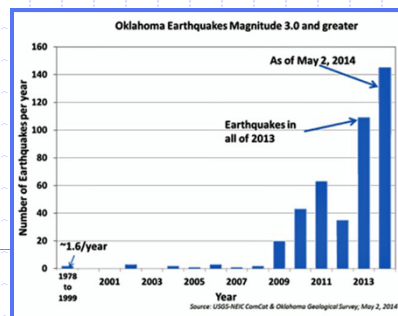


Presentation for use with the textbook, *Algorithm Design and Applications*, by M. T. Goodrich and R. Tamassia, Wiley, 2015

Fast Sorting and Selection



USGS NEIC. Public domain government image.

1

1

A Lower Bound for Worst Case

Theorem: Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Proof:

- Suffices to determine the height of a decision tree.
- The number of leaves is at least $n!$ (# outputs)
- The number of internal nodes $\geq n! - 1$
- The height is at least $\log(n! - 1) = \Omega(n \lg n)$

2

Can we do better?

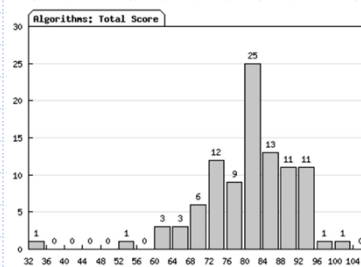
- Linear sorting algorithms
 - Bucket Sort
 - Counting Sort (special case of Bucket Sort)
 - Radix Sort
- Make certain assumptions about the data
- Linear sorts are NOT “comparison sorts”

3

3

Application: Constructing Histograms

- One common computation in data visualization and analysis is computing a **histogram**.
- For example, n students might be assigned integer scores in some range, such as 0 to 100, and are then placed into ranges or “buckets” based on these scores.



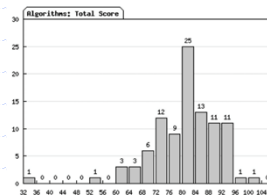
A histogram of scores from a recent Algorithms course.

4

4

Application: An Algorithm for Constructing Histograms

- When we think about the algorithmic issues in constructing a histogram of n scores, it is easy to see that this is a type of sorting problem.
- But it is not the most general kind of sorting problem, since the keys being used to sort are simply integers in a given range.
- So a natural question to ask is whether we can sort these values faster than with a general comparison-based sorting algorithm.
- The answer is "yes." In fact, we can sort them in $O(n)$ time.



5

5

Bucket-Sort

- Let be S be a sequence of n (key, element) items with keys in the range $[0, r - 1]$
- Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)
 - Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$
 - Phase 2: For $i = 0, \dots, r - 1$, move the entries of bucket $B[i]$ to the end of sequence S
- Analysis:
 - Phase 1 takes $O(n)$ time
 - Phase 2 takes $O(n + r)$ time
 Bucket-sort takes $O(n + r)$ time

Algorithm bucketSort(S):

Input: Sequence S of entries with integer keys in the range $[0, r - 1]$

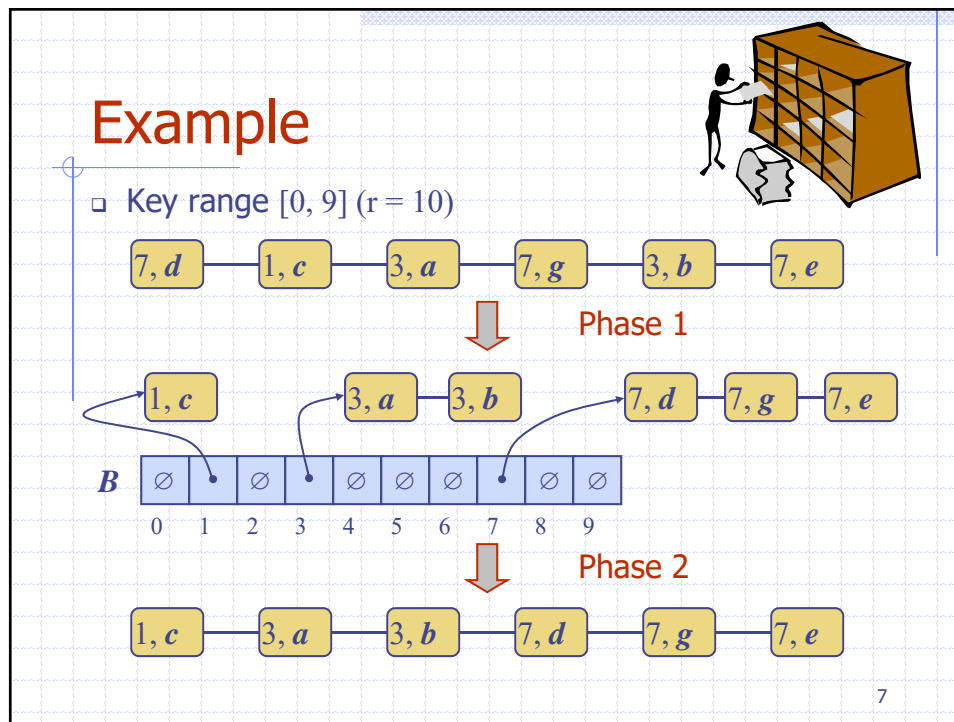
Output: Sequence S sorted in nondecreasing order of the keys
 let B be an array of N sequences, each of which is initially empty

```

for each entry  $e$  in  $S$  do
     $k$  = the key of  $e$ 
    remove  $e$  from  $S$ 
    insert  $e$  at the end of bucket  $B[k]$ 
for  $i = 0$  to  $r-1$  do
    for each entry  $e$  in  $B[i]$  do
        remove  $e$  from  $B[i]$ 
        insert  $e$  at the end of  $S$ 
    
```

6

6



7

Array-based Implementation: Counting Sort

□ Assumptions:

- n integers which are in the range $[0 \dots r-1]$
- r has the same growth rate as n , that is, $r = O(n)$

□ Idea:

- For each element x , find the number of occurrences of x and store it in the counter
- Place x into its correct position in the output array using the counter.

8

8

Step 1

Find the number of times $A[i]$ appears in A (i.e., frequencies)

input array A:

3	6	4	1	3	4	1	4
---	---	---	---	---	---	---	---

allocate C

1	2	3	4	5	6
0	0	0	0	0	0

 ($r=7$)

$i=1, A[1]=3$

1	2	3	4	5	6
0	0	1	0	0	0

 $C[A[1]]=C[3]=1$

$i=2, A[2]=6$

1	2	3	4	5	6
0	0	1	0	0	1

 $C[A[2]]=C[6]=1$

$i=3, A[3]=4$

1	2	3	4	5	6
0	0	1	1	0	1

 $C[A[3]]=C[4]=1$

⋮

$i=8, A[8]=4$

1	2	3	4	5	6
2	0	2	3	0	1

 $C[A[8]]=C[4]=3$

$C[i]$ = number of times element i appears in A

```
for (i = 0; i < n; i++)
    C[A[i]]++;
```

9

Step 2

1. `int index = 0`
2. `For i = 0 to r-1`
3. `For j = 1 to C[i]`
4. `A[index++] = i`
 `// Copy value i into the array C[i] times`


Example:

```
i: 0 1 2 3 4 5 6
C = [ 0 2 0 2 3 0 1 ]
A = [ 1 1 3 3 4 4 4 6 ]
```

10

10

Properties and Extensions



- Key-type Property
 - The keys are used as indices into an array and cannot be arbitrary objects
 - No external comparator
- Stable Sort Property
 - The relative order of any two items with the same key is preserved after the execution of the algorithm

Extensions

- Integer keys in the range $[a, b]$
 - ◆ Put entry (k, o) into bucket $B[k - a]$
- Float numbers round to integers
- String keys from a set D of possible strings, where D has constant size (e.g., names of the 50 U.S. states)
 - ◆ Sort D and compute the rank $r(k)$ of each string k of D in the sorted sequence
 - ◆ Put entry (k, o) into bucket $B[r(k)]$

11

11

Example - Bucket Sort $R = [0..0.99]$

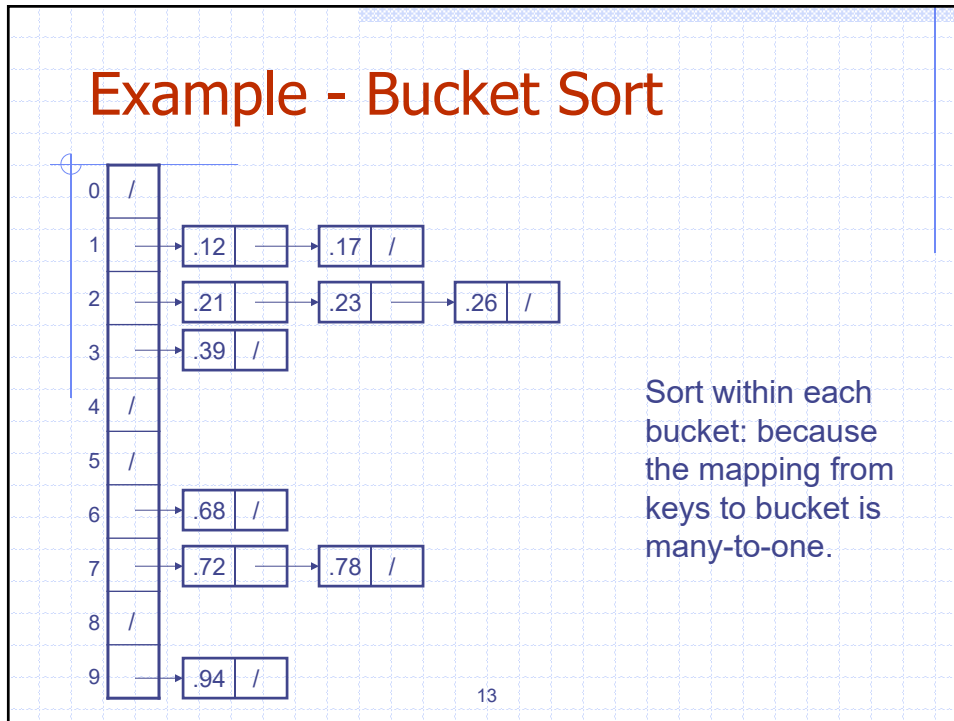
A	1	.78
	2	.17
	3	.39
	4	.26
	5	.72
	6	.94
	7	.21
	8	.12
	9	.23
	10	.68

B	0	/
	1	→ .17 → .12 /
	2	→ .26 → .21 → .23 /
	3	→ .39 /
	4	/
	5	/
	6	→ .68 /
	7	→ .78 → .72 /
	8	/
	9	→ .94 /

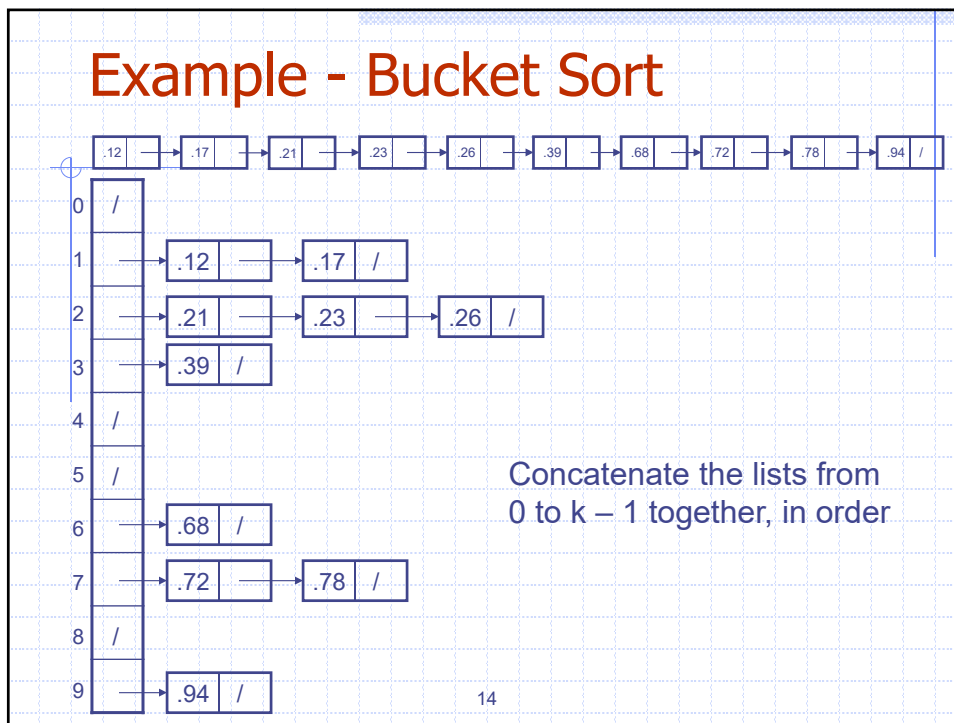
Distribute
Into buckets

12

12



13



14

Analysis of Extended Bucket Sort

Alg.: BUCKET-SORT(A, n)

for $i \leftarrow 1$ to n	}	O(n)
do insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$		
for $i \leftarrow 0$ to $r - 1$	}	k O(n/r log(n/r)) =O(nlog(n/r)) (average case)
do sort list $B[i]$ with merge sort		
concatenate lists $B[0], B[1], \dots, B[r - 1]$	}	O(n+r)
together in order		
return the concatenated lists		

Note: If the mapping from keys to buckets is 1-to-1, there is no need to sort each bucket, and the time is the worst case, not the average case. O(n) (if $r = \Theta(n)$)

15

Lexicographic Order

- A d -tuple is a sequence of d keys (k_1, k_2, \dots, k_d) , where key k_i is said to be the i -th dimension of the tuple
- Example:
 - The Cartesian coordinates of a point in 3D space are a 3-tuple
- The lexicographic order of two d -tuples is recursively defined as follows

$$(x_1, x_2, \dots, x_d) <_{\text{lex}} (y_1, y_2, \dots, y_d)$$

$$\Leftrightarrow x_1 < y_1 \vee x_1 = y_1 \wedge (x_2, \dots, x_d) <_{\text{lex}} (y_2, \dots, y_d)$$

I.e., the tuples are compared by the first dimension, then by the second dimension, etc.

16

16

Lexicographic-Sort

- Let C_i be the comparator that compares two tuples by their i -th dimension
- Let $stableSort(S, C)$ be a stable sorting algorithm that uses comparator C
- Lexicographic-sort sorts a sequence of d -tuples in lexicographic order by executing d times algorithm $stableSort$, one per dimension
- Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of $stableSort$

Algorithm *lexicographicSort(S)*

Input sequence S of d -tuples
Output sequence S sorted in lexicographic order

```

for  $i \leftarrow d$  downto 1
   $stableSort(S, C_i)$ 
  //  $C_i$  compares  $i$ -th dimension
  
```

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)
 (2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)
 (2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)
 (2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

17

Correctness of Alg. *lexicographicSort(S)*

Theorem: Alg. *lexicographicSort(S)* sorts S by lexicographic order.

Proof: Induction on d .

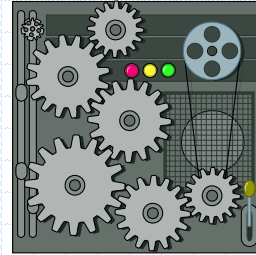
- Base case: $d=1$, $stableSort(S, C_1)$ will do the job.
- Induction hypothesis: Theorem is true for $d' < d$.
- Inductive case:
 - Suppose $(x_1, x_2, \dots, x_d) <_{lex} (y_1, y_2, \dots, y_d)$.
 - If $x_1 < y_1$, then the last round places (x_1, x_2, \dots, x_d) before (y_1, y_2, \dots, y_d) .
 - If $x_1 = y_1$, then $(x_2, \dots, x_d) <_{lex} (y_2, \dots, y_d)$.
 - By induction hypothesis, the previous rounds will place (x_2, \dots, x_d) before (y_2, \dots, y_d) . And we use a stable sort, so (x_1, x_2, \dots, x_d) goes before (y_1, y_2, \dots, y_d) .

18

18

Radix-Sort

- Radix-sort is a special case of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension.
- Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range $[0, r - 1]$
- Radix-sort runs in time $O(d(n + r))$
- If d is constant and r is $O(n)$, then this is $O(n)$.



Algorithm *radixSort(S, N)*

Input sequence S of d -tuples such that $(0, \dots, 0) \leq (x_1, \dots, x_d)$ and $(x_1, \dots, x_d) \leq (N - 1, \dots, N - 1)$ for each tuple (x_1, \dots, x_d) in S

Output sequence S sorted in lexicographic order

for $i \leftarrow d$ **downto** 1
 bucketSort(S, N)

19

19

Radix Sort Example

- Represents keys as d -digit numbers in some base- r
 $\text{key} = x_1x_2\dots x_d$ where $0 \leq x_i \leq r-1$
- Example: key=15
 $\text{key}_{10} = 15, d=2, r=10$ where $0 \leq x_i \leq 9$

20

20

Radix Sort Example

- Sorting looks at one column at a time
 - For a **d** digit number, sort the least significant digit first
 - Continue sorting on the next least significant digit, until all digits have been sorted
 - Requires only **d** passes through the list

326
453
608
835
751
435
704
690

21

21

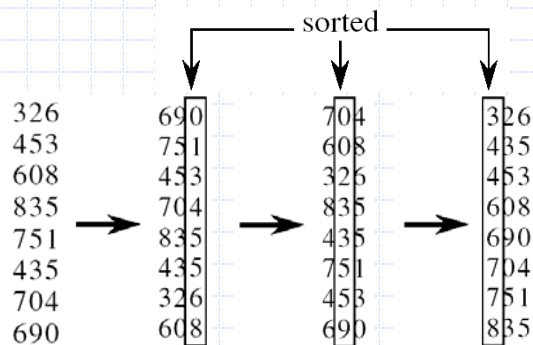
RADIX-SORT

Alg.: RADIX-SORT(*A*, *d*)

for *i* ← 1 **to** *d*

do use a **stable** bucket sort of array *A* on digit *i*

(stable sort: preserves order of identical elements)



22

22

Analysis of Radix Sort

- Given n numbers of d digits each, where each digit may take up to k possible values, RADIX-SORT correctly sorts the numbers in $O(d(n+k))$
 - One pass of sorting per digit takes $O(n+k)$ assuming that we use **bucket sort**
 - There are d passes (for each digit)

23

23

Summary: Beating the lower bound

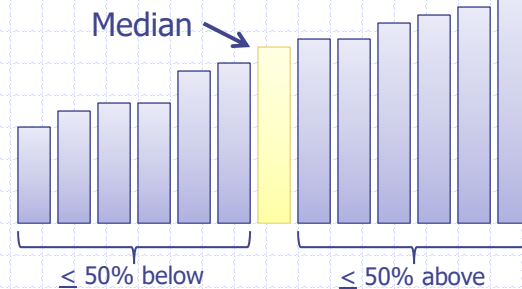
- We can beat the lower bound if we don't base our sort on comparisons:
 - **Counting sort** for keys in $[0..k]$, $k=O(n)$
 - **Bucket sort** for keys which can map to small range of integers (uniformly distributed)
 - **Radix sort** for keys with a fixed number of "digits"



24

Finding Medians

- A common data analysis tool is to compute a **median**, that is, a value taken from among n values such that there are at most $n/2$ values larger than this one and at most $n/2$ elements smaller.
- Of course, such a number can be found easily if we were to sort the scores, but it would be ideal if we could find medians in $O(n)$ time without having to perform a sorting operation.



25

25

Selection: Finding the Median and the k th Smallest Element

- The median of a sequence of n sorted numbers $A[1 \dots n]$ is the “middle” element.
- If n is odd, then the middle element is the $(n+1)/2^{\text{th}}$ element in the sequence.
- If n is even, then there are two middle elements occurring at positions $n/2$ and $n/2+1$. In this case, we will choose the $n/2^{\text{th}}$ smallest element.
- Thus, in both cases, the median is the $\lceil n/2 \rceil^{\text{th}}$ smallest element.
- The k th smallest element is a general case.

26

The Selection Problem

- Given an integer k and n elements x_1, x_2, \dots, x_n taken from a total order, find the k -th smallest element in this set.
- Of course, we can sort the set in $O(n \log n)$ time and then index the k -th element.

$k=3$ 7 4 9 6 2 → 2 4 6 7 9

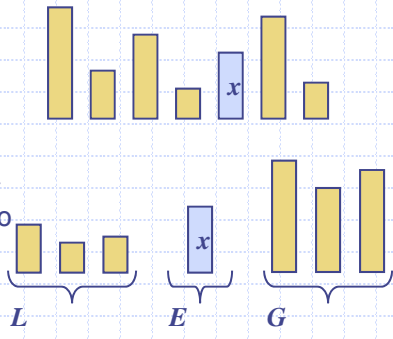
- We want to solve the selection problem faster.

27

27

Quick-Select

- Quick-select is a randomized selection algorithm based on the prune-and-search paradigm:
 - **Prune:** pick a random element x (called **pivot**) and partition S into
 - ♦ L : elements less than x
 - ♦ E : elements equal x
 - ♦ G : elements greater than x
 - **Search:** depending on k , either answer is in E , or we need to recur in either L or G



$k \leq |L|$ $k > |L| + |E|$
 $k' = k - |L| - |E|$

$|L| < k \leq |L| + |E|$
 (done)

28

28

Pseudo-code

Algorithm quickSelect(S, k):

Input: Sequence S of n comparable elements, and an integer $k \in [1, n]$

Output: The k th smallest element of S

if $n = 1$ **then**

return the (first) element of S

pick a random element x of S

remove all the elements from S and put them into three sequences:

- L , storing the elements in S less than x
- E , storing the elements in S equal to x
- G , storing the elements in S greater than x .

if $k \leq |L|$ **then**

 quickSelect(L, k)

else if $k \leq |L| + |E|$ **then**

return x // each element in E is equal to x

else

 quickSelect($G, k - |L| - |E|$)

- Note that partitioning takes $O(n)$ time.

29

29

Quick-Select Visualization

- An execution of quick-select can be visualized by a recursion path
 - Each node represents a recursive call of quick-select, and stores k and the remaining sequence

$k=5, S=(7\ 4\ 9\ \underline{3}\ 2\ 6\ 5\ 1\ 8)$

$k=2, S=(7\ 4\ 9\ 6\ 5\ \underline{8})$

$k=2, S=(7\ \underline{4}\ 6\ 5)$

$k=1, S=(7\ 6\ \underline{5})$

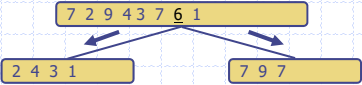
5

30

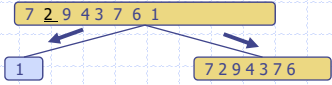
30

Expected Running Time

- Consider a recursive call of quick-select on a sequence of size s
 - **Good call:** the sizes of L and G are each less than $3s/4$
 - **Bad call:** one of L and G has size greater than $3s/4$




Good call



Bad call

- A call is **good** with probability $1/2$
 - $1/2$ of the possible pivots cause good calls:



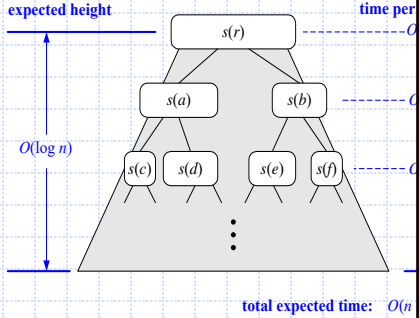
Bad pivots Good pivots Bad pivots

31

31

Expected Running Time, Part 2

- **Probabilistic Fact:** The expected number of coin tosses required in order to get k heads is $2k$
- For a node of depth i , we expect
 - $i/2$ ancestors are good calls
 - The size of the input sequence for the current call is at most $(3/4)^{i/2}n$
- ◆ Therefore, we have
 - For a node of depth $2\log_{4/3}n$, the expected input size is one
 - The expected height of the quick-sort tree is $O(\log n)$
- ◆ The amount of work done at the nodes of the same depth is $O((3/4)^{i/2}n)$
- ◆ Thus, the expected running time of quick-sort is $O(n)$



total expected time: $O(n)$

32

32

Expected Running Time



- Let $T(n)$ denote the expected running time of quick-select.
- By Fact #2,
 - $T(n) \leq T(3n/4) + bn \cdot (\text{expected \# of calls before a good call})$
- By Fact #1,
 - $T(n) \leq T(3n/4) + 2bn$
- That is, $T(n)$ is a geometric series:
 - $T(n) \leq 2bn + 2b(3/4)n + 2b(3/4)^2n + 2b(3/4)^3n + \dots$
- So $T(n)$ is $O(n)$.
- We can solve the selection problem in $O(n)$ expected time.

33

33

Linear Time Selection Algorithm

- Also called Median Finding Algorithm.
- Find k^{th} smallest element in $O(n)$ time in worst case.
- Uses Divide and Conquer strategy.
- Uses elimination in order to cut down the running time substantially.

34

Deterministic Selection

- If we select an element m among A , then A can be divided into 3 parts:

$$L = \{ a \mid a \text{ is in } A, a < m \}$$

$$E = \{ a \mid a \text{ is in } A, a = m \}$$

$$G = \{ a \mid a \text{ is in } A, a > m \}$$

- According to the number elements in L, E, G , there are following three cases. In each case, where is the k -th smallest element?

Case 1: $ L \geq k$	The k -th element is in L
Case 2: $ L + E \geq k > L $	The k -th element is in E
Case 3: $ L + E < k$	The k -th element is in G

35

Deterministic Selection



- We can do selection in $O(n)$ worst-case time.
- Main idea: recursively use the selection algorithm itself to find a good pivot for quick-select:
 - Divide S into $n/5$ groups of 5 each
 - Find a median in each group
 - Recursively find the median of the “baby” medians.

Min size
for L

1	1	1	1	1	1	2	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5	5	5

Min size
for G

36

36

Steps to solve the problem

- Step 1: If n is small, for example $n < 45$, just sort and return the k^{th} smallest number in constant time i.e; $O(1)$ time.
- Step 2: Group the given numbers in subsets of 5 in $O(n)$ time.
- Step 3: Sort each of the group in $O(n)$ time. Find median of each group.

37

Example:

- Given a set
(.....2,6,8,19,24,54,5,87,9,10,44,32,21,13,3,4,
18,26,36,30,25,39,47,56,71,91,61,44,28.....)
having n elements.

38

Arrange the numbers in groups of five

.....	2	54	44	4	25
.....	6	5	32	28	39
.....	8	87	21	36	47
.....	19	9	13	16	56
.....	24	10	3	30	71

39

Sort each group of 5 from top to bottom

.....	2	5	3	4	25
.....	6	9	13	16	39
.....	8	10	21	28	47
.....	19	54	32	30	56
.....	24	87	44	36	71

Each group of 5 is sorted

40

Step 4: Find median of $n/5$ group medians recursively

.....	2	5	3	4	25
.....	6	9	13	16	39
.....	8	10	21	28	47
.....	19	54	32	30	56
.....	24	87	44	36	71

Median of each group

41

There are $s = n/5$ groups, there are $s/2$ groups on the left of m and $s/2$ groups on the right of m .

So there are $3/2s - 1 = 3n/10 - 1$ numbers less than m and $3n/10 - 1$ numbers greater than m .

.....	2	5	3	4	25
.....	6	9	13	16	39
.....	8	10	21	28	47
.....	19	54	32	30	56
.....	24	87	44	36	71

Find m , the median of medians

42

Step 5: Find the sets L, E, and G

- Compare each (n-1) elements in the top-right and bottom-left regions with the median m and find three sets L, E, and G such that every element in L is smaller than m, every element in E is equal to m, and every element in G is greater than m.

$$3n/10 - |E| \leq |L| \leq 7n/10 - |E|$$

(|L| is the size or cardinality of L)

$$3n/10 - |E| \leq |G| \leq 7n/10 - |E|$$

43

Pseudo code: Finding the k-th Smallest Element

- Input:** An array $A[1 \dots n]$ of n elements and an integer k , $1 \leq k \leq n$;
- Output:** The k th smallest element in A ;
- 1. $select(A, n, k)$;

44

Pseudo code: Finding the k -th Smallest Element

- $select(A, n, k)$
- 2. if $n < 45$ then sort A and return $(A[k])$;
- 3. Let $q = \lceil n/5 \rceil$. Divide A into q groups of 5 elements each.
 - If 5 does not divide n , then add max element;
- 4. Sort each of the q groups individually and extract its median.
 - Let the set of medians be M .
- 5. $m \leftarrow select(M, q, \lceil q/2 \rceil)$;
- 6. Partition A into three arrays:
 - $L = \{a \mid a < m\}$, $E = \{a \mid a = m\}$, $G = \{a \mid a > m\}$;
- 7. case
 - $|L| \geq k$: return $select(L, |L|, k)$;
 - $|L| + |E| \geq k$: return m ;
 - $|L| + |E| < k$: return $select(G, |G|, k - |L| - |E|)$;
- 8. end case;

45

Complexity: Finding the k -th Smallest Element (Bound time: $T(n)$)

- $select(A, n, k)$
- 2. if $n < 45$ then sort A and return $(A[k])$; O(1)
- 3. Let $q = \lceil n/5 \rceil$. Divide A into q groups of 5 elements each. O(n)
 - If 5 does not divide n , then add max element;
- 4. Sort each of the q groups individually and extract its median. O(n)
 - Let the set of medians be M .
- 5. $m \leftarrow select(M, q, \lceil q/2 \rceil)$; T(n/5)
- 6. Partition A into three arrays:
 - $L = \{a \mid a < m\}$, $E = \{a \mid a = m\}$, $G = \{a \mid a > m\}$; O(n)
- 7. case
 - $|L| \geq k$: return $select(L, |L|, k)$; T(7n/10)
 - $|L| + |E| \geq k$: return m ; O(1)
 - $|L| + |E| < k$: return $select(G, |G|, k - |L| - |E|)$; T(7n/10)
- 8. end case;

Summary: $T(n) = T(n/5) + T(7n/10) + a*n$

46

Analysis: Finding the k -th Smallest Element

- What is the best case time complexity of this algorithm?
- $O(n)$ when $|L| < k \leq |L| + |E|$
- $T(n)$: the worst case time complexity of $\text{select}(A, n, k)$

$$T(n) = T(n/5) + T(7n/10) + a*n$$
- The k -th smallest element in a set of n elements drawn from a linearly ordered set can be found in $\Theta(n)$ time.

47

Recursive formula

$$T(n) = T(n/5) + T(7n/10) + a*n$$

We will solve this equation in order to get the complexity.

We guess that $T(n) \leq Cn$ for a constant, and then by induction on n .

The base case when $n < 45$ is trivial.

$$T(n) = T(n/5) + T(7n/10) + a*n$$

$$\leq C*n/5 + C*7*n/10 + a*n \quad (\text{by induction hypothesis})$$

$$= ((2C + 7C)/10 + a)n$$

$$= (9C/10 + a)n$$

$$\leq Cn \quad \text{if } C \geq 9C/10 + a, \text{ or } C/10 \geq a, \text{ or } C \geq 10a$$

So we let $C = 10a$.

Then $T(n) \leq Cn$.

So $T(n) = O(n)$.

48

Why group of 5??

- If we divide elements into groups of 3 then we will have
 $T(n) = a*n + T(n/3) + T(2n/3)$
so $T(n)$ cannot be $O(n)$
- If we divide elements into groups of more than 5, finding the median of each group will be more, so grouping elements into 5 is the optimal situation.