Algorithms

Ch 7: Quicksort

Ming-Te Chi

Sorting algorithm

- Quick sort: (on an input array of n numbers)
  - Based on the divide-and-conquer mechanism (like merge sort)
  - Worst-case time complexity $O(n^2)$
  - Average time complexity $O(n \log n)$
  - Constants hidden in $O(n \log n)$ are small
  - Sorts in place

7.1 Description of Quicksort

- To sort the subarray A[p..r]
  - Divide: PARTITION A[p..r] into A[p..q-1] & A[q+1..r]
    - $a \in A[p..q-1] \Rightarrow a \le A[q]$
    - $b \in A[q+1..r] \Rightarrow A[q] \le b$
  - Conquer: sort the two subarrays by recursive calls to QUICKSORT
  - Combine: no work is needed, because they are sorted in place.

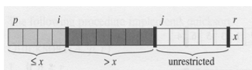

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

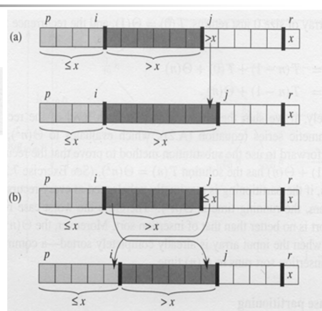

Partition(A, p, r)

Partition subarray A[p..r] by the following procedure:

```
x = A[r]
i = p - 1
for j = p to r -1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i +1] with A[r]
return i +1
```


Two cases for one iteration of procedure *Partition*



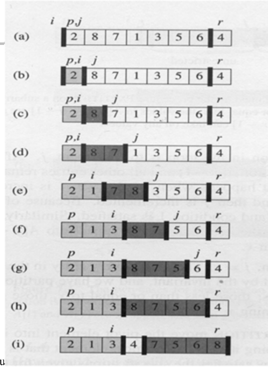


Complexity:
Partition on A[p…r] is $\Theta(n)$
where $n = r - p + 1$

The operation of *Partition* on a sample array

Partition subarray A[p..r] by the following procedure:

```
1  x = A[r]
2  i = p - 1
3  for j = p to r -1
4     if A[j] ≤ x
5         i = i + 1
6         exchange A[i] with A[j]
7  exchange A[i +1] with A[r]
8  return  i +1
```



Partition(A, p, r)

- PARTITION always selects the last element *A*[*r*] in the subarray *A*[*p . . r*] as the pivot — the element around which to partition.
- As the procedure executes, the array is partitioned into four regions, some of which may be empty:
  - All entries in *A*[*p . . i*] are ≤ pivot.
  - All entries in *A*[*i* + 1 . . *j* -1] are > pivot.
  - *A*[*r*] = pivot.
  - It's not needed as part of the loop invariant, but the fourth region is *A*[*j . . r* -1], whose entries have not yet been examined, and so we don't know how they compare to the pivot.

Loop invariant

At the beginning of each iteration of the loop of lines 3-6, for any array index k,

1. if p ≤ k ≤ i, then A[k] ≤ x.
2. if i + 1 ≤ k≤ j -1, then A[k] > x.
3. if k = r, then A[k] = x.

Correctness: Use the loop invariant to prove correctness of PARTITION

We have to show that

- the loop invariant is true prior to the first iteration,
- each iteration of the loop maintains the invariant, and
- the invariant provides useful property to show the correctness when the loop terminates.

Correctness: Use the loop invariant to prove correctness of PARTITION —— continue

Idea of loop invariant: similar to the mathematical induction(歸納法), so we have to "prove"

- The initial case
- The induction step

  If the statement is true at the n-1th step, it will hold for the nth step

As indicated in Cormen's book:

- Initialization
- Maintenance
- Termination

Correctness: Use the loop invariant to prove correctness of PARTITION —— continue

- Initialization:

  Before the loop starts, all the conditions of the loop invariant are satisfied, because *r* is the pivot and the subarrays *A*[*p .. i*] and *A*[*i* +1 .. *j* -1] are empty. (i=p-1, j=p)
- Maintenance

  While the loop is running, if *A*[*j*] ≤ pivot, then *A*[*j*] and *A*[*i* +1] are swapped and then *i* and *j* are incremented. If *A*[*j*] > pivot, then increment only *j*.
- Termination

  When the loop terminates, *j* = *r*, so all elements in *A* are partitioned into one of the three cases: *A*[*p..i*] ≤ pivot, *A*[*i* +1 .. *r* -1] > pivot, and *A*[r] = pivot.

Correctness: Use the loop invariant to prove correctness of PARTITION —— continue

- The last two lines of PARTITION move the pivot element from the end of the array to between the two subarrays.
- This is done by swapping the pivot and the first element of the second subarray, i.e., by swapping $A[i+1]$ and $A[r]$.

7.2 Performance of quicksort

- The running time of quicksort depends on the partitioning of the subarrays:
  - If the subarrays are balanced, then quicksort can run as fast as mergesort.
  - If they are unbalanced, then quicksort can run as slowly as insertion sort.

Worst case

- Occurs when the subarrays are completely unbalanced.
- Have 0 elements in one subarray and n-1 elements in the other subarray.

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= \sum_{k=1}^{n} \Theta(k) = \Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$$

- Occurs when quicksort takes a sorted array as input
  - but insertion sort runs in $O(n)$ time in this case.

Best case

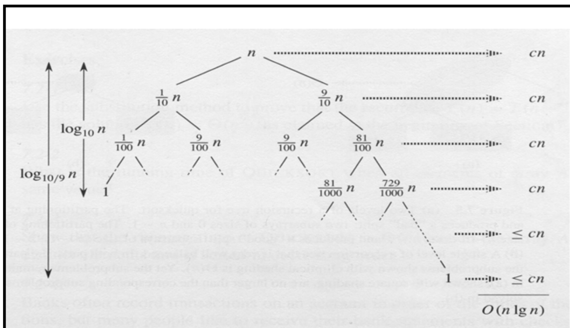
- Occurs when the subarrays are completely balanced every time.
- Each subarray has ≤ n/2 elements.

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \log n)$$

Balanced partitioning

- Quicksort's average running time is much closer to the best case than to the worst case.
  - Imagine that PARTITION always produces a 9-to-1 split.

$$T(n) \le T(9n/10) + T(n/10) + \Theta(n)$$
$$= \Theta(n \log n)$$



Balanced partition $T(n) = \Theta(n \log n)$

$$T(n) = T(9n/10) + T(n/10) + \Theta(n)$$
$$\Rightarrow T(n) = \Theta(n \log n)$$

Balanced partitioning —— continue

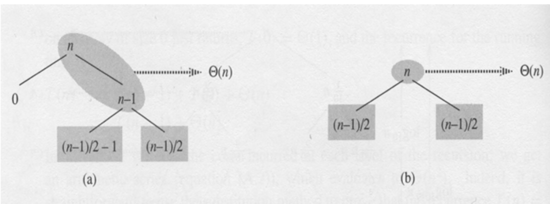
Look at the recursion tree:

- It's like the one for T(n) = T(n/3)+T(2n/3)+O(n) in Section 4.2.
- Except that here the constants are different; we get $\log_{10}$ n full levels and $\log_{10/9}$ n levels that are nonempty.
- As long as it's a constant, the base of the log doesn't matter in asymptotic notation.
- Any split of constant proportionality will yield a recursion tree of depth Θ(log n).

Intuition for the average case

- Splits in the recursion tree will not always be constant.
- There will usually be a mix of good and bad splits throughout the recursion tree.
- To see that this doesn't affect the asymptotic running time of quicksort, assume that levels alternate between best-case and worst-case splits.

Intuition for the average case $T(n) = \Theta(n \lg n)$



Intuition for the average case ——continue

- The extra level in the left-hand figure only adds to the constant hidden in the Θ-notation.
- There are still the same number of subarrays to sort, and only twice as much work was done to get to that point.
- Both figures result in $O(n \log n)$ time, though the constant for the figure on the left is higher than that of the figure on the right.

7.3 Randomized versions of partition

- We could randomly permute the input array.
- Instead, we use random sampling, or picking one element at random.
- Don't always use $A[r]$ as the pivot. Instead, randomly pick an element from the subarray that is being sorted.
- Randomly selecting the pivot element will, on average, cause the split of the input array to be reasonably well balanced.

Randomized partition

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

Randomized quicksort

```
RANDOMIZED_QUICKSORT(A,p,r)
if p < r
    q= RANDOMIZED_PARTITION(A,p,r)
    RANDOMIZED_QUICKSORT(A,p,q-1)
    RANDOMIZED_QUICKSORT(A,q+1,r)
```


- Randomization of quicksort stops any specific type of array from causing worstcase behavior.
  - For example, an already-sorted array causes worst-case behavior in non-randomized QUICKSORT, but not in RANDOMIZED-QUICKSORT.

7.4 Analysis of quicksort

- We will analyze
  - the worst-case running time of QUICKSORT and RANDOMIZED-QUICKSORT (the same), and
  - the expected (average-case) running time of RANDOMIZED-QUICKSORT.

7.4.1 Worst-case Analysis

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

guess $T(n) \le cn^2$

$$T(n) \le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n)$$

$$= c \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

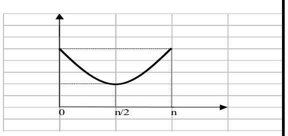
$$\le cn^2 - c(2n-1) + \Theta(n)$$

$$\le cn^2$$

pick the constant c large enough so that the $c(2n-1)$ term dominates the $\Theta(n)$ term.

$\Rightarrow T(n) = \Theta(n^2)$

Show that $q^2 + (n-q-1)^2$ **achieves a maximum over** $q = 0,1,2,.....,n-1$ **when** $q = 0$ **or** $q = n-1$



ans: 先令 $f(q) = q^2 + (n-q)^2$

一次微分: $f'(q) = 2q - 2(n-q) = 4q - 2n$

令 $f'(q) = 0 \Rightarrow 4q - 2n = 0 \Rightarrow q = \frac{n}{2}$ (極小值)

二次微分: $f''(q) = 4$ (開口向上)

因為 $0 \le q \le n-1$ 所以 $f(0) = f(n-1) = (n-1)^2$ (相對極大值)

7.4.2 Expected (average) running time

- The dominant cost of the algorithm is partitioning.
- PARTITION removes the pivot element from future consideration each time.
  - ➔ PARTITION is called at most *n* times.
- QUICKSORT recurses on the partitions.
- The amount of work that each call to PARTITION does is a constant plus the number of comparisons that are performed in its for loop.
- Let X = the total number of comparisons performed in all calls to PARTITION.

➔ the total work done over the entire execution is O(n + X).

7.4.2 Expected running time

- Lemma 7.1
  - Let X be the number of comparisons performed in line 4 of *partition* over the entire execution of *Quicksort* on an *n*-element array. Then the running time of *Quicksort* is $O(n+X)$

Goal: compute X

- Not to compute the number of comparison in each call to PARTITION.
- Derive an overall bound on the total number of comparision.
- For easy of analysis:
  - Rename the elements of *A* as $z_1, z_2, \ldots, z_n$, with z_i being the i^{th} smallest element.
  - Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$ to be the set of elements between z_i and z_j, inclusive.

Goal: compute X —— continue

- Each pair of elements is compared at most once, why?
  - because elements are compared only to the pivot element, and then the pivot element is never in any later call to PARTITION.

we define

$$X_{ij} = I\ \{z_i \text{ is compared to } z_j\},$$

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr\{z_i \text{ is compared to } z_j\}$$

$$\begin{aligned}\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is first pivot chosen from } Z_{ij}\}\\ &= \Pr\{z_i \text{ is first pivot chosen from } Z_{ij}\}\\ &\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\}\\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1}\\ &= \frac{2}{j-i+1}\end{aligned}$$

$$\therefore E[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1}.$$

Goal: compute X —— continue

$$E[X] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1}\sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1}\sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log n)$$

$$= O(n \log n)$$

- (Ref: Eq. A.7 Harmonic series)
- Expected running time of quicksort is $O(n \log n)$