파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

퀵 정렬의 전체적인 코드를 이해하고

시간복잡도를 계산할 수 있다

Data Structures in Python
Chapter 5 - 2

- Merge sort
- Quick sort Algorithm
- **Quick sort Analysis**
- Empirical Analysis

Agenda & Readings

- Agenda
  - Quicksort Analysis
  - Time Complexity

- Reference:
  - Problem Solving with Algorithms and Data Structures
    - Chapter 5 Search, Sorting and Hashing: Quicksort
    - Wikipedia Quick sort
    - [알고리즘] 퀵정렬

Partition Code

```
# This function takes the last element as pivot, places the pivot element at its
# correct position in sorted array, and places all smaller (less than pivot) to
# left of pivot and all greater elements to right of pivot.

def partition(a, lo, hi):
    pivot = a[hi]                       # pivot
    i = lo - 1;                         # last index of smaller element on the left
    j = lo
    while j <= hi - 1:                  # traverse the array
        if a[j] < pivot:                # look for element less than pivot
            i += 1                      # increment only when a[j] is less than pivot
            if i != j:                  # swap: smaller to left and greater to right
                a[j], a[i] = a[i], a[j]
        j += 1

    a[hi], a[i+1] = a[i+1], a[hi]   # move the pivot at the position sorted
    return i + 1                        # return index where pivot moved and sorted
```

Partition Code

```
# qsort helper function for recursive operation
def qsort(a, lo, hi):
    if lo >= hi: return              # done, we have an empty array

    pi = partition(a, lo, hi)        # partition, get index of the pivot sorted
    qsort(a, lo, pi - 1)             # soft left of the pivot
    qsort(a, pi + 1, hi)             # soft right of the pivot

def quicksort(a):
    qsort(a, 0, len(a) - 1)

if __name__ == "__main__":
    a = [32, 23, 81, 43, 92, 39, 57, 16, 75, 65]
    print('                   input:', a)
    quicksort(a)
    print('                  output:', a)
```

while **j** traverses from **low** to **hi-1**
i increments only when **a[j] < pivot**

```
def partition(a, lo, hi):
    pivot = a[hi]
    i = lo - 1;
    j = lo
    while j <= hi - 1:                       scan
        if a[j] < pivot:
            i += 1
            if i != j:                       swap
                a[j], a[i] = a[i], a[j]
        j += 1
    a[hi], a[i+1] = a[i+1], a[hi]           sorted
    return i + 1
```

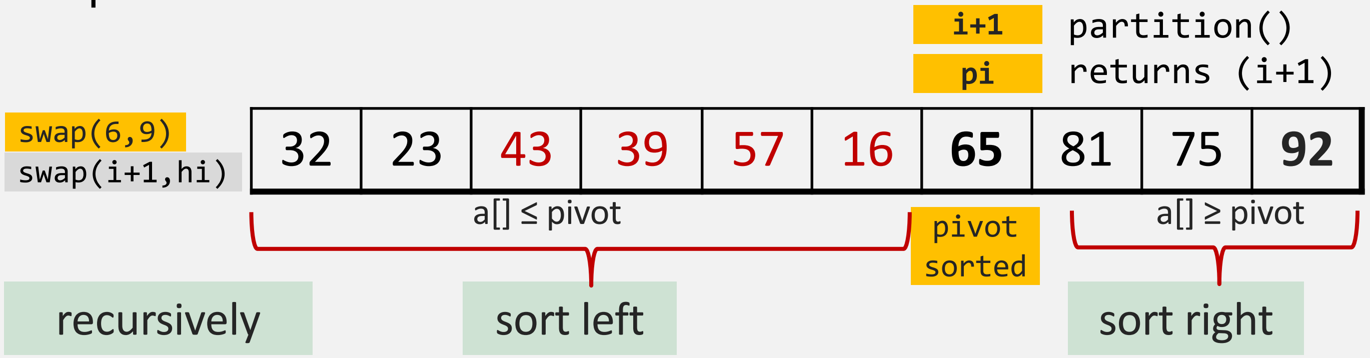
Partition Code

```
# qsort helper function for recursive operation
def qsort(a, lo, hi):
    if lo >= hi: return

    pi = partition(a, lo, hi)
    qsort(a, lo, pi - 1)
    qsort(a, pi + 1, hi)

def quicksort(a):
    qsort(a, 0, len(a) - 1)

if __name__ == "__main__":
    a = [32, 23, 81, 43, 92, 39, 57, 16, 75, 65]
    print('           input:', a)
    quicksort(a)
    print('          output:', a)
```



while j traverses from low to hi-1
i increments only when a[j] < pivot

```
def partition(a, lo, hi):
    pivot = a[hi]
    i = lo - 1;
    j = lo
    while j <= hi - 1:                      scan
        if a[j] < pivot:
            i += 1
            if i != j:                      swap
                a[j], a[i] = a[i], a[j]
        j += 1
    a[hi], a[i+1] = a[i+1], a[hi]          sorted
    return i + 1
```

Quick sort Code

```
           input: [32, 23, 81, 43, 92, 39, 57, 16, 75, 65]
   partition: 0 ~ 9
swap (2 3):(43 81) [32, 23, 43, 81, 92, 39, 57, 16, 75, 65]
swap (3 5):(39 81) [32, 23, 43, 39, 92, 81, 57, 16, 75, 65]
swap (4 6):(57 92) [32, 23, 43, 39, 57, 81, 92, 16, 75, 65]
swap (5 7):(16 81) [32, 23, 43, 39, 57, 16, 92, 81, 75, 65]
   move pivot: 9->6
   partitioned(65) [32, 23, 43, 39, 57, 16, 65, 81, 75, 92] returns 6
   sort(le): 0 ~ 5
   partition: 0 ~ 5
   move pivot: 5->0
   partitioned(16) [16, 23, 43, 39, 57, 32] returns 0
    sort(le): 0 ~ -1
    sort(ri): 1 ~ 5
   partition: 1 ~ 5
   move pivot: 5->2
   partitioned(32) [23, 32, 39, 57, 43] returns 2
    sort(le): 1 ~ 1
    sort(ri): 3 ~ 5
   partition: 3 ~ 5
   move pivot: 5->4
   ...
```

Quick sort Code

```
 partitioned(43) [39, 43, 57] returns 4
 sort(le): 3 ~ 3
 sort(ri): 5 ~ 5
 sort(ri): 7 ~ 9
partition: 7 ~ 9
move pivot: 9->9
partitioned(92) [81, 75, 92] returns 9
 sort(le): 7 ~ 8
partition: 7 ~ 8
move pivot: 8->7
partitioned(75) [75, 81] returns 7
 sort(le): 7 ~ 6
 sort(ri): 8 ~ 8
 sort(ri): 10 ~ 9
       output: [16, 23, 32, 39, 43, 57, 65, 75, 81, 92]
```

Choosing a better pivot

- Choosing the first or last element as the pivot leads to very poor performance on certain inputs (ascending, descending) does **not** the array into roughly-equal size chunks.
- Alternative methods of picking a pivot:
  - random: Pick a random index from `[min .. max]`
  - median-of-3: look at left/middle/right elements and pick the one with the medium value of the three:
    - `a[min], a[(max+min)/2], and a[max]`
    - better performance than picking random numbers every time
    - provides near-optimal runtime for almost all input orderings

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
value	8	18	91	-4	27	30	86	50	65	78	5	56	2	25	42	98	**31**

Stable sorting

- Stable sort: Sorting that maintains relative order of "equal" elements.
  - It is important for secondary sorting, e.g.
    - E.g., in Excel, we may sort by name, then sort again by age, then by salary, ..., then we want to maintain relative orders of others.
- Many sorting algorithms are **stable** such as bubble sort, insertion sort, shell sort and merge sort.
- Quick sort is **not** stable.
  - The partitioning algorithm can reverse the order of "equal" elements.
  - For this reason, Java's Arrays/Collections.sort() use merge sort.

Unstable sort example

- Suppose you want to sort these points (x, y) **by y first, then by x**:
  - `[(4, 2), (5, 7), (3, 7), (3, 1)]`

- A stable sort like merge sort would do it this way:

  - `[(3, 1), (4, 2), (5, 7), (3, 7)]` sort by y
  - `[(3, 1), (3, 7), (4, 2), (5, 7)]` sort by x

  - Note that the relative order of (3, 1) and (3, 7) is maintained.

- Quick sort might leave them in the following state:

  - `[(3, 1), (4, 2), (5, 7), (3, 7)]` sort by y
  - `[(3, 7), (3, 1), (4, 2), (5, 7)]` sort by x

  - Note that the relative order of (3, 1) and (3, 7) has reversed.

Time Complexity - the worst case

- The most unbalanced (worst) partition occurs when one of the sublists returned by the partitioning routine is of size n − 1.
  - This may occur if the pivot happens to be the smallest or largest element in the list.
  - If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. Consequently, we can make n − 1 nested calls before we reach a list of size 1.
  - A single quicksort call involves O(n) work plus two recursive calls on lists of size 0 and n−1, so the recurrence relation is

$$
\begin{aligned}
T(n) &= n + T(0) + T(n-1) && (1)\\
&= n + T(n-1) && (2)\\
&= n + (n-1) + T(n-2) && (3)\\
&\dots && (4)\\
&= n + (n-1) + (n-2) + \dots + T(0) && (5)
\end{aligned}
$$

- Therefore, it solves to the worst case $T(n) = O(n^2)$.

Time Complexity - the most balanced case

- In this case, a single quicksort call involves O(n) work plus two recursive calls on lists of size n/2, so the recurrence relation is. The timing for a list of size 1 is constant, i.e., $T(1) = 1$.

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + n && (1)\\
&= 2(2T(\frac{n}{2^2})) + n + n && (2)\\
&= 2(2(2T(\frac{n}{2^3}))) + n + n + n && (3)\\
&= 2^4 T(\frac{n}{2^4}) + 4n && (4)\\
&= \ldots && (5)\\
&= 2^k T(\frac{n}{2^k}) + kn && (6)
\end{aligned}
$$

- Since the base case, $T(1) = T(\frac{n}{2^k})$, occurs when $n = 2^k$. That is, $k = \log n$.

$$T(n) = n \cdot T(\frac{n}{n}) + n \cdot \log_2 n = n + n \cdot \log_2 n \qquad (1)$$

- Therefore, it solves to the balanced (or average) case is $O(n \log_2 n)$.

Quick sort - Exercise

- Suppose we have an array of size 7 that consists of values from 11 to 17. For example, `a = [12, 11, 13, 17, 15, 14, 16].`
- Find two sequences that may show the worst case and the best case of the quick sort.
  - Count the number of partition function calls.
  - For each partition, show the result, number of comparisons and swaps.

Summary

Algorithm	Best	Worst	Average	Extra Memory	
Bubble	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	slow
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Good if often almost sorted
Shell	$O(n)$	$O(n\ (\log n)^2)$	$O(n\ (\log n)^2)$	$O(1)$	
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Good for very large datasets
Quick	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(n)$	**Faster than merge sort in general**
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	Best if $O(n \log n)$ required
Tim	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	used in Python, hybrid of merge sort and nsertion sort

- Note: A comparison-based sorting algorithm cannot be better than $O(n \log n)$ in the average and worst case.

학습 정리

1) 퀵 정렬의 평균적 시간복잡도는 O($n \log n$)이지만,
최대 O(n^2)까지 증가할 수 있다

2) 퀵 정렬은 일반적으로 병합 정렬보다 빠르지만 불완전 정렬이다


파이썬으로 배우는
데이터 구조
수고했습니다
곧 다음 시간에
다시 뵙겠습니
다