Divide & Conquer (1)

Sep 27th, 2018

Algorithm Design and Analysis

YUN-NUNG (VIVIAN) CHEN HTTP://ADA.MIULAB.TW

國立臺灣大學 National Taiwan University

Slides credited from Hsueh-I Lu, Hsu-Chun Hsiao, & Michael Tsai

Announcement

- Mini-HW 2 released
  - Due on 10/4 (Thu) 14:20
  - Submit to NTU COOL before the lecture starts
- Homework 1 released
  - Due on 10/18 (Thur) 14:20 (total 3 weeks)
  - Writing: print out the A4 hard copy and submit to NTU COOL before the lecture starts
  - Programming: submit to Online Judge – http://ada18-judge.csie.org
    - Account and password were sent

程式的參考資料請加在註解中

Mini-HW 2

$$T(n) = \begin{cases} 4T(\frac{n}{4}) + \frac{n}{\lg n}, & n > 1 \\ 1, & n \leq 1 \end{cases}$$

Prove or disprove that $T(n) = O(n \log^2 n)$.

Homework 1

Homework #1

Due Time: 2018/10/18 (Thu.) 14:20
Contact TAs: `ada-ta@csie.ntu.edu.tw`

Instructions and Announcements

- There are **three programming problems** and **two hand-written problems**.
- **Programming.** The judge system is located at `https://ada18-judge.csie.org`. Please login and submit your code for the programming problems (i.e., those containing "Programming" in the problem title) by the deadline. NO LATE SUBMISSION IS ALLOWED.
- **Hand-written.** For other problems (also known as the "hand-written problems"), please turn in **a printed/written version** of your answers to the instructor at the beginning of the class. In case that your homework is lost during the grading, you can also upload your homework to the NTU COOL system. NO LATE SUBMISSION IS ALLOWED.
- **Collaboration policy.** Discussions with others are strongly encouraged. However, you should write down your solutions **in your own words**. In addition, for **each and every** problem you have to specify the references (e.g., the Internet URL you consulted with or the people you discussed with) on the first page of your solution to that problem. You may get zero point due to the lack of references.

Algorithm Design Strategy

- Do not focus on “specific algorithms”
- But “some strategies” to “design” algorithms

- First Skill: Divide-and-Conquer (各個擊破)

Outline

- Recurrence (遞迴)
- Divide-and-Conquer
- D&C #1: Tower of Hanoi (河內塔)
- D&C #2: Merge Sort
- D&C #3: Bitonic Champion
- D&C #4: Maximum Subarray
- Solving Recurrences
  - Substitution Method
  - Recursion-Tree Method
  - Master Method
- D&C #5: Matrix Multiplication
- D&C #6: Selection Problem
- D&C #7: Closest Pair of Points Problem

Divide-and-Conquer 首部曲

Divide-and-Conquer 之神乎奇技

What is Divide-and-Conquer?

- Solve a problem recursively
- Apply three steps at each level of the recursion
  1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem (比較小的同樣問題)
  2. **Conquer** the subproblems by solving them recursively
     If the subproblem sizes are *small enough*
     - then solve the subproblems — base case
     - else recursively solve itself — recursive case
  3. **Combine** the solutions to the subproblems into the solution for the original problem

Divide-and-Conquer Benefits

- Easy to solve difficult problems
  - Thinking: solve easiest case + combine smaller solutions into the original solution
- Easy to find an efficient algorithm
  - Better time complexity
- Suitable for parallel computing (multi-core systems)
- More efficient memory access
  - Subprograms and their data can be put in cache in stead of accessing main memory

9 Recurrence (遞迴)

Recurrence Relation

- Definition

  A ***recurrence*** is an equation or inequality that describes a function in terms of its value on smaller inputs.

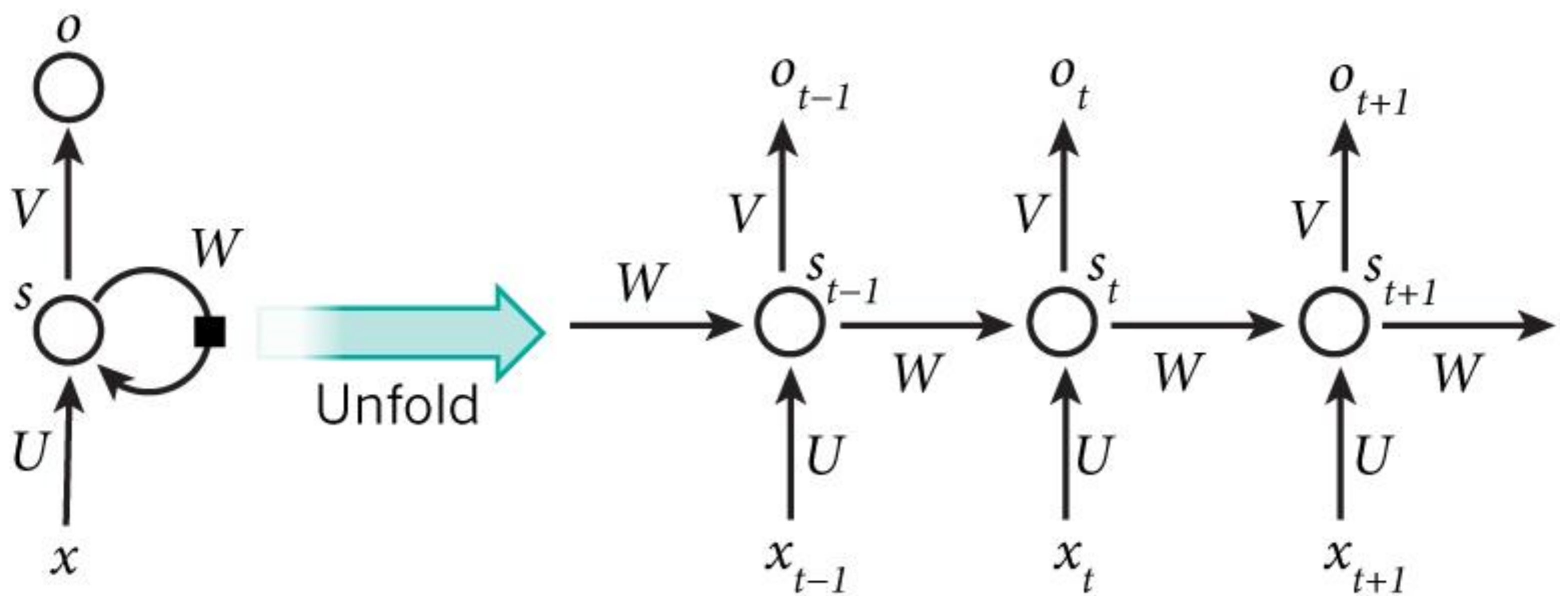
- Example

  Fibonacci sequence (費波那契數列)
  - Base case: F(0) = F(1) = 1
  - Recursive case: F(n) = F(n-1) + F(n-2)

n	0	1	2	3	4	5	6	7	8	…
F(n)	1	1	2	3	5	8	13	21	34	…

Recurrent Neural Network (RNN)

$$s_t = \sigma(W s_{t-1} + U x_t)$$
$$o_t = \mathrm{softmax}(V s_t)$$



http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/

Recurrence Benefits

- Easy & Clear
  - Define base case and recursive case
  - Define a long sequence

Base case
Recursive case

F(0), F(1), F(2)..............
unlimited sequence

a program for solving F(n)

```
Fibonacci(n) // recursive function:程式中會呼叫自己的函數
  if n < 2 // base case: termination condition
    return 1

  // recursive case: call itself for solving subproblems
  return Fibonacci(n-1) + Fibonacci(n-2)
```

important otherwise the program cannot stop

Recurrence v.s. Non-Recurrence

```
Fibonacci(n)
    if n < 2 // base case
        return 1
    // recursive case
    return Fibonacci(n-1) + Fibonacci(n-2)
```

Recursive function
- Clear structure 👍
- Poor efficiency 👎

```
Fibonacci(n)
    if n < 2
        return 1
    a[0] <- 1
    a[1] <- 1
    for i = 2 … n
        a[i] = a[i-1] + a[i-2]
    return a[n]
```

Non-recursive function
- Better efficiency 👍
- Unclear structure 👎

Recurrence Benefits

- Easy & Clear
  - Define base case and recursive case
  - Define a long sequence

Base case
Recursive case

F(0), F(1), F(2)...............
unlimited sequence

a program for solving F(n)

If a problem can be simplified into a **base case** and a **recursive case**, then we can find a algorithm that solves this problem.

Hanoi(n) is not easy to solve.
✓ It is easy to solve when *n* is small
✓ we can find the relation between Hanoi(n) & Hanoi(n-1)

Base case
Recursive case

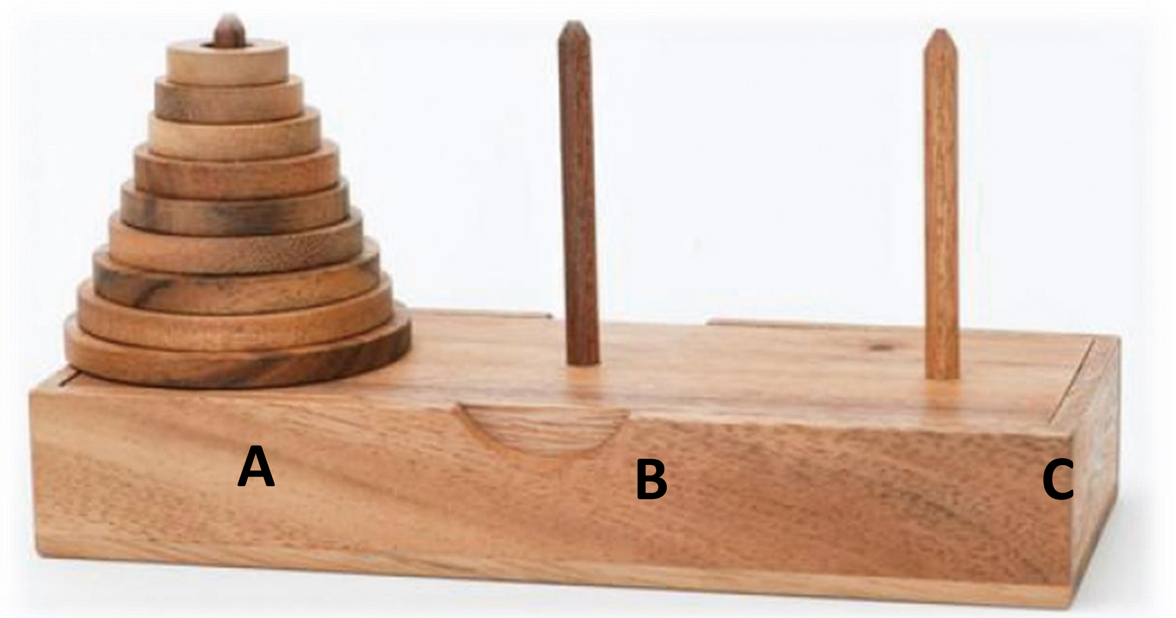
a program for solving Hanoi(n)

15

D&C #1: Tower of Hanoi

Tower of Hanoi (河内塔)

- Problem: move *n* disks from A to C
- Rules
  - Move one disk at a time
  - Cannot place a larger disk onto a smaller disk

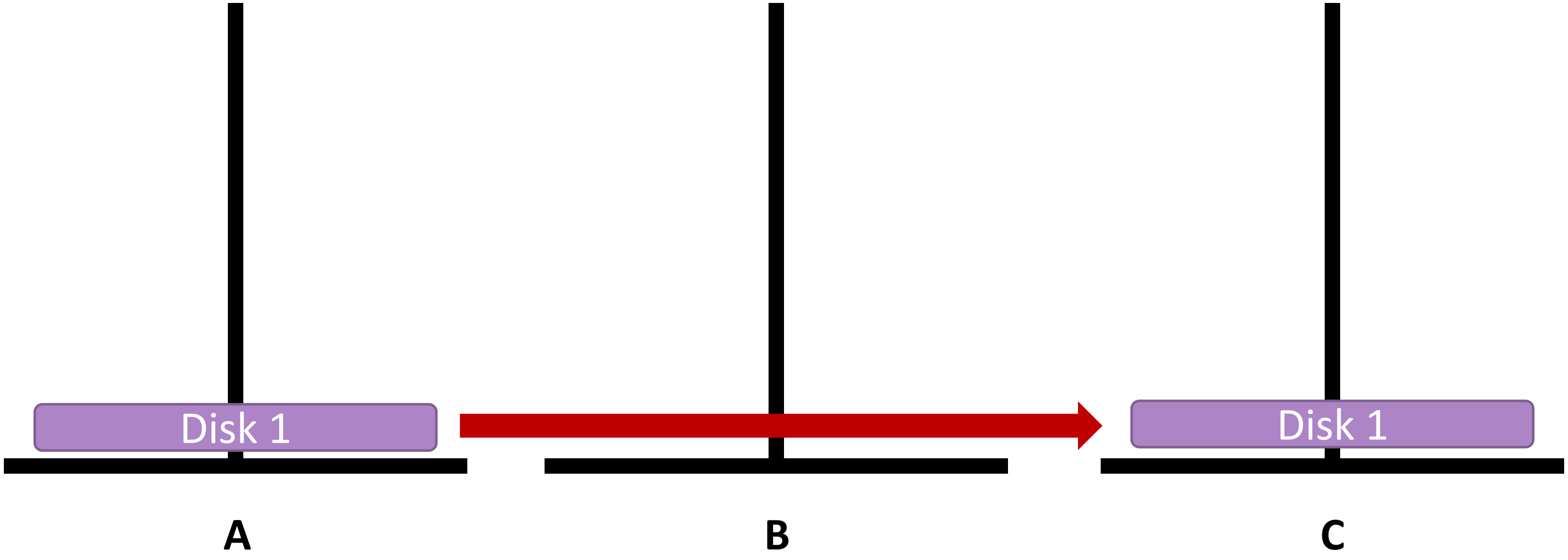


Play online: https://www.mathsisfun.com/games/towerofhanoi.html

Hanoi(1)

- Move 1 from A to C

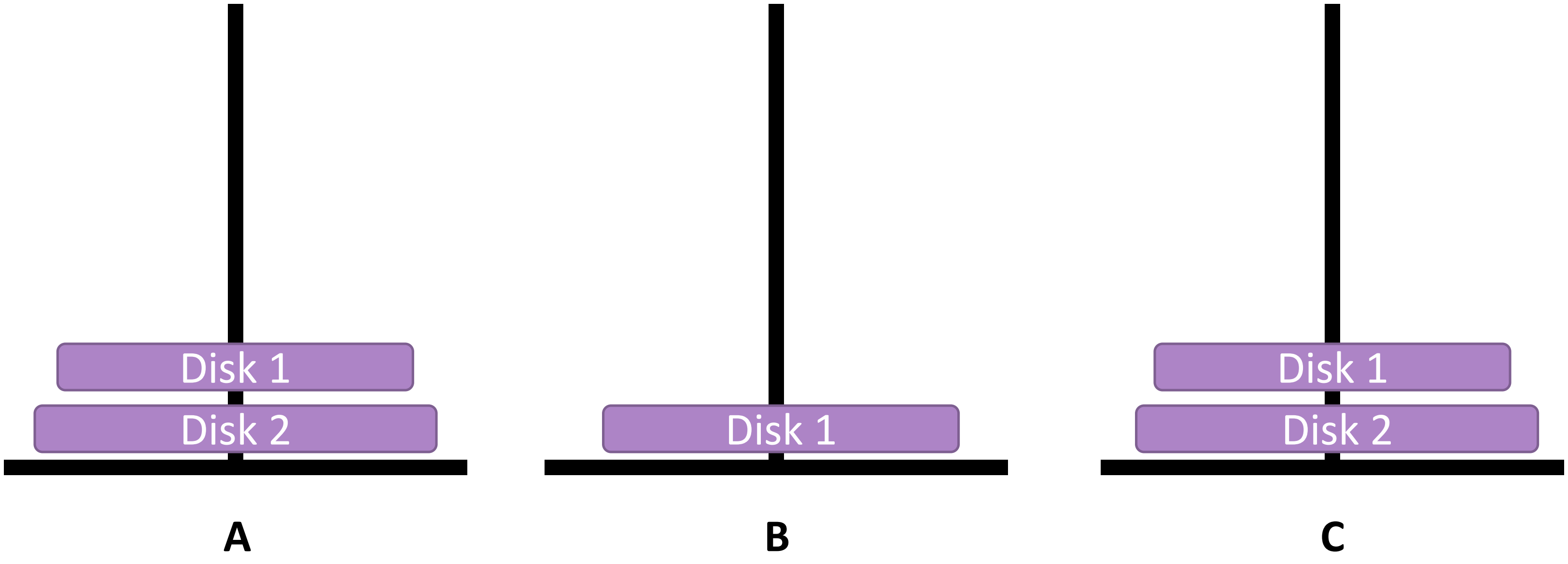
→ 1 move in total
Base case



Hanoi(2)

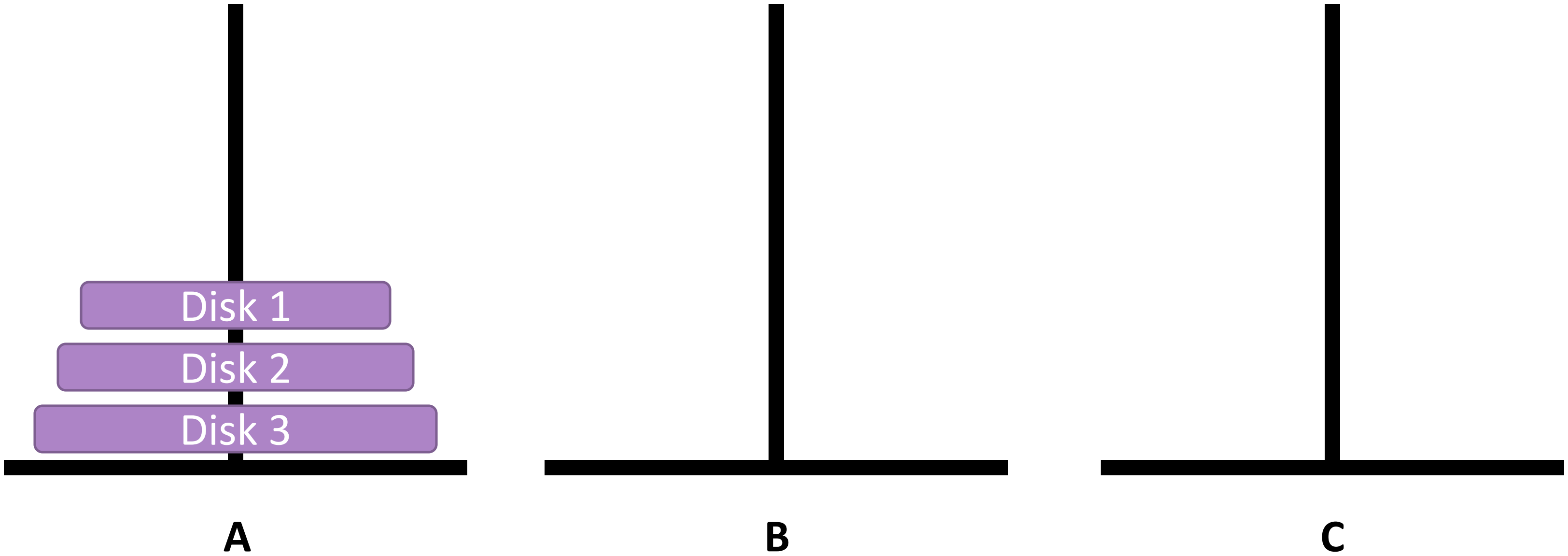
- Move 1 from A to B
- Move 2 from A to C
- Move 1 from B to C

→ 3 moves in total



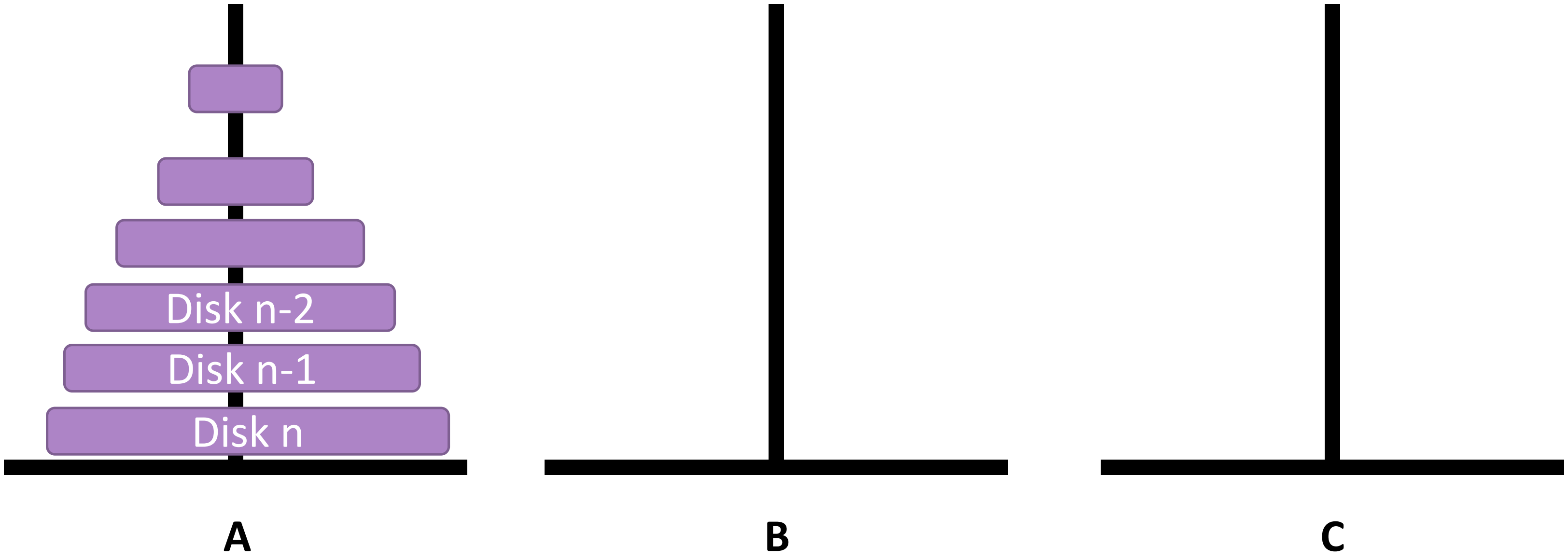
Hanoi(3)

- How to move 3 disks?
- How many moves in total?



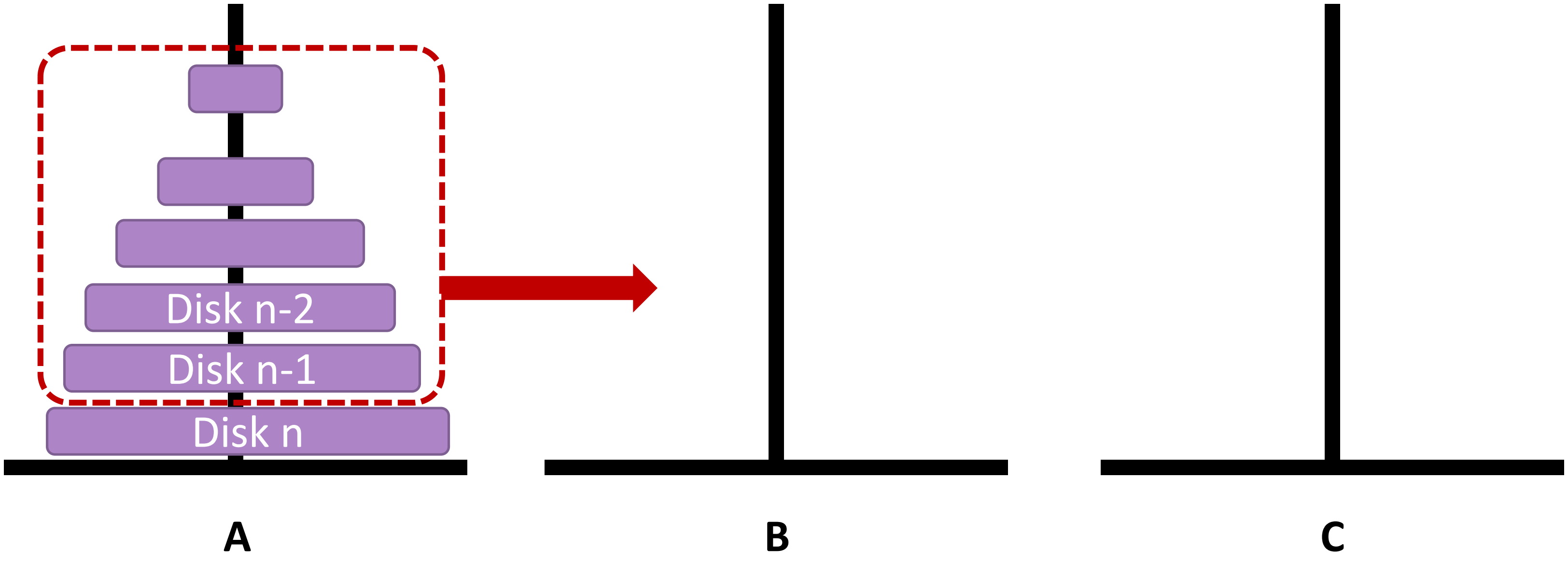
Hanoi(n)

- How to move n disks?
- How many moves in total?



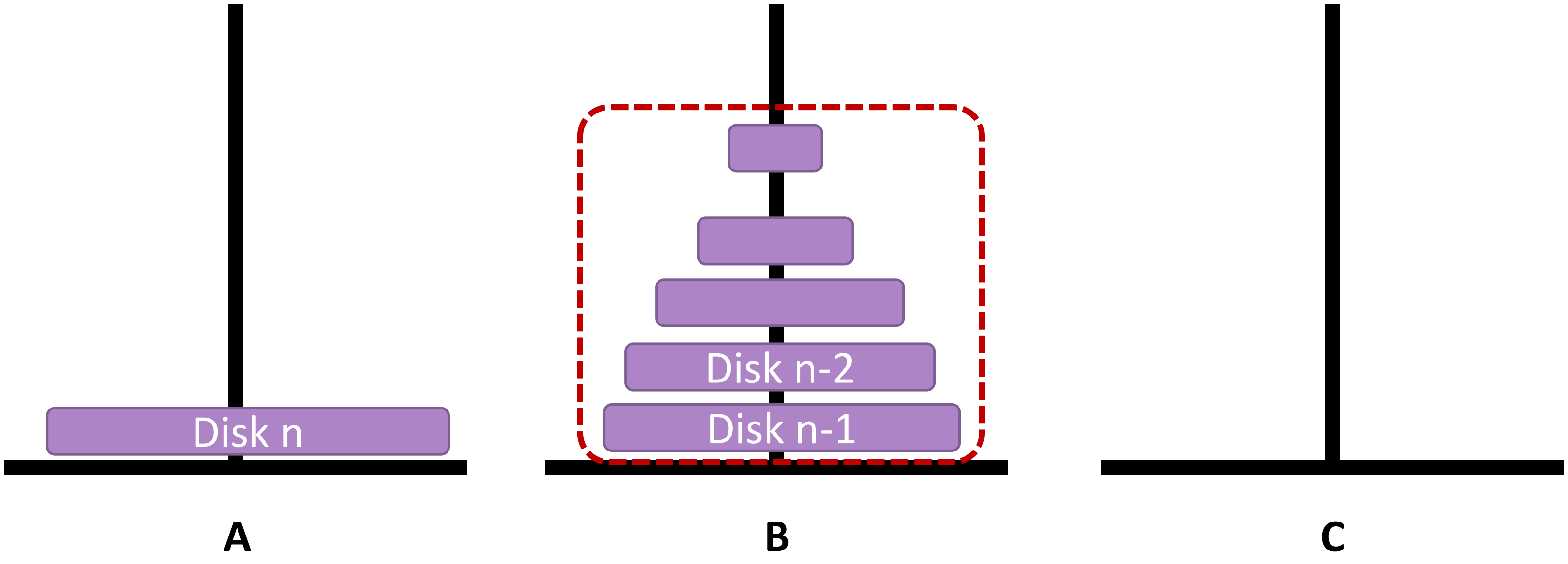
Hanoi(n)

- To move n disks from A to C (for n > 1):
  1. Move Disk 1~n-1 from A to B



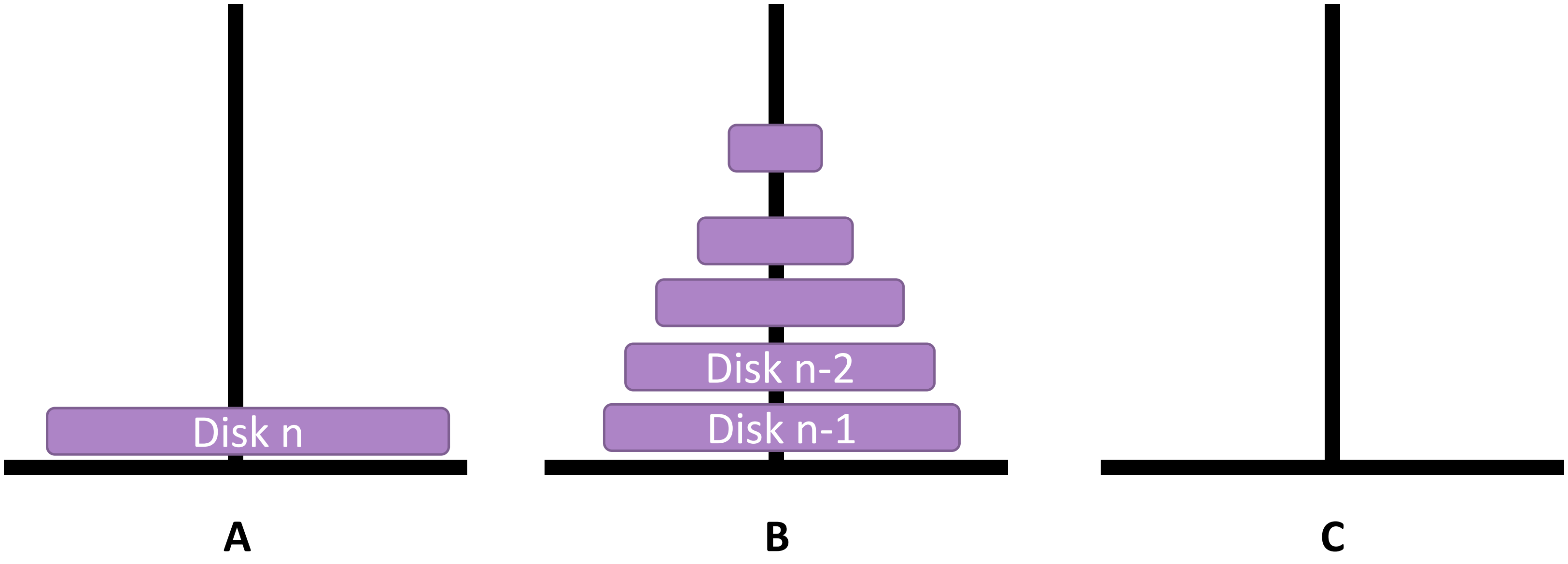
Hanoi(n)

- To move n disks from A to C (for n > 1):
  1. Move Disk 1~n-1 from A to B



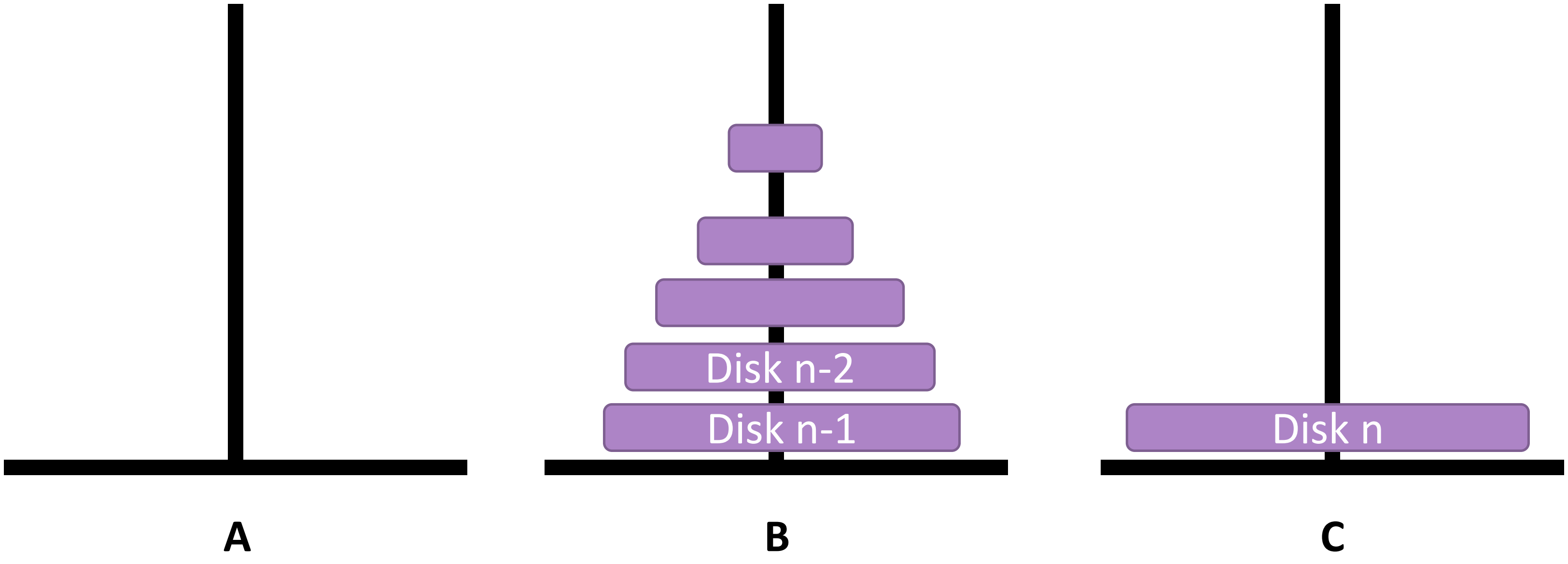
Hanoi(n)

- To move n disks from A to C (for n > 1):
  1. Move Disk 1~n-1 from A to B
  2. Move Disk n from A to C



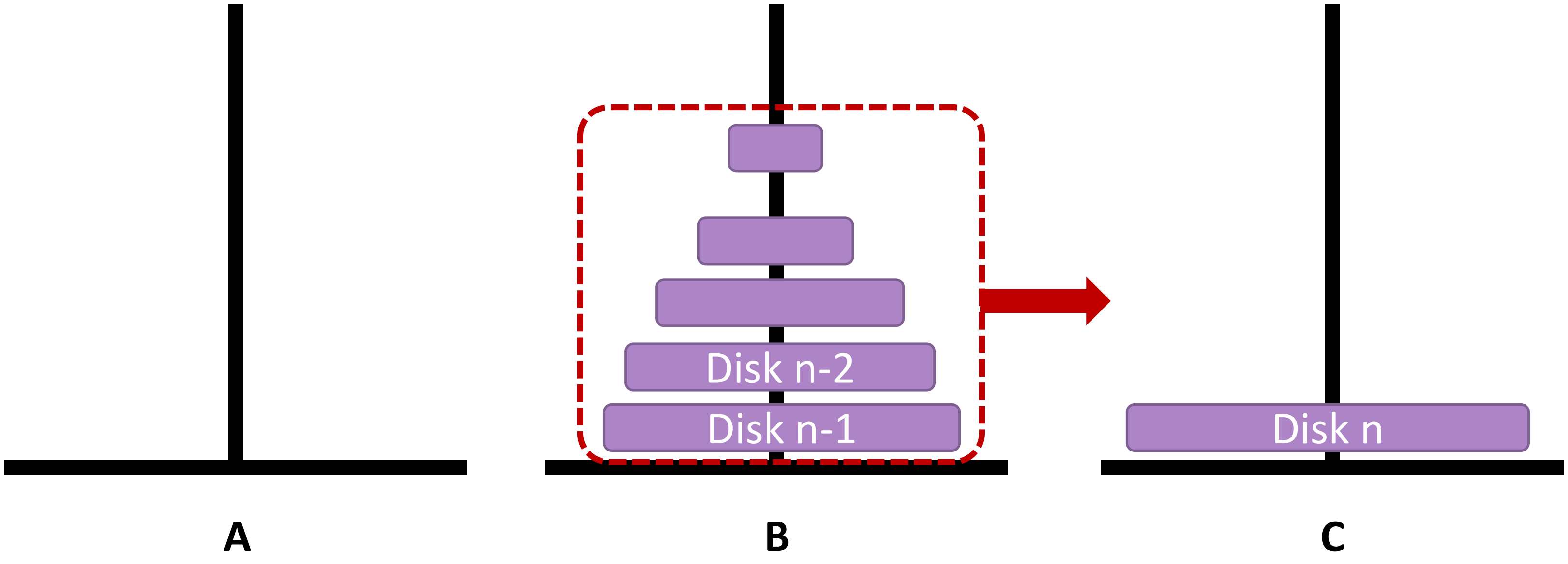
Hanoi(n)

- To move n disks from A to C (for n > 1):
  1. Move Disk 1~n-1 from A to B
  2. Move Disk n from A to C



Hanoi(n)

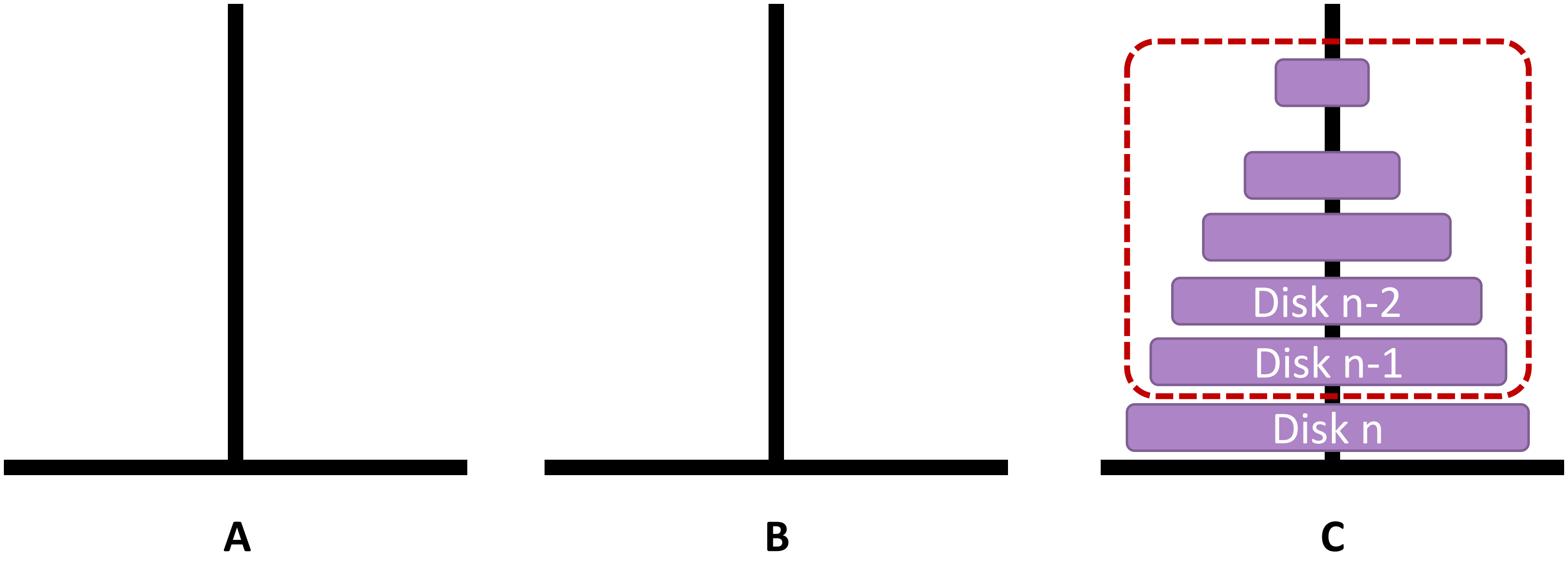
- To move n disks from A to C (for n > 1):
  1. Move Disk 1~n-1 from A to B
  2. Move Disk n from A to C
  3. Move Disk 1~n-1 from B to C



Hanoi(n)

- To move n disks from A to C (for n > 1):
  1. Move Disk 1~n-1 from A to B
  2. Move Disk n from A to C
  3. Move Disk 1~n-1 from B to C

→ 2Hanoi(n-1) + 1 moves in total
recursive case



Pseudocode for Hanoi

```
Hanoi(n, src, dest, spare)
  if n==1 // base case
    Move disk from src to dest
  else // recursive case
    Hanoi(n-1, src, spare, dest)
    Move disk from src to dest
    Hanoi(n-1, spare, dest, src)
```

No need to combine the results in this case

- Call tree

```
                  Hanoi(3, A, C, B)
       Hanoi(2, A, B, C)      Hanoi(2, B, C, A)
Hanoi(1,A,C,B)  Hanoi(1,C,B,A)  Hanoi(1,B,A,C)  Hanoi(1,A,C,B)
```

Algorithm Time Complexity

```
Hanoi(n, src, dest, spare)
  if n==1 // base case
    Move disk from src to dest
  else // recursive case
    Hanoi(n-1, src, spare, dest)
    Move disk from src to dest
    Hanoi(n-1, spare, dest, src)
```

- $T(n)$ = #moves with n disks
  - Base case: $T(1) = 1$
  - Recursive case ($n > 1$): $T(n) = 2T(n-1) + 1$
- We will learn how to derive $T(n)$ later

$$T(n) = 2^n - 1 = O(2^n)$$

Further Questions

- Q1: Is $O(2^n)$ tight for Hanoi? Can $T(n) < 2^n - 1$?
- Q2: What about more than 3 pegs?
- Q3: Double-color Hanoi problem
  - Input: 2 interleaved-color towers
  - Output: 2 same-color towers

30

D&C #2: Merge Sort

Textbook Chapter 2.3.1 – The divide-and-conquer approach

Sorting Problem

Input: unsorted list of size *n*

6 3 5 1 8 7 2 4

What are the **base case** and **recursive case**?

1 2 3 4 5 6 7 8

Output: sorted list of size *n*

Divide-and-Conquer

- Base case (n = 1)
  - Directly output the list
- Recursive case (n > 1)
  - Divide the list into two sub-lists
  - Sort each sub-list recursively
  - Merge the two sorted lists **How?**

1 3 5 6   2 4 7 8   2 sublists of size ***n/2***

# of comparisons = $\Theta(n)$

Illustration for n = 10

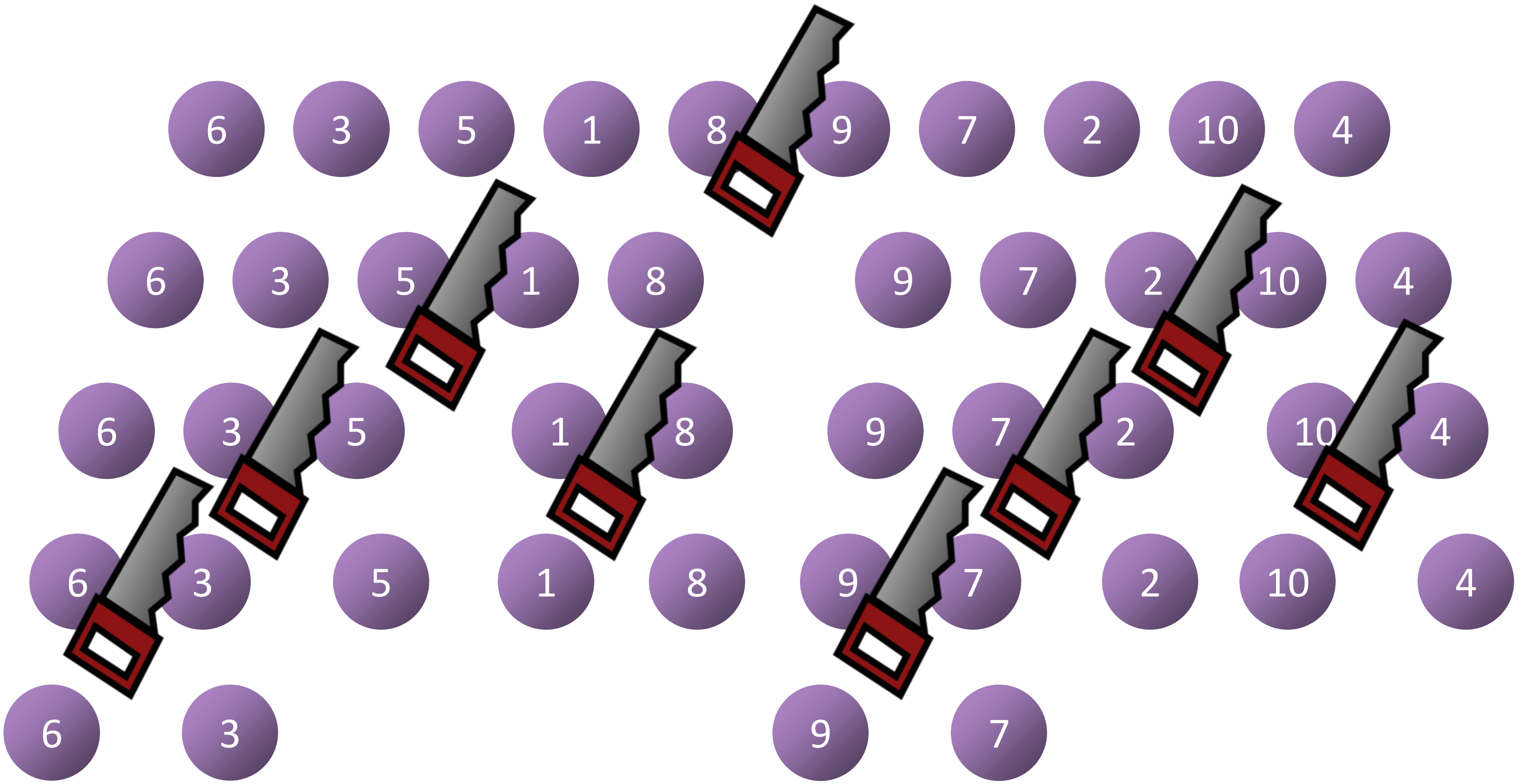
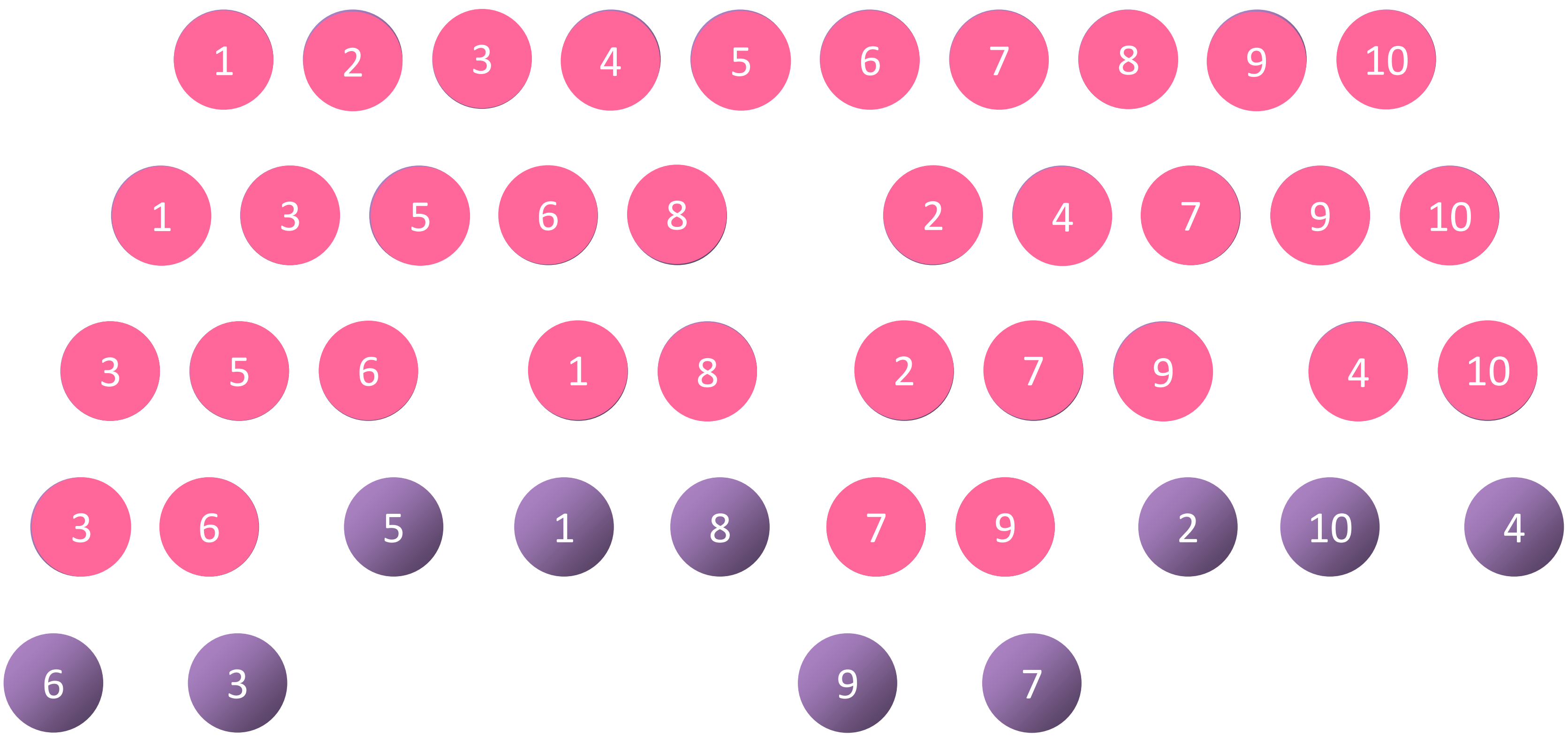
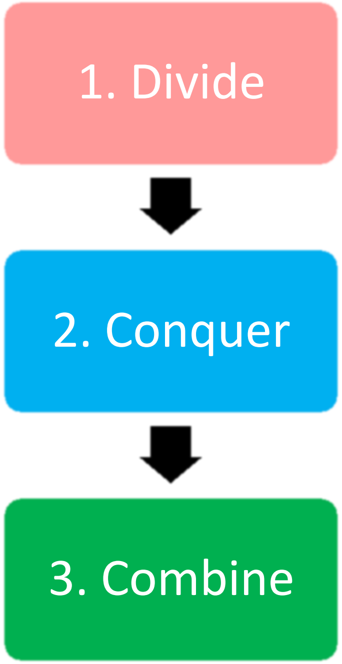


Illustration for n = 10



Pseudocode for Merge Sort

```
MergeSort(A, p, r)
  // base case
  if p == r
   return
  // recursive case
  // divide
  q = [(p+r-1)/2]
  // conquer
  MergeSort(A, p, q)
  MergeSort(A, q+1, r)
  // combine
  Merge(A, p, q, r)
```



- Divide a list of size *n* into 2 sublists of size *n/2*
- <u>Recursive case ($n > 1$)</u>
  - Sort 2 sublists ***recursively*** using ***merge sort***
- <u>Base case ($n = 1$)</u>
  - Return itself
- Merge 2 sorted sublists into one sorted list in **linear** time

Time Complexity for Merge Sort

```
MergeSort(A, p, r)
  // base case
  if p == r
   return
  // recursive case
  // divide
  q = [(p+r-1)/2]
  // conquer
  MergeSort(A, p, q)
  MergeSort(A, q+1, r)
  // combine
  Merge(A, p, q, r)
```

1. Divide

2. Conquer

3. Combine

- Divide a list of size *n* into 2 sublists of size *n/2* $\Theta(1)$
- Recursive case ($n > 1$)
  - Sort 2 sublists ***recursively*** using ***merge sort*** $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$
- Base case ($n = 1$)
  - Return itself $\Theta(1)$
- Merge 2 sorted sublists into one sorted list in **linear** time $\Theta(n)$

- $T(n)$ = time for running `MergeSort(A, p, r)` with $r - p + 1 = n$

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

Time Complexity for Merge Sort

- Simplify recurrences
- Ignore floors and ceilings (boundary conditions)
- Assume base cases are constant (for small *n*)

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

$$\begin{aligned} T(n) &\leq 2T(\frac{n}{2}) + cn \\ &\leq 2[2T(\frac{n}{4}) + c\frac{n}{2}] + cn = 4T(\frac{n}{4}) + 2cn \quad \text{1st expansion} \\ &\leq 4[2T(\frac{n}{8}) + c\frac{n}{4}] + 2cn = 8T(\frac{n}{8}) + 3cn \quad \text{2nd expansion} \\ &\vdots \\ &\leq 2^k T(\frac{n}{2^k}) + kcn \quad \text{kth expansion} \end{aligned}$$

The expansion stops when $2^k = n$

$$\begin{aligned} T(n) &\leq nT(1) + cn\log_2 n \\ &= O(n) + O(n\log n) \\ &= O(n\log n) \end{aligned}$$

Theorem 1

- Theorem

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

- Proof
  - There exists positive constant a, b s.t. $T(n) \leq \begin{cases} a & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + b \cdot n & \text{if } n \geq 2 \end{cases}$
  - Use induction to prove $T(n) \leq 2b \cdot n \log_2 n + a \cdot n$
    - n = 1, trivial
    - n > 1, $\lceil \frac{n}{2} \rceil \leq \frac{n}{\sqrt{2}}$

$$\begin{aligned} T(n) &\leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + b \cdot n \\ &\leq 2b \cdot (\lceil n/2 \rceil \log_2 \lceil n/2 \rceil) + a \cdot \lceil n/2 \rceil + 2b \cdot (\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor) + a \cdot \lfloor n/2 \rfloor + b \cdot n \\ &\leq 2b \cdot (\lceil n/2 \rceil \log_2 \frac{n}{\sqrt{2}} \rceil) + a \cdot \lceil n/2 \rceil + 2b \cdot (\lfloor n/2 \rfloor \log_2 \frac{n}{\sqrt{2}}) + a \cdot \lfloor n/2 \rfloor + b \cdot n \\ &= 2b \cdot n(\log n - \log_2 \sqrt{2}) + a \cdot n + b \cdot n = 2b \cdot n \log_2 n + a \cdot n \end{aligned}$$

Inductive hypothesis

How to Solve Recurrence Relations?

1. **Substitution Method** (取代法)
   - Guess a bound and then prove by induction
2. **Recursion-Tree Method** (遞迴樹法)
   - Expand the recurrence into a tree and sum up the cost
3. **Master Method** (套公式大法/大師法)
   - Apply Master Theorem to a specific form of recurrences

Let’s see more examples first and come back to this later

D&C #3: Bitonic Champion Problem

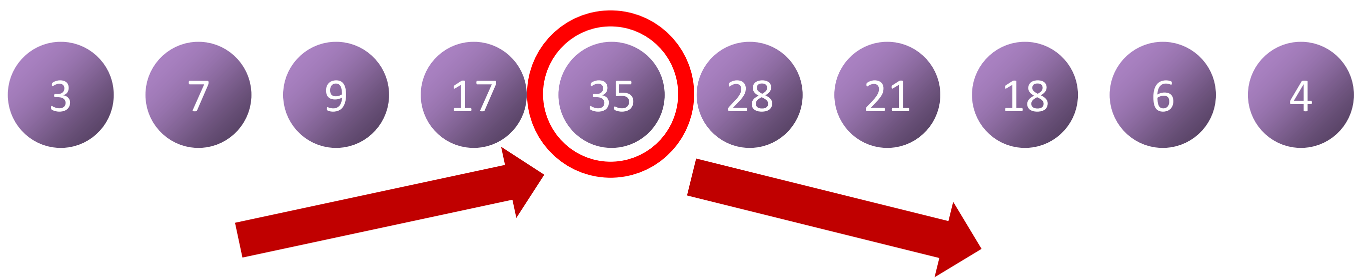
Bitonic Champion Problem

The bitonic champion problem

- Input: A bitonic sequence $A[1], A[2], \ldots, A[n]$ of distinct positive integers.
- Output: the index i with $1 \leq i \leq n$ such that

$$A[i] = \max_{1 \leq j \leq n} A[j].$$

The bitonic sequence means "increasing before the champion and decreasing after the champion" (冠軍之前遞增、冠軍之後遞減)



Bitonic Champion Problem Complexity

Upper bound $= O(n)$

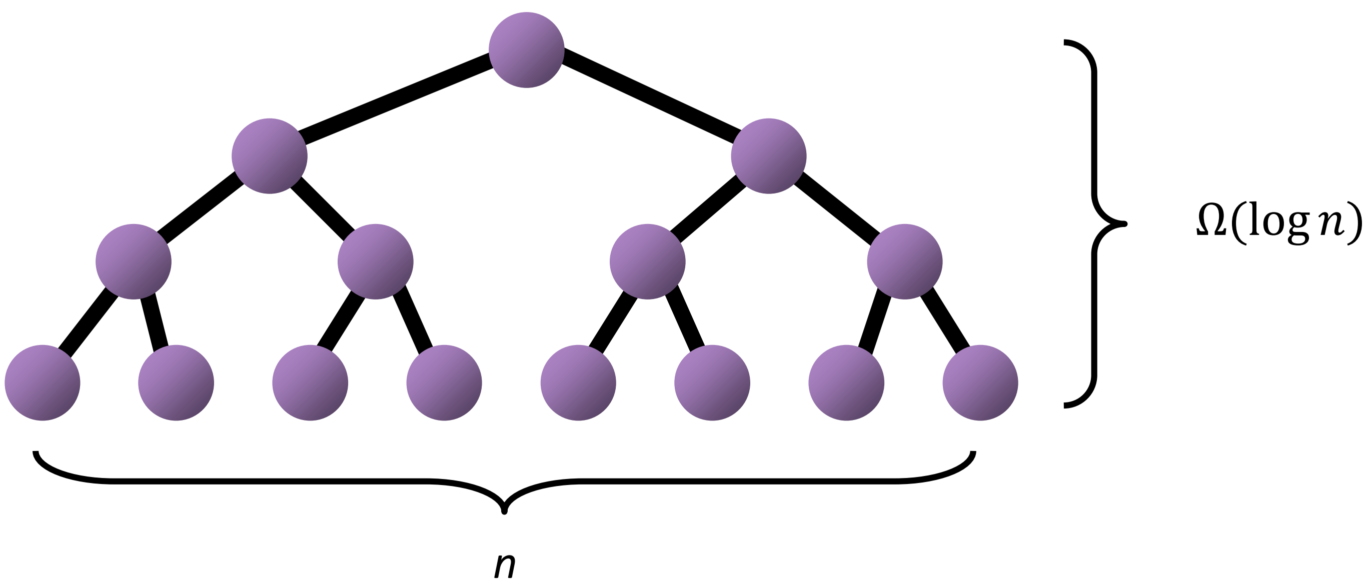
Why?

Lower bound $= \Omega(1)$

Why not Ω(n)?

Bitonic Champion Problem Complexity

- When there are *n* inputs, any solution has *n* different outputs
- Any comparison-based algorithm needs $\Omega(\log n)$ time in the worst case



Bitonic Champion Problem Complexity

Upper bound $= O(n)$

Lower bound $= \Omega(\log n)$

Lower bound $= \Omega(1)$

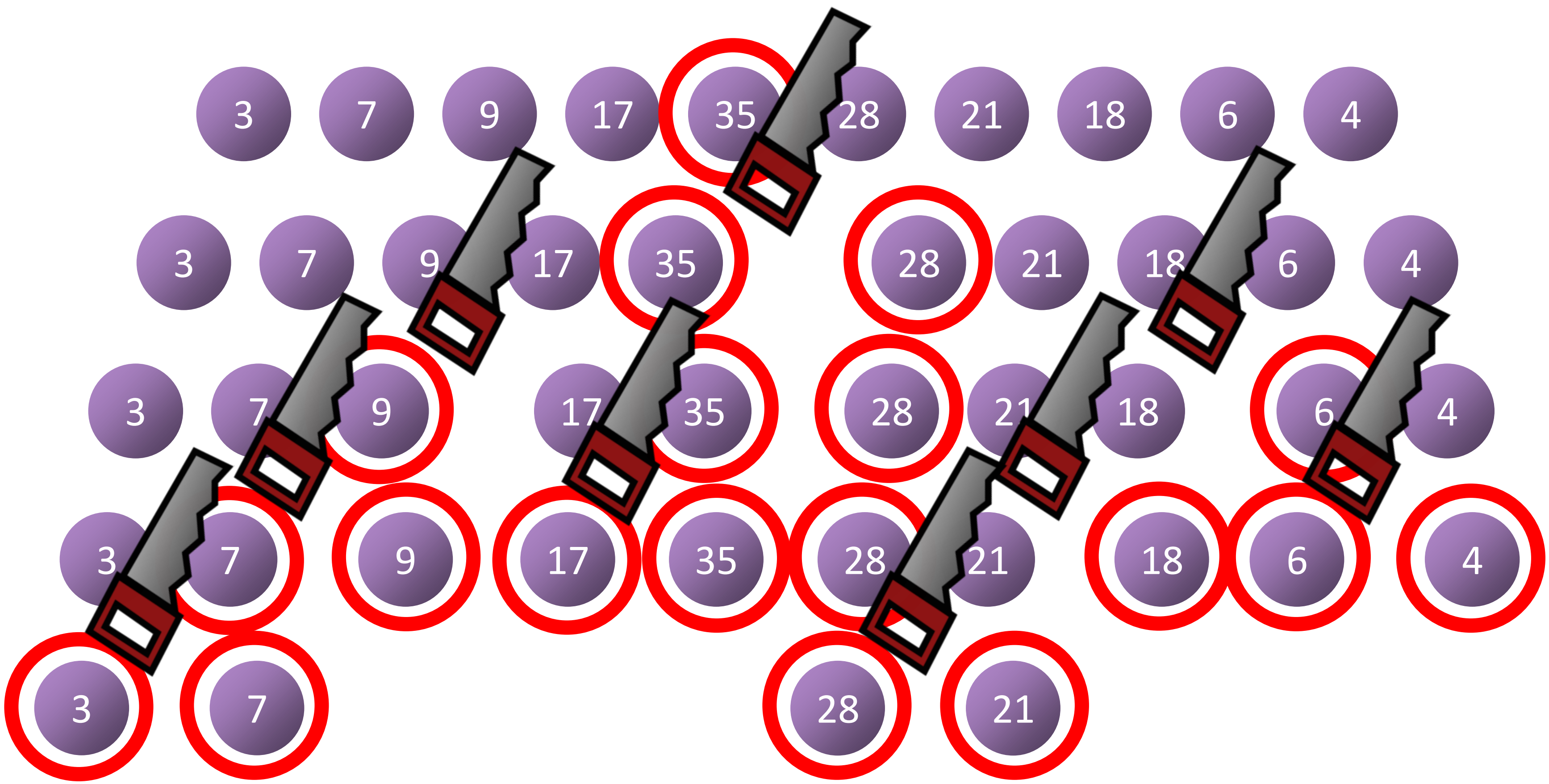
Divide-and-Conquer

- Idea: divide *A* into two subproblems and then find the final champion based on the champions from two subproblems

```
Output = Champion(1, n)
```

```
Champion(i, j)
  if i==j // base case
    return i
  else // recursive case
    k = floor((i+j)/2)
    l = Champion(i, k)
    r = Champion(k+1, j)
    if A[l] > A[r]
      return l
    if A[l] < A[r]
      return r
```

Illustration for $n = 10$



Proof of Correctness

- Practice by yourself!

```
Output = Chamption(1, n)
```

```
Champion(i, j)
  if i==j // base case
    return i
  else // recursive case
    k = floor((i+j)/2)
    l = Champion(i, k)
    r = Champion(k+1, j)
    if A[l] > A[r]
      return l
    if A[l] < A[r]
      return r
```

Hint: use induction on (j – i) to prove `Champion(i, j)` can return the champion from A[i … j]

Algorithm Time Complexity

```
Champion(i, j)
  if i==j // base case
    return i
  else // recursive case
    k = floor((i+j)/2)
    l = Champion(i, k)
    r = Champion(k+1, j)
    if A[l] > A[r]
      return l
    if A[l] < A[r]
      return r
```

1. Divide

- Divide a list of size *n* into 2 sublists of size *n/2* $\Theta(1)$

2. Conquer

$T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$

- Recursive case
  - Find champions from 2 sublists ***recursively***
- Base case $\Theta(1)$
  - Return itself

3. Combine

- Choose the final champion by a single comparison $\Theta(1)$

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

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

Theorem 2

- Theorem

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

- Proof
  - There exists positive constant a, b s.t.

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

  - Use induction to prove $T(n) \leq a \cdot n + b \cdot (n-1)$
    - n = 1, trivial
    - n > 1,

$$\begin{aligned} T(n) &\leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + b \\ &\leq a \cdot \lceil n/2 \rceil + b \cdot (\lceil n/2 \rceil - 1) + a \cdot \lfloor n/2 \rfloor + b \cdot (\lfloor n/2 \rfloor - 1) + b \\ &\leq a \cdot n + b \cdot (n-1) \end{aligned}$$

Inductive hypothesis

Bitonic Champion Problem Complexity

$$\text{Upper bound} = O(n)$$

Can we have a better algorithm by using the bitonic sequence property?

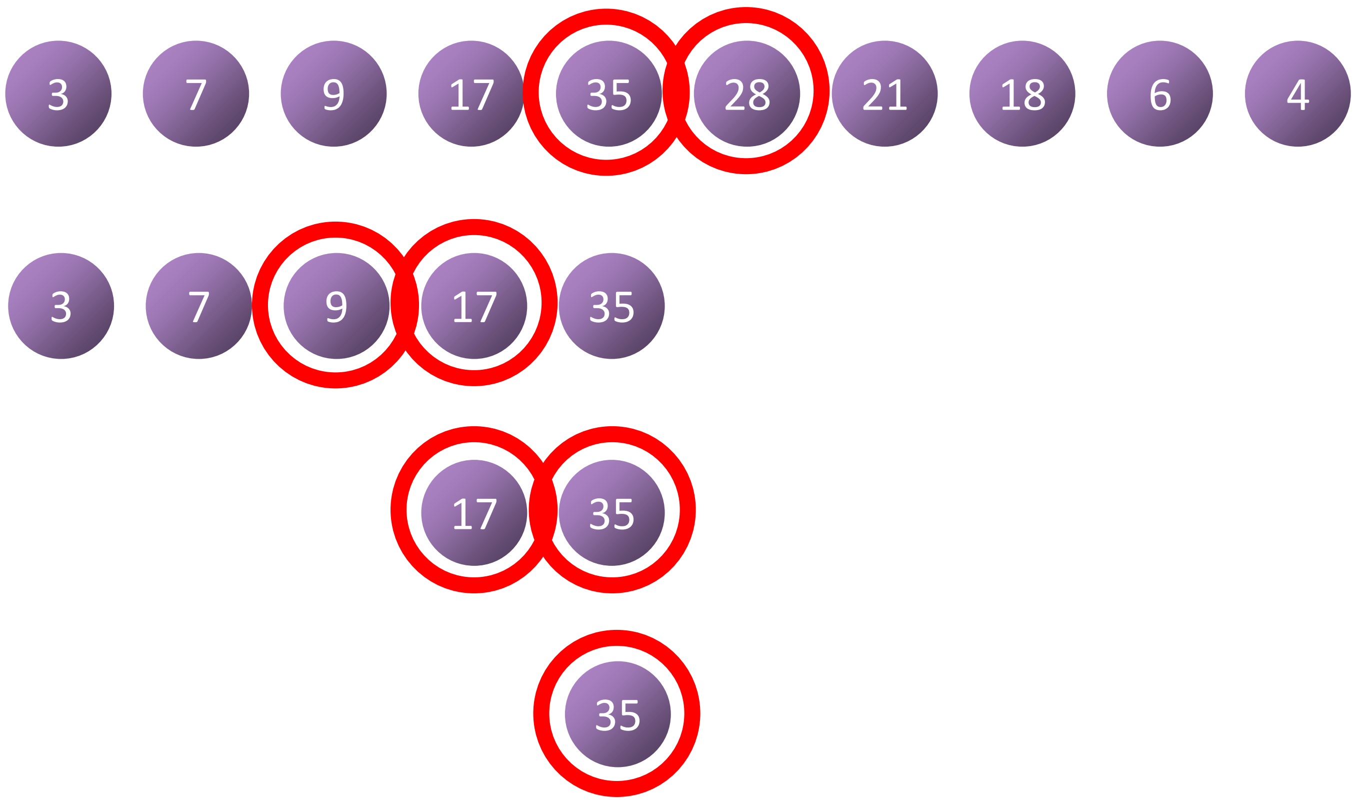
$$\text{Lower bound} = \Omega(\log n)$$

Improved Algorithm

```
Champion(i, j)
  if i==j // base case
    return i
  else // recursive case
    k = floor((i+j)/2)
    l = Champion(i, k)
    r = Champion(k+1, j)
    if A[l] > A[r]
      return l
    if A[l] < A[r]
      return r
```

```
Champion-2(i, j)
  if i==j // base case
    return i
  else // recursive case
    k = floor((i+j)/2)
    if A[k] > A[k+1]
      return Champion(i, k)
    if A[k] < A[k+1]
      return Champion(k+1, j)
```

Illustration for n = 10



Correctness Proof

- Practice by yourself!

```
Output = Champion-2(1, n)
```

```
Champion-2(i, j)
  if i==j // base case
    return i
  else // recursive case
    k = floor((i+j)/2)
    if A[k] > A[k+1]
      return Champion(i, k)
    if A[k] < A[k+1]
      return Champion(k+1, j)
```

Two crucial observations:

- If $A[1 \dots n]$ is bitonic, then so is $A[i, j]$ for any indices i and j with $1 \leq i \leq j \leq n$.
- For any indices i, j, and k with $1 \leq i \leq j \leq n$, we know that $A[k] > A[k+1]$ if and only if the maximum of $A[i \dots j]$ lies in $A[i \dots k]$.

Algorithm Time Complexity

```
Champion-2(i, j)
  if i==j // base case
    return i
  else // recursive case
    k = floor((i+j)/2)
    if A[k] > A[k+1]
      return Champion(i, k)
    if A[k] < A[k+1]
      return Champion(k+1, j)
```

1. Divide

- Divide a list of size *n* into 2 sublists of size *n/2* $\Theta(1)$

2. Conquer

- Recursive case $T(\lceil n/2 \rceil)$
  - Find champions from 1 sublists ***recursively***
- Base case $\Theta(1)$
  - Return itself

3. Combine

- Return the champion $\Theta(1)$

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

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

Algorithm Time Complexity

```
Champion-2(i, j)
  if i==j // base case
    return i
  else // recursive case
    k = floor((i+j)/2)
    if A[k] > A[k+1]
      return Champion(i, k)
    if A[k] < A[k+1]
      return Champion(k+1, j)
```

The algorithm time complexity is $O(\log n)$

- each recursive call reduces the size of (j - i) into half
- there are $O(\log n)$ levels
- each level takes $O(1)$

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

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

Theorem 3

- Theorem

$$T(n) \le \begin{cases} O(1) & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + O(1) & \text{if } n \ge 2 \end{cases} \Rightarrow T(n) = O(\log n)$$

- Proof

Practice to prove by induction

Bitonic Champion Problem Complexity

$\text{Upper bound} = O(n)$

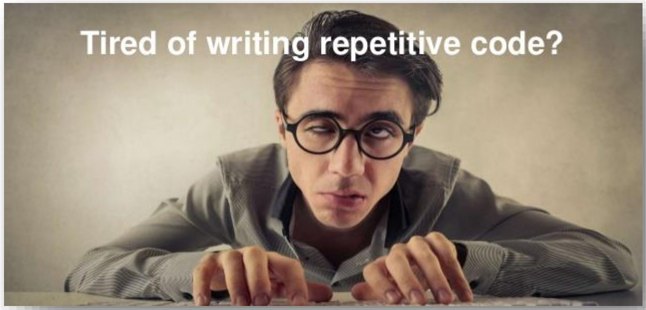
$\text{Upper bound} = O(\log n)$

$\text{Lower bound} = \Omega(\log n)$

D&C #4: Maximum Subarray Problem

Textbook Chapter 4.1 – The maximum-subarray problem

Coding Efficiency



- How can we find the most efficient time interval for continuous coding?

Coding power
戰鬥力 (K)

7pm-2:59am
Coding power= 8k

Maximum Subarray Problem

- Input: A sequence $A[1], A[2], \ldots, A[n]$ of integers.
- Output: Two indicex i and j with $1 \leq i \leq j \leq n$ that maximize

$$A[i] + A[i+1] + \cdots + A[j].$$

3	7	9	17	5	28	21	18	6	4
-3	7	-9	17	-5	28	-21	18	-6	4
-3	-7	-9	-17	-5	-28	-21	-18	-6	-4

$O(n^3)$ Brute Force Algorithm

```
MaxSubarray-1(i, j)
  for i = 1,…,n
    for j = 1,…,n
      S[i][j] = -∞

  for i = 1,…,n
    for j = i,i+1,…,n
      S[i][j] = A[i] + A[i+1] + … + A[j]

  return Champion(S)
```

$O(n^2)$

$O(n^3)$

$O(n^2)$

$O(n^2)$ Brute Force Algorithm

```
MaxSubarray-2(i, j)
  for i = 1,…,n
    for j = 1,…,n
      S[i][j] = - ∞
```
$O(n^2)$

```
  R[0] = 0
  for i = 1,…,n
    R[i] = R[i-1] + A[i]
```
`R[n]` is the sum over `A[1…n]`

$O(n)$

```
  for i = 1,…,n
    for j = i+1,i+2,…,n
      S[i][j] = R[j] - R[i-1]
```
$O(n^2)$

```
  return Champion(S)
```
$O(n^2)$

Max Subarray Problem Complexity

Upper bound $= O(n^2)$

Lower bound $= \Omega(n)$

Divide-and-Conquer

- Base case (n = 1)
  - Return itself (maximum subarray)
- Recursive case (n > 1)
  - Divide the array into two sub-arrays
  - Find the maximum sub-array recursively
  - Merge the results **How?**

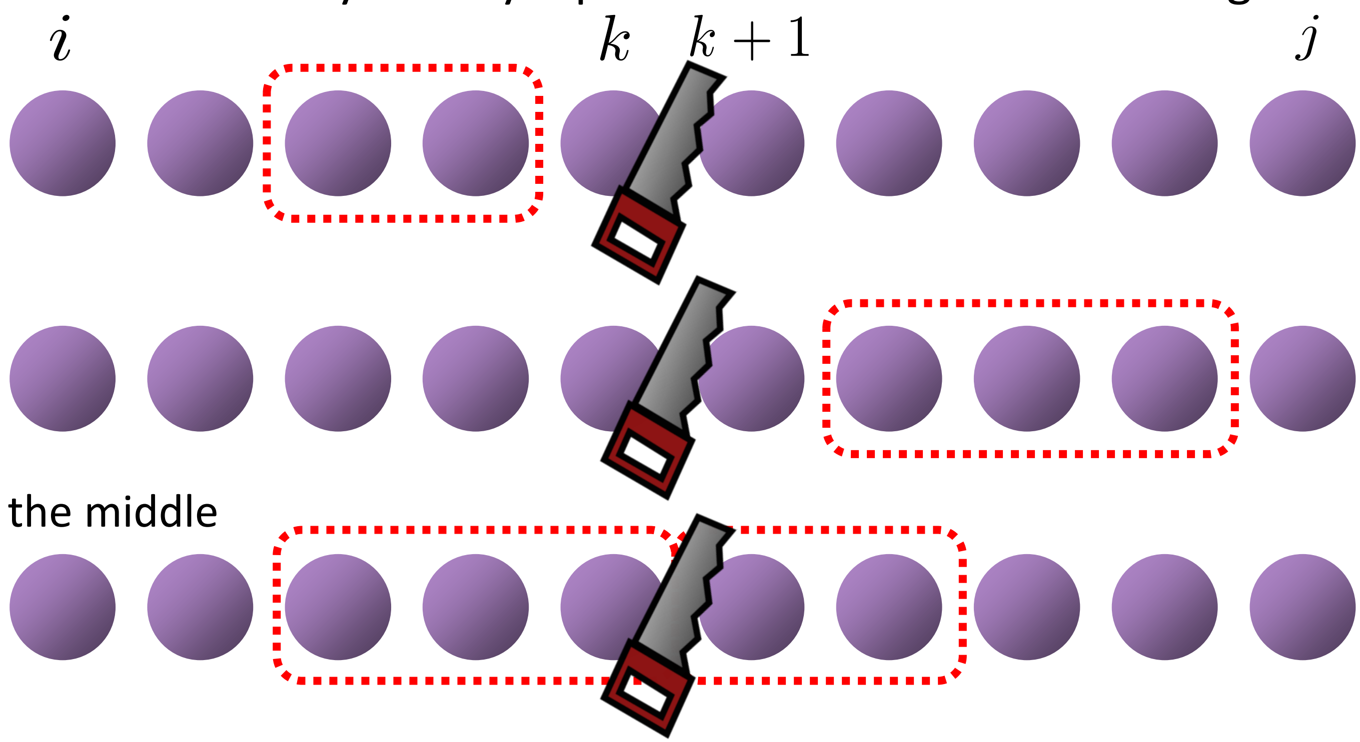
Where is the Solution?

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

Case 1: left

Case 2: right

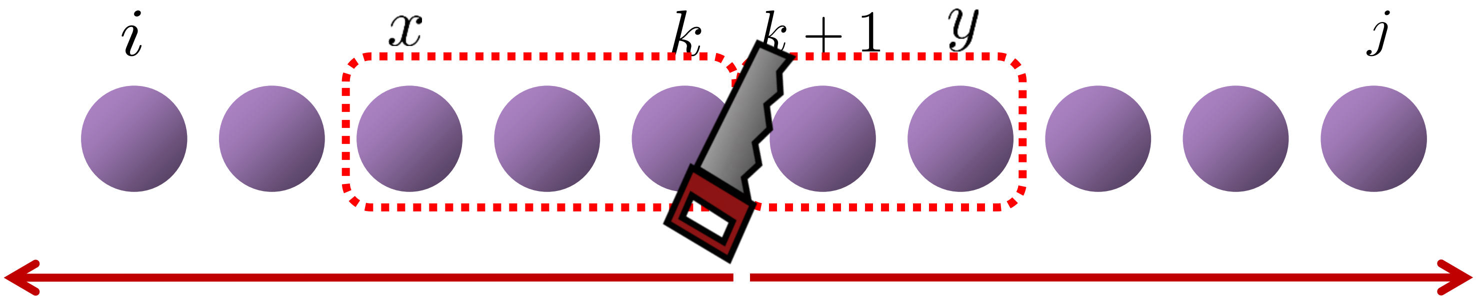
Case 3: cross the middle



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 → $\Theta(n)$

Divide-and-Conquer Algorithm

```
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)
```

$$\left.\begin{array}{l} O(k-i+1) \\ \\ O(j-k) \end{array}\right\} = O(j-i+1)$$

Divide-and-Conquer Algorithm

```
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 Algorithm

```
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)
```

- `if i == j // base case`: $O(1)$
- `(l_low, l_high, l_sum) = MaxSubarray(A, i, k)`: $T(k-i+1)$
- `(r_low, r_high, r_sum) = MaxSubarray(A, k+1, j)`: $T(j-k)$
- `(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`: $O(1)$
- `else if r_sum >= l_sum and r_sum >= c_sum // case 2`: $O(1)$
- `else // case 3`: $O(1)$

Algorithm Time Complexity

1. Divide

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

2. Conquer

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

3. Combine

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

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

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

Theorem 1

- Theorem

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

- Proof
  - There exists positive constant a, b s.t. $T(n) \leq \begin{cases} a & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + b \cdot n & \text{if } n \geq 2 \end{cases}$
  - Use induction to prove $T(n) \leq 2b \cdot n \log_2 n + a \cdot n$
    - n = 1, trivial
    - n > 1, $\frac{n+1}{2} \leq \frac{n}{\sqrt{2}}$

$$\begin{aligned} T(n) &\leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + b \cdot n \\ &\leq 2b \cdot (\lceil n/2 \rceil \log_2 \lceil n/2 \rceil + a \cdot \lceil n/2 \rceil) + 2b \cdot (\lfloor n/2 \rfloor \log_2 \lfloor n/2 \rfloor + a \cdot \lfloor n/2 \rfloor) + b \cdot n \\ &\leq 2b \cdot (\lceil n/2 \rceil \log_2 \frac{n}{\sqrt{2}} \rceil + a \cdot \lceil n/2 \rceil) + 2b \cdot (\lfloor n/2 \rfloor \log_2 \frac{n}{\sqrt{2}} + a \cdot \lfloor n/2 \rfloor) + b \cdot n \\ &= 2b \cdot n(\log n - \log_2 \sqrt{2}) + a \cdot n + b \cdot n = 2b \cdot n \log_2 n + a \cdot n \end{aligned}$$

Inductive hypothesis

Theorem 1 (Simplified)

- Theorem

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases} \Rightarrow T(n) = O(n \log n)$$

- Proof
  - There exists positive constant a, b s.t.

$$T(n) \leq \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$$

  - Use induction to prove $T(n) \leq b \cdot n \log n + a \cdot n$
    - n = 1, trivial
    - n > 1,

$$\begin{aligned} T(n) &\leq 2T(n/2) + bn \\ &\leq 2[b \cdot \frac{n}{2} \log \frac{n}{2} + a \cdot \frac{n}{2}] + b \cdot n \\ &= b \cdot n \log n - b \cdot n + a \cdot n + b \cdot n \\ &= b \cdot n \log n + a \cdot n \end{aligned}$$

Inductive hypothesis

Max Subarray Problem Complexity

Upper bound $= O(n^2)$

Upper bound $= O(n \log n)$

Lower bound $= \Omega(n)$

Max Subarray Problem Complexity

Upper bound $= O(n \log n)$

Upper bound $= O(n)$

Exercise 4.1-5
page 75 of textbook

Next topic!

Lower bound $= \Omega(n)$

Solving Recurrences

Textbook Chapter 4.3 – The substitution method for solving recurrences

Textbook Chapter 4.4 – The recursion-tree method for solving recurrences

D&C Algorithm Time Complexity

- $T(n)$: running time for input size n
- $D(n)$: time of **Divide** for input size n
- $C(n)$: time of **Combine** for input size n
- a: number of subproblems
- n/b: size of each subproblem

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Solving Recurrences

1. **Substitution Method** (取代法)
   - Guess a bound and then prove by induction
2. **Recursion-Tree Method** (遞迴樹法)
   - Expand the recurrence into a tree and sum up the cost
3. **Master Method** (套公式大法/大師法)
   - Apply Master Theorem to a specific form of recurrences

- Useful simplification tricks
  - Ignore floors, ceilings, boundary conditions (proof in Ch. 4.6)
  - Assume base cases are constant (for small *n*)

Review

- Time Complexity for Merge Sort
- Theorem

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases} \Rightarrow T(n) = O(n \log n)$$

- Proof
  - There exists positive constant a, b s.t. $T(n) \leq \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + bn & \text{if } n \geq 2 \end{cases}$
  - Use induction to prove $T(n) \leq b \cdot n \log n + a \cdot n$
    - n = 1, trivial
    - n > 1, $T(n)$

$$\begin{aligned} T(n) &\leq 2T(n/2) + bn \\ &\leq 2[b \cdot \frac{n}{2} \log \frac{n}{2} + a \cdot \frac{n}{2}] + b \cdot n \\ &= b \cdot n \log n - b \cdot n + a \cdot n + b \cdot n \\ &= b \cdot n \log n + a \cdot n \end{aligned}$$

Substitution Method (取代法)
guess a bound and then prove by induction

Review

- Time Complexity for Merge Sort
- Theorem

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n \geq 2 \end{cases} \quad \Rightarrow \quad T(n) = O(n \log n)$$

- Proof

Recursion-Tree Method (遞迴樹法)

Expand the recurrence into a tree and sum up the cost

$$\begin{aligned} T(n) &\leq 2T(\frac{n}{2}) + cn \\ &\leq 2[2T(\frac{n}{4}) + c\frac{n}{2}] + cn = 4T(\frac{n}{4}) + 2cn \\ &\leq 4[2T(\frac{n}{8}) + c\frac{n}{4}] + 2cn = 8T(\frac{n}{8}) + 3cn \\ &\vdots \\ &\leq 2^k T(\frac{n}{2^k}) + kcn \end{aligned}$$

1st expansion

2nd expansion

kth expansion

The expansion stops when $2^k = n$

$$\begin{aligned} T(n) &\leq nT(1) + cn \log_2 n \\ &= O(n) + O(n \log n) \\ &= O(n \log n) \end{aligned}$$

80

To Be Continued…

Question?

Important announcement will be sent to @ntu.edu.tw mailbox
& post to the course website

Course Website: http://ada.miulab.tw

Email: ada-ta@csie.ntu.edu.tw