CISC 320 Introduction to Algorithms Fall 2005

Lecture 4

QuickSort & MergeSort

- Terminologies
  - Comparison-based sorting
  - In-place
  - Stable sorting

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

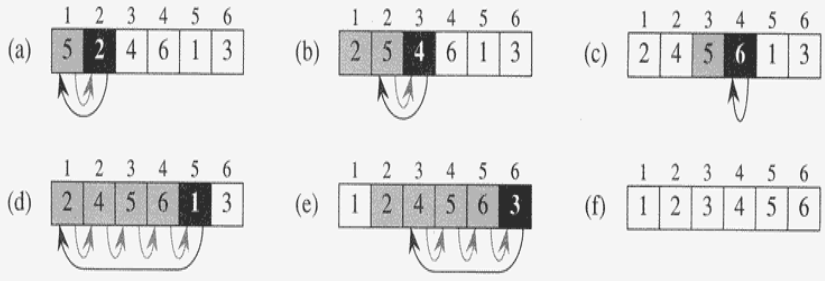


Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

Insertion Sort: complexity analysis

- Worst-case

$$T_{wc}(n) \le \sum_{i=1 \text{ to } n-1} i = n(n-1)/2$$

for loop to see whether A[i] needs to move

while loop to move A[i] at most i times to the front of the array


```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

Why we save? Because elements in range [p, q-1] and range [q+1, r] will never be compared in this divide-and-conquer algorithm.

Given any pivot element A[r], it takes n-1 comparisons to partition array A into two subarrays and the right location for A[r] such that all smaller keys are to its left and larger keys to the right. The partition, as shown below, is “in-place”. Is it also stable?

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

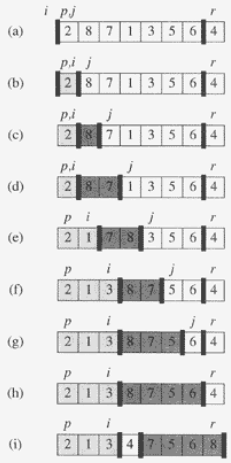



Figure 7.1 The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is “swapped with itself” and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 8 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6 and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

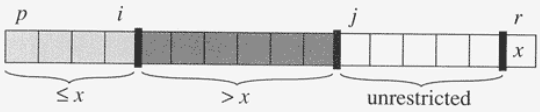


Figure 7.2 The four regions maintained by the procedure PARTITION on a subarray $A[p..r]$. The values in $A[p..i]$ are all less than or equal to x, the values in $A[i+1..j-1]$ are all greater than x, and $A[r] = x$. The values in $A[j..r-1]$ can take on any values.

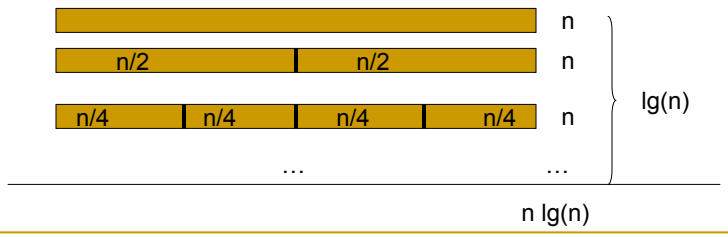
Quicksort: ideal-case

Each call to the partition subroutine will return a splitPoint which is right at the middle of the range, namely, divide the range into two equal subranges.

In doing so, at most n comparisons are needed to ensure one subrange contains only keys that are smaller than the pivot, and the other subrange only keys larger than the pivot.

Therefore,

$$T(n) = 2T(n/2) + n$$



Quicksort: worst-case

Each call to the partition subroutine will return a splitPoint which is just the left boundary of the range, namely, no keys are smaller than the pivot.

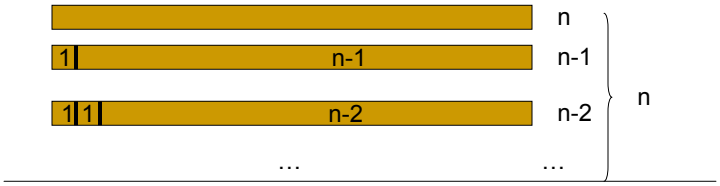
How many comparisons needed to know this? Still, all elements in the range need to compare with the pivot.

When does this happen? Ironically when the array is a sorted one.

In this case, divide-and-conquer decays into chip-and-conquer as the 2nd recursive call is quickSort(E, splitPoint + 1, last), chipping off one element from the range.

Therefore,

T(n) = T(n-1) + n



Θ (n²)

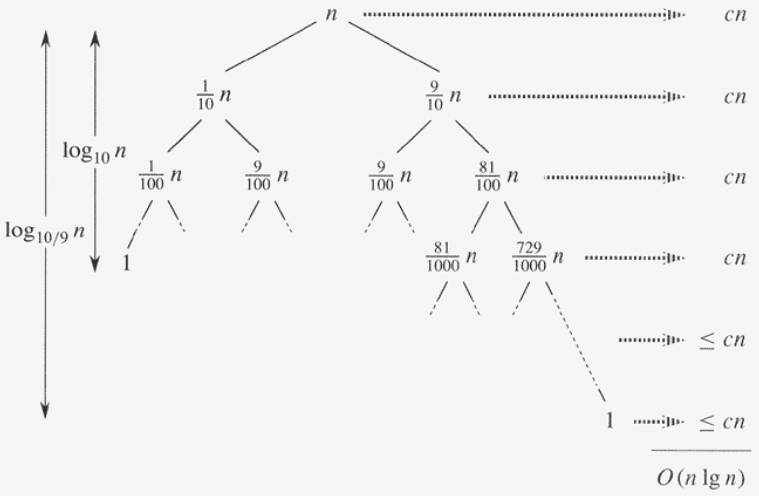


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Quicksort: Average-case

Assumption:

Prob(splitPoint = i| splitPoint = partiton[1 to n]) = 1/n

Then

$T(n) = n-1 + \sum_{i=0 \text{ to } n-1} (1/n) [T(i) + T(n-1-i)]$

$= n-1 + (1/n) \sum_{i=1 \text{ to } n-1} [T(i) + T(n-1-i)]$

$= n-1 + (2/n) \sum_{i=1 \text{ to } n-1} T(i)$

Claim: $T(n) \le c\, n \lg(n)$ holds for any $n \ge 1$, where c is a constant.

Proof: induction on n.

Base case n = 1. One single element array is sorted, i.e., T(1) = 0.

As $c\, 1 \ln(1) = 0$, the theorem holds for the base case.

Assume for any n, $T(n) \le c\, n \lg(n)$, then

$T(n+1) = (n+1) -1 + (2/(n+1)) \sum_{i=1 \text{ to } n} T(i)$

$\le n + (2/(n+1) \sum_{j=1 \text{ to } n} c\, i \lg(i)$

$\le n + (2c/(n+1) \int_1^{n+1} x \lg(x)\, dx$

$= n + 2c [(1/2)(n+1)^2 \lg (n+1) – (1/4) (n+1)^2 + (1/4)] / (n+1)$

$= c(n+1) \ln(n+1) + (n+1) [1- (c/2) + (c/2) (n+1)^{-2}] - 1$

$\le 2(n+1) \ln(n+1)$ if c = 2

QED

Using a randomly chosen element as pivot will enforce the equal distribution assumption made at the average-case analysis.

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

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q − 1)
4          RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Quicksort: space complexity

Quicksort is "in place" sort, because no extra array is needed. Yet there are hidden space usage: stacks for recursive calls -> $\Theta(n)$ for worst case, and $\lg(n)$ on average. The following modified algorithm guarantees this $\lg(n)$ usage of space.

```
QUICKSORT'(A, p, r)
1  while p < r
2        do ▷ Partition and sort left subarray.
3           q ← PARTITION(A, p, r)
4           QUICKSORT'(A, p, q − 1)
5           p ← q + 1
```


Mergesort

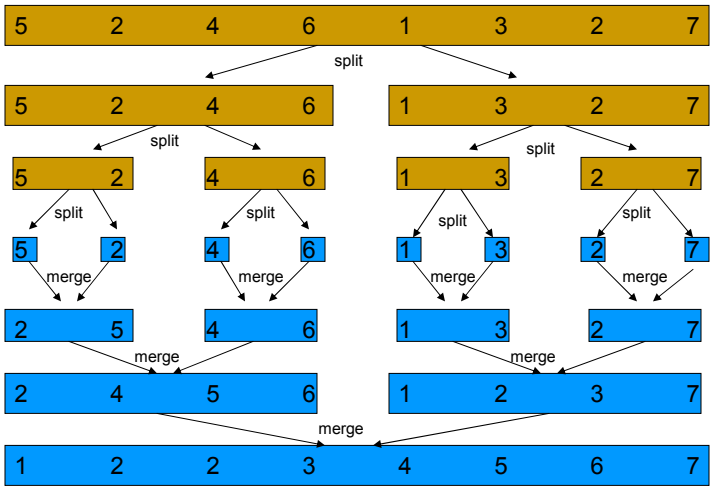
```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

- Unlike quickSort, mergeSort guarantees equal division each time.
- Array with just a single element is already sorted!
- Work is done at the combining steps

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 .. n1 + 1] and R[1 .. n2 + 1]
 4  for i ← 1 to n1
 5       do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7       do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13       do if L[i] ≤ R[j]
14             then A[k] ← L[i]
15                  i ← i + 1
16             else A[k] ← R[j]
17                  j ← j + 1
```


Mergesort: example

Is it a stable sorting?



Mergesort: analysis

Worst-case (Q: when?; and when is best case)

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1$

$T(1) = 0$

Master Theorem => $T(n) \in \Theta(n \log n)$

Note: 1st algorithm by far does nlog(n) for worst-case.
Can we do better? Or will be better off on average?