Lecture 02
Divide and Conquer

Wei-Che Chien(簡暐哲)
Department of Computer Science and Information Engineering
National Dong Hwa University

Prove by contradiction(反證法)

$$\boldsymbol{n}^2 \in \boldsymbol{O}\ (\boldsymbol{n})\ ?$$

Disproof.

Assume for a contradiction that there exist positive constants c and N s.t.

$$\boldsymbol{n}^2 \leq cn$$

holds for any integer n with $n \geq N$.

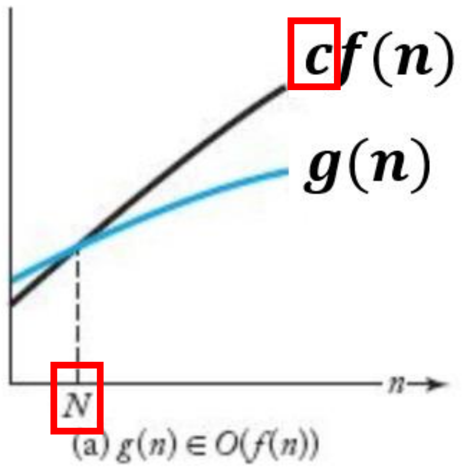
Assume $$\boldsymbol{n} = \mathbf{1} + \lceil \boldsymbol{max}\ (\boldsymbol{N}, \boldsymbol{c}) \rceil$$

and because $\boldsymbol{n} > N, n > c$, it follows that

$$\boldsymbol{n}^2 \leq cn$$

Due to contradiction, we know that

$$\boldsymbol{n}^2 \neq \boldsymbol{O}\ (\boldsymbol{n})$$



$c\boldsymbol{f}(\boldsymbol{n})$
$\boldsymbol{g}(\boldsymbol{n})$
N
$n \rightarrow$
(a) $g(n) \in O(f(n))$

Recursion(遞迴)

What is Recursion?

The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function. Using recursive algorithm, certain problems can be solved quite easily.

Types of Recursion

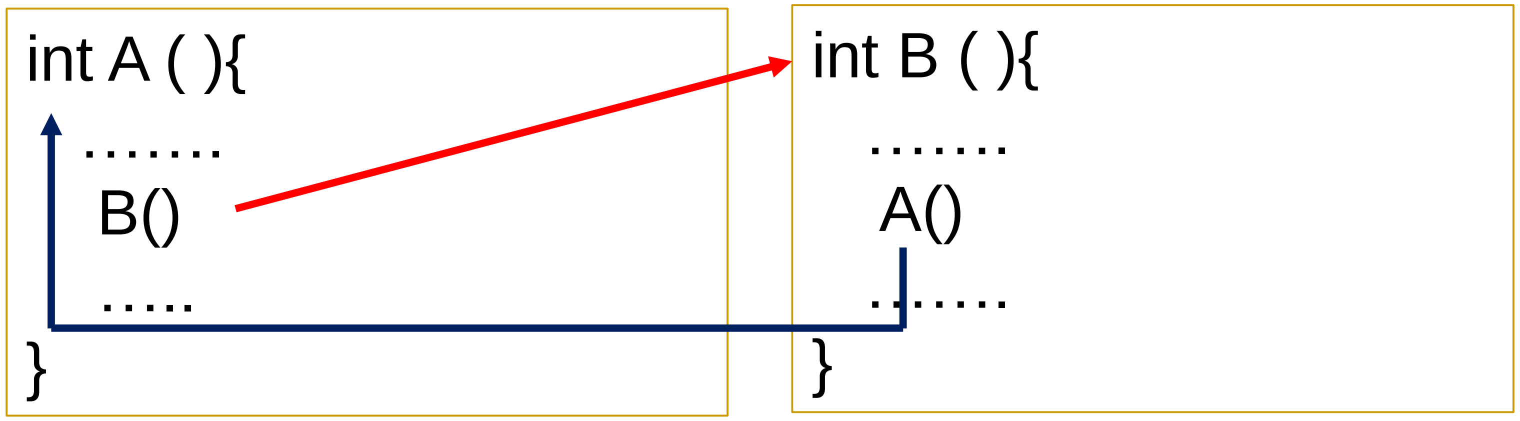
Direct Recursion(直接遞迴)

```
int A ( ){
   …….
    A()
    ……
}
```

Tail Recursion (尾端遞迴)

```
int A ( ){
   …….
    A()

}
```

Indirect Recursion(間接遞迴)



```
int A ( ){
   …….
    B()
    …..
}
```

```
int B ( ){
   …….
   A()
   …….
}
```

Designing Recursive Algorithms

1. **Define base case (終止條件)**
2. **Define recursive case (遞迴關係)**

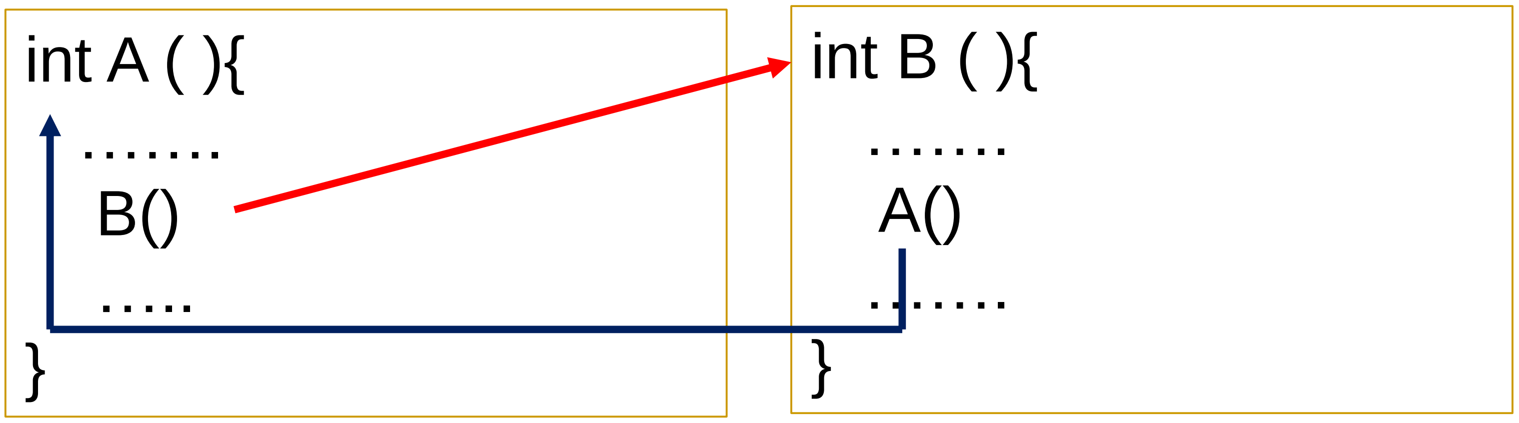
Fibonacci sequence:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } 2 \\ f(n-1) + f(n-2) & \text{if } n \geq 3 \end{cases}$$

```
Fib (n) {
     if n < 2 // base case
     return 1
    // recursive case
     return Fib (n-1) + Fib (n-2) }
```

How Recursion Works

Indirect Recursion(間接遞迴)



Stack

Parameter
Variable
Address

Push & Pop

Recurrence v.s. Non-Recurrence

```
Fib (n) {
      if n < 2 // base case
            return 1
      // recursive case
      return Fib (n-1) + Fib (n-2)}
```

Recursive function
- Clear structure
- Poor efficiency
- Better storage

```
Fib (n) {
      if n < 2
            return 1
      a[0] <- 1
      a[1] <- 1
      for i = 2 to n
            a[i] = a[i-1] + a[i-2]
       return a[n] }
```

Non-recursive function
- Better efficiency
- Unclear structure
- Poor storage

Recursion analysis

1. **Define base case (終止條件)**
2. **Define recursive case (遞迴關係)**

Fibonacci sequence:
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233

$$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } 2 \\ f(n-1) + f(n-2) & \text{if } n \geq 3 \end{cases}$$

$$T(n) = \underline{T(n\text{-}1) + T(n\text{-}2)}\ \underline{+1}$$

Recursion analysis

1. **Define base case (終止條件)**
2. **Define recursive case (遞迴關係)**

Factorial:

0, 1, 2, 6, 24, 120 => 0!, 1!, 2!, 3!, 4!, 5!

$$factorial(n) = \begin{cases} 1 & n = 0 \\ n * factorial(n-1) & n > 0 \end{cases}$$

$$T(n) = \underline{T(n-1)} + \underline{1}$$

Recursion analysis

1. Mathematics -based Method(數學解法)
2. Recursion-Tree Method (遞迴樹法)
3. Master Method(支配理論)
4. Substitution method (替代法)

Mathematics -based Method

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

$T(1) = 1$

Sol:

Mathematics -based Method

$$T(n) = T(\sqrt{n}) + 1$$
$$T(2) = 1$$

Sol:

Recursion-Tree Method

1. **Expand :**

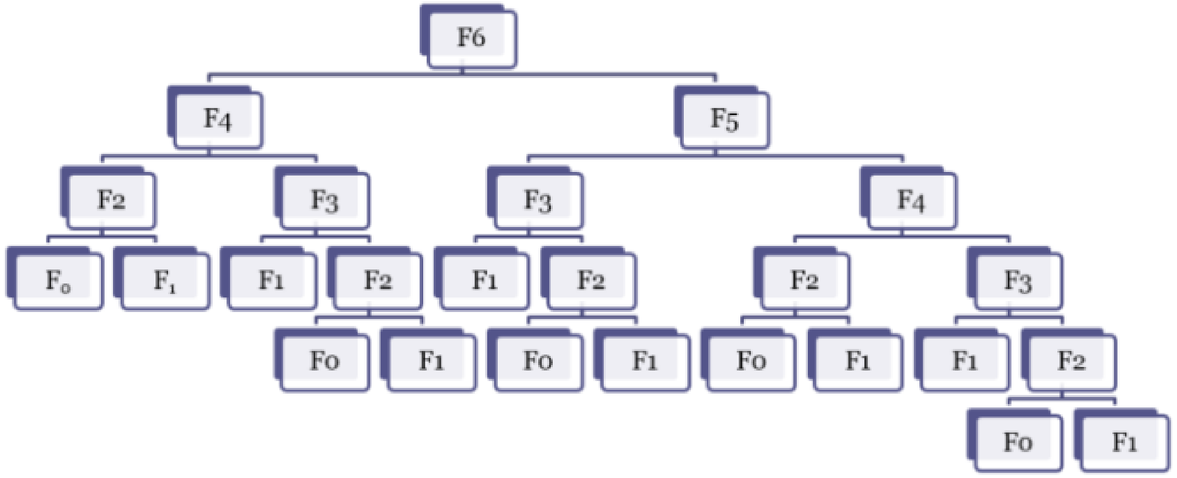
   Expand a recurrence into a tree

2. **Sumup :**

   Sum up the cost of each layer of sub- problem

3. **Total cost :**

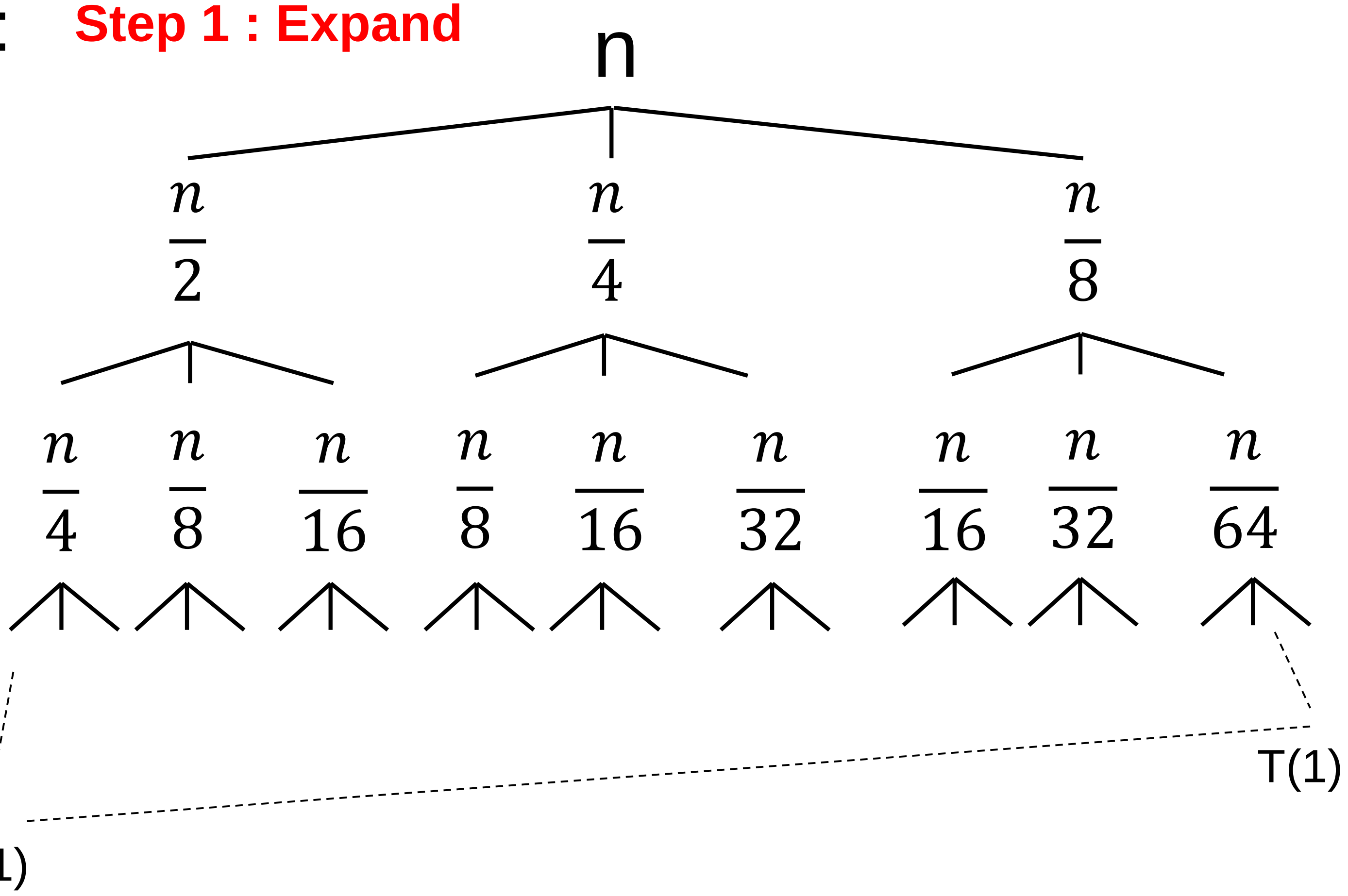
   Sum up the cost of all nodes



Recursion-Tree Method

$T(n) = T(n/2) + T(n/4) + T(n/8) + n, \ T(n) = \theta(n)$

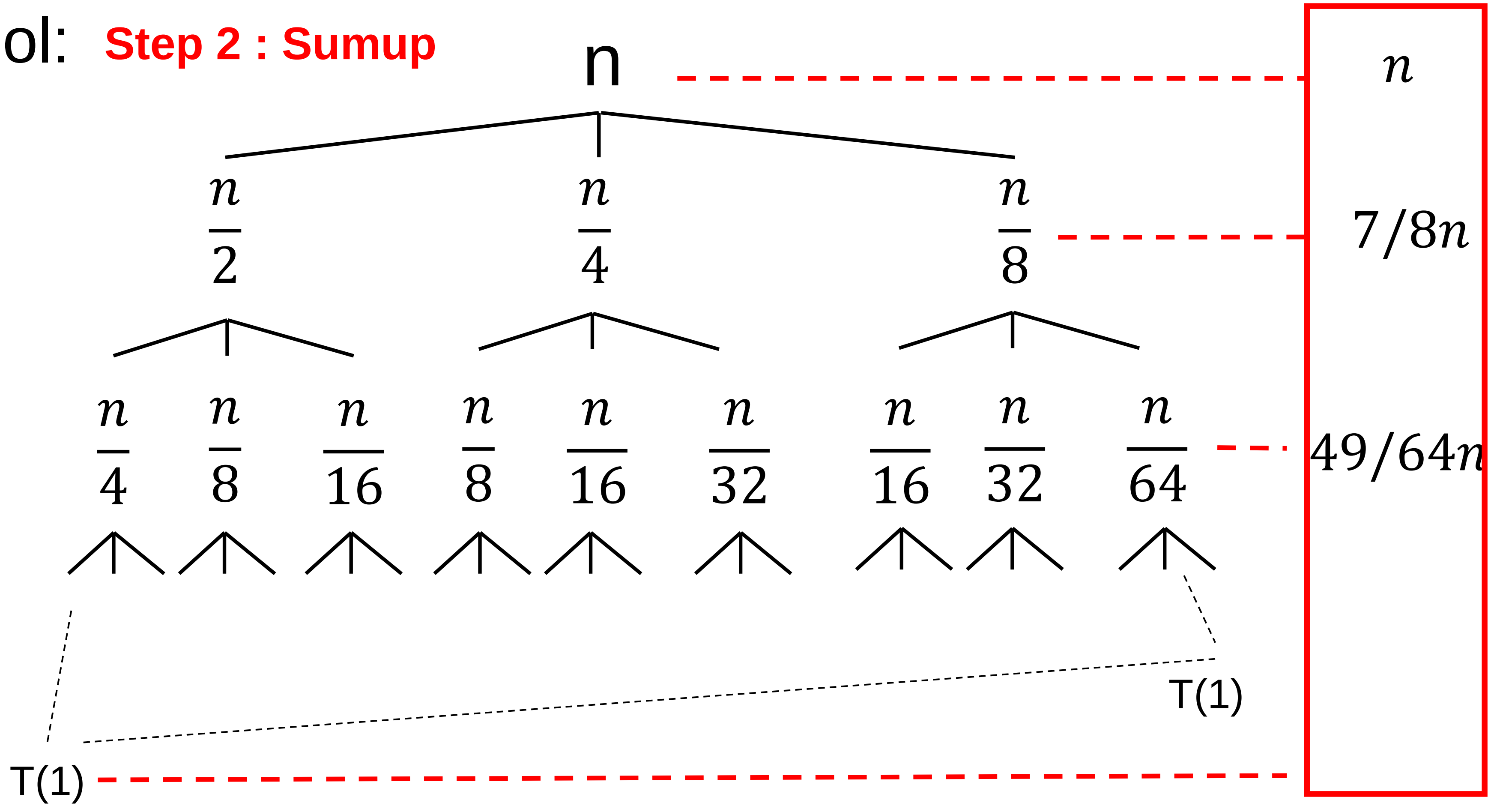
Sol: **Step 1 : Expand**



Recursion-Tree Method

T(n) = T(n/2) + T(n/4) + T(n/8) + n, T(n) = θ(n)

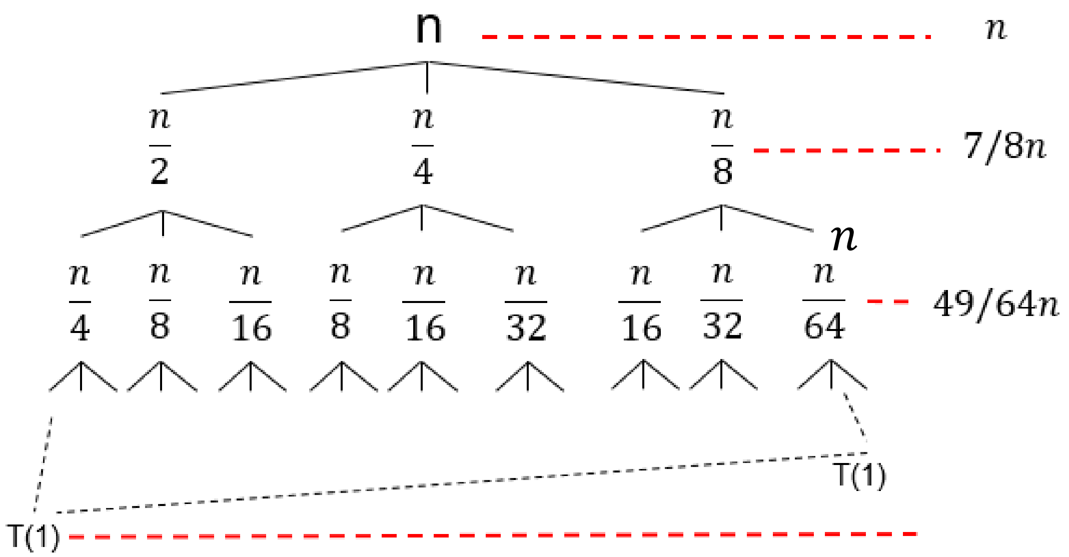
Sol: **Step 2 : Sumup**



Recursion-Tree Method

$T(n) = T(n/2) + T(n/4) + T(n/8) + n$, $T(n) = \theta(n)$

Sol: **Step 3 : Total cost**



n
n
$\frac{n}{2}$ $\frac{n}{4}$ $\frac{n}{8}$
$7/8n$
n
$\frac{n}{4}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{64}$
$49/64n$
T(1)
T(1)

Geometric sequence

$$\boxed{a + ar + ar^2 + \ldots + ar^{(n-1)}}$$

$$\sum_{k=0}^{n-1}(ar^k) = a\left(\frac{1-r^n}{1-r}\right)$$

$$\sum_{k=0}^{\infty}(ar^k) = a\left(\frac{1}{1-r}\right)$$

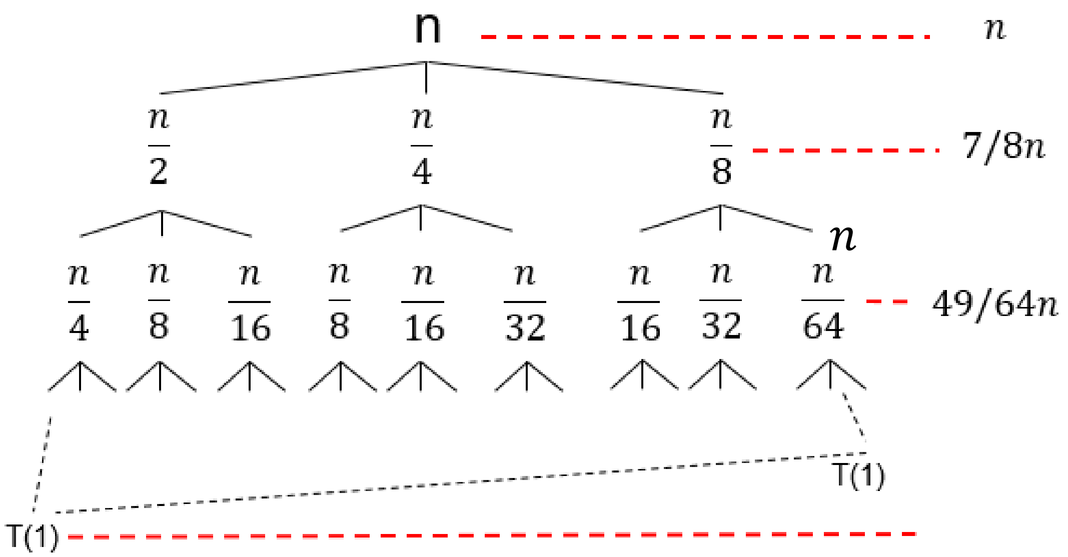
Total cost : n + 7/8n + 49/64n + … $\leq f(n) = \frac{n}{1-\frac{7}{8}} = 8n$

$\therefore T(n) = O(n)$ (1)

Recursion-Tree Method

T(n) = T(n/2) + T(n/4) + T(n/8) + n, T(n) = θ(n)

Sol: **Step 3 : Total cost**


n
n
$\frac{n}{2}$
$\frac{n}{4}$
$\frac{n}{8}$
$7/8n$
n
$\frac{n}{4}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{8}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{16}$ $\frac{n}{32}$ $\frac{n}{64}$
$49/64n$
T(1)
T(1)

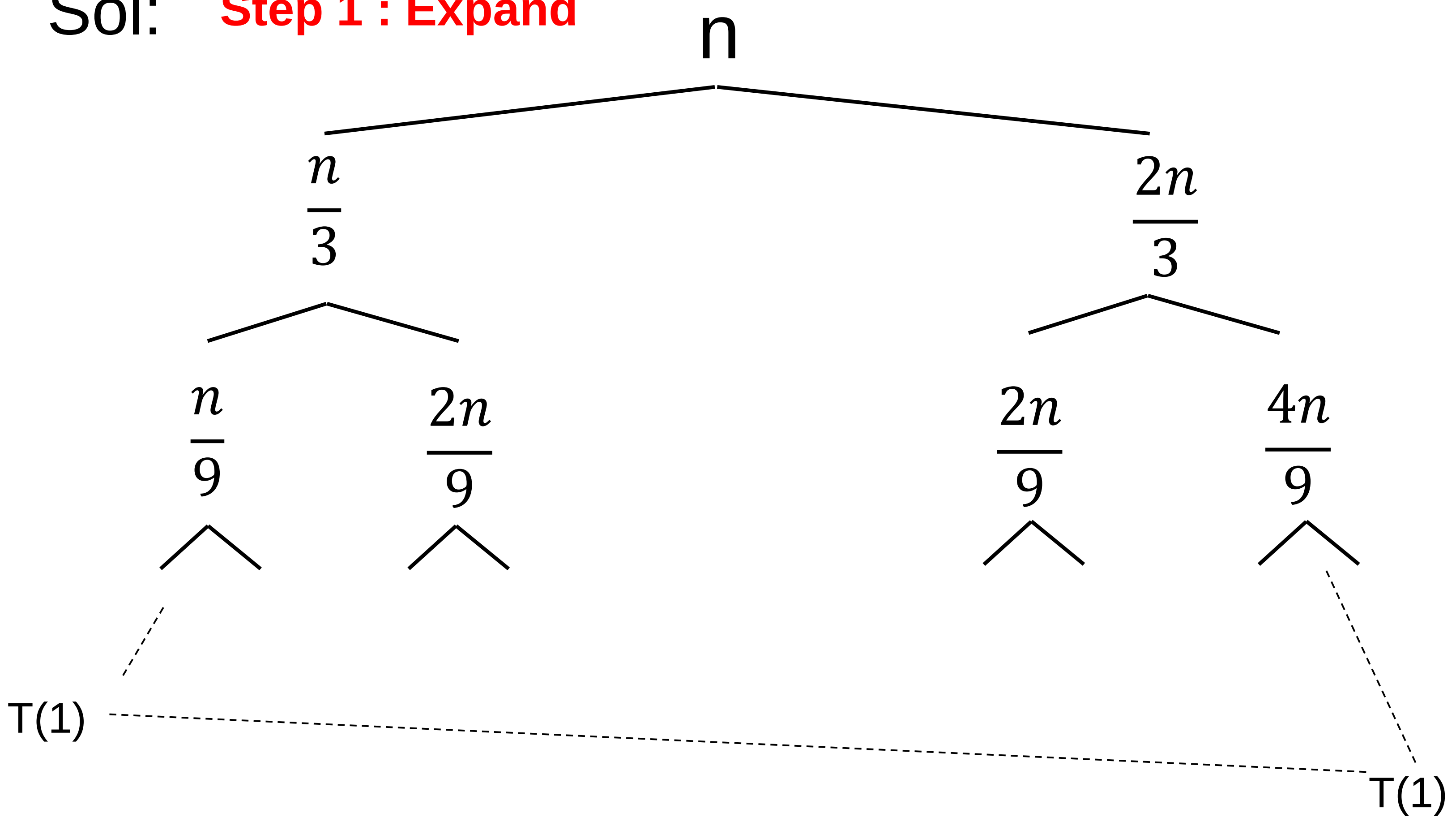
Total cost : n + 7/8n + 49/64n + … $\geq \boldsymbol{f(n) = n}$

$\therefore T(n) = \Omega(n) \quad (2)$ $\therefore (1) + (2) \Rightarrow$ T(n) = θ(n)

Recursion-Tree Method

T(n) = T(n/3) + T(2n/3) + n, T(n) = θ(?)

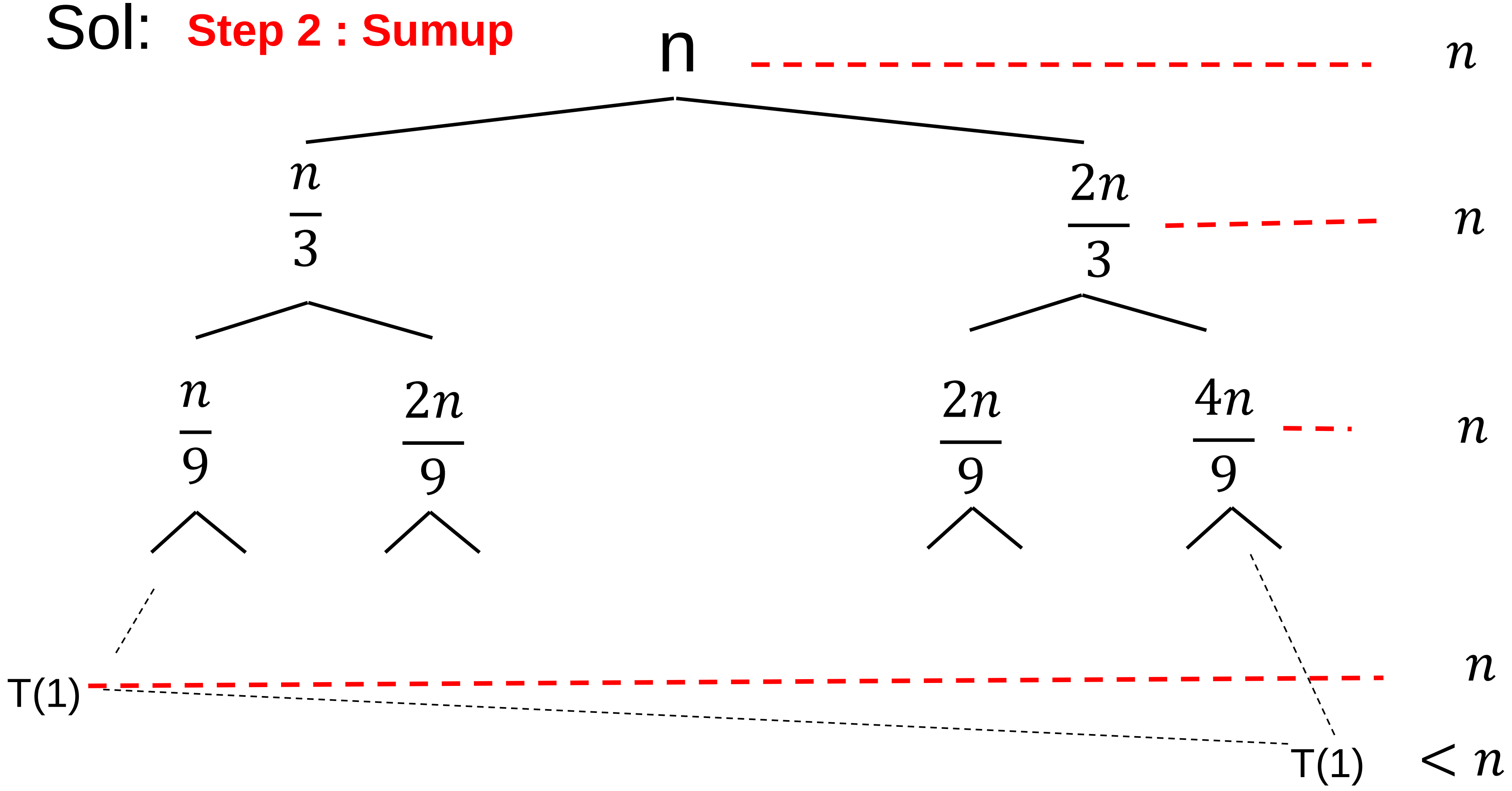
Sol: **Step 1 : Expand**



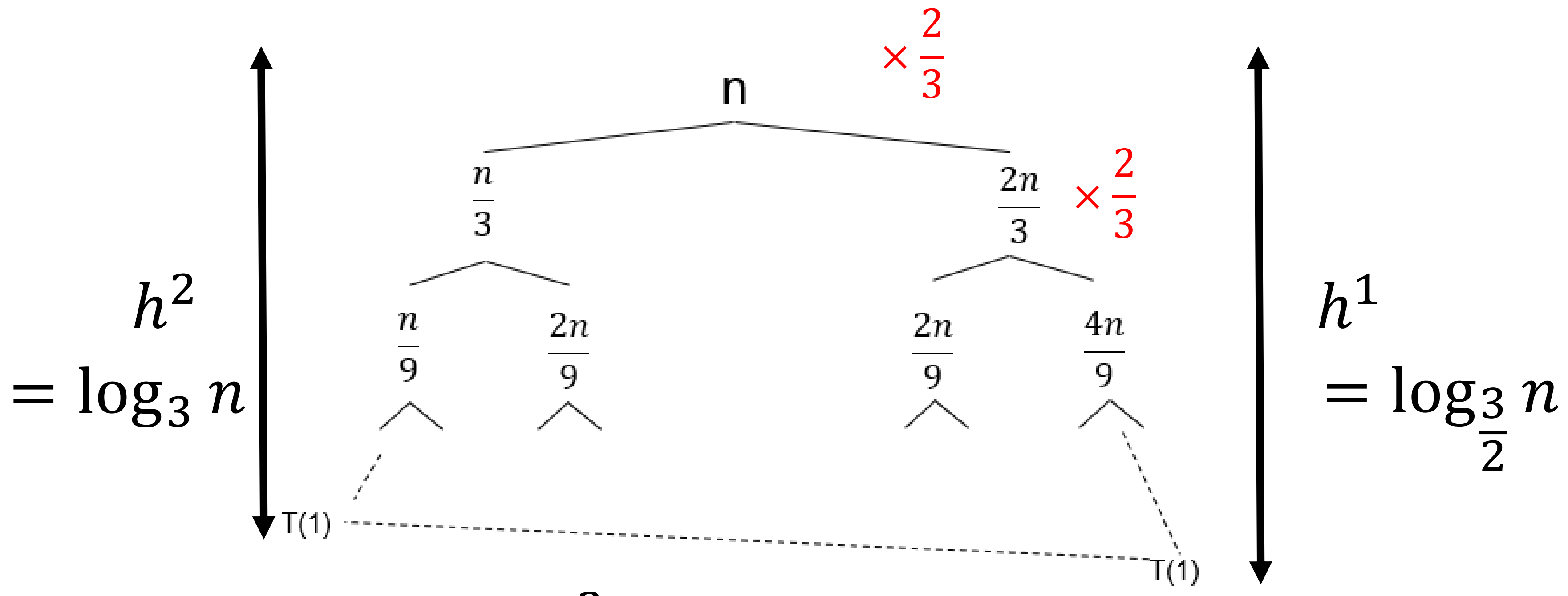
Recursion-Tree Method

$T(n) = T(n/3) + T(2n/3) + n$, $T(n) = \theta(?)$

Sol: **Step 2 : Sumup**



Recursion-Tree Method



$$n \times \left(\frac{2}{3}\right)^h = 1$$

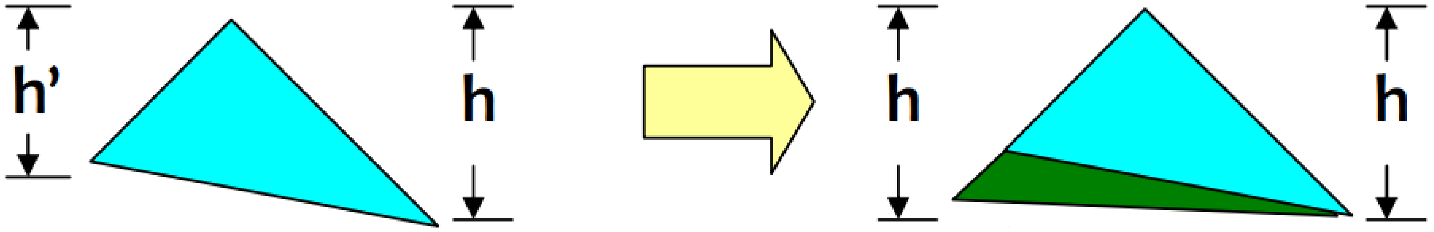
$$\Rightarrow n = \left(\frac{3}{2}\right)^h$$

$$\Rightarrow \log_{\frac{3}{2}} n = h \times \log_{\frac{3}{2}} \frac{3}{2} = h$$

Recursion-Tree Method

T(n) = T(n/3) + T(2n/3) + n,   T(n) = θ(?)

Sol: **Step 3 : Total cost**



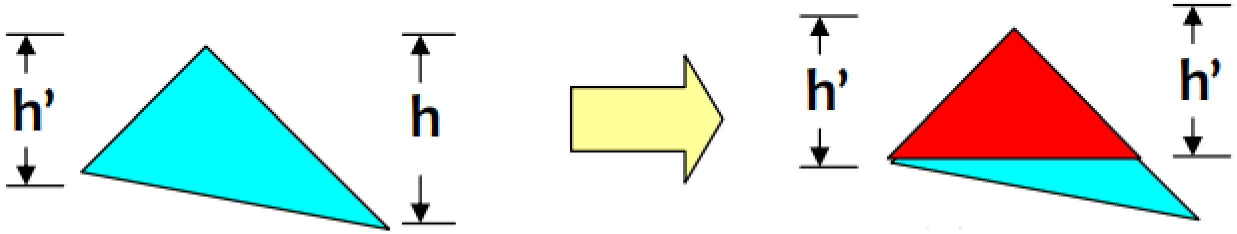
Total cost : $n + n + \ldots \leq \underbrace{\boldsymbol{n} + \boldsymbol{n} + \ \ldots + \boldsymbol{n}}_{\log_{\frac{3}{2}} n + 1} = \boldsymbol{n}\left(\log_{\frac{3}{2}} n + 1\right)$

$$\therefore T(n) = O(n \log n) \quad (1)$$

Recursion-Tree Method

T(n) = T(n/3) + T(2n/3) + n, T(n) = θ(?)

Sol: **Step 3 : Total cost**

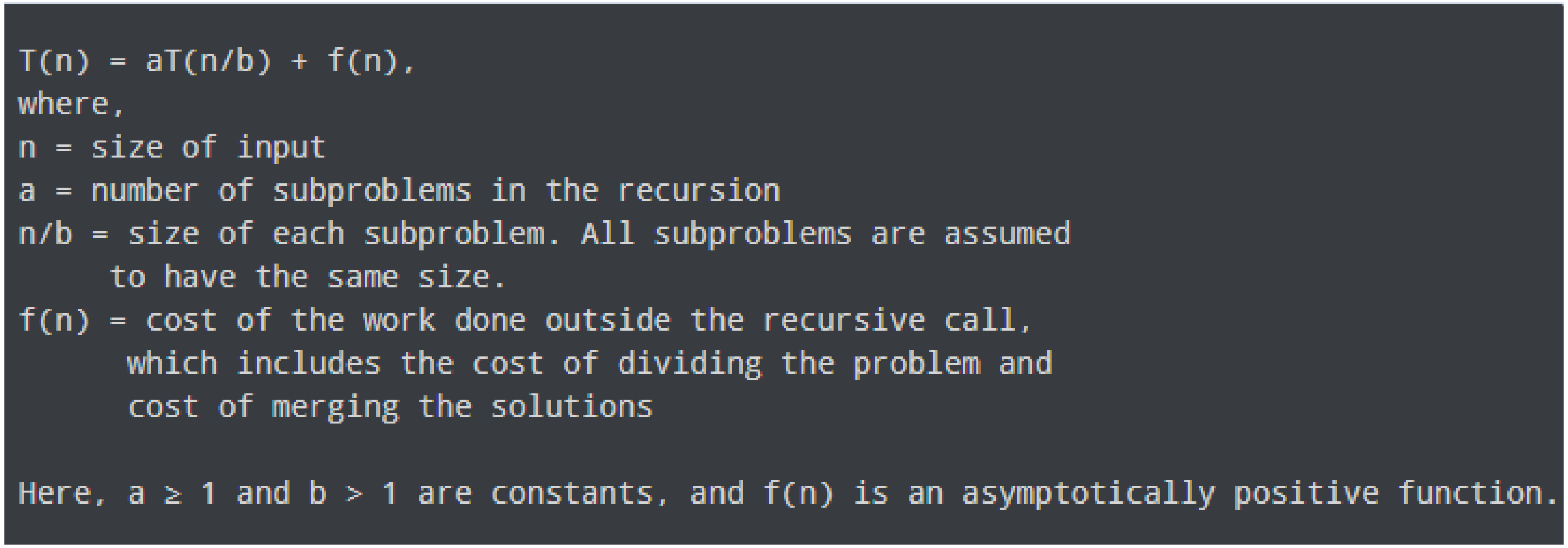


Total cost : $n + n + \ldots \geq \underbrace{\boldsymbol{n} + \boldsymbol{n} + \ \ldots + \boldsymbol{n}}_{\log_3 n + 1} = \boldsymbol{n}\,(\log_3 n + 1)$

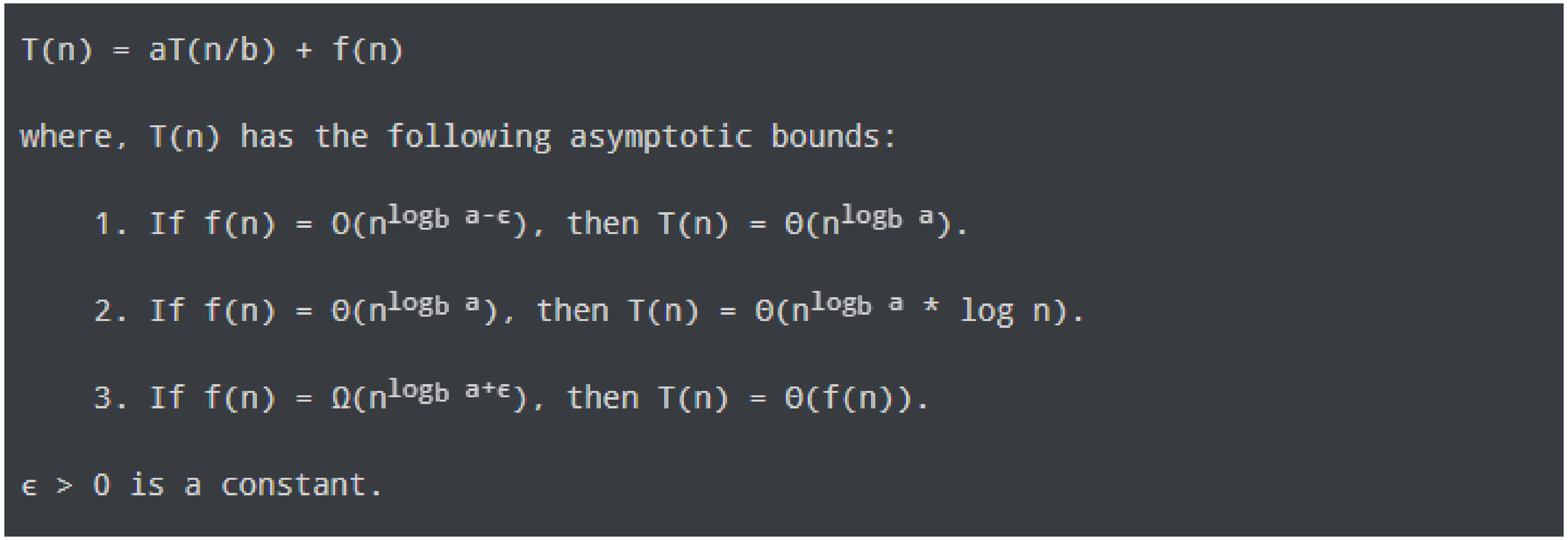
$\therefore T(n) = \Omega\,(n \log n) \quad (2)$

$\therefore (1) + (2) \Rightarrow$ T(n) = θ($n \log n$)

Master Method



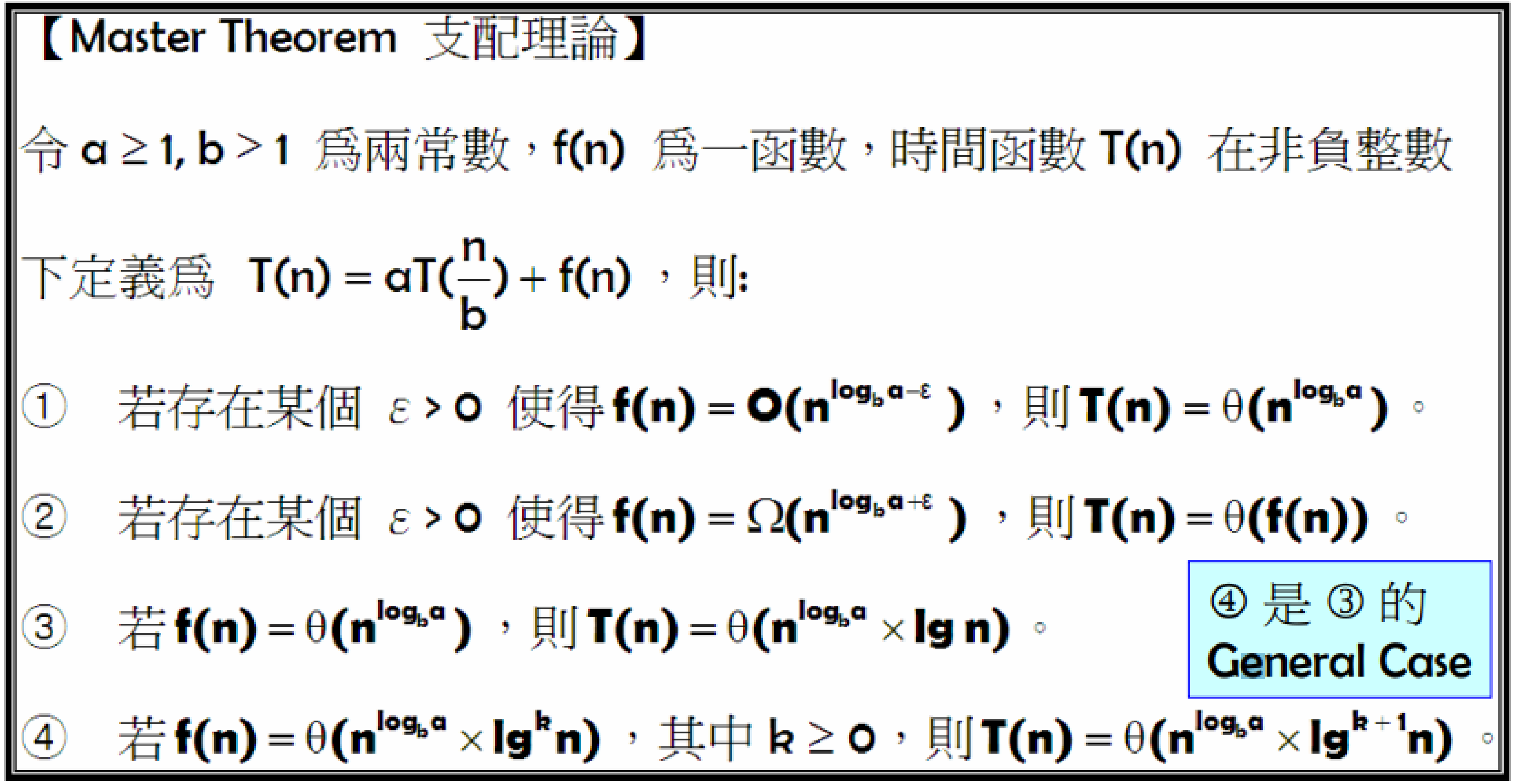
```
T(n) = aT(n/b) + f(n),
where,
n = size of input
a = number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed
     to have the same size.
f(n) = cost of the work done outside the recursive call,
      which includes the cost of dividing the problem and
      cost of merging the solutions

Here, a ≥ 1 and b > 1 are constants, and f(n) is an asymptotically positive function.
```



```
T(n) = aT(n/b) + f(n)

where, T(n) has the following asymptotic bounds:

    1. If f(n) = O($n^{\log_b a-\epsilon}$), then T(n) = Θ($n^{\log_b a}$).

    2. If f(n) = Θ($n^{\log_b a}$), then T(n) = Θ($n^{\log_b a}$ * log n).

    3. If f(n) = Ω($n^{\log_b a+\epsilon}$), then T(n) = Θ(f(n)).

ϵ > 0 is a constant.
```

Source : https://www.programiz.com/dsa/master-theorem

Master Method



【Master Theorem 支配理論】

令 $a \geq 1, b > 1$ 爲兩常數，f(n) 爲一函數，時間函數 T(n) 在非負整數

下定義爲 $T(n) = aT(\frac{n}{b}) + f(n)$ ，則:

① 若存在某個 $\varepsilon > 0$ 使得 $f(n) = O(n^{\log_b a - \varepsilon})$ ，則 $T(n) = \theta(n^{\log_b a})$ 。

② 若存在某個 $\varepsilon > 0$ 使得 $f(n) = \Omega(n^{\log_b a + \varepsilon})$ ，則 $T(n) = \theta(f(n))$ 。

③ 若 $f(n) = \theta(n^{\log_b a})$ ，則 $T(n) = \theta(n^{\log_b a} \times \lg n)$ 。

④ 若 $f(n) = \theta(n^{\log_b a} \times \lg^k n)$ ，其中 $k \geq 0$ ，則 $T(n) = \theta(n^{\log_b a} \times \lg^{k+1} n)$ 。

④ 是 ③ 的 General Case

1. $n^{\log_b a}$ 2. Compare $n^{\log_b a}$ and f(n)

Master Method

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

1. T(n) = 8T(n /2) + n^2 2 , T(n) = 4T(n/2) + n^2

1. a= 8, b=2, f(n) = n^2

$\Rightarrow\ n^{\log_b a} = n^{\log_2 8} = n^3$

$\Rightarrow\ \because n^2 \le\ n^{3-0.5}$

$\Rightarrow$ f(n) = O ($n^{\log_b a-\varepsilon}$) - 1

$\Rightarrow\therefore T(n) =\ \theta\left(n^{\log_b a}\right) = \theta(n^3)$

2. a= 4, b=2, f(n) = n^2

$\Rightarrow\ n^{\log_b a} = n^{\log_2 4} = n^2$

$\Rightarrow\ \because n^2 = n^2$

$\Rightarrow$ f(n) = O ($n^{\log_b a}$) - 3

$\Rightarrow\therefore T(n) =\ \theta\left(n^{\log_b a} \times \lg n\right) = \theta(n^2 lgn)$

Master Method

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

3. T(n) = 3T(n /2) + n^2 4. T(n) = 4T(n/2) + $n^2 lgn$

3. a= 3, b=2, f(n) = n^2

$\Rightarrow n^{\log_b a} = n^{\log_2 3} = n^{1.5}$

$\Rightarrow \because n^2 \geq n^{1.5+0.3}$

$\Rightarrow$ f(n) = Ω ($n^{\log_b a+\varepsilon}$) - 2

$\Rightarrow \therefore T(n) = \theta(\text{f}(n)) = \theta(2)$

4. a= 4, b=2, f(n) = $n^2 lgn$, k=1

$\Rightarrow n^{\log_b a} = n^{\log_2 4} = n^2$

$\Rightarrow \because n^2 \approx n^2 lgn$

$\Rightarrow$ f(n) =$\theta(n^{\log_b a} \times \lg^k n)$ - 4

$\Rightarrow \therefore T(n) = \theta\left(n^{\log_b a} \times \lg^k n\right) = \theta(n^2 lg^2 n)$

Substitution method

1. **Guess :**

   Guess the form of the solution

2. **Verify :**

   Verify by mathematical induction

3. **Solve:**

   Solve constants to show that the solution works

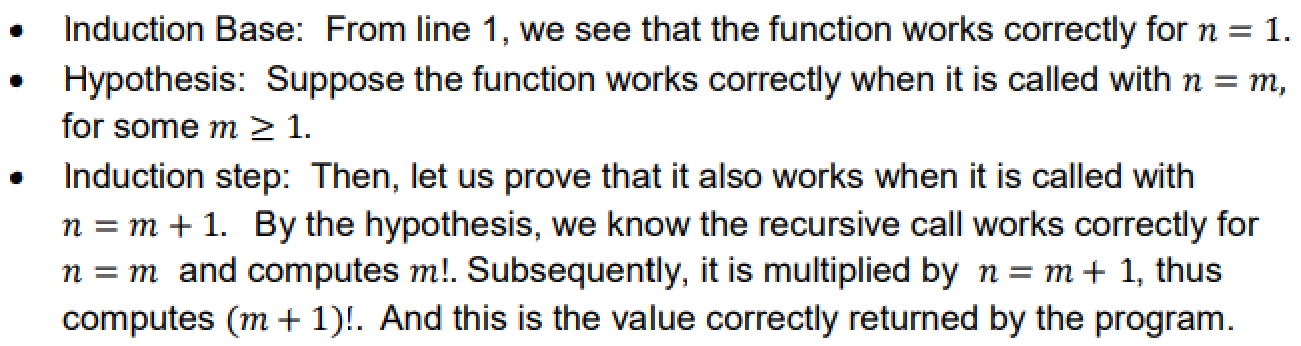
Substitution method

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

Proof:

1. $T(n) = n$

2. Verify by mathematical induction



- Induction Base: From line 1, we see that the function works correctly for $n = 1$.
- Hypothesis: Suppose the function works correctly when it is called with $n = m$, for some $m \geq 1$.
- Induction step: Then, let us prove that it also works when it is called with $n = m + 1$. By the hypothesis, we know the recursive call works correctly for $n = m$ and computes $m!$. Subsequently, it is multiplied by $n = m + 1$, thus computes $(m + 1)!$. And this is the value correctly returned by the program.

⇨ $T(0)=0$

⇨ $T(n)=n$

⇨ $T(n+1)$
$=n+1$

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

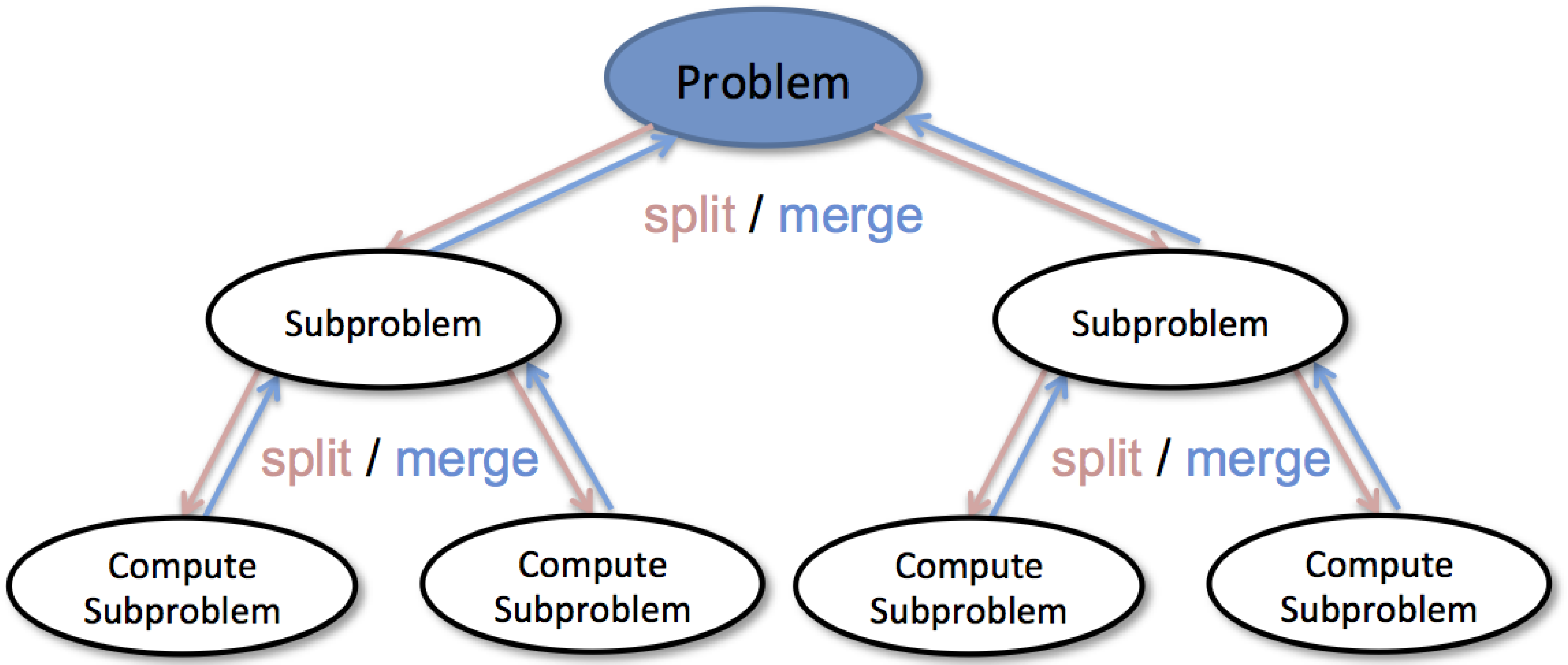
Objectives

- Describe the Divide-and-Conquer(D&C) strategy for solving problems(**what**)
- Apply the divide-and-conquer approach(**how**)
- Determine **when** to apply the divide-and-conquer strategy
- Complexity analysis of divide-and-conquer algorithms
- Contrast worst-case and average-case complexity analysis

The Military Tactic of "Divide and Conquer"

- The famous Battle of Austerlitz on December 2, 1805 between Napoleon and Austro-Russian coalition army.
- Napoleon's army was outnumbered by 15,000.
- Napoleon split the Austro-Russian army in two and conquer the smaller armies individually.
- Divide an instance of a hard problem into 2 or more smaller(easier) instances.
- Repeat the strategy until solvable instances.
- Top-down approach

Principle of Divide-and-Conquer



Advantage: concise and easy to understand
Disadvantage : Poor efficiency (Time complexity and space complexity)

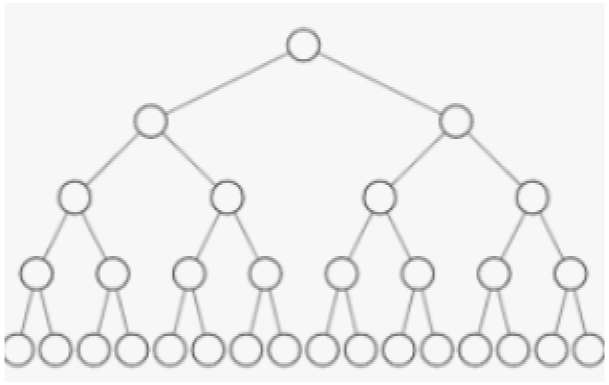
Determine when to apply the D &C strategy

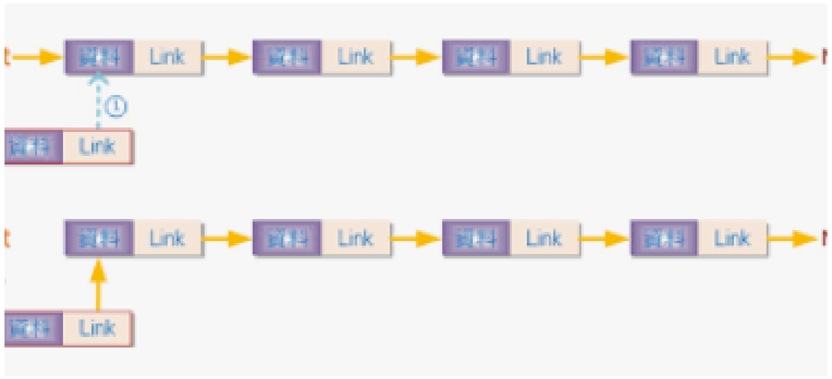
The problem is the recursive relationship

1. Binary search, Fibonacci sequence, Tower of Hanoi

The data structure belongs to the recursive definition

1. Binary tree, Link list

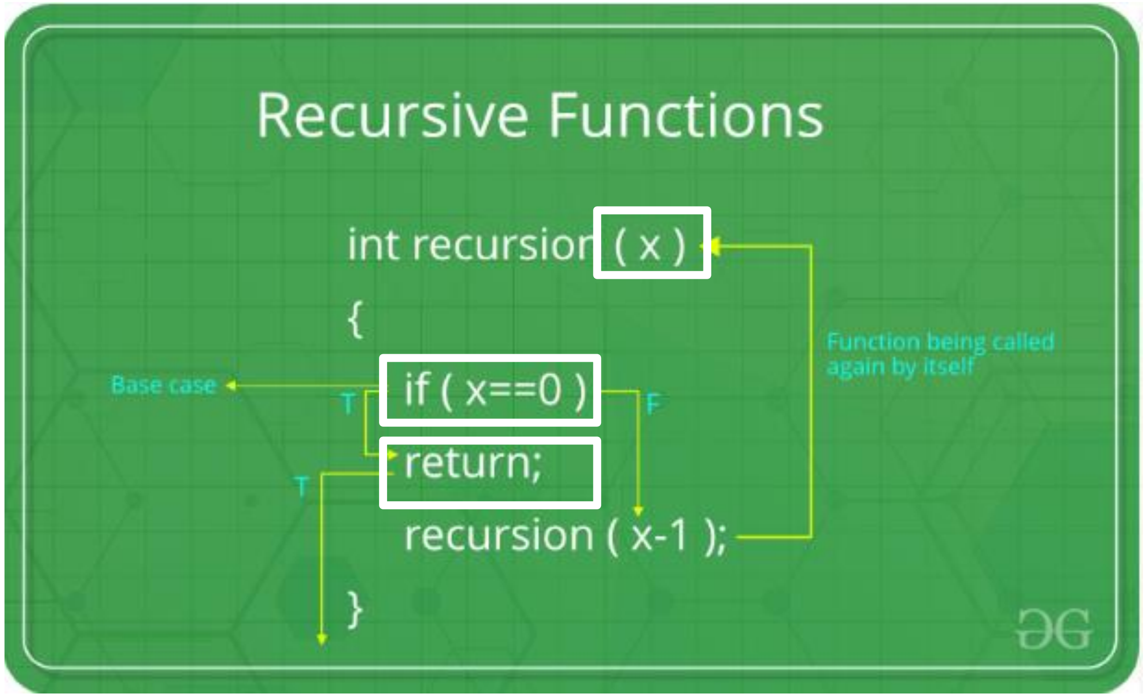




Designing Recursive Algorithms

Skill:

1. Termination condition
2. Recursive relationship

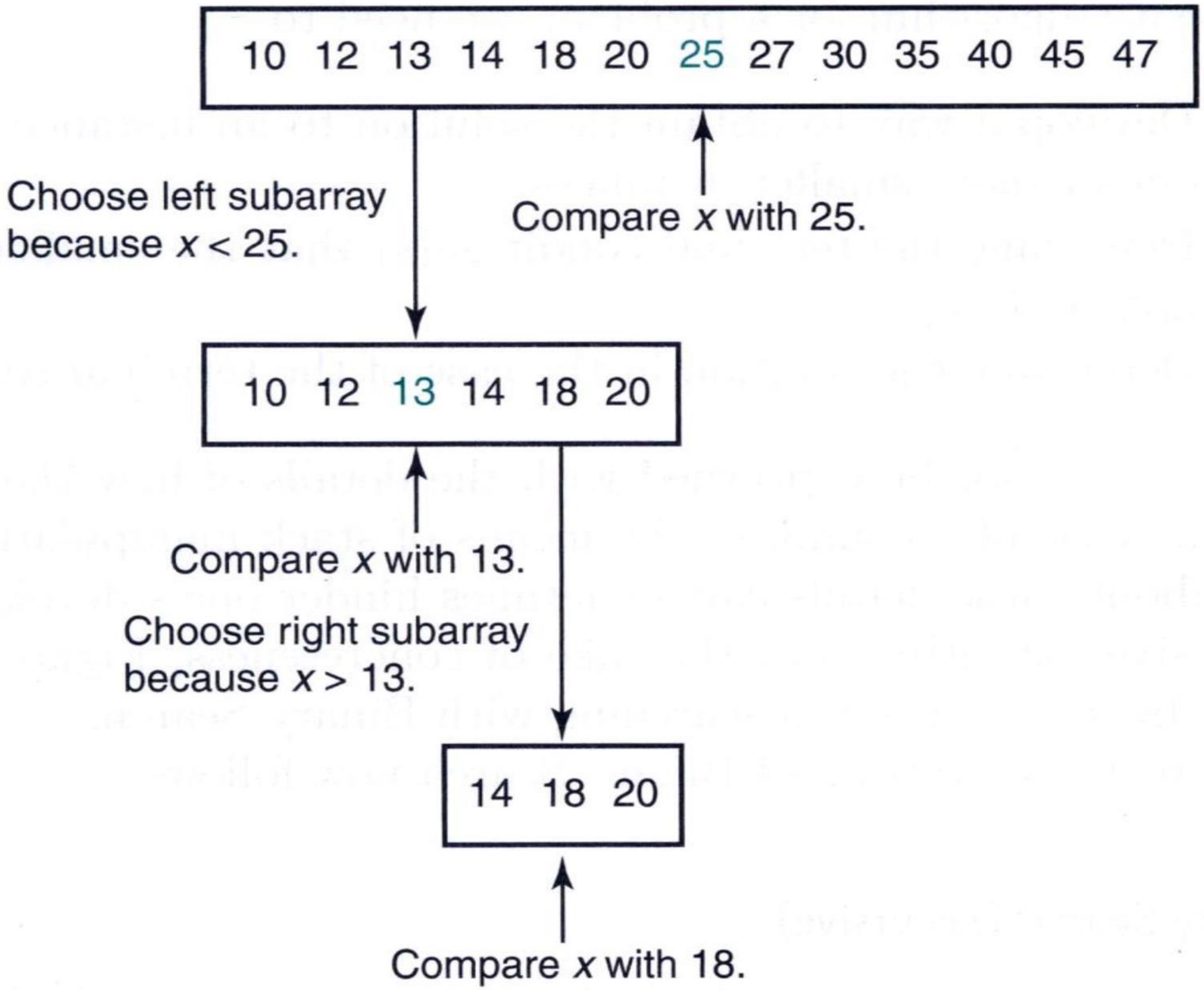


```
int recursion ( x )
{
    if ( x==0 )
    return;
    recursion ( x-1 );
}
```

Binary Search

- **Problem**: Locate key *x* in a sorted(non-decreasing) array of size *n*.
- If *x* equals the middle item *m* – found – quit. Else
  - *Divide* the array into two sub-arrays approximately in half
    - If *x* is smaller than *m*, select left sub-array
    - If *x* is larger than *m*, select right sub-array
  - *Conquer* (solve) the sub-array: Is *x* in the sub-array using recursion until the sub-array is sufficiently small (can be solved directly).
  - *Obtain* the solution to the array from the solution to the subarray.

Binary Search Example (x = 18)



Binary Search (Recursive)

```
index location (index low, index high) {
  index mid;

  if (low > high)
    return 0;  // Not found.
  else {
    mid = (low + high) / 2  // Integer div. Split in half.
    if (x == S[mid])
      return mid;  //  Found.
    else if (x < S[mid])
      return location(low, mid-1);   // Choose the left half.
    else
      return location(mid+1, high);  // Choose the right half.
  }
}
// Call as follows:  locationout = location(1, n);
```

Observations

- Reason for using a local variable *locationout*
  - Parameters n, S, x, are not changed during execution.
  - Dragging them over recursive calls are unnecessary.
- Tail-recursion
  - No operations are done after the recursive call.
  - Straightforward to produce an iterative version.
  - Recursion clearly illustrates the D&C process.
  - However, recursions is overburdensome due to excessive uses of activation records.
  - Memory can be saved by eliminating the stack for activation records. (reason for preferring to iteration)
  - Iterative version is better only as constant factor. Order is same.

Worst-Case Analysis (Binary Search)

- **Baisc operation**: compare x with S[n]
- **Input size**: n, the number of items in S
- **Analysis**:
  - Let *n* be a power of 2. Worst case occurs when *x* > S[n].
  - W(n) = W(n/2) + 1 for n>1 and n power of 2
    - W(n/2) = the no. of comparisons in the recursive call
    - 1 comparison at the top level
  - W(1) = 1
  - Example B1 in Appendix B: W(n) = lg n + 1
  - If n not a power of 2
    - $W(n) = \lfloor \lg n \rfloor + 1 \in \theta(\lg n)$

Worst-Case Analysis (Binary Search)

- Time function

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

$$\boxed{\frac{n}{2^i} = 1 \Rightarrow n = 2^i \Rightarrow \log_2 n = i}$$

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

$$=(T\left(\frac{n}{4}\right) + c)+c = T\left(\frac{n}{4}\right) + 2c$$

$$= (T\left(\frac{n}{8}\right) + c) + 2c = T\left(\frac{n}{8}\right) + 3c$$

$$= \mathrm{T}\,\boxed{(n/2^i)} + i * c = \mathrm{T}\,(1) + (\log_2 n) * c$$

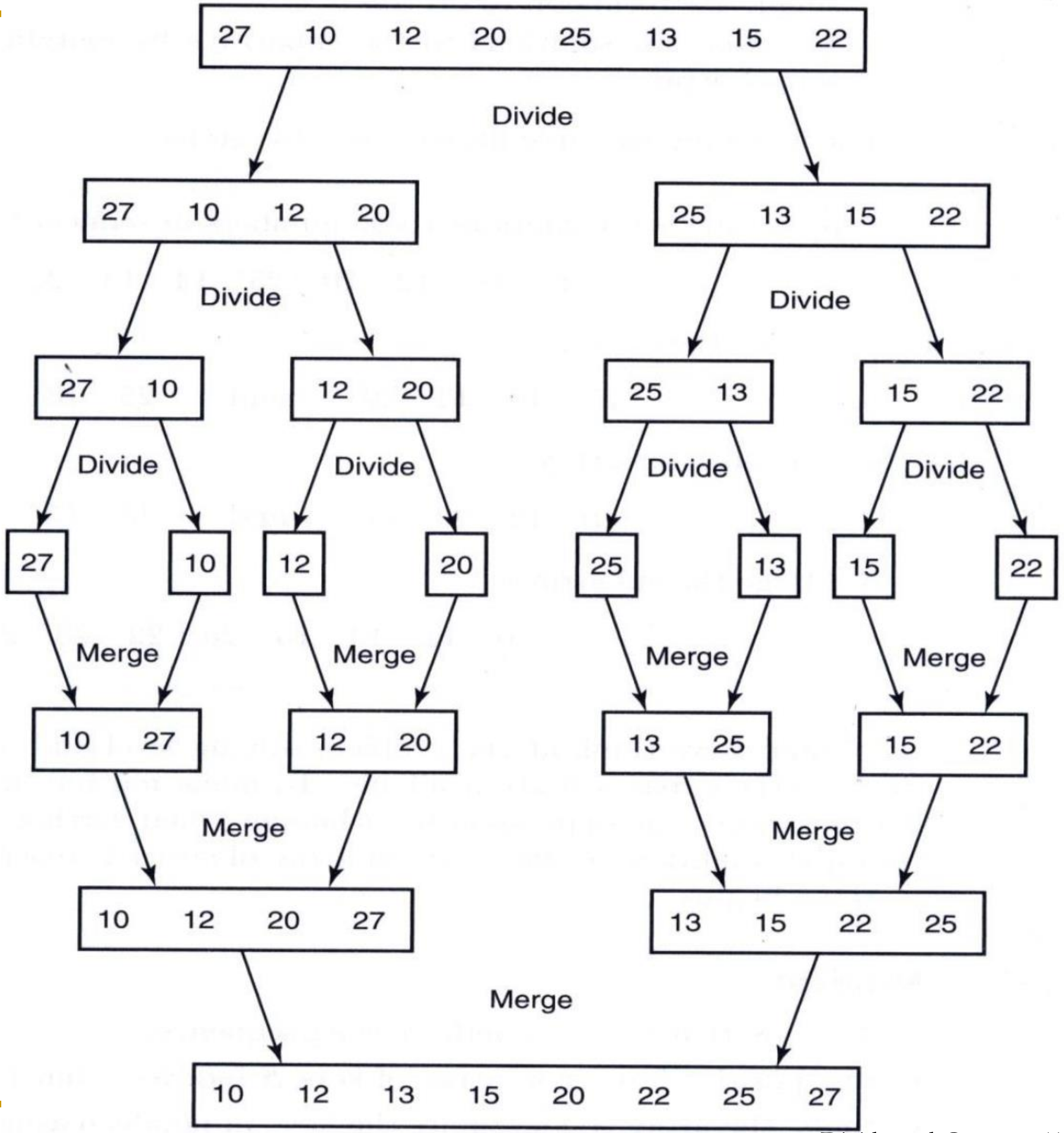
$$= 1 + c\log_2 n$$

$$\therefore T(n) = O(\log_2 n) = O\,(lg\,n)$$

Mergesort (Recursive)

- **Problem**: Sort an array S of size n (for simplicity, let n be a power of 2)
- *Divide* S into 2 sub-arrays of size n/2
- *Conquer* (solve) recursively sort each sub-array until array is sufficiently small (size 1)
- *Combine* (merge) the solutions to the sub-arrays into a single sorted array.

Mergesort Example



Mergesort: the merge Function

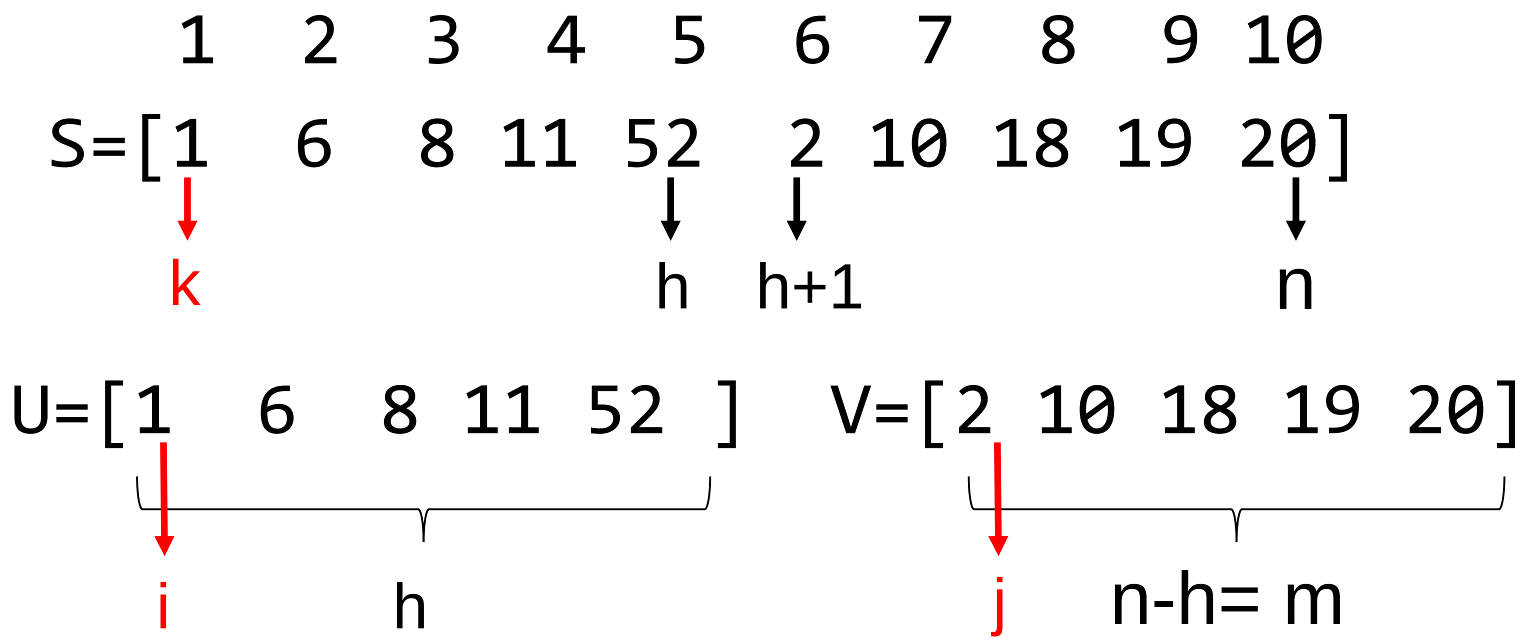
```
void merge(int h, int m, const keytype U[],
           const keytype V[], keytype S[]) {
  index i = 1 , j = 1, k = 1;
  while (i <= h && j <= m) {
    if (U[i] < V[j]) { S[k] = U[i]; i++; }
    else { S[k] = V[j]; j++; }
    k++;
  }
  if (i > h)
    copy V[j] ~ V[m] to S[k] ~ S[h+m];
  else
    copy U[i] ~ U[h] to S[k] ~ S[h+m];
}
```

Mergesort Algorithm

```
void mergesort(int n, keytype S[])
{
  if (n>1) {
    const int h=⌊n/2⌋, m = n - h;
    keytype U[1..h], V[1..m];
    copy S[1] ~ S[h] to U[1] ~ U[h];
    copy S[h+1] ~ S[n] to V[1] ~ V[m];
    mergesort(h, U);
    mergesort(m, V);
    merge(h, m, U, V, S);
  }
}
```

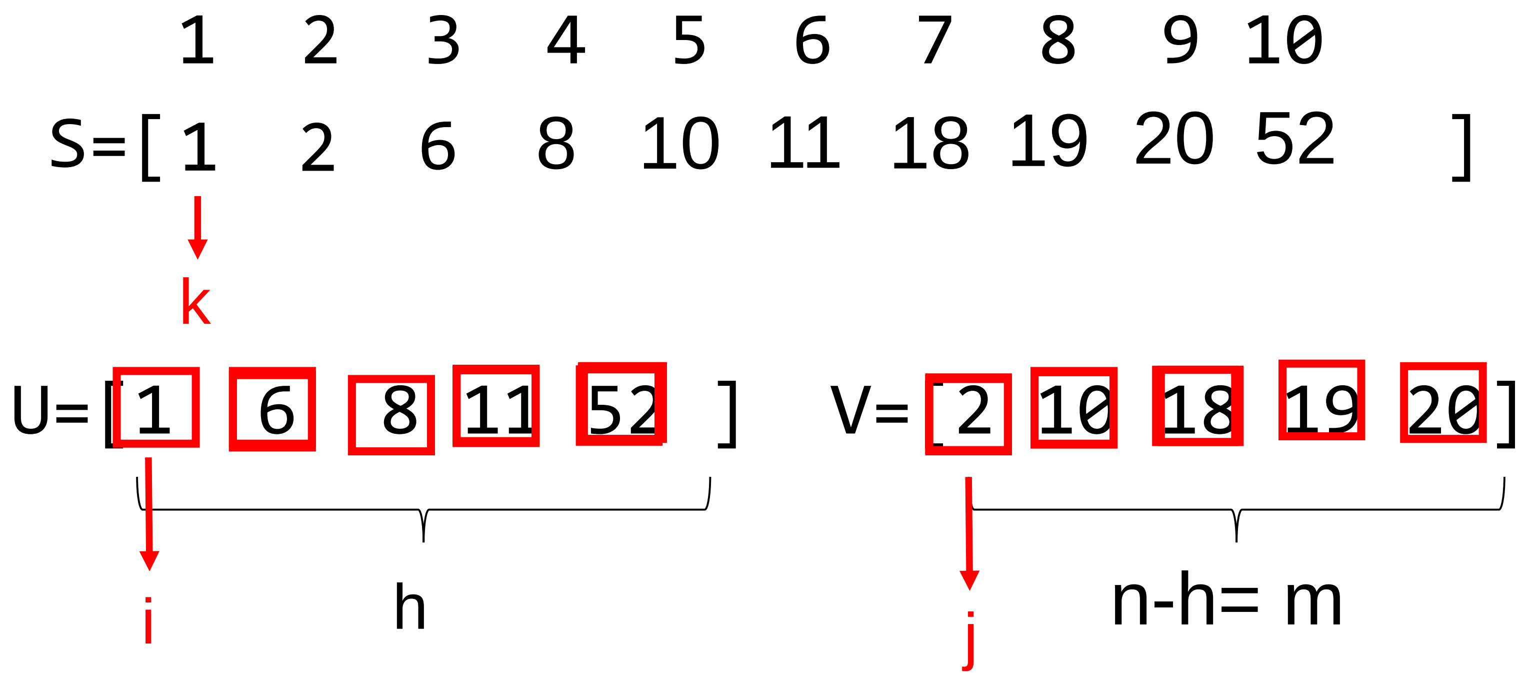
Mergesort: the `merge` Function

```
int i, j , k , U[], S[] , V[]
```



Mergesort: the `merge` Function

```
int i, j , k , U[], S[] , V[]
```



Mergesort Analysis

- Merges the two arrays U and V created by the recursive calls to mergesort
- **Input size**
  - h the number of items in U
  - m the number of items in V
- **Basic operation**: Comparison of U[I] to V[j]
- Worst case:
  - Loop exited with one index at exit point and the other one less than the exit point

Worst-Case Analysis (Mergesort) 1

- W(n) = time_sort_U + time_sort_V + time_merge
- $W(n) = W(h) + W(m) + h+m-1$
- First analysis assumes n is a power of 2
    - $h = \lfloor n/2 \rfloor = n/2$
    - $m = n – h = n – n/2 = n/2$
    - $h + m = n/2 + n/2 = n$
- $W(n) = W(n/2) + W(n/2) + n – 1 = 2W(n/2) + n-1$
- $W(1) = 0$
- From B19 in Appendix B
    - $W(n)=n \lg n - (n-1) \in \theta(n \lg n)$

Worst-Case Analysis (Mergesort) 2

- If n is not a power of 2
- $W(n)=W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n-1$
- From Theorem B4: $W(n) \in \theta(n \lg n)$

Space Analysis (Mergesort)

- New arrays U and V will be created when *mergesort* is called.
- The total number of extra array items created is

$$n + \frac{n}{2} + \frac{n}{4} + \cdots = 2n$$

- In other words, the space complexity is 2n $\in \theta(n)$
- We may reduce the extra space to *n.* (Read textbook on Mergesort 2, Algorithm 2.4)
- But it is not possible to make mergesort algorithm to be an in-place sort.

Divide-and-Conquer Strategy

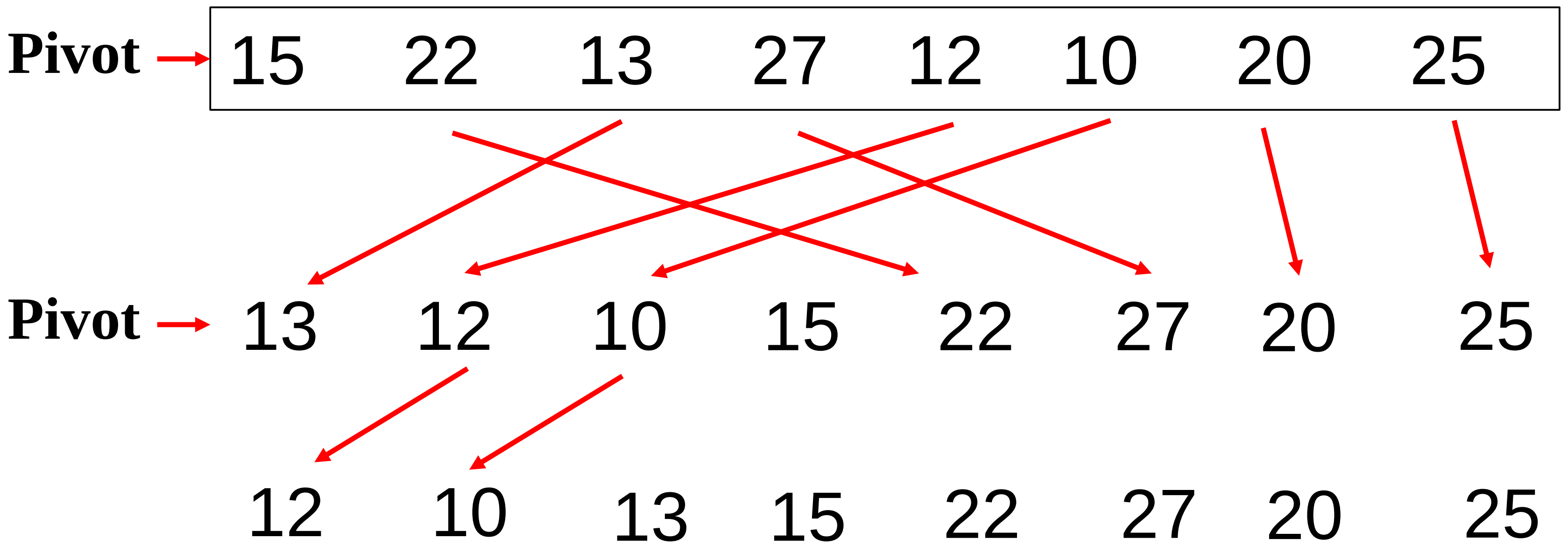
1. ***Divide*** an instance of a problem into one or more smaller instances.
2. ***Conquer*** (solve) each of the smaller instances. Unless a smaller instance is sufficiently small, use recursion to do this.
3. If necessary, ***combine*** the solutions to the smaller instances to obtain the solution to the original instance.

- One of the most widely used design stragety.

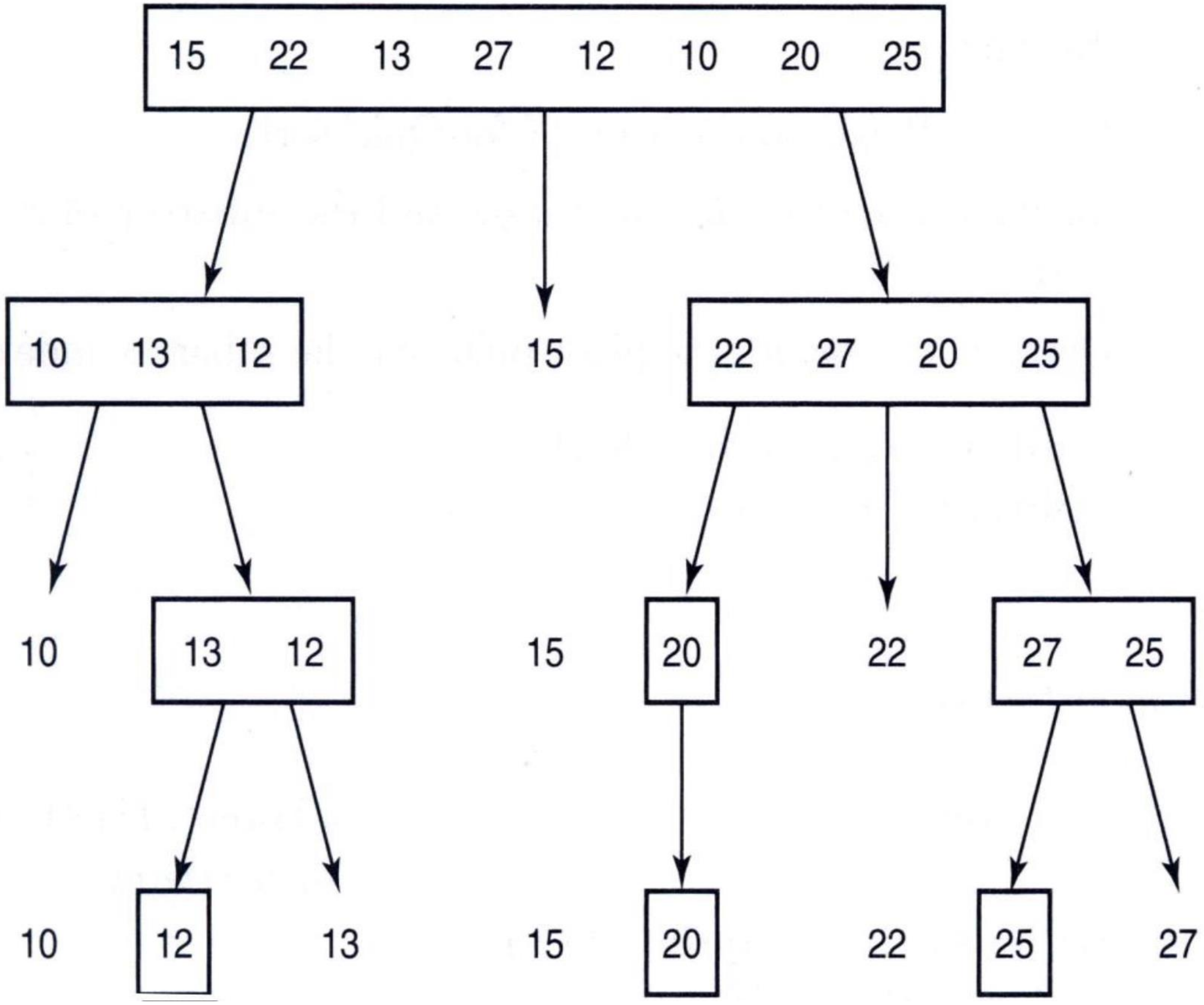
Quicksort

- Array recursively divided into two partitions and recursively sorted.
- Division based on a pivot.
- The pivot divides the two sub-arrays.
- All items < pivot placed in sub-array before pivot.
- All items >= pivot placed in sub-array after pivot.

Quicksort Example 1



Quicksort Example 2



Quicksort Algorithm

```
void quicksort(index low, index high) {
   index pivotpoint;

   if (high > low){
      partition(low, high, pivotpoint);
      quicksort(low, pivotpoint - 1);
      quicksort(pivotpoint + 1, high);
   }
}
```

The partition Function

```
void partition(index low, index high,
               index& pivotpoint) {
  index i, j;
  keytype pivotitem;

  pivotitem = S[low];  // Choose 1st item as pivot
  j = low;
  for (i = low + 1; i <= high; i++)
    if (S[i] < pivotitem) {
      j++;
      swap S[i] and S[j];
    }
  pivotpoint = j;
  swap S[low] and S[pivotpoint];  // Place pivotitem
}
```

Every-Case Analysis (Partition)

- **Baisc operation**: Comparison of S[i] with pivotitem
- **Input size**: n = high – low + 1, no. items in subarray
- **Analysis**:
  - Every item except the first is compared.
  - $T(n) = n - 1$

Worst-Case Analysis (Quicksort) 1

- Occurs when the array is already sorted in non-decreasing order.
- The pivot(1st item) is always the smallest.
- Array is repeatedly sorted into an empty sub-array which is less than the pivot and a sub-array of n-1 containing items greater than pivot.
- If there are k keys in the current sub-array, k-1 key comparisons are executed.

Worst-Case Analysis (Quicksort) 2

- T(n) is used because analysis is for the every-case complexity for the class of instances already sorted in non-decreasing order
- T(n) = time to sort left sub-array + time to sort right sub-array + time to partition
- $T(n) = T(0) + T(n-1) + n – 1$
- $T(n) = T(n – 1) + n – 1$ for $n > 0$
- $T(0) = 0$
- From B16
  - $T(n) = n(n-1)/2$

Worst-Case Analysis (Quicksort) 3

- From T(n) above, we know that worst-case is at least n(n-1)/2.
- By induction, we can show it is the worst case
  - $W(n) = n(n-1)/2 \in \theta(n^2)$

Average-Case Analysis (Quicksort)1

- Value of pivotpoint is equally likely to be any of the numbers from 1 to n.
- The probability for the pivot position to be the p-th is 1/n.
- The average time to sort if the pivot position is the p-th is [A(p-1) + A(n-p)] and the time to partition is n-1.
- Therefore, the average time complexity is

$$A(n) = \sum_{p=1}^{n} \frac{1}{n}[A(p-1) + A(n-p)] + n - 1$$

$$= \frac{2}{n}\sum_{p=1}^{n} A(p-1) + n - 1$$

Average-Case Analysis (Quicksort)2

$$nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1) \quad (1)$$

$$(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2) \quad (2)$$

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$$

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$

$$a_n = \frac{A(n)}{n+1} \qquad a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \qquad n > 0$$

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \quad a_{n-1} = a_{n-2} + \frac{2(n-2)}{(n-1)n} \quad a_2 = a_1 + \tfrac{1}{3} \quad a_1 = a_0 + 0$$

Average-Case Analysis (Quicksort)3

$$a_n = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)}$$

$$= 2\left(\sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)}\right) \approx 2\ln n$$

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n$$

$$\begin{aligned} A(n) &\approx (n+1)2\ln n \\ &= (n+1)2(\lg n)/(\lg e) \\ &\approx 1.38(n+1)\lg n \\ &\in \Theta(n\lg n) \end{aligned}$$

Matrix Multiplication Problem

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

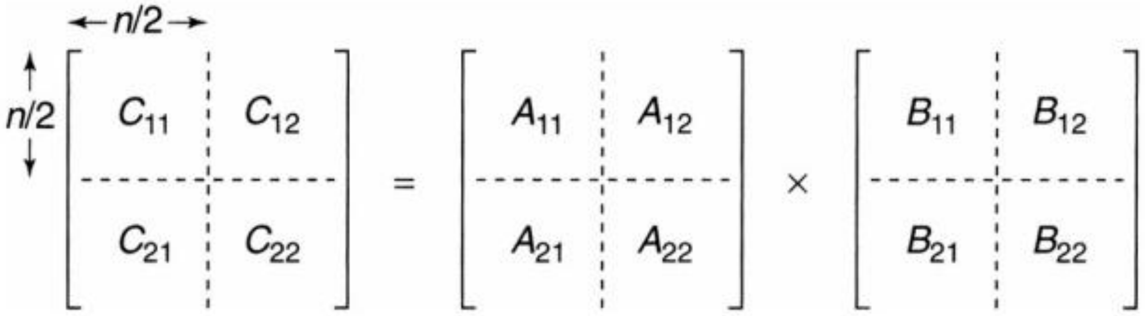
$c_{11} = a_{11} \times b_{11} + a_{12} \times b_{21}$

$c_{12} = a_{11} \times b_{12} + a_{12} \times b_{22}$

$c_{21} = a_{21} \times b_{11} + a_{22} \times b_{21}$

$c_{22} = a_{21} \times b_{12} + a_{22} \times b_{22}$

$$\boxed{c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad \text{for } 1 \le i, j \le n.}$$



$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21}$

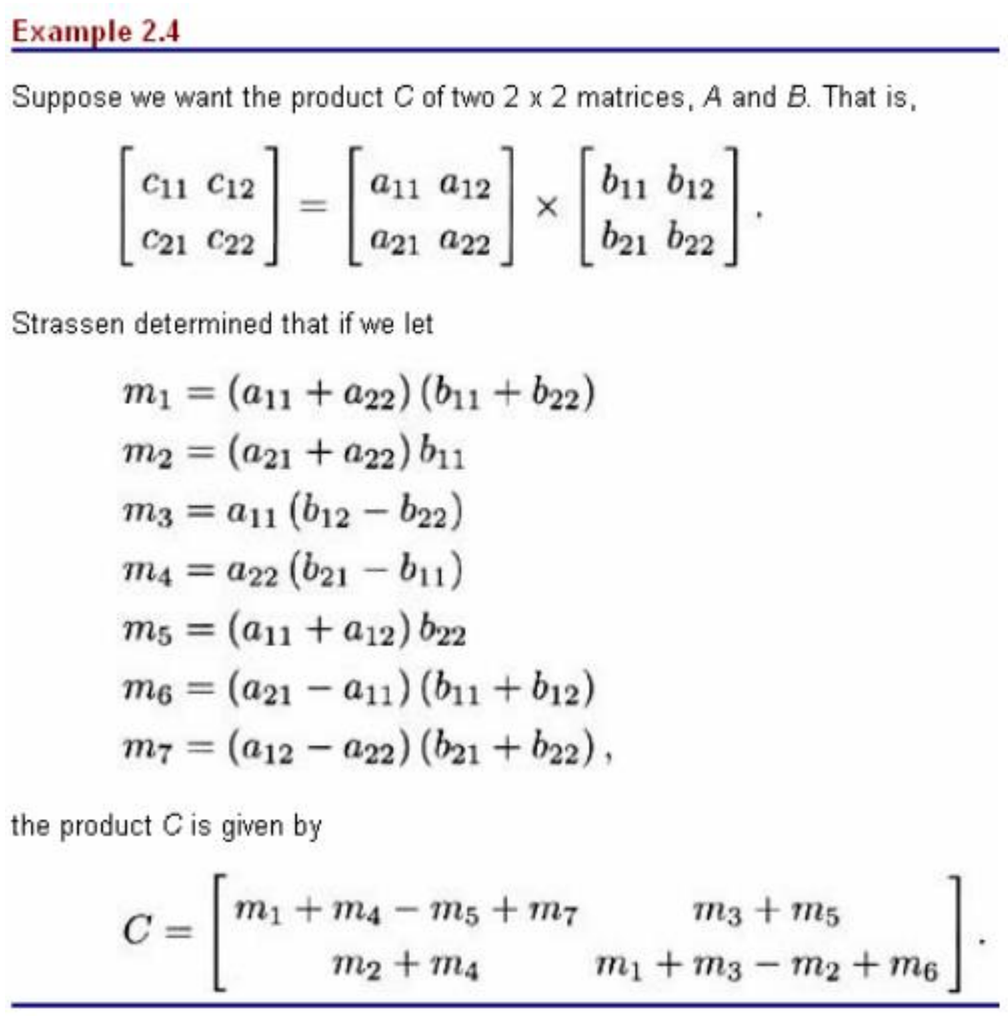
$C_{12} = A_{11} \times B_{12} + A_{12} \times B_{22}$

$C_{21} = A_{21} \times B_{11} + A_{22} \times B_{21}$

$C_{22} = A_{21} \times B_{12} + A_{22} \times B_{22}$

$T(n) = 8T(n/2) + 4 \times (\frac{n}{2})^2$ Based on Master Method : $T(n) = (n^3\ )$

Matrix Multiplication Problem



Example 2.4

Suppose we want the product *C* of two 2 x 2 matrices, *A* and *B*. That is,

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Strassen determined that if we let

$$\begin{aligned} m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ m_2 &= (a_{21} + a_{22})\, b_{11} \\ m_3 &= a_{11}(b_{12} - b_{22}) \\ m_4 &= a_{22}(b_{21} - b_{11}) \\ m_5 &= (a_{11} + a_{12})\, b_{22} \\ m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\ m_7 &= (a_{12} - a_{22})(b_{21} + b_{22}), \end{aligned}$$

the product *C* is given by

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}.$$

$$T(n) = 7T(n/2) + 18\left(\frac{n}{2}\right)^2$$

Maximum Subarray Sum Problem

- **Problem**: Given an array A[1…n] of integers (positive and negative), compute a subarray A[i*… j*] with maximum sum.
- If s(i,j) denotes the sum of the elements of a subarray A[i…j], that is $$s(i,j) = \sum_{k=i}^{j} A[k]$$
- We want to compute indices i* ≤ j* such that
$$s(i^*, j^*) = \max\{s(i,j) \mid 1 \le i \le j \le n\}$$
- Example: $3\quad -4\quad \underbrace{5\quad -2\quad -2\quad 6\quad -3\quad 5}_{\text{max sum = 9}}\quad -3\quad 2$

Brute Force Solution

- Compute the sum of every subarray and pick the maximum.
- Try every pair of indices i, j with $1 \leq i \leq j \leq n$, and for each one compute s(i, j).
- Time complexity: $\theta(n^3)$. (why?)
- With a little more care, can improve to $\theta(n^2)$: can compute the sums of all the subarrays in time $\theta(n^2)$.

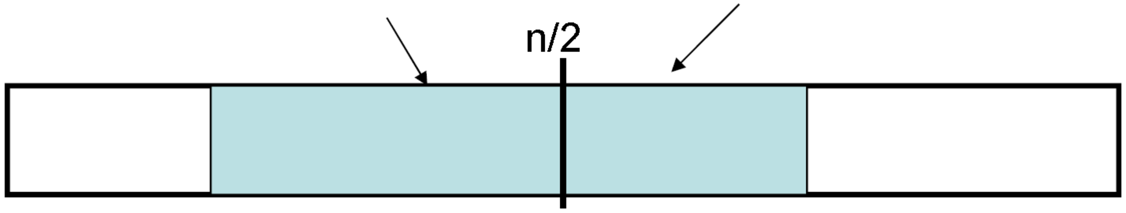
Brute Force Solution (improved)

- With a little more care, can improve to $\theta(n^2)$.
- Can compute the sums for all subarrays with same left end in O(n) time $\Rightarrow$ compute the sums of all the subarrays (there are n(n-1)/2 + n subarrays) in time $O(n^2)$.

```
for i = 1 to n {
   s(i,i)=A[i];
   for j = i+1 to n
      s(i,j) = s(i,j-1)+A[j];
}
```

Divide-and-Conquer Solution

- A subarray A[i*…j*] with maximum sum is
  - Either contained entirely in the left half , i.e. j* ≤ n/2
  - Or contained entirely in the right half , i.e. i* ≥ n/2
  - Or cross the mid element: i* ≤ n/2 ≤ j*
- We can compute the best subarray of the first two types with recursive calls.
- The best subarray of the third type consists of the best subarray that ends at n/2 and the best subarray that starts at n/2. We can compute these in O(n) time.



Maximum Subarray Sum Algorithm

- It is a good exercise to write the pseudo code for the D&C solution.
- Also quite easy to convert the pseudo code into any programming language.
- Do it by yourself first w/o searching the Internet.
- Then search for a solution on the Internet.
- Compare your solution with the found one.

Worst-Case Analysis

- $W(n) = 2W(n/2) + O(n)$
- $W(n) \in O(n \log n)$

- It is possible to do even better: can compute the maximum subarray sum in O(n) time. (exercise)
- *Note: Not divide and conquer.*

Advantages of Divide-and-Conquer

- **Solving difficult problems**
- **Algorithm efficiency**: often help in the discovery of efficient algorithms.
- **Parallelism**: D&C algorithms can be easily executed on parallel machines.
- **Memory access**: D&C algorithms tend to make efficient use of memory caches.
- **Widely applicable**: D&C turns out to be a good strategy for many different problems. (Try to identify the D&C algorithms in Lecture 1)

When Not to Use Divide-and-Conquer

- Avoid using D&C in the following two cases:
  - An instance of size *n* is divided into two or more instances each almost of size n.
  - An instance of size *n* is divided into almost *n* instances of size *n/c*, where *c* is a constant.
- The first type leads to an exponential-time algorithm.
- The second type leads to an $n^{\theta(\lg n)}$ algorithm.
- If Napoleon did this, he would have met his Waterloo much sooner.

Assignment 2: Divide-and-Conquer

1. Design and implement the improved and D&C version of the maximum subarray sum algorithm.
2. The Closest Pair of Points problem is to find the closest pair of points in a set of points in x-y plane. Design and implement a D&C algorithm to solve the problem.
3. Textbook exercises: 2.6, 2.7, 2.13

Due date: two weeks.