Search部分

Linear (Sequential) Search(線性、循序搜尋)

定義:從頭到尾，一一比較各資料之鍵值，直到找到符合者，或搜尋完整資料皆找不到為止

分析:

1. Data不須事先經過牌序，即可Search
2. Data之保存在Random Access => Array，Sequential Access => Link List皆可
3. Time = O(n)

Binary Search(二分搜尋法)

前提:

1. Data必須事先排序過(小=>大)
2. Data保存在Random Access(例:Array)機制上

觀念:

每次都跟中間位置之Data比較:

1. “==”: 找到
2. “<”: 落在左半部
3. “>”: 落在右半部

N筆Data用Binary Search 找X最多比較次數?

Ans:最多比較次數 = 樹高，而Decision Tree是Binary Tree，且高度最小化，

所以n個Data最小高度=log2(n+1)取上限，也就是等於最多比較次數

N筆Data以Binary Search Tree Search X:

Worst case: O(n)，for skewed tree

Best case: O(logn)

N筆Data以Binary Search，Search X:

Worst case: O(logn)

Sort部分

Internal and External Sort

Internal Sorting: 資料量少，可以一次全部置於memory中，進行排序工作

External Sorting: 資料量太大，無法一次全部置入memory中，必須借助外部儲存體(for Disk)保存，再行排序工作

常用的External Sorting Method: Merge Sort、Selection Tree、m-way search Tree、B tree

Stable and Unstable Sorting Method

定義:

在input Data中，可能會存在多個鍵值相同之Data

例: input: …,k,k\*,….

排序後，若保證:…k,k\*,… => 此sorting method = stable

反之，k\*可能會在k前面 => 此sorting method = unstable

Note:

1. Unstable代表產生不必要的SWAP
2. 所有Unstable sorting method必定比 stable sorting method還慢
   * False，因為quicksort為unstable, but 初等排序:Insert、Bubble皆為stable，但都比quicksort慢

初等排序:

Insertion Sort(插入排序)

觀念: i = 2 to n 做(n-1)回合

將第i筆Data插入到前面(i-1)筆排好之串列中之正確位置，形成n筆排好之串列

Time:

1.Best case:O(n) => 當input Data剛好是小->大呈現時

2.Worst case:O(n平方) => 當input Data剛好是大->小呈現時

3.Avg case:O(n平方)

Space:

O(1)

Stable

Selection Sort(選擇排序)

觀念:

自第i到n筆Data中，找出最小值，然後再跟第i筆做交換

做(n-1)個合併

Time:

Best、Avg、Worst = O(n平方)

說明:任何情況下，A[j]<A[min]的比較次數皆為(n-1)+(n-2)+.....+(1) = O(n平方)

Space:

額外空間需求:固定O(1)，與n無關

Unstable

補充:

1. Selection Sort每一回合，頂多SWAP一次
2. 適用在排序大型紀錄(紀錄是由很多欄位組成)上 for database

Bubble Sort(氣泡排序)

定義:

由左而右，依序兩兩相互比較，若前者>後者，則SWAP(前、後)

每一回合的效果:最大值開到最高位置，上述回合數頂多做到(n-1)回合，因為如果某一回合如果都沒有SWAP，則代表Sort完成，可直接Exit

Time:

Best case:O(n)，當input為小->大呈現時

Worst case:O(n平方)，當input為大->小呈現時

Avg case:O(n平方)

Space:

O(1)，額外空間需求是固定的 => flag、Swap函數之temp

Stable

Shell Sort

觀念:

比較A[i]與A[i+span]，若前者>後者，則Swap(前、後)

1. 每一回合須持續做到No Swap，才可進入下一回合
2. 回合數由Span型式控制
3. 最後回合Span要訂1

Time:

Avg case:O(n平方)

Worst case:O(n平方)

Best case:依Span型而定，到目前為止沒有定論 => 考試寫 O(n的2分之3的次方)

Space:

O(1)

Unstable

高等排序(avg case: O(nlogn))

Quick Sort(快速排序法)[DS版本]

Avg case下，實際執行時間最快的方法

採用”Divide-and-Conquer”策略

Divide-and-Conquer定義:

將problem切成幾個獨立的subproblems(無Overlap)，每個Subproblems各自求解，最後合併所有subproblems solutions成為整個problem之solution

Note:與Dynamic Programming相比，有2點不同

1. D.P之subproblem可以overlay(重疊)
2. D.P之Subproblem之solution有保存，將來可以reuse，不用再重算，但Divide-and-Conquer沒有，即相同subproblem要重算一次

Note: Quick Sort、Merge Sort、Binary Search 皆採Divide-and-Conquer

Quick Sort之觀念

1. 最左邊Data視為Pivot Key(PK)
2. 經過Partition處理後，PK會被置於最正確位置上，即左右兩個sublists可各自排序，當左右兩邊皆排好，則整個list也Sorted

分析:

Time:

Best case:O(nlogn) => Pk之位置剛好切割成左、右兩等分時

Worst case:O(n平方) => 切割無效果，當pk剛好是最小值或最大值的時候

形成 => 0筆,PK,(n-1)筆 或 (n-1筆),PK,0筆 => 只有一個Subproblem left

如何避免worst case:

原則:防止PK為min or Max值

1. Algo版 => Randomized Quick Sort，用亂數挑PK，但已被證出無效
2. Algo版 => 使用medium-of-mediums 作為PK
3. 使用middle-of-three作為PK

Steps:

1. m = (l+u)/2;
2. 比較A[l]、A[m]、A[u]，這三筆Data，取中間值跟A[l]互相SWAP成為PK
   * 時間變成O(nlogn)

Space:

O(logn)~O(n)

額外空間來自於recursion所需之Stack space，而Stack size取決於recursive call depth(次數)

Note:Data都一樣，algo版視為Worst case => O(n平方)

如何改善:

1.先花O(n)time check是否所有data皆一樣，若一樣，則Exit

2.改用DS版本的partition => O(nlogn) => 因為DS版本視為Best case

到底排序可達多快?

1. 假設排序技巧是採用Comparison-based skill (or Comparison & swap)，則sorting 最快可達Ω(nlogn)

初等及高等排序方法皆是Comparison-based skill

1. 若排序技巧並非採用Comparison Skill，則不受此限制(有可能突破)

For Linear-time sorting method

Merge Sort (合併排序)

1. External Sorting常用的方法
2. Run: 排序好的片段資料串列
3. Run長度:Run中的Data個數
4. K-way merge: 一次合併K個Runs成一個Run

For 2-way、4-way、8-way….

1. 分為兩個版本 => Iterative、Recursive

Iterative 2-way Merge Sort

若Run1,Run2長度分別是m,n

則合併兩個Runs成一個Run

1. 最少比較次數: m or n次 => 若某個Run的Data皆小於另一個Run之Data
2. 最多比較次數: m + n -1 次 => 某個Run Scan完了，另一個Run剩下一個Data
3. Merge Sort每一回合之合併處理時間為O(n)
4. Iterative 2-way merge Sort n個Data，需要log2n取上限回合
   * 高度of BT = log2n取上限+1，回合數=高度-1
   * Merge Sort Time = log2n取上限 \* O(n) = O(nlogn)

Recursive Merge Sort

1. 採用”Divide-and-Conquer”策略
2. Steps:
3. Data list一律對切二半
4. 左、右半部各自Merge Sort
5. 合併左、右兩個Runs成一個Run

分析:

Avg/Best/Worst case: O(nlogn)

Space: O(n) => 額外空間需求是用來保存每一回之合併結果

Selection Tree(選擇樹)

1. 目的: 協助加速k-way merge(k>>2)之進行
2. 在沒有Selection Tree情況下:

假設k-way merge，k個Runs之Data數=n

分析:每次從K個Runs中挑出最小值，要花k-1次比較，最多要找(n-1)次min動作，total花O(n\*k)time

1. Selection Tree如何加入?

分為兩種 => Winner Tree、Loser Tree，兩者之Time皆相同

Winner Tree

Time分析:

1. 建Tree部分 => O(k) Time
   1. 每個Run中最小資料Copy成為Leaf => 花O(k)time
   2. 經過k-1次比較後，決定出Root(兩兩sibling比) => 花O(k)time
   3. 輸出root值到new run中，此時剩下n-1個資料
2. 要做n-2次挑min(決定Root)之動作

每一次挑min值經過log2k取上限次比較而決定

Total:花O(n\*logk)time

Loser Tree

同Winner Tree，輸家罰站

Heap Sort

Steps:

1. 先將n個Data以Button-Up方式建立Max-Heap
2. 執行(n-1)個回合之排序工作

每一回合執行類似”Del-Max”工作，即Swap(root,當時the lastNode)，對剩下的Data調成Heap

分析:

Time => Avg/Best/Worst case:O(nlogn)

說明:

1.建立Heap:花O(n)time

2.執行(n-1)次回合，每一回合花O(logn)time執行類似"Del-Max"運作，所以花O(nlogn)

所以total: O(n)+O(nlogn) => O(nlogn)

Space: O(1) => 額外空間需求固定

Unstable

Linear-time Sorting Method

1. 當排序技巧並非採用”Comparison-based” Skill時，則有可能來到O(n)之linear-time Sorting method
2. 主要方法:
   * 1. Radix Sort
     2. Bucket Sort
     3. Counting Sort

Radix Sort、Bucket Sort => DS版本視為相同

Radix Sort(DS版)(基數排序法也叫Bucket Sort)

1. 採用Distribution and Merge的排序技巧
2. 分為兩種:
3. LSD radix sort
4. MSD radix sort

LSD radix Sort

1. 假設r是採用的基底(Base or 進制)，則準備r個Buckets，編號:0~(r-1)
2. 假設d是最大值之位數個數，則表要做d回合完成Sort，或者，已知input Data之值域範圍或值域<=多少 => 即知d回合

=>回合數卡死了=>常數與data n數無關

1. 從最低位數開始，往最高位數，依序執行各回合工作

例: Pass1:個位數 Pass2:十位數 Pass3:百位數

1. 每一回合做2個工作
2. 分派:依Data之某位數值，將他分派到對應的Bucket中
3. 合併:由0到(r-1)合併Bucket內Data成一串列

分析:

Time Complexity: O(d\*(r+n))

因為執行d回合，每一回合主要有2個工作

1. 分派: O(n)

2. 合併: O(r)

每一回合花O(n+r)time

所以total = d\*O(n+r) = O(d\*(n+r))

說明:

1. r是base, 可視為常數:C1

2. 因為值域受到限制，所以d=回合數是固定值，與n無關，故也視為常數C2

所以O(d\*(n+r)) => O(n)

Space: 額外空間需求即Buckets size，有r個buckets，每個bucket size = n，所以O(r\*n)

Stable => FIFO

MSD Radix Sort

Steps:

1.依照最高位數值，將Data分派到對應的Bucket中

2.每個Bucket各自排序

3.合併Bucket結果，from 0~(r-1)號

特色:

與LSD不同在於:分派及合併都各做一次，不像LSD要做d次，MSD適用於最大鍵值位數之個數太大之case

Ds版本 Alg版本

LSD radix Sort = Radix Sort

MSD radix Sort = Bucket Sort

Bucket Sort[Alg版本]

Step:

1. 先將每個資料化成純小數

2. 以小數點後第一位數之值，分派Data到Bucket中

3. 每個Bucket之Data List均維持是ordered list

4. 最後合併r個buckets from 0~(r-1)號

Note: 此種作法也可視為Hashing(雜湊)，用來作為sorting之方法

Counting Sort(計數排序法)

假設n為Data個數，k是值域範圍限制

操作:

1. 統計每個值域之出現次數，記錄在Count[1..k]中

2. 求出每個值域之值未來擺放的起始位置，記錄在Start[1…k]中

3. 依Start之指示將input Data置入到Output Array中之正確起始位置，且Start值加1

分析:

Time => total time:O(k)+O(n)+O(k)+O(n)=>O(n+k)

說明:

因為值域受到限制，k視為常數c => O(n+k)=O(n+c)=>O(n)=>Linear-time Sorting method

Space => 額外空間需求是 = Count[1...k] + Start[1...k] + Output[1...n] = O(n+k)

Stable

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Summary | Time | | | | Space | Stable/Unstable |
| Best | Worst | | Avg |
| Insertion Sort | O(n) | | O | O | O(1) | Stable |
| Selection Sort | O | | O | O | O(1) | Unstable |
| Bubble Sort | O(n) | | O | O | O(1) | Stable |
| Shell Sort | O | | O | O | O(1) | Unstable |
| Quick Sort | O(nlogn) | | O | O(nlogn) | O(logn)~O(n) | Unstable |
| Merge Sort | O(nlogn) | | O(logn) | O(nlogn) | O(n) | Stable |
| Heap Sort | O(nlogn) | | O(nlogn) | O(nlogn) | O(1) | Unstable |
|  | Time | | | | Space | Stable/Unstable |
| Radix Sort | O(d\*(n+r)) | | | | O(n\*r) | Stable |
| Counting Sort | O(n+k) | | | | O(n+k) | Stable |

[補充]

Algo版本 => Sorting in-place

在原本input Data space藉由Swap即可完成排序

例: Not Sorting in-place => Merge Sort、Radix Sort、Counting Sort、Bucket Sort

[補充]

Algo版本 => Select ith smallest data among n unsorted Data list

BST如何找出ith smallest Data，而現在這個是based on array

[法一]

Steps:

1. 先sort => 採用quick sort

2. return A[i] (Assume Data皆不同) Time: O(nlogn) in best/avg case

[法二]

利用Quick Sort中的Partition技巧來完成

(可將pk製於正確位置上)

分析:

Time => Best case: 當Partition恰好將Data切成兩等分=>O(n)

=> Avg case: O(n)

=> Worst case: 當PK恰巧是min or Max時候 => O(n平方)

[法三]

在worst case下，仍為O(n)之solution

原則: 避免PK是min or Max值，使用median-of-medians(中間鍵s之中間鍵)作為PK

Algo:

Select(A,p,r,i)

// A[p]~A[r]有n個Data

{

1.先將n個Data分成[n/5]取上限個groups，每個group為5個Data，可能有一個group不足5個Data

=>Time:O(n)

2.每個Group之5個Data各自sort(小->大)

=>Time:O(n)

3.每個Group之第三個Data為該Group之中間鍵，在這些中間鍵中，運用Select遞迴，找出他們

的中間鍵，即為median-of-medians，此為PK

=>Time:O([n/5])

4. q = Partition(A,p,r);

=>Time:O(n)

5. k = q - p + 1; // pk是第kth小

if (i == k)

{

return A[q];

}

else if (i < k)

{

return Select(A,p,q-1,i);

}

else

{

return Select(A,q+1,r,i-k);

}

}

Worst case => O(n)

Note: group元素要>=5個，才會O(n)，若 < 5個=>O(nlogn)