程式設計
Ch13. Introduction to Data Structure

Chuan-Chi Lai 賴傳淇

Department of Communications Engineering
National Chung Cheng University

Spring Semester, 2024

Outline

什麼是資料結構？(What is data structure?)

什麼是資料結構？
What is data structure?

什麼是資料結構？(What is data structure?)

為何我們需要資料結構？

什麼是資料結構？

- 可在電腦中以特定方式儲存和組織資料，以便有效使用資料的。
- 不同類型的資料結構適合不同種類的應用。
- 例如：
  - B 樹 (B-Tree) 適用於資料庫應用。
  - 雜湊表 (Hash table) 在編譯器中被使用來快速查詢識別字/保留字等。

什麼是資料結構？(What is data structure?)

以一個簡單搜尋為例

- 假受給定 8 個數字儲存在陣列中並按升序排列。($n = 8$)

1	3	5	8	9	18	31	49

- 想要檢查/搜尋 “10” 是否在陣列中，怎麼做？
  - 直覺的方法：循序檢查陣列中每個元素是否為 “10”，需要檢查 n 次。
  - 聰明的方法：使用二分搜尋 (binary search)，檢查的步數少於 $\log_2(n)$。

什麼是資料結構？(What is data structure?)

資料結構的定義

- 資料結構包含資料的表示 (representation) 和操作 (manipulation)。
- **Representation:**
  - 我們將資料建立/組職成特定的結構，以便日後可以有效率地使用。
- **Manipulation:**
  - 使用演算法來操作資料！

什麼是資料結構？(What is data structure?)

為何資料結構很重要？

- 假設您必須維護一個個人通訊錄，其中包含您朋友的 100 筆記錄，每筆記錄都儲存一個名稱和一個位址。
- 如果你想找某個特定朋友的記錄，你會怎麼做？
- 您可以按順序循序地瀏覽每筆記錄，直到找到目標！
- 但是，如果您維護的資料是一個是城市的通訊錄，包含接近 10^6 筆資料，怎麼辦？
- 甚至，如果需要在每筆記錄附加更多資訊，例如性別、電話、工作等，怎麼辦？

什麼是資料結構？(What is data structure?)

- 當真實問題規模變得越來越大時，真正的難題就會出現！
- 因此，我們如何建構/組織資料使其適合搜尋 (searching)，以利後續的資料維護？

什麼是資料結構？(What is data structure?)

資料結構的重要性

- 資料結構很重要，因為它決定了
  - 可以對資料執行的運算類型
  - 這些運算的執行效率如何
  - 處理資料時能有多大的彈性/動態性
    - 例如，我們是否可以動態新增額外的資料，或者我們是否需要預先了解所有資料？
- 資料結構組織的方式會決定解決問題的方式
- 並且，解決問題的方式也會決定解決問題的效率

什麼是資料結構？(What is data structure?)

後續章節內容

- 我們談 c 語言實作一些基礎的資料結構，包含：
  - 資料抽象與封裝、演算法設計、效能分析與測量
  - 資料的基本資料結構：
    - Arrays, Stacks, Queues, Linked lists, Trees, and so on.
  - 操作上述資料結構的基本演算法:
    - Sorting, Searching, Tree traversal, and so on.

抽象資料型別
Abstract Data Type

抽象資料型別 (Abstract Data Type)

如何創建程式？

- 應用/問題的需求
- 分析方法：bottom-up vs. top-down
- 設計：資料物件/結構和操作
- 重構和程式撰寫
- 驗證：
  - 程式證明 (Program Proving)
  - 測試 (Testing)
  - 除錯 (Debugging)

資料結構 vs 演算法

- 資料結構：
  - 組織和存取資料的系統化方法。
- 演算法：
  - 在有限時間內執行特定任務的逐步過程。
- 演算法的特徵歸納：
  - 輸入 (input)：一個演算法必須有零個或以上輸入量。
  - 輸出 (output)：一個演算法應有一個或以上輸出量，輸出量是演算法計算的結果。
  - 明確性 (definiteness): 演算法的描述必須無歧義，以保證演算法的實際執行結果是精確地符合要求或期望，通常要求實際執行結果是確定的。
  - 有限性 (finiteness): 規定演算法必須在有限個步驟內完成任務。
  - 可行性 (effectiveness): 演算法中描述的操作都是可以通過已經實現的基本運算執行有限次來實現。

資料型別 (Data Type)

- 一種資料型別 (Data Type) 的定義，會包含定義特定數值 (資料物件) 的集合，以及這些數值的一系列操作/運算的定義。
- 例如：

```
int x, y; // x, y, a, b are identifiers
float a, b;
x = x + y; // integer addition
a = a + b; // floating-point addition
```

上述兩種加法是否相同？

抽象資料型別 (Abstract Data Type, ADT)

- 原生資料型別 (Native Data Type)：
  - int (not realistic integer), float (real number), char
  - 硬體支援/實現
- 抽象資料型別
  - 資料物件型別和運算的規範分別定義成物件表示 (object representation) 和運算實作 (operation implementation)
  - 由現有資料型別定義和組成
  - 抽象資料型別的內部物件表示和運算實作對外是隱藏的。
  - 實例：stack, queue, list, set, ...

自然數的抽象資料型別 (ADT of NaturalNumber)

ADT *NaturalNumber* is

- objects: An ordered subrange of the integers starting at zero and ending at the maximum integer (MAXINT) on the computer.
- functions: for all $x, y \in$ *NaturalNumber*, TRUE, FALSE $\in$ Boolean and where +, -, <, ==, and = are the usual integer operations.

```
Nat_Num Zero() ::=      0
Boolean IsZero(x) ::=   if (x == 0) return true
                        else return false
Equal(x,y) ::=          if (x == y) return true
                        else return false
Successor ::=           ...
Add(x, y) ::=           if (x + y <= MAXINT) Add = x + y
                        else Add = MAXINT
Subtract(x,y) ::=       ...
```

end *NaturalNumber*

抽象資料型別 (Abstract Data Type)

評判一個程式的標準

- 程式是否做我們想要它做的事？
- 程式是否按照任務的原始規範正確運作？
- 是否有文件描述如何使用此程式以及它是如何運作的？
- 我們是否有效地使用函式來建立邏輯單元與運算？
- 程式是否具可讀性？
- 程式是否有效地使用了主要儲存媒體和次要儲存媒體？
- 運轉時間可以接受嗎？

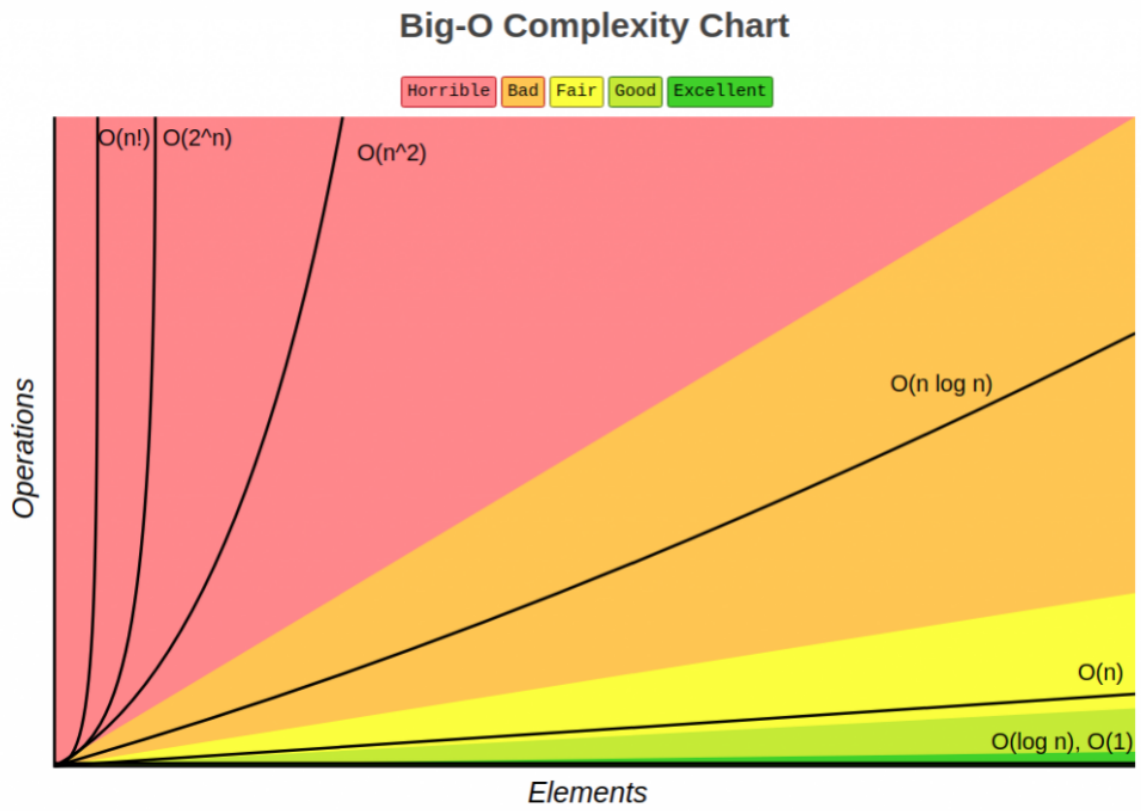
度量衡 (Measurements)

- 效能分析 (Performance Analysis)
  - 此方法與機器無關 (Machine Independent)
  - 空間複雜度 (Space Complexity): 程式執行佔用空間的需求
  - **時間複雜度 (Time Complexity): 程式的計算步數/時間**
- 效能量測 (Machine Measurement)
  - 此方法與機器本身運算能力高度相關 (Machine Dependent)

時間複雜度
Time Complexity

時間複雜度 (Time Complexity)

- 時間複雜度是一個衡量演算法運行時間的指標，它可以告訴我們隨著資料量規模的增加，演算法運行時間會如何增長。



Source: https://www.freecodecamp.org/news/all-you-need-to-know-about-big-o-notation-to-crack-your-next-coding-interview-9d575e7eec4/

時間複雜度 (Time Complexity)

- 用於描述時間複雜度的常見符號如下：
  - O (Big-O)：演算法的上界複雜度
  - Ω (Big-Omega)：演算法的下界複雜度
  - Θ (Theta)：表示一時間複雜度同時屬於演算法的上界與下界
- 我們最常使用 O(Big-O) 來評斷一個演算法的運行時間趨勢，後面只會以此作為舉例。

時間複雜度 (Time Complexity)

- 常見的時間複雜度量級：
  - $O(1)$：常數時間
  - $O(\log n)$：對數時間
  - $O(n)$：線性時間
  - $O(n \log n)$：線性對數時間
  - $O(n^2)$：平方時間
  - $O(n^k)$：(k 次方) 多項式時間
  - $O(2^n)$：指數時間
  - $O(n!)$：階乘時間
- 通常探討計算複雜度使用的 log 事實上是以 2 為底的 ($\log_2$)，以上提到的時間複雜度是由上至下逐漸增加的，因此執行效率也逐漸降低。

常數時間 $O(1)$

- 不管程式碼執行了多少行，只要沒有循環或其他複雜的結構，這段程式碼的時間複雜度就一直是 $O(1)$。
- 單行程式碼不一定是 $O(1)$，看到函式的時候，要了解函式背後的實作細節，才能計算複雜度。

```
1 int i = 10, j = 3;
2 int k = i++;
3 int n = i + j;
4 isdigit('1');
5
6 //注意：以下函式並非O(1)
7 int len = strlen("abcde");
8 pow(2, n);
9 sqrt(n);
```

線性時間 $O(1)$

- EX1 迴圈從 1 執行到 n，因此時間複雜度為 $O(n)$。
- EX3 這邊有三個迴圈，但不代表時間複雜度會是 $O(n^3)$。第一個迴圈跑 3 次，第二個迴圈跑 n 次，第三個迴圈跑 5 次，相乘起來是 $O(3*n*5)$。然而隨著 n 的增加，常數對於整體時間複雜度的影響變得越來越不重要，因此時間複雜度綜合而言是 $O(n)$。

```
1  //EX1:
2  for (int i = 1; i <= n; i++) {
3      ... //some O(1) operations
4  }
5  //EX2:
6  for (int i = 1; i <= 3; i++) {
7      for (int i = 1; i <= n; i++) {
8          for (int i = 1; i <= 5; i++) {
9              ... //some O(1) operations
10         }
11     }
12 }
```

對數時間 $O(\log n)$

- $O(\log n)$ 一個經典的例子就是二分搜尋。
- 假設陣列大小為 n，陣列中的數值以遞增排列，每次迴圈查找都會將陣列大小減半，因此總共需要 $\log_2 n$ 次查找。故二分搜尋的時間複雜度是 $O(\log n)$。

```
1  int binarySearch(int* nums, int n, int target) {
2      int left = 0, right = n - 1;
3      while (left <= right) {
4          int mid = (right - left) / 2;
5          if (nums[mid] < target)
6              left = mid + 1;
7          else if (target < nums[mid])
8              right = mid - 1;
9          else
10             return mid;
11     }
12 }
```

時間複雜度 (Time Complexity)

指數時間 $O(2^n)$

- 用遞迴計算費式數列的 fib(n) 時，需要先計算 fib($n-1$) 和 fib($n-2$)，每次遞迴需要解決兩個子問題，因此其時間複雜度為 $O(2^n)$。

```
1 int fib(int n) {
2     if (n <= 1) {
3         return n;
4     }
5     return fib(n - 1) + fib(n - 2);
6 }
```

時間複雜度 (Time Complexity)

- 綜合以上所學，我們再來看這個例子，上半部的部分複雜度為 $O(n^2)$，下半部的部分複雜度為 $O(n)$，加起來複雜度為 $O(n^2+n)$。
- 但隨著 n 的數值越來越大，n 對比 n^2 變得十分苗小，因此 $O(n)$ 可以省略，故綜合起來複雜度為 $O(n^2)$。

```
1  //上半部 O(n^2)
2  for(int i=0;i<n;i++){
3      for(int j=0;j<n;j++){
4          int num = i + j;
5      }
6  }
7  //下半部 O(n)
8  for(int i=0;i<n;i++){
9      int num = i;
10 }
```

時間複雜度 (Time Complexity)

- 學習如何估算時間複雜度後，各位未來在寫題目時，可以透過題目給定的測資範圍，思考自己的解題方法是否符合題目的時間限制要求。

- 註: 以大部分 Online Judge 的設備而言，C 語言執行一秒的速度，大約可以容忍演算法複雜度在 10^8~10^9 左右。

Q & A