Temasek Junior College

JC H2 Computing

Problem Solving & Algorithm Design 14B – Quicksort

1 Faster Sorting - divide and conquer (D&C)

A common computer programming tactic is to divide a problem into sub-problems of the **same type** as the original, solve those problems, and combine the results. This is often referred to as the divide-and-conquer method; when combined with a lookup table that stores the results of solving sub-problems (to avoid solving them repeatedly and incurring extra computation time), it can be referred to as **dynamic programming** or **memorization**.

In computer science, divide and conquer (D&C) is an important algorithm design paradigm based on multi-branched recursion.

A divide and conquer algorithm works by **recursively** breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

Basic idea: divide and conquer
» Divide into 2 (or more) sub-problems.
» Solve each sub-problem

divide–and-conquer **strategy**.

That is, each algorithm finds a way of breaking the list into smaller sublists. These sublists are then **sorted recursively**. Ideally, if the number of these subdivisions is $\log_2(N)$ and the amount of work needed to rearrange the data on each subdivision is N, then the total complexity of such a sort algorithm is $O(N \log_2 N)$. In Table 1, you can see that the growth rate of work of an $O(N \log_2 N)$ algorithm is much slower than that of an $O(N^2)$ algorithm.

Table 1 Comparing $N \log N$ and N^2

N	$N \log_2 N$	N^2
512	4,608	262,144
1,024	10,240	1,048,576
2,048	22,458	4,194,304
8,192	106,496	67,108,864
16,384	229,376	268,435,456
32,768	491,520	1,073,741,824

2 Quick Sort (Partition-Exchange Sort)

Considering a sorting method which performs **very well on larger tables**. At each step in the method, the goal is to place a particular record [**pivot**] in its **final position** within the table. In so doing, all records that precede this record have smaller keys, while all records that follow it have larger keys. This technique essentially partitions the table into two subtables. The same process can then be applied to each of these subtables and repeated until all records are placed in their final positions.

Each time a table is partitioned into two smaller subtables, these can be processed in turn using either an iterative or recursive approach. A **general algorithm** based on a recursive approach follows:

1. Partition the current table into two subtables
2. Invoke quicksort to sort the left subtable
3. Invoke quicksort to sort the right subtable

The **method** can be described by the following recursive algorithm:

1. If the array Arr has no elements or only one element, then it is sorted. Otherwise, perform the following steps:
2. Choose an arbitrary element in the array and call it **pivot**.
3. Move the elements around in the array so that two groups are formed. The element pivot should be placed between the two groups. All the elements that are less than or equal to pivot should be placed in the group to the left of pivot and all the rest in the group to the right.
4. Sort the part of the array to the left of pivot using this algorithm.
5. Sort the part of the array to the right of pivot using this algorithm.

Example 1 Consider the following key set:

first									last
1	2	3	4	5	6	7	8	9	10
42	23	74	11	65	58	94	36	99	87

Using two index variables, `first` and `last`, with initial values of `1` and `10`, respectively.

Choose of index of pivot = `(first+ last) / 2` //integer division

```
pivotIndex = (1+10) / 2
           = 5
```

The sequence of exchanges for placing 65 [**pivot**] in its final position is given as follows, where the encircled entries on each line denote the keys being compared:

1	2	3	4	5	6	7	8	9	10	low	high	
										1	**10**	first = 1, last = 10
42	23	74	11	65•	58	94	36	99	87			**pivotIndex = 5, pivot = 65•**
42►	23	74	11	65•	58	94	36	99	87			Scan left, 42►, 42<65 True
42	23►	74	11	65•	58	94	36	99	87	**2**		Scan left, 23►, 23<65 True
42	23	74►	11	65•	58	94	36	99	87	**3**		Scan left, 74►, 74<65 False
42	23	74	11	65•	58	94	36	99	◄87			Scan right, ◄87, 87>65 True
42	23	74	11	65•	58	94	36	◄99	87		**9**	Scan right, ◄99, 99>65 True
42	23	74	11	65•	58	94	◄36	99	87		**8**	Scan right, ◄36, 36>65 False
42	23	**74►**	11	65•	58	94	**◄36**	99	87	**[3]**	**[8]**	interchange 74► and ◄36
42	23	36Δ	11	65•	58	94	74Δ	99	87			36Δ ↔74Δ
										4	**7**	Low <= high True
42	23	36	11►	65•	58	94	74	99	87			Scan left, 11►, 11<65 True
42	23	36	11	65•►	58	94	74	99	87	**5**		Scan left, 65►, 65<65 False
42	23	36	11	65•	58	◄94	74	99	87			Scan right, ◄94, 94>65 True
42	23	36	11	65•	58◄	94	74	99	87		**6**	Scan right, ◄58, 58>65 False
42	23	36	11	65•►	58◄	94	74	99	87	**[5]**	**[6]**	interchange 65► and ◄58
42	23	36	11	58Δ	65•Δ	94	74	99	87			65Δ ↔58Δ
										6	**5**	Low <= high **False**
{42	23	36	11	58}	65♥	{94	74	99	87}			65♥ final position: 6th
{42	23	36	11	58}	65♥	{94	74	99	87}	**1**	**5**	first = 1, last = 5
42•	23	36•	11	58	65♥							**pivotIndex = 3, pivot = 36•**
42	23	36•	11	58•	65♥							
42	23	36•	11•	58	65♥							
11Δ	23	36•	42Δ	58	65♥							interchange 42 and 11
11	23•	36•	42	58	65♥							
{11	23}	36♥	{42	58}	65♥							36♥ final position: 3rd
										1	**2**	first = 1, last = 2
11•	23	36♥	{42	58}	65♥							**pivotIndex = 1, pivot = 11•**
11♥	{23}	36♥	{42	58}	65♥							11♥ final position: 1st
										2	**2**	first = 2, last = 2
11♥	23•	36♥	{42	58}	65♥							**pivotIndex = 2, pivot = 23•**
11♥	23♥	36♥	{42	58}	65♥							23♥ final position: 2nd
										4	**5**	first = 4, last = 5
			42•	58								**pivotIndex = 4, pivot = 42•**

The original key set has been partitioned into the subtables, namely, the sets {42, 23, 36, 11, 58} and {94, 74, 99, 87}. The same process can be applied to each of these sets until the table is completely sorted.

2.1 A detailed algorithm based on this recursive formulation can now be given.

```
quickSort(Arr, first, last)
//variable used:  temp, low, high, pivot
      low ← first
      high ← last
      // pivotIndex: index of pivot.
      //Value can be any of
      //first, last, median, random between first and last
      pivotIndex ← (first+last)/2     // integer division
      pivot ← Arr[pivotIndex]
      DOWHILE(low <= high)            //Using REPEAT
            //Scan left
            DOWHILE  Arr[low] < pivot
                  low ← low + 1
            ENDDO
            //Scan right
            DOWHILE  Arr[high] > pivot
                  high ← high -1
            ENDDO
            //Swapping
            IF low <= high THEN
                  temp ← Arr[low]
                  Arr[low] ← Arr[high]
                  Arr[high] ← temp
                  //Shift right by 1 element
                  low ← low + 1
                  //Shift left by 1 element
                  high ← high -1
            EDNIF
      ENDDO                           //Using UNTIL low > high
      IF first < high THEN
            quickSort(Arr, first, high)
      ENDIF
      IF low < last THEN
            quickSort(Arr, low, last)
      ENDIF
END
```

2.2 Picking an element as pivot

Some examples of selecting an element as pivot:

- first element as pivot // `pivotIndex ← first`
- last element as pivot // `pivotIndex ← last`
- random-value element as pivot. // `pivotIndex ←` random.randrange(first, last)
- middle element as pivot // `pivotIndex ← (first+last) // 2` [integer division]

2.3 Analysis of Quick Sort

In average, to sort a list of size N, the quick sort algorithm has a running time of order $N \log_2 N$, i.e. the average number of comparisons is approximately a constant factor of $N \log_2 N$. This is much better than insertion sort which takes an average of order N^2. For example, when N = 512, N^2 is 262,144, whereas $N \log_2 N$ is 512 * 9 = 4,608. As N gets larger, $N \log_2 N$ is much smaller than N^2.

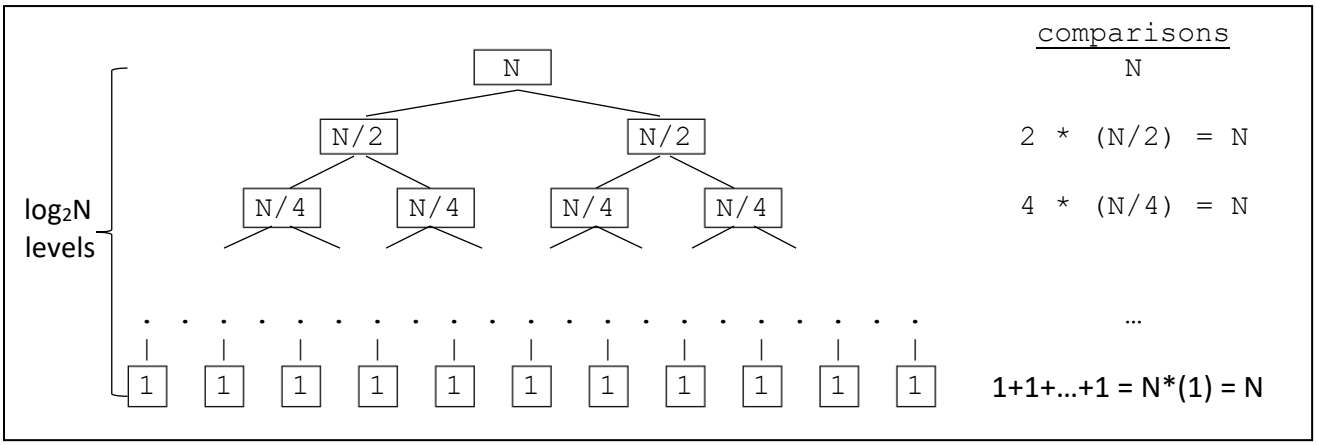
Quick Sort has its drawback.
It can take a running time of order N^2 when the list is completely or almost ordered. This is because each time scanning the list to left or right, it takes $N - 1$ comparisons, whereas in average it should take about $\log_2 N$ comparisons.

2.3.1 Best Case

Quick sort's best case occurs when the partitions are as **evenly balanced** (ie. left and right sub-array sizes are equal or different by 1 element). For instance after partitioning:

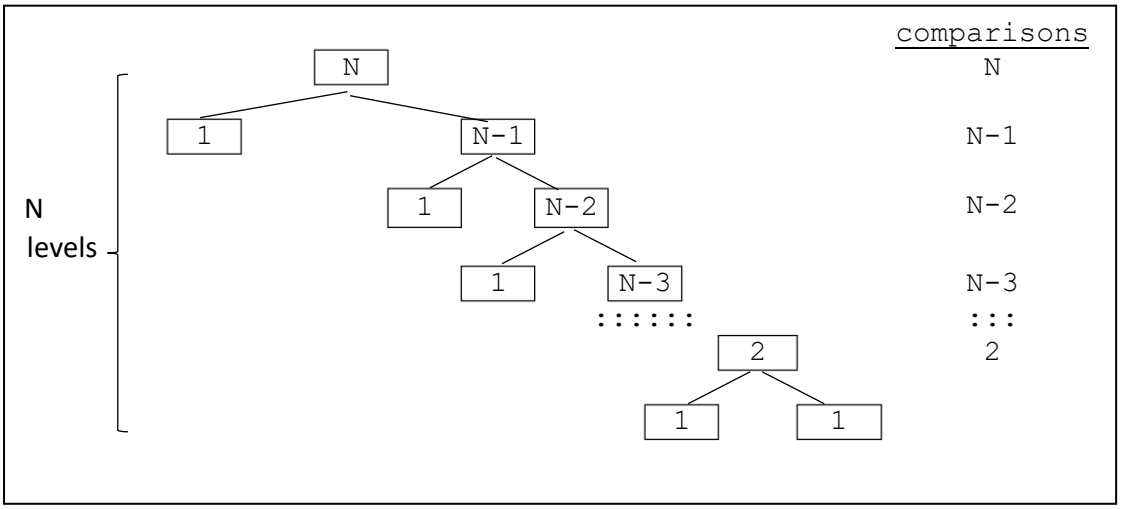
- an odd number of elements (i.e. *2k + 1*, where *k = 1, 2, 3,…*), the **pivot is in the middle** with a total of ½(2k+1) elements in each partition.
- an even number of elements (i.e. *2k*, where *k = 2, 3, 4,…*), the pivot is in the middle with one partition of elements and another partition of *k*-1 elements.

To partition array from one level to another, N comparisons have to be made, as each element is compared to the pivot.



2.3.2 Worst Case

- Pivot when placed at its correct position is always the smallest or largest element.
- one subarray has 1 element, the other has `N-1` elements



No. of comparisons: `2 + 3 + 4 +…+ (N-1) + N ≈ N * (N + 1) / 2`

Worst Case Time Complexity: $O(N^2)$

Quick Sort Time Complexity			
Best case	Average case	Worse case	Concept
$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N^2)$	*Array, Recursion, Divide and Conquer*

2.3.3 What are some techniques to choose a pivot?

Choose the left most or rightmost element.
Pros: Simple to code, fast to calculate
Cons: If the data is sorted or nearly sorted, quick sort will degrade to $O(N^2)$

Choose the middle element:
Pros: Simple to code, fast to calculate, but slightly slower than the above methods
Cons: Still can degrade to $O(N^2)$. Easy for someone to construct an array that will cause it to degrade to $O(N^2)$

Choose the median of three:
Pros: Fairly simple to code, reasonably fast to calculate, but slightly slower than the above methods
Cons: Still can degrade to $O(N^2)$. Fairly easy for someone to construct an array that will cause it to degrade to $O(N^2)$.

Choose the pivot randomly (using built in random function):
Pros: Simple to code. Harder for someone to construct an array that will cause it to degrade to $O(N^2)$
Cons: Selecting a random pivot is fairly slow. Still can degrade to $O(N^2)$

Choose the pivot randomly (using a custom built random function):
Pros: Much harder for someone to construct an array that will cause it to degrade to O(N2), if they don't know how you are choosing the random numbers.
Cons: May be complicated to code. Selecting a random pivot is fairly slow. Still theoretically possible that it can degrade to $O(N^2)$.

Use the median of medians method to select a pivot
Pros: The pivot is guaranteed to be good. Quick sort is now $O(N \log_2 N)$ worst case !
Cons: Complicated code. Typically, a lot slower than the above methods.

Which method should I use?

-If it is unlikely that the data will be sorted, and you are willing to accept $O(N^2)$ in the rare cases when the array is sorted then use the leftmost or rightmost element.

-If there is a reasonable chance your data is sorted use the middle element or median of threes.

-If you are somewhat worried about malicious users giving you bad arrays to sort (used as a Denial of Service attack) then use random pivots.

-If you are really worried about malicious users or you need to guarantee that the quicksort runs is $O(N \log_2 N)$ then use the median of medians. At this point you may want to seriously consider using a different sorting method like merge sort.

2.3.4 QuickSort with partitions

```
PROCEDURE QuickSort (ARRAY, low, high)
    IF low < high THEN
        p ← partition(ARRAY, low, high)
    ENDIF
        quickSort(ARRAY, low, p - 1)
        quickSort(ARRAY, p + 1, high)
ENDPROCEDURE

FUNCTION partition(ARR, low, high)
    pivotIndex = high              //Use last element
    pivot ← ARR[pivotIndex]
    I ← low
    FOR J ← low to high
        IF ARR[J] < pivot THEN
            swap ARR[I] with ARR[J]
            I ← I + 1
        ENDIF
    ENDFOR
    swap ARR[I] with ARR[high]
    RETURN I
ENDFUNCTION
```

2.3.5 QuickSort with additional memory in each recursion

```
import random
def quicksort(array):
    if len(array) < 2:
        print(array)
     ##Base case: arrays with 0 or 1 element are already sorted.
        return array
    else:
        pivotpos = (len(array)-1)//2   #or 0 or len(array)-1 or random
        pivot = array[pivotpos]   ##Recursive case
           ##Sub-array of all the elements less than the pivot
        less = [i for i in array[0:len(array)] if i < pivot]
           ##Sub-array of all the elements greater than the pivot
        greater = [i for i in array[0:len(array)] if i > pivot]
    return quicksort(less) + [pivot] + quicksort(greater)
```