Algorithms

授課老師：張景堯

Program Analysis

Chapter.2-Algorithm Analysis

Reasoning About Efficiency

- Grossly reasoning about the running time of an algorithm is usually easy given a precise-enough written description of the algorithm.(精確描述演算法讓推導時間複雜度更加容易)
- When you *really* understand an algorithm, this analysis can be done in your head. However, recognize there is always implicitly a written algorithm/program we are reasoning about.

Selection Sort

- https://www.youtube.com/watch?v=Ns4TPTC8whw

```
def selection_sort(s):
    min_index = 0
    for i in range(len(s)):
        min_index = i
        for j in range(i+1,len(s)):
            if s[j] < s[min_index]:
                min_index = j
        if min_index != i:
            s[min_index], s[i] = s[i], s[min_index]
    return s
```

Worst Case Analysis

- The outer loop goes around n times.
- The inner loop goes around at most n times for each iteration of the outer loop
- Thus selection sort takes at most $n \times (n-1) \rightarrow O(n^2)$ time in the worst case.
- In fact, it is $\Theta(n^2)$, because at least $n/2$ times it scans through at least $n/2$ elements, for a total of at least $n^2/4$ steps $\rightarrow \Omega(n^2)$.

More Careful Analysis

- An exact count of the number of times the *if* statement is executed is given by:

$$S(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1} (n - i - 1) = \sum_{i=0}^{n-1} i$$

$$S(n) = (n-1) + (n-2) + (n-3) + \cdots + 2 + 1 = n(n+1)/2$$

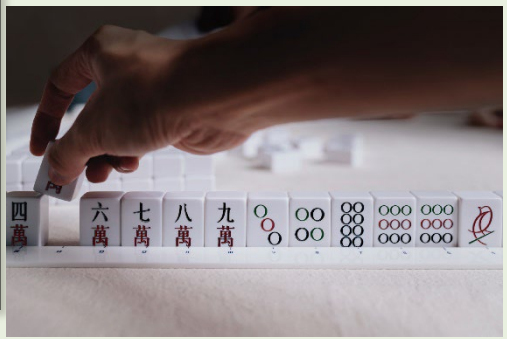
- Thus the worst case running time is $\Theta(n^2)$.

Insertion Sort

- https://www.youtube.com/watch?v=ROalU379l3U

```
def insertion_sort(s):
    for i in range(1,len(s)):
        j = i
        while j > 0 and s[j] < s[j-1]:
            s[j], s[j-1] = s[j-1], s[j]
            j = j-1
    return s
```

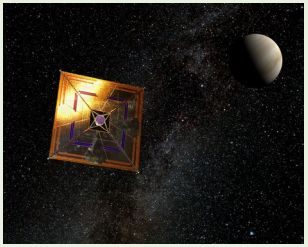
I N S E R T I O N S O R T
I N S E R T I O N S O R T
I N S E R T I O N S O R T
E I N S R T I O N S O R T
E I N R S T I O N S O R T
E I N R S T I O N S O R T
E I I N R S T O N S O R T
E I I N O R S T N S O R T
E I I N N O R S T S O R T
E I I N N O R S S T O R T
E I I N N O O R S S T R T
E I I N N O O R R S S T T
E I I N N O O R R S S T T



Insertion Sort Analysis

- This involves a while loop, instead of just for loops, so the analysis is less mechanical.
- But n calls to an inner loop which takes at most n steps on each call is $O(n^2)$.
- The reverse-sorted permutation proves that the worst-case complexity for insertion sort is $\Theta(n^2)$.
- $(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)$

Solar Sails vs. Rockets(太陽帆與火箭)



- The bad-ass rocket hits a high speed before it runs out of fuel, then coasts at constant speed v_r.
- The solar sail slowly accelerates from the force of radiation/solar wind hitting it, but its speed of $v_s = at$ must eventually exceed the bad-ass rocket.
- This is asymptotic dominance in action.

Asymptotic Dominance in Action

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Implications of Dominance(支配)

- Exponential algorithms get hopeless fast.
- Quadratic algorithms get hopeless at or before 1,000,000.
- $O(n \log n)$ is possible to about one billion.
- $O(\log n)$ never sweats.

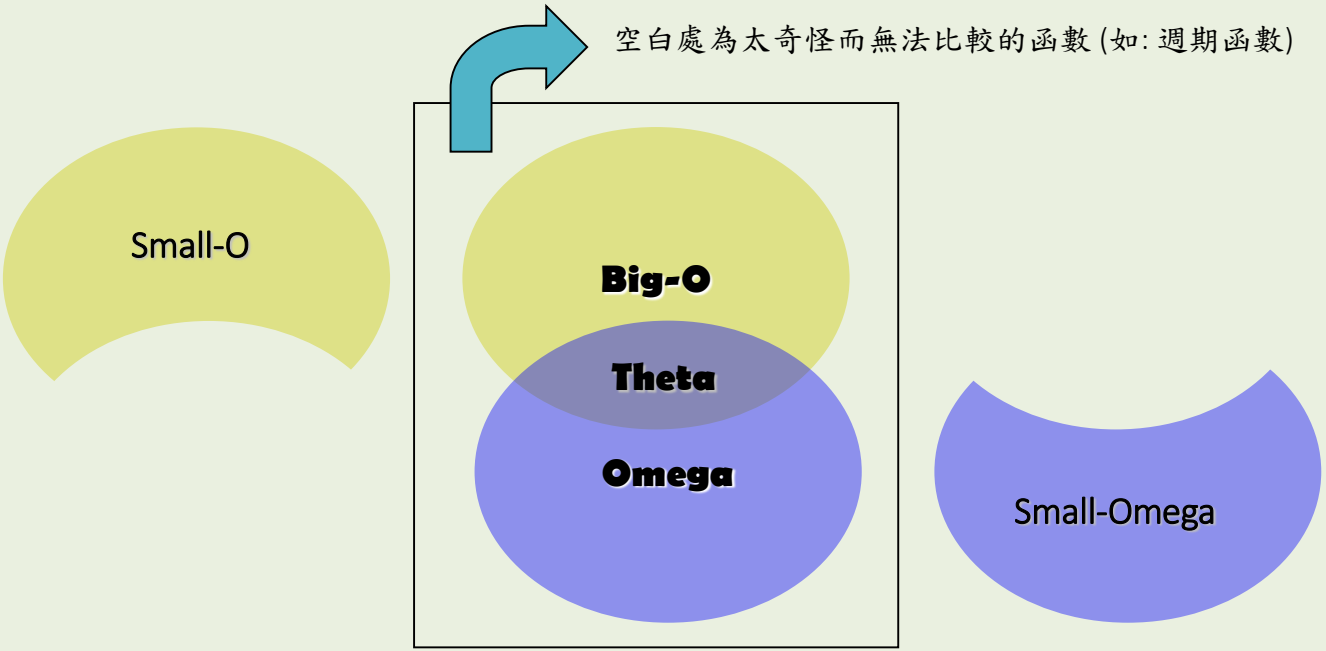
Testing Dominance(檢驗支配性)

- $g(n)$ dominates $f(n)$ if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

, which is the same as saying $f(n) = o(g(n))$.

- Note the little-oh – it means “grows strictly slower than”.


空白處為太奇怪而無法比較的函數 (如: 週期函數)
Small-O
Big-O
Theta
Omega
Small-Omega

Properties of Dominance

- n^a dominates n^b if $a > b$ since

$$\lim_{n\to\infty} \frac{n^b}{n^a} = n^{b-a} \to 0$$

- $n^a + o(n^a)$ doesn't dominate n^a since

$$\lim_{n\to\infty} \frac{n^a}{n^a + o(n^a)} \to 1$$

Dominance Rankings

- You must come to accept the dominance ranking of the basic functions:

$$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

Advanced Dominance Rankings

- Additional functions arise in more sophisticated analysis than we will do in this course:

$$n! \gg c^n \gg n^3 \gg n^2 \gg n^{1+\epsilon} \gg n \log n \gg n \gg \sqrt{n} \gg \log^2 n \gg \log n \gg \log n / \log\log n \gg \log\log n \gg \alpha(n) \gg 1$$

Logarithms

Chapter.2-Algorithm Analysis

Logarithms(對數)

- It is important to understand deep in your bones what logarithms are and where they come from.
- A logarithm is simply an inverse exponential function. Saying $b^x = y$ is equivalent to saying that $x = \log_b y$.
- Logarithms reflect how many times we can double something until we get to n, or halve something until we get to 1.

Binary Search(二元搜尋)

- In binary search we throw away half the possible number of keys after each comparison. Thus twenty comparisons suffice to find any name in the million-name Manhattan phone book! How many time can we halve n before getting to 1?

https://www.youtube.com/watch?v=iP897Z5Nerk

- Answer: $\lceil \log_2 n \rceil$

Logarithms and Trees(二元樹)

- How tall a binary tree do we need until we have n leaves?
  The number of potential leaves doubles with each level.
  How many times can we double 1 until we get to n?

- Answer: $\lceil \log_2 n \rceil$

Logarithms and Bits

- How many bits do you need to represent the numbers from 0 to $2^i - 1$?

- Each bit you add doubles the possible number of bit patterns, so the number of bits equals $\log_2 2^i = i$.

Logarithms and Multiplication

- Recall that

$$\log_a(xy) = \log_a(x) + \log_a(y)$$

- Because,

$$a^{\log_a(xy)} = xy = a^{\log_a x} \cdot a^{\log_a y} = a^{(\log_a x + \log_a y)}$$

- This is how people used to multiply before calculators, and remains useful for analysis.

The Base is not Asymptotically Important

- Recall the definition, $c^{\log_c x} = x$ and that

$$a = b^{\log_b a} = \left(c^{(\log_c b)}\right)^{\log_b a} = c^{(\log_c b \cdot \log_b a)} = a = c^{(\log_c a)}$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

- Thus $\log_2 n = \left(\frac{1}{\log_{100} 2}\right) \times \log_{100} n$. Since $\frac{1}{\log_{100} 2} = 6.643$ is just a constant, it does not matter in the Big Oh.

Elementary Data Structures-Array

Chapter.3-Data Structures

Elementary Data Structures

- "Mankind's progress is measured by the number of things we can do without thinking."(人類的進步幅度取決於我們可以不假思索地做多少事情。)
- Elementary data structures such as stacks, queues, lists, and heaps are the "off-the-shelf" components we build our algorithm from.(很多基本資料結構都已經是"現成"的了)
- There are two aspects to any data structure:
  - The abstract operations which it supports.(操作方式)
  - The implementation of these operations.(方法建置)

Data Abstraction(抽象概念)

- That we can describe the behavior of our data structures in terms of abstract operations is why we can use them without thinking.
  - $Push(x, s)$– Insert item x at the top of stack s.
  - $Pop(s)$ – Return (and remove) the top item of stack s.
- That there are different implementations of the same abstract operations enables us to optimize performance in difference circumstances. (相同操作方式會根據現實環境考慮最優效能而有不同的實際建置作法)

Contiguous vs. Linked Data Structures (連續型與鏈結型資料結構)

- Data structures can be neatly classified as either *contiguous* or *linked* depending upon whether they are based on arrays(tuples, lists) or pointers(references):
  - Contiguously-allocated structures are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
  - Linked data structures are composed of multiple distinct chunks of memory bound together by *pointers(references)*, and include lists, trees, and graph adjacency lists.

Arrays(陣列)

- An array is a structure of fixed-size data records such that each element can be efficiently located by its *index* or (equivalently) address.
- Advantages of contiguously-allocated arrays include:
  - Constant-time access given the index. O(1)
  - Arrays consist purely of data, so no space is wasted with links or other formatting information.
  - Physical continuity (memory locality) between successive data accesses helps exploit the high-speed cache memory on modern computer architectures.

Dynamic Arrays(動態陣列)

- Unfortunately we cannot adjust the size of simple arrays in the middle of a program's execution.(C-like Language)
- Compensating by allocating extremely large arrays can waste a lot of space.(陣列無法在執行期改變大小，若預先分配極大的空間給陣列，不見得都用到會很浪費)
- With *dynamic arrays* we start with an array of size 1, and double its size from m to $2m$ each time we run out of space.
- How many times will we double for n elements?

- Only $\lceil \log_2 n \rceil$.
(Python程式以可變物件list串列提供動態陣列功能)

Linked List

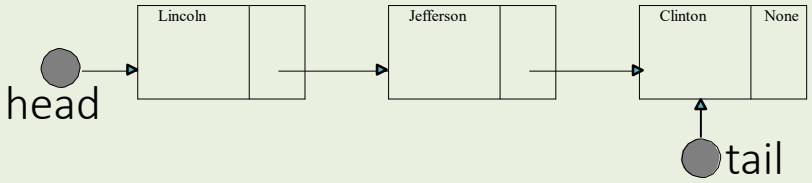
Chapter.3-Data Structures

Pointers and Linked Structures

- Pointers represent the address of a location in memory.
- A cell-phone number can be thought of as a pointer to its owner as they move about the planet.
- In Python, ListNode can be represented as a class. The ListNode class contains a reference of ListNode class type. That is, each node has two entries - a data attribute, and a next attribute, which points to the next node in the list.
- A special **`None`** value is used to denote structure-terminating or unassigned pointers.

Linked List Structures(鏈接串列)

```
class ListNode:
    def __init__(self, data):
        self.data = data
        self.next = None
```



Searching a List

- Searching in a linked list can be done iteratively or recursively. O(n)

```
def search(self, key):
    L = self
    while L and L.data != key:
        L = L.next
    return L #If key was not present in the linked list, L will have become None.
def search_rec(self, key):
    if self.data == key:
        return self
    elif self.next is None:
        return None
    else:
        return self.next.search_rec(key)
```

Insertion into a List

- Since we have no need to maintain the list in any particular order, we might as well insert each new item at the head. O(1)

```
def insert_after(self, new_node):
    new_node.next = self.next
    self.next = new_node
```

Deleting from a List

```
#Delete the node immediately following a_node.
#Assumes a_node is not a tail.
def delete_nextnode(self):
    self.next = self.next.next
def delete_thisnode(self):
    self.data = self.next.data
    self.next = self.next.next
```

Print All Nodes in a Linked List

- Print the data of all subsequent nodes include itself.

```
def print_list(self):
        L = self
        while L:
            print(L.data, end=' ')
            L = L.next
        print()
```

Implement iterator for ListNodes

- Print the data of all subsequent nodes include itself.

```
def __iter__(self):
    current = self
    while current is not None:
        yield current # suspend and output current ListNode object
        current = current.next
```

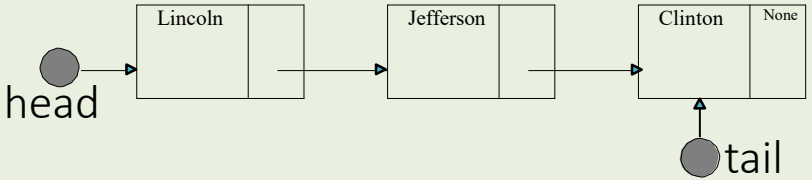
Print All Nodes in a Linked List v2

- Print the data of all subsequent nodes include itself.

```
def __str__(self):
    return "->".join([L.data for L in self])
```

Create a Linked List

- Construct a linked list as depicted.



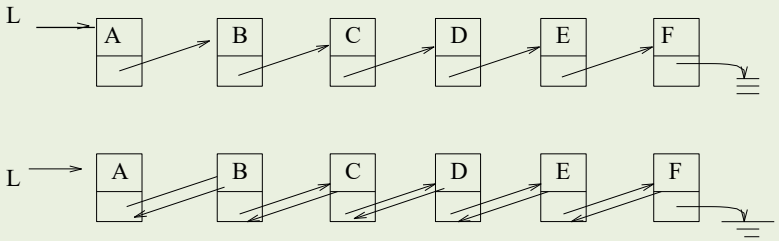

```
presidents=['Lincoln', 'Jefferson', 'Clinton']
head = None
for i in presidents:
    if head is None:
        head = ListNode(i)
        tail = head
    else:
        tail.next = ListNode(i)
        tail = tail.next
```

Advantages of Linked Lists

- The relative advantages of linked lists over static arrays include:
  1. Overflow on linked structures can never occur unless the memory is actually full.
  2. Insertions and deletions are *simpler* than for contiguous array(python tuple).
  3. With large records, moving pointers is easier and faster than moving the items themselves.
- Dynamic memory allocation provides us with flexibility on how and where we use our limited storage resources.
(動態配置讓我們能靈活的運用有限的記憶體存儲資源)

Singly or Doubly Linked Lists

- We gain extra flexibility on predecessor queries at a cost of doubling the number of pointers by using doubly-linked lists.



- Since the extra big-Oh costs of doubly-linked lists is zero, we will usually assume they are so maintained, although it might not always be necessary.

Singly or Doubly Linked Lists (Cont.)

- Singly linked to doubly-linked list is as a conga line is to a can-can line.

Stacks and Queues

Chapter.3-Data Structures

Stacks and Queues(堆疊與佇列)

- Sometimes, the order in which we retrieve data is independent of its content, being only a function of when it arrived.
- A *stack* supports last-in, first-out operations:
  - $Push(x, s)$– Insert item x at the top of stack s.
  - $Pop(s)$ – Return (and remove) the top item of stack s.
- A *queue* supports first-in, first-out operations:
  - $Enqueue(x, q)$ – Insert item x at the back of queue q.
  - $Dequeue(q)$ – Return (and remove) the front item from queue q.

Stack/Queue Implementations

- Stacks are more easily represented as an array, with push/pop incrementing/decrementing a counter.
- Queues are more easily represented as a linked list, with enqueue/dequeue operating on opposite ends of the list.

- All operations can be done in $O(1)$ time for both structures, with both arrays and lists.

Why Stacks and Queues?

- Both are appropriate for a container class where order(content) doesn't matter, but sometime it(processing order) does matter.
- Lines in banks are based on queues, while food in my refrigerator is treated as a stack.
- The entire difference between *depth-first search (DFS)* and *breadth-first search (BFS)* is whether a stack or a queue holds the vertices/items to be processed.

Dictionaries

Chapter.3-Data Structures

Dictonary / Dynamic Set Operations

- Perhaps the most important class of data types maintain a set of items, indexed by *keys*.
  - $Search(S, k)$ – A query that, given a set S and a key value k, returns a pointer x to an element in S such that $key[x] = k$ ($S[k]$ is x), or nil ($None$) if no such element belongs to S.
  - $Insert(S, x)$ – A modifying operation that augments the set S with the element x.
  - $Delete(S, x)$ – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key value.

Dictonary / Dynamic Set Operations (Cont.)

- $Min(S), Max(S)$ – Returns the element of the totally ordered set S which has the smallest (largest) key(,element).
- Logical $Predecessor(S, x), Successor(S, x)$ – Given an element x whose key is from a totally ordered set S, returns the next smallest (largest) element in S, or nil (None) if x is the maximum (minimum) element.

- There are a variety of implementations of these *dictionary* operations, each of which yield different time bounds for various operations.
- There is an inherent **tradeoff** between these operations. We will see that no single implementation will achieve the best time bound for all operations.

Array Based Sets: Unsorted Arrays

- $Search(S, k)$ – sequential search, $O(n)$
- $Insert(S, x)$ – place in first empty spot, $O(1)$
- $Delete(S, x)$ – copy n-th item to the x-th spot, $O(1)$
- $Min(S, x), Max(S, x)$ – sequential search, $O(n)$
- $Successor(S, x), Predecessor(S, x)$ – sequential search, $O(n)$

Array Based Sets: Sorted Arrays

- $Search(S, k)$ – binary search, $O(\log n)$
- $Insert(S, x)$ – search, then move to make space, $O(n)$
- $Delete(S, x)$ – move to fill up the hole, $O(n)$
- $Min(S, x), Max(S, x)$ – first or last element, $O(1)$
- $Successor(S, x), Predecessor(S, x)$ – Add or subtract 1 from pointer, $O(1)$

Exercise

Dictonary / Dynamic Set Operations

- Perhaps the most important class of data structures maintain a set of items, indexed by *keys*.
  - $Search(S, k)$ – A query that, given a set S and a key value k, returns a pointer x to an element in S such that $key[x] = k$, or nil if no such element belongs to S.
  - $Insert(S, x)$ – A modifying operation that augments the set S with the element x.
  - $Delete(S, x)$ – Given a pointer x to an element in the set S, remove x from S. Observe we are given a pointer to an element x, not a key value.
  - $Min(S), Max(S)$ – Returns the element of the totally ordered set S which has the smallest (largest) key.
  - Logical $Predecessor(S, x), Successor(S, x)$ – Given an element x whose key is from a totally ordered set S, returns the next smallest (largest) element in S, or nil if x is the maximum (minimum) element.

Problems of the Day

- What is the asymptotic worst-case running times for each of the seven fundamental *dictionary* operations when the data structure is implemented as
  - A singly-linked unsorted list,
  - A doubly-linked unsorted list,
  - A singly-linked sorted list, and finally
  - A doubly-linked sorted list.

Solution Blank

Dictionary operation	singly unsorted	doubly unsorted	singly sorted	doubly sorted
Search(L, k)	$O(n)$	$O(n)$	$O(n)$	Q1
Insert(L, x)	Q2	$O(1)$	$O(n)$	$O(n)$
Delete(L, x)	$O(n)$*	$O(1)$	$O(n)$*	$O(1)$
Successor(L, x)	$O(n)$	$O(n)$	Q3	$O(1)$
Predecessor(L, x)	$O(n)$	$O(n)$	$O(n)$*	Q4
Minimum(L)	$O(1)$*	Q5	$O(1)$	$O(1)$
Maximum(L)	$O(1)$*	$O(n)$	$O(1)$*	$O(1)$

Problems of the Day (Cont.)

- We can find two functions $f(n)$ and $g(n)$ that satisfy the following relationship. It's True or False.
  - $f(n) = o(g(n))$ and $f(n) \neq \Theta(g(n))$
  - $f(n) = \Theta(g(n))$ and $f(n) \neq O(g(n))$
  - $f(n) = \Omega(g(n))$ and $f(n) \neq O(g(n))$
    - *Recall that little oh o(n) means "grows strictly small than".*
- What is the Big-O function of the shown equation?

$$2^n + n^4 + n^2$$

Program Exercises (Moodle CodeRunner)

- Exercise 02 (close at 3/11 23:59)
  - Sudoku Validation:
    - *Check whether a 9 x9 2D array representing a partially completed Sudoku is valid. Specifically, check that no row, column, or 3 x 3 2D subarray contains duplicates. A 0-value in the 2D array indicates that entry is blank; every other entry is in 1,…,9.*
  - Test for Cyclicity:
    - *Write a method that takes the head of a singly linked list and returns None if there does not exist a cycle, and the node at the start of the cycle, if a cycle is present, (You do not know the length of the list in advance.)*