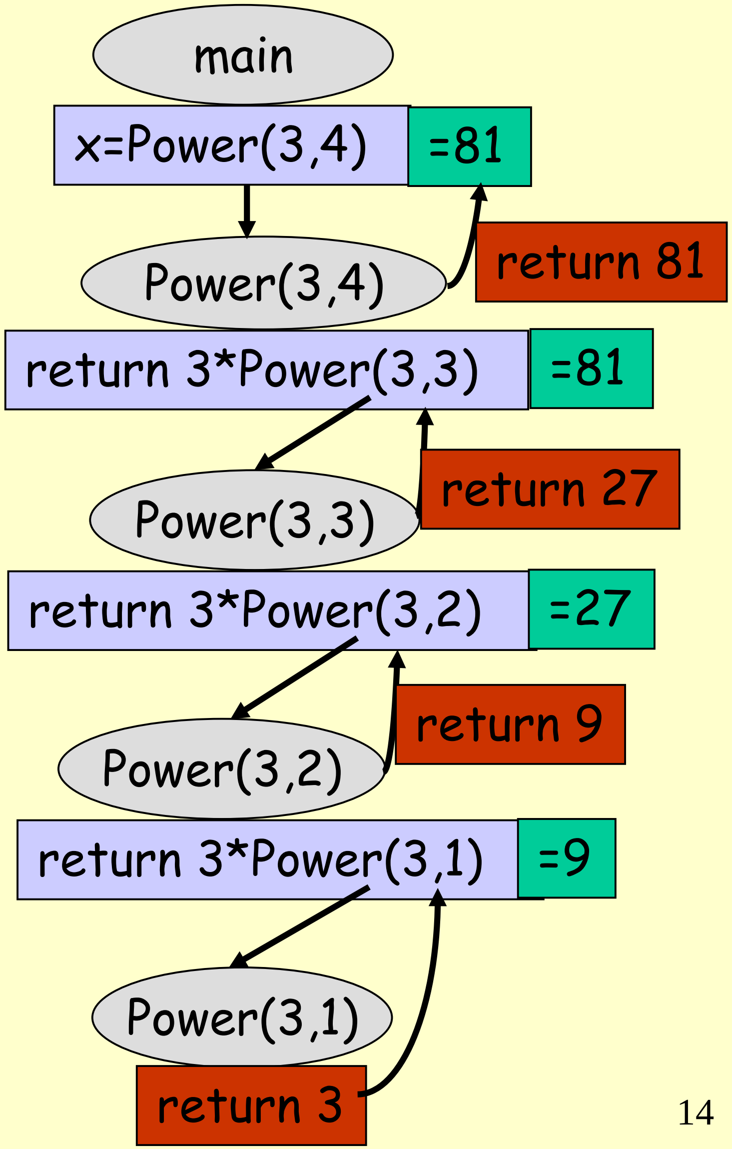
```
/* Computes a^n */
double Power(double a, int n){
  double partialResult;

  /* Base cases */
  if (n == 0) return 1;
  else if (n == 1) return a;

  /* partialResult = a^(n-1) */
  partialResult = Power(a, n-1);

  /* Merge */
  return partialResult*a;
} /* end-Power */
```

- We can combine divide, conquer & merge into a single statement

```
/* Computes a^n */
double Power(double a, int n){
  /* Base cases */
  if (n == 0) return 1;
  else if (n == 1) return a;

  return Power(a, n-1)*a;
} /* end-Power */
```

Recursion Tree for Power(3, 4)

```
/* Computes a^n */
double Power(double a, int n){
  /* Base cases */
  if (n == 0) return 1;
  else if (n == 1) return a;

  return a * Power(a, n-1);
} /* end-Power */

main(String args[]){
  double x;

  x = Power(3, 4);
} /* end-main */
```



Running Time for Power(a, n)

```
/* Computes a^n */
double Power(double a, int n){
  /* Base cases */
  if (n == 0) return 1;
  else if (n == 1) return a;

  return a * Power(a, n-1);
} /* end-Power */
```

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

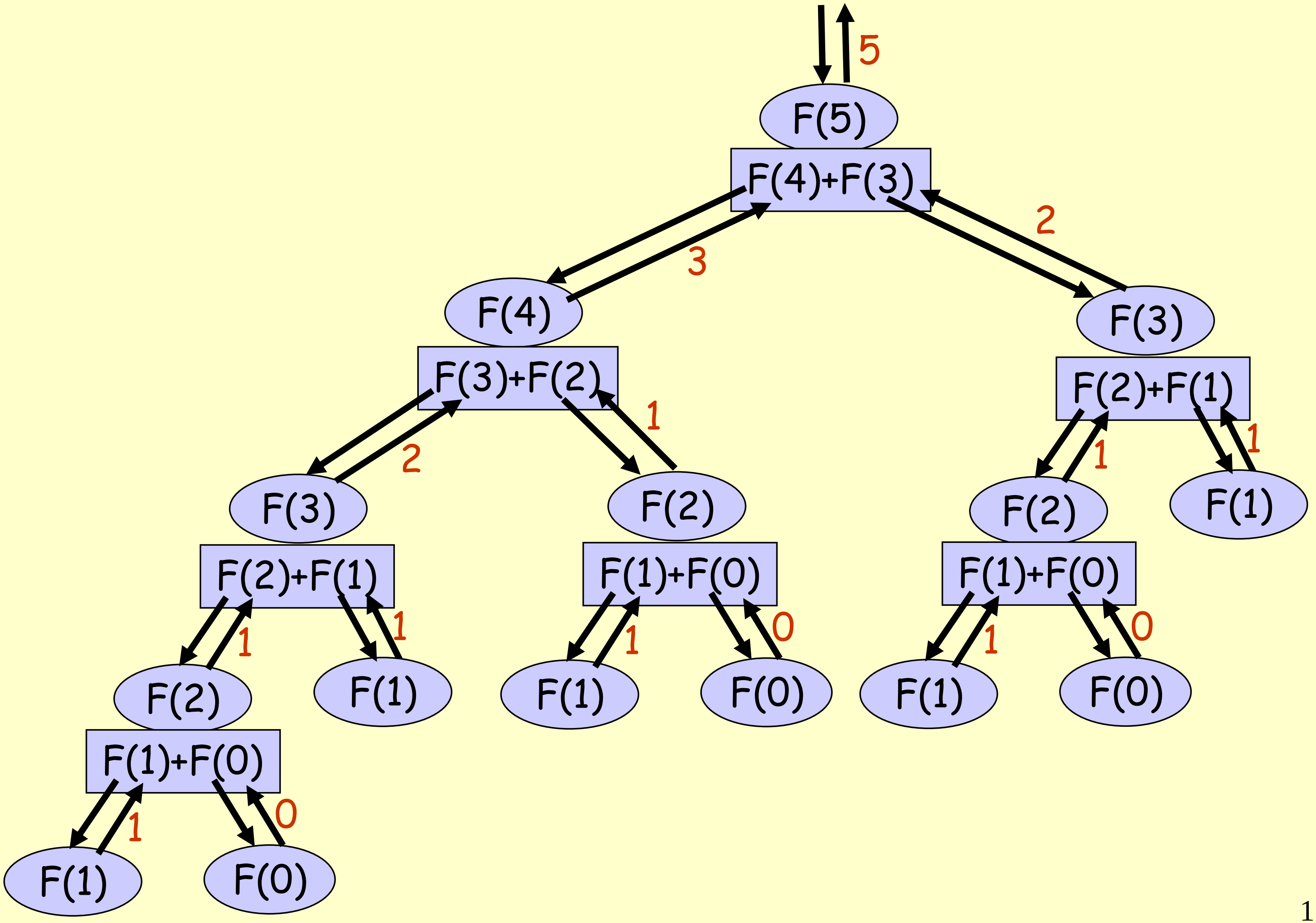
Fibonacci Numbers

- 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 */
```


Example: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Recursion Tree for F(5)



Linear Search

- Find a key in an array of numbers A[0..N-1]

- Stopping rules (Base Cases):
  - if (N == 0) return false;
  - if (key == A[0]) return true;

- Key Step
  - Divide & Conquer
    - Search key in A[1..N-1]

Linear Search - Code

```
/* Searches a key in A[0..N-1] */
bool LinearSearch(int A[], int index, int N, int key){
    /* Base cases */
    if (N == 0) return false; /* Unsuccessful search */
    if (key == A[0]) return true;    /* Success */

    /* Divide & Conquer & Merge */
    return LinearSearch(A, index+1, N-1, key);
} //end-LinearSearch
```

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

Binary Search

- Find a key in a sorted array of numbers A[0..N-1]

- Stopping rules (Base Cases):
  - if (N == 0) return false;
  - if (key == A[N/2]) return true;

- Key Step
  - if (key < A[N/2]) Search A[0..N/2-1]
  - else Search A[N/2+1..N-1]

Binary Search - Code

```
/* Searches a key in sorted array A[0..N-1] */
bool BinarySearch(int A[], int index1, int index2, int key){
    int middle = (index1+index2)/2;
    int N = index2-index1;

    /* Base cases */
    if (key == A[middle]) return true; /* Success */
    if (N == 1) return false; /* Unsuccessful search */

    /* Conquer & Merge */
    else if (key < A[middle])
        return BinarySearch(A, index1, middle, key);
    else
        return BinarySearch(A, middle, index2, key);
} //end-BinarySearch
```

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

Solving Recurrences

- So far we have expressed the running time of our recursive algorithms in terms of recurrences
  - $T(n) = T(n-1) + 1$
  - $T(n) = 2*T(n/2) + 1$
  - $T(n) = T(n/2) + 1$
  - $T(n) = 2*T(n/2) + n$
  - …
- We need to solve these recurrences and express the running time as a function of the input size N
  - How do we solve recurrences?

Method1: Master Theorem

- Let a >= 1, b >= 1 be constants and let $T(n) = aT(n/b) + cn^k$ defined for n >=0
  - Case1: If $a > b^k$ then $T(n) = \Theta(n^{\log b(a)})$
  - Case2: If $a = b^k$ then $T(n) = \Theta(n^k * \log n)$
  - Case3: If $a < b^k$ then $T(n) = \Theta(n^k)$

- $T(n) = 2*T(n/2) + n$
  - a = 2, b = 2, c = k = 1 -- > Falls to Case 2
  - $T(n) = \Theta(n*\log n)$
- $T(n) = 2*T(n/2) + 1$
  - a = 2, b = 2, c = 1, k = 0 -- > Falls to Case 1
  - $T(n) = \Theta(n)$

Method2: Repeated Expansions

- How do we solve $T(n) = 2*T(N/2) + N$
  - $T(n) = 2*T(n/2) + n$
  - $T(n) = 2*(2*T(n/4) + n/2) + n$
  - $T(n) = 2^2*T(n/2^2) + n + n$
  - $T(n) = 2^2*(2*T(n/2^3) + n/2^2) + n + n$
  - $T(n) = 2^3*T(n/2^3) + n + n + n$
  - ……
  - $T(n) = 2^k*T(n/2^k) + \sum_{i=1}^{k} n$
  - We want $n/2^k = 1 \rightarrow k = \log n$
  - $T(n) = 2^{\log n} + \sum_{i=1}^{k} n$
  - $T(n) = n + n*\log n = O(n\log n)$

Solving Recurrences - Example

- How do we solve $T(n) = T(n-1) + 1$
- Master theorem does not help. Do repeated expansion
  - $T(n) = T(n-1) + 1$
  - $T(n) = (T(n-2)+1) + 1 = T(n-2) + 1 + 1$
  - $T(n) = (T(n-3)+1) + 1 + 1$
  - $T(n) = T(n-3) + 1 + 1 + 1$
  - …
  - …
  - Do the rest yourself…

Solving Recurrences - Example

- How do we solve $T(n) = 2*T(N/2) + nlogn$
- Master theorem does not help. Do repeated expansion
  - $T(n) = 2*T(n/2) + nlogn$
  - $T(n) = 2*(2*T(n/4) + n/2*log(n/2)) + nlogn$
  - $T(n) = 2^2*T(n/2^2) + nlog(n/2) + nlogn$
  - ……
  - Do the rest yourself…