**問題：**

1. **證明基於比較的排序演算法不可能快過 O(n \* log n)。**
2. **將遞迴版 Binary Search 的時間複雜度寫出來，並用 Master Theorem 分析。**
3. **將遞迴版 Binary Search 改寫成非遞迴（使用 while）的版本。**
4. **分析使用 Intermediate Value Theorem 解方程時的最壞情況時間複雜度。**
5. **在 k 分搜尋中，什麼樣的 k 可以讓搜尋最快？**
6. **任一基於比較的排序演算法可視為決策樹，需判斷 n! 種排列**

**最少需要 log₂(n!) ≈ n log n 個比較**

**所以不可能快於 O(n log n)**

1. **遞迴式：T(n) = T(n/2) + 1  
   用 Master Theorem 分析：**

**a = 1, b = 2, f(n) = 1**

**f(n) = O(n^log\_b a - ε) = O(n^0 - ε)**

**屬於 Master Theorem Case 1**

**時間複雜度：Θ(log n)**

1. **template <typename T>**

**int find(const T data[], int n, const T key) {**

**int left = 0, right = n - 1;**

**while (left <= right) {**

**int mid = (left + right) / 2;**

**if (data[mid] == key)**

**return mid;**

**else if (data[mid] < key)**

**left = mid + 1;**

**else**

**right = mid - 1;**

**}**

**return n; // 沒找到**

**}**

1. **解精度為 ε 時所需次數為  
   log₂((b - a)/ε)**

**若每次計算 f(x) 花 O(k) 時間，則：**

**時間複雜度為 O(k \* log((b - a)/ε))**

1. **設 T(k) = (k - 1) \* logₖ(n)，最小化此值**

* **當 k 趨近於 e ≈ 2.718 時最小**
* **實際使用中 k 必須為整數，最小值在 k = 2**
* **所以 Binary Search（k = 2）仍是最有效的搜尋策略**

**Lab 03：分治法（二）- 二元搜尋（Binary Search）**

排序是將一組資料依照某種意義上的順序進行排列的過程，例如按照從小到大的順序。

定義：遞增排序

對於一個串列 A = {a1, a2, ..., an}，如果對所有的 i 滿足 1 < i <= n，都有 a(i-1) <= a(i)，那麼我們就稱這個串列是「遞增排序」的。

排序過的資料通常能讓我們的工作變得更簡單。

舉例來說，在一個串列 A 中找出最大值，最壞情況下必須花費 O(n) 時間，因為至少要看過所有資料一次，才能保證正確性。因此時間複雜度不可能比 O(n) 更快。

一個線性時間找最大值的方法如下：

**template <typename T>**

**T& max(const T data[], int n) {**

**int ret = 0;**

**for (int i = 1; i < n; ++i) {**

**if (data[ret] < data[i])**

**ret = i;**

**}**

**return data[ret];**

**}**

但如果我們的資料已經是遞增排序，只要回傳最後一個元素就行了，時間是 O(1)：

**template <typename T>**

**T& max(const T sorted\_data[], int n) {**

**return sorted\_data[n-1];**

**}**

不過要注意排序本身也是一筆成本。上面找最大值只需 O(n)，但排序需要 O(n \* log n)，反而更慢**。**

**問題：試著證明任何基於「比較」的排序演算法不可能快過 O(n \* log n)。**

提示：n 個元素會有 n! 種排列方式，而每次比較只有兩種結果（true 或 false）。

**二元搜尋法（Binary Search）**

現在我們討論一個更實用的問題：給定一個串列 A = {a1, a2, ..., an} 和一個查詢值 k，想找出是否存在某個元素 ai 等於 k。

用最簡單的方法是線性搜尋，時間為 O(n)：

**template <typename T>**

**int find(const T data[], int n, const T key) {**

**for (int i = 0; i < n ; ++i) {**

**if (data[i] == key)**

**return i;**

**}**

**return n; // 沒找到就回傳 n 當作「沒找到」的標記**

**}**

如果資料是排序好的，我們可以使用更快的方法。考慮選一個中間的元素 ak：

* 對於所有 i < k，一定有 ai <= ak
* 對於所有 i > k，一定有 ai >= ak

根據這個性質可以寫出遞迴版本的二元搜尋：

**template <typename T>**

**int find(const T sorted\_data[], int n, const T key) {**

**if (n <= 0)**

**return 0;**

**int k = n/2;**

**if (sorted\_data[k] == key)**

**return k;**

**else if (sorted\_data[k] > key) {**

**int tmp = find(sorted\_data, k, key);**

**return (tmp == k) ? tmp : n;**

**} else {**

**int tmp = find(sorted\_data + k + 1, n - k - 1, key);**

**return tmp + k + 1;**

**}**

**}**

這個做法叫做「Binary Search（二分搜尋）」，每次都切一半搜尋，因此時間複雜度為 O(log\_2 n)，比 O(n) 快非常多。

雖然排序的前處理要 O(n \* log n)，但如果你要查詢的次數超過 log n 次，那麼整體效能會優於線性搜尋。

實務上，資料量越大，被重複查詢的可能性就越高。因此 Binary Search 是許多資料庫系統的首選演算法。

**問題：請寫出遞迴演算法時間複雜度遞迴式，並用 Master Theorem 分析其時間。**

**問題：請將上面的程式改寫成使用 while 迴圈的版本（非遞迴）。**

**二分解連續函數解（Intermediate Value Theorem）**

Binary Search 不只可以搜尋元素，也可以應用於**連續函數的根的求解**。

中間值定理（Intermediate Value Theorem）：

對於一個連續函數 f: [a, b] → ℝ，如果 (f(a) - u)(f(b) - u) <= 0，那一定存在c ∈ [a, b]，使得 f(c) = u 根據這個定理，我們可以設計出以下二分法的程式來求解：

**#include <cmath>**

**double solve(double (\*f)(double), double a, double b, double val, double epi) {**

**if ((f(a) - val) \* (f(b) - val) > 0)**

**return NAN;**

**double mid = (a + b) \* 0.5;**

**if (abs(f(mid) - val) <= epi)**

**return mid;**

**else if ((f(a) - val) \* (f(mid) - val) <= 0)**

**return solve(f, a, mid, val, epi);**

**else**

**return solve(f, mid, b, val, epi);**

**}**

**問題：上面這個函數最壞情況的時間複雜度是什麼？**

提示 1：跟精度 epi 有關 提示 2：不要忘了 f 本身的運算時間也是成本

**k 分搜尋（k-ary search）**

我們不一定只能將資料切成 2 半來搜尋，理論上也可以切成 k 半。

以下是 k 分搜尋的做法（以 vector 為例）：

**#include <vector>**

**template <typename T>**

**int find(const std::vector<T> &data, const T key, int k, int lb, int ub) {**

**if (ub - lb < k) {**

**for (int i = lb; data[i] <= key && i < ub; ++i)**

**if (data[i] == key)**

**return i;**

**return data.size();**

**}**

**int periods[k - 1];**

**for (int i = 1; i < k; ++i)**

**periods[i-1] = lb \* i + ub \* (k-i) / k;**

**if (key < data[periods[0]])**

**return find(data, key, k, lb, periods[0]);**

**if (key == data[periods[0]])**

**return periods[0];**

**for (int i = 1; i < k-1; ++i) {**

**if (key < data[periods[i]])**

**return find(data, key, k, periods[i-1]+1, periods[i]);**

**if (key == data[periods[i]])**

**return periods[i];**

**}**

**return find(data, key, k, periods[k-2]+1, ub);**

**}**

**int find(const std::vector<T> &data, const T key, int k) {**

**return find(data, key, k, 0, data.size()); }**

**問題：在搜尋 n 筆資料時，什麼樣的 k 可以讓搜尋最快？**