Searching, Sorting, Medians

part 2

Week 4 Objectives

- QuickSort
- QuickSort Running time
- Median Statistics
- Sorting in linear time

QuickSort - pseudocode

- QuickSort(A,b,e) *//array A , sort between indices b and e*

```
– q = Partition(A,b,e) // returns pivot q, b<=q<=e
–                      // Partition also rearranges A so that if i<q then A[i]<=A[q]
–                      //                                   and if i>q then A[i]>=A[q]
– if(b<q-1) QuickSort(A,b,q-1)
– if(q+1<e) QuickSort(A,q+1,e)
```

- After Partition the pivot index contains the right value:

b=0 q=3 e=9

b=0			q=3						e=9
-3	0	5	7	18	8	7	29	21	10

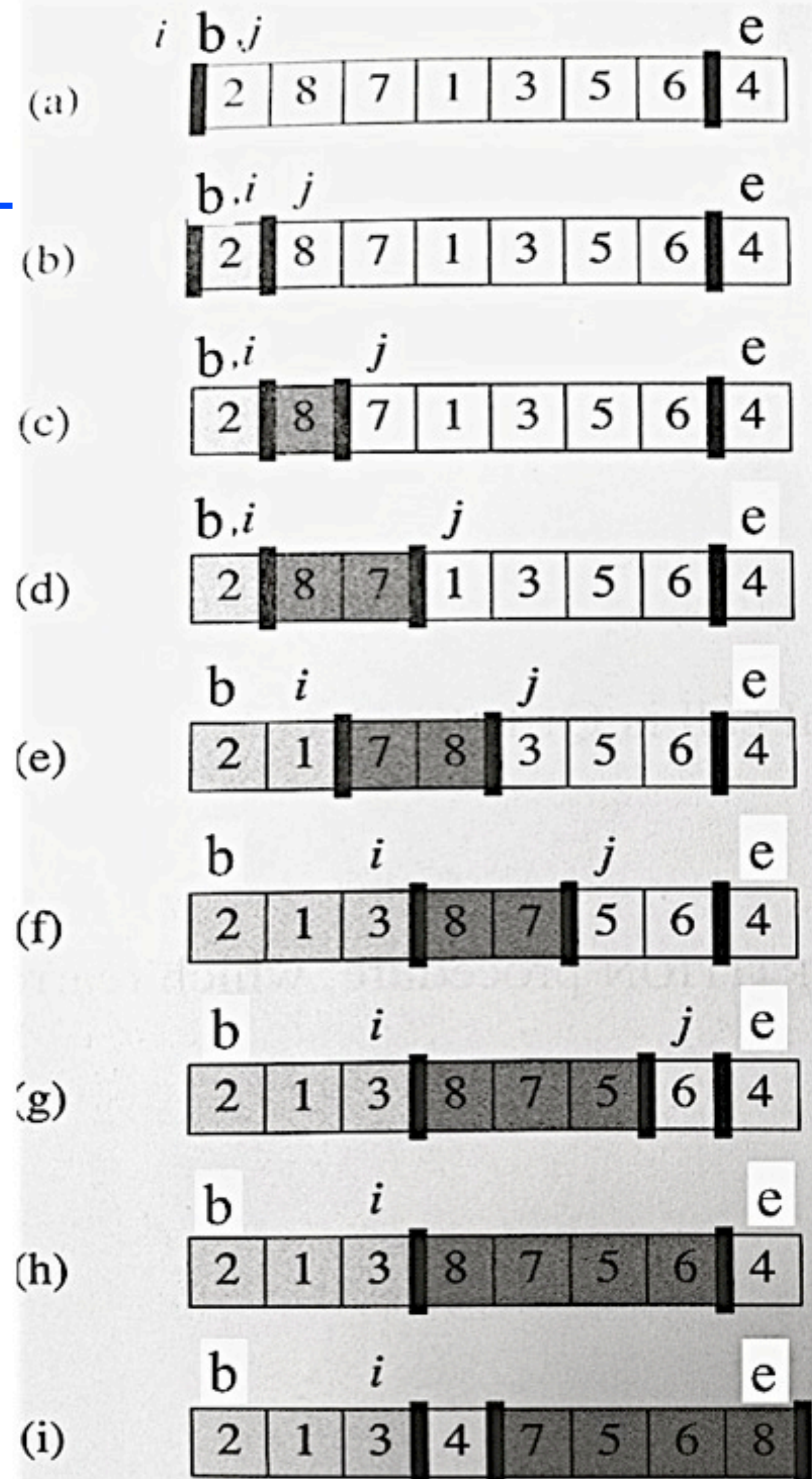
QuickSort Partition

- TASK: rearrange A and find pivot q, such that
  - all elements before q are smaller than A[q]
  - all elements after q are bigger than A[q]
- Partition (A, b, e)

```
– x=A[e]//pivot value
– i=b-1
– for j=b TO e-1
    • if A[j]<=x then
    •     i++; swap A[i]<->A[j]
  swap A[i+1]<->A[e]
– q=i+1; return q
```

Partition Example

- set pivot value x = A[e], // x=4
  - i =index of last value < x
  - i+1 = index of first value > x
- run j through array indices b to e-1
  - if A[j] <= x //see steps (d),(e)
    - swap (A[j] , A[i+1]);
    - i++; //advance i
- move pivot in the right place
  - swap (pivot=A[e] , A[i+1])
- return pivot index
  - return i+1



QuickSort time

- Partition runs in linear time
  - If pivot position is q, the QuickSort recurrence is $T(n) = n + T(q) + T(n-q)$
- Best case q is always in the middle
  - $T(n)=n+2T(n/2)$, overall $\Theta(n*\log n)$
- Worst case: q is always at extreme, 1 or n
  - $T(n) = n + T(1) + T(n-1)$, overall $\Theta(n^2)$

QuickSort Running Time

- Depends on the Partition balance
- Worst case: Partition produces unbalanced split n = (1, n-1) most of the time
  - results in $O(n^2)$ running time
- Average case: most of the time split balance is not worse than n = (cn, (1-c)n) for a fixed c
  - for example c=0.99 means balance not worse than (1/100*n, 99/100*n)
  - results in O(n*logn) running time
  - can prove that on expectation (average), if pivot value is chosen randomly, running time is Θ(n*logn), see book.

Median Stats

- Task: find k-th element
  - k=n is same as "find MAX", or "find highest"
  - k=2 means "find second-smalles"
  - k=1 is same as "finding MIN"
- naive approach, based on selection sort:
  - find first smallest (MIN)
  - then find second smallest, third smallest, etc; until the k-th smallest element
  - Running Time: average case $k=\Theta(n)$, and each "finding" min takes $\Theta(n)$ time, so total $\Theta(n^2)$

Median Stats

- "find k-th element"
- better approach, based on QuickSort
- Median(A,b,e,k) *//find k-th greatest in array A , sort between indices b=1 and e=n*

```
  – q = Partition(A,b,e) //returns pivot index q, b<=q<=e
  –                      //Partition also rearranges A so that if i<q then A[i]<=A[q]
  –                      //                                and if i>q then A[i]>=A[q]
  – if(q==k) return A[q] //found the k-th greatest
  – if(q>k) Median(A,b,q-1,k)
      – else Median(A,q+1,e,q-k)
```

- Not like Quiksort, Median recursion goes only on one side, depending on the pivot
- why the second Median call has $k_{(new)}=q-k_{(old)}$?

Median Stats

- Running Time of Median
- the recursive calls makes $T(n) = n + \max(T(q), T(n-q))$
  - "max" : assuming the recursion has to call the longer side
  - just like QuickSort, average case is when q is "balanced", i.e. $cn<q<(1-c)n$ for some constant $0<c<1$
  - balanced case: $T(n) = n + T(cn)$; Master Theorem gives linear time $\Theta(n)$
  - expected (average) case can be proven linear time (see book); worst case $\Theta(n^2)$
- worst case can run in linear time with a rather complicated choice of the pivot value before each partition call (see book)

Linear-time Sorting: Counting Sort

- Counting Sort (A[]) : count values, NO comparisons
- STEP 1 : build array C that counts A values
  - init C[]=0 ;
  - run index i through A
    - value = A[i]
    - C[value] ++; *//counts each value occurrence*
- STEP 2: assign values to counted positions
  - init position=0;
  - for value=0:RANGE
    - for i=1:C[value]
      - position = position+1;
    - OUTPUT[position]=value;

Counting Sort

- n elements with values in k-range of $\{v_1, v_2, \ldots v_k\}$
  - for example: 100,000 people sorted by age: n=100,000; k = {1,2,3,...170} since 170 is maximum reasonable age in years.
- Linear Time $\Theta(n+k)$
  - Beats the bound? YES, linear $\Theta(n)$, not $\Theta(n*\log n)$, if k is a constant
  - Definitely appropriate when k is constant or increases very slowly
  - Not good when k can be large. Example: sort pictures by their size; n=10000 (typical picture collection), size range k can be any number from 200Bytes to 40MBytes.
- Stable (equal input elements preserve original order)

Radix Sort

- Counting sort on each digit

Radix Sort

- Counting sort on each digit

329
457
657
839
436
720
355

Radix Sort

- Counting sort on each digit

329	720
457	355
657	436
839	457
436	657
720	329
355	839

Radix Sort

- Counting sort on each digit

329	720	720
457	355	329
657	436	436
839	457	839
436	657	355
720	329	457
355	839	657

Radix Sort

- Counting sort on each digit

329	720	720
457	355	329
657	436	436
839	457	839
436	657	355
720	329	457
355	839	657

Still sorted (due to stability) if the current sort column does not

Radix Sort

- Counting sort on each digit

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Still sorted (due to stability) if the current sort column does not

Radix Sort Analysis

- critical that the digit-sorting procedure is stable
  - (329, 355) remain properly sorted when the third digit is used
- counting sort fits the bill: stable, also linear when the range is fixed, like base 10 digits {0-9}
- each digit-sort is linear. But how many digits ?
  - quick informal answer: log(n) digits with fixed range, so O(n*logn) total.

Radix Sort Analysis

- each digit-sort is linear. We can represent items with few/many bits by choosing representation base
- b bits per item, n items. (b,n fixed).
  - for example computers typically represent integers on b=32 bits and long integers on b=64 bits
  - limit to 2^b items total
- use r bits per digit -> number of digits d = b/r. (r,d up to us, variables)
  - each digit sort $\Theta(n+2^r)$, d=b/r digits, so total $\Theta(b/r*(n+2^r))$
  - choosing $r \approx \log(n)$, total is $\Theta(b/\log(n)*(n+n)) = \Theta(bn/\log(n))$

Sorting : stable; in place

- stable: preserve relative order of elements with same value
- in place: dont use significant additional space (arrays)

	time	in-place	stable
Bubble	n^2	✔	✔
Insertion	n^2	✔	✔
Selection	n^2	✘	?
QuickSort	n*log(n)	✔	?
MergeSort	n*log(n)	✘	✔