Lecture 12 – Quick sort

Review

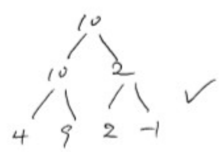
- Heap sort
    - heap structure properties
    - heapify
    - heapsort idea

e.g.
heap sort the following array: 2 4 -1 10 9 12 10

index 0 1 2 3 4 5 6 7
2 4 -1 10 9 2 10

Step1: make a heap $\Theta(n)$

2 4 10 10 9 2 -1
2 10 10 4 9 2 -1
10 10 2 4 9 2 -1 ✓



Step 2: root removal
index 1 2 3 4 5 6 7
-1 10 2 4 9 2 | 10
10 -1 2 4 9 2 | 10
10 9 2 4 -1 2 | 10 ✓
2 9 2 4 -1 | 10 10
9 2 2 4 -1 | 10 10
9 4 2 2 -1 | 10 10 ✓
-1 4 2 2 | 9 10 10
4 -1 2 2 | 9 10 10
4 2 2 -1 | 9 10 10 ✓
-1 2 2 | 4 9 10 10
2 2 -1 | 4 9 10 10 ✓
-1 2 | 2 4 9 10 10
2 -1 | 2 4 9 10 10 ✓
-1 | 2 2 4 9 10 10 #

Today's topics

- Quick sort

1. Quick sort
Q:/ What is the idea for merge sort?
A:/ partition the list into 2 halves based on **positions**.

Idea of quick Sort: Partition the list into 2 halves based on **values**.

- Place the "pivot" item in its final position in the sorted list → break the list into two parts
- All elements before the pivot are smaller than or equal to the pivot and all elements after the pivot are greater than or equal to the pivot

Illustration

A[0]	...	A[s-1]	A[s]	A[s+1]	...	A[n-1]
all are ≤A[s]				all are ≥A[s]		

Thus we can recursively sort the two sub-arrays (i.e. before and after A[s]) using the same method).

So **if we can find this position s**, then we can develop an algorithm called quick sort.
Quicksort (A,p, q)
if(p<q)
    x=A[p];//pivot
    partition A into a left subarray and right subarray at position s where elements on the left subarray <=x<=elements in the right subarray
    Quicksort(A,p,s-1)
    Quicksort(A,s+1,q)

So now the key is how to find this position s → Partition
Without losing generality, we will pick the first element in A to be the value of the pivot. The method to find where the position is called **double scan**.
Double scan: one scan from the left → right and one from right → left

- Left to Right Scan: use index *i* starting with the second element
- Right to Left Scan: use index *j* starting with the last element

	→*i*				*j*←
p	A[*l*+1]	A[*l*+2]	…	A[r-1]	A[r]

Goal: elements ≥ the pivot are in the second part of the array and elements ≤ the pivots are in the first part of the array

- *i*: skips elements <pivot, stop on encountering the first element ≥pivot
- *j:* skips elements >pivot, stop on encountering the first element ≤pivot

Details:

There are three situations depending on whether or not the scanning indexes have crossed
1. i and j haven't crossed each other

		$\rightarrow i$		$j\leftarrow$	
p	all are <p	≥p	……	≤p	all are>p

example

			$\rightarrow i$		$j\leftarrow$		
5	4	2	6	……	4	9	8

Q:/ What should we do?

A:/ Swap A[i] with A[j], resume the scans

2. i and j cross over

		$j\leftarrow$	$i\rightarrow$	
p	all are <p	≤p	≥p	all are >p

example

			$\leftarrow j$	$\rightarrow i$			
11	4	2	5	12	20	14	19

Q:/ What should we do?
A:/ swap pivot with p[j]

3. i and j meet at the same it

		$\rightarrow i = j \leftarrow$	
p	all are <p	=p	all are >p

example

			$\rightarrow i = j \leftarrow$			
11	4	2	11	20	14	19

Q:/ What should we do?
A:/either do nothing or swap pivot with A[j] (though these two values are the same)

Thus the algorithm for partitioning A[l,…,r] is the following
Partition (A, p, q)
p ← A[p]

i←p+1
j←q
repeat
    repeat i++ until A[i]>=p
    repeat j++ until A[j]<=p
    swap A[i] with A[j]
until i>=j

swap A[p] with A[j]
return j

Example: quick sort 5 3 1 9 8 2 4 7

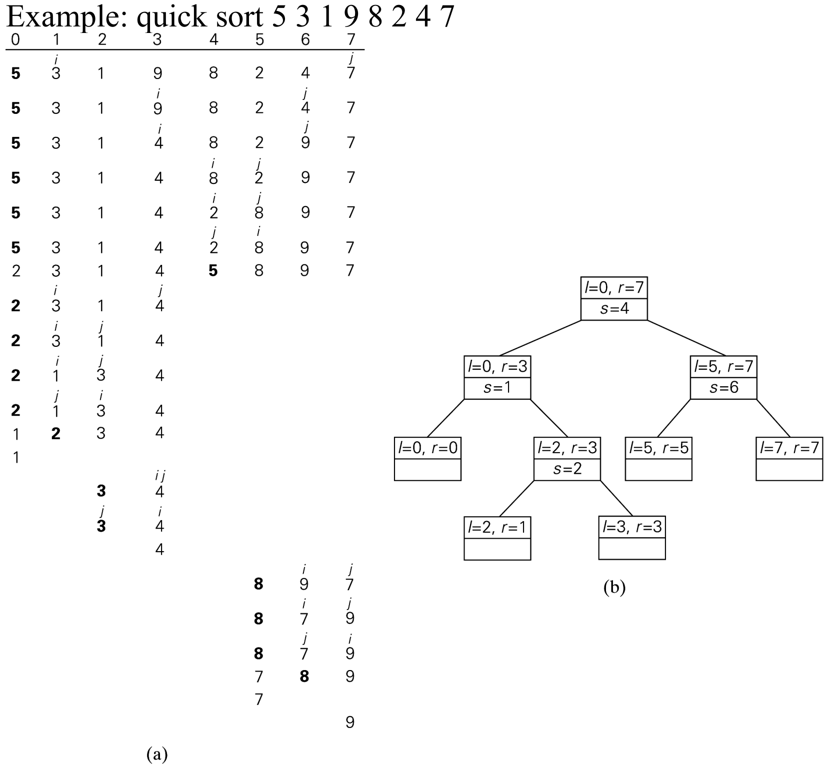


FIGURE 4.3 Example of Quicksort operation. (a) The array's transformations with pivots shown in bold. (b) The tree of recursive calls to *Quicksort* with input values l and r of subarray bounds and split position s of a partition obtained.

Efficiency analysis

Input size: # of items n

Q:/ Are there three cases?
A:/ Yes, sometimes we do n checkings and sometimes we do n+1 checkings

Q:/ When does best case happen?
A:/ when all splits happen in the middle of corresponding sub-arrays
T(n)=2T(n/2)+Θ(n) → efficiency $\Theta(n\log n)$

Q:/ When does worst case happen?
A:/ when the increasing list has been sorted!

$j \leftarrow$	$\rightarrow i$		
A[0]	A[1]	……	A[n-1]

$T(n)=T(n-1)+\Theta(n)$
Using recursion tree, we can find the efficiency is $\Theta(n^2)$

Q:/ Why is quick sort a good sorting algorithm since its worst case is the same as insertion sort?
A:/ It's because it's average case has very good properties.

The average case is $T(n) = 1.38n\log n$ and it can also be improved by 20-25%.

Q:/ How can we avoid the worst case scenario?
A:/ avoid sorted array → randomly pick the pivot!!