

CSE 3101: Introduction to the Design and Analysis of Algorithms

Instructor: Suprakash Datta (datta[at]cse.yorku.ca) ext 77875

Lectures: Tues, BC 215, 7–10 PM

Office hours: Wed 4-6 pm (CSEB 3043), or by appointment.

Textbook: Cormen, Leiserson, Rivest, Stein.
Introduction to Algorithms (**3nd Edition**)

Minimum and Maximum

Problem: Find the maximum and the minimum of n elements.

- Naïve algorithm 1: Find the minimum, then find the maximum -- $2(n-1)$ comparisons.
- Naïve algorithm 2: Find the minimum, then find the maximum of $n-1$ elements -- $(n-1) + (n-2) = 2n - 3$ comparisons.

Minimum and Maximum – better algorithms

Problem: Find the maximum and the minimum of n elements.

Approach 1

- Sort $n/2$ pairs. Find min of losers, max of winners.

comparisons: $n/2 + n/2 - 1 + n/2 - 1 = 3n/2 - 2$.

Is this the best possible?

Approach 2

- Divide into $n/2$ pairs. Compare the first pair, set winner to current max, loser to current min.

- Sort next pair, compare winner to current max, loser to current min.

#comparisons: $1 + 3(n/2 - 1) = 3n/2 - 2$.

Lower bounds for the MIN and MAX

Claim: Every comparison-based algorithm for finding both the minimum and the maximum of n elements requires at least $(3n/2)-2$ comparisons.

Idea: Use similar argument as for the minimum

Max = maximum and Min=minimum only if:

Every element other than min has won at least 1

Every element other than max has lost at least 1

A proof?

“Proof” from the web: For each comparison, $x < y$, score a point if this is first comparison that x loses or if y wins and 2 points if both occur. Before the algorithm can terminate $n-2$ must both win and lose (since they aren't min or max) and 2 elements must either win or lose. Thus, $2(n-2)+2$ points are scored before termination.

Define A to be the set of elements that have not won or lost a comparison. All comparisons between elements in A must score 2 points. All other comparisons can score at most 1 point. Let X be A - A comparisons. Let Y be number of other comparisons. We want to minimize $X+Y$ such that $2X+Y \geq 2n-2$ & $X \leq n/2$ (assume n is even). Given the constraints we want to make X as big as possible. So set $X=n/2$. Then $Y \geq 2n-2-2X \Rightarrow Y \geq 2n-2-n \Rightarrow Y \geq n-2 \Rightarrow X+Y \geq n/2 + n - 2$.

Is the previous proof correct?

Lower bounds for the MIN and MAX

Idea: Define 4 sets: U: has not participated in a comparison

W: has won all comparisons

L: has lost all comparisons

N: has won and lost at least one comparison

Note: All these sets are disjoint.

1. Initially all elements in U.
2. Finally no elements in U, 1 each in W, L and $n-2$ in N.
3. Each element in N comes from U via W or L.

Lower bounds for the MIN and MAX - contd

Idea: Score a point when an element enters W or L or N for the first time.

Question: Can we ensure that only U-U comparisons result in two points being scored?

Answer: YES! The adversary argument!

The adversary constructs a worst-case input by revealing as little as possible about the inputs.

Lower bounds for the MIN and MAX - contd

Adversary strategy:

U-U: any

U-W: make element of W winner

U-L: make element of L loser

U-N: any

W-W: any (be consistent with before)

W-L/N: make element of W winner

L-L: any (be consistent with before)

L-N: make element of L loser

Lower bounds for the MIN and MAX – contd.

We need to score $2n-2$ points. At most $n/2$ U-U comparisons can be made – gives n points.

To move $n-2$ elements to N , we need another $n-2$ comparisons.

Next: Linear sorting

Q: Can we beat the $\Omega(n \log n)$ lower bound for sorting?

A: In general no, but in some special cases
YES!

Ch 7: Sorting in linear time

Non-Comparison Sort – Bucket Sort

- Assumption: uniform distribution
 - Input numbers are **uniformly distributed** in $[0,1)$.
 - Suppose input size is n .
 - Idea:
 - Divide $[0,1)$ into n equal-sized subintervals (buckets).
 - Distribute n numbers into buckets
 - Expect that each bucket contains few numbers.
 - Sort numbers in each bucket (insertion sort as default).
 - Then go through buckets in order, listing elements
- Can be shown to run in linear-time on average

Example of BUCKET-SORT

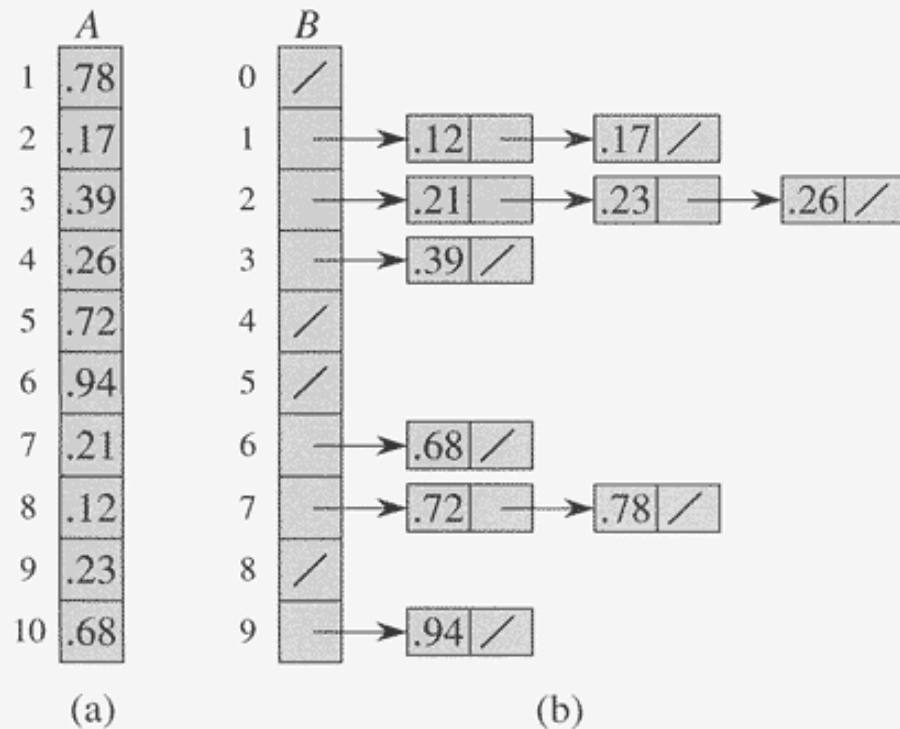


Figure 8.4 The operation of BUCKET-SORT. (a) The input array $A[1 \dots 10]$. (b) The array $B[0 \dots 9]$ of sorted lists (buckets) after line 5 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i + 1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \dots, B[9]$.

Bucket Sort - generalizations

- What if input numbers are NOT uniformly distributed?
- What if the distribution is not known a priori?

Non-Comparison Sort – Counting Sort

- Assumption: n input numbers are integers in the range $[0, k]$, $k = O(n)$.
- Idea:
 - Determine the number of elements less than x , for each input x .
 - Place x directly in its position.

Counting Sort - pseudocode

Counting-Sort(A, B, k)

- **for** $i \leftarrow 0$ **to** k
- do $C[i] \leftarrow 0$
- **for** $j \leftarrow 1$ **to** $\text{length}[A]$
- do $C[A[j]] \leftarrow C[A[j]] + 1$
- // $C[i]$ contains number of elements equal to i .
- **for** $i \leftarrow 1$ **to** k
- do $C[i] = C[i] + C[i-1]$
- // $C[i]$ contains number of elements $\leq i$.
- **for** $j \leftarrow \text{length}[A]$ **downto** 1
- do $B[C[A[j]]] \leftarrow A[j]$
- $C[A[j]] \leftarrow C[A[j]] - 1$

Counting Sort - example

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

(a)

	0	1	2	3	4	5
C	2	2	4	7	7	8

(b)

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

(c)

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

(d)

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

(e)

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

(f)

Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. (a) The array A and the auxiliary array C after line 4. (b) The array C after line 7. (c)–(e) The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. (f) The final sorted output array B .

Counting Sort - analysis

- | | | |
|-----|--|---|
| 1. | for $i \leftarrow 0$ to k | $\Theta(k)$ |
| 2. | do $C[i] \leftarrow 0$ | $\Theta(1)$ |
| 3. | for $j \leftarrow 1$ to $\text{length}[A]$ | $\Theta(n)$ |
| 4. | do $C[A[j]] \leftarrow C[A[j]] + 1$ | $\Theta(1)$ ($\Theta(1) \Theta(n) = \Theta(n)$) |
| 5. | // $C[i]$ contains number of elements equal to i . | $\Theta(0)$ |
| 6. | for $i \leftarrow 1$ to k | $\Theta(k)$ |
| 7. | do $C[i] = C[i] + C[i-1]$ | $\Theta(1)$ ($\Theta(1) \Theta(n) = \Theta(n)$) |
| 8. | // $C[i]$ contains number of elements $\leq i$. | $\Theta(0)$ |
| 9. | for $j \leftarrow \text{length}[A]$ downto 1 | $\Theta(n)$ |
| 10. | do $B[C[A[j]]] \leftarrow A[j]$ | $\Theta(1)$ ($\Theta(1) \Theta(n) = \Theta(n)$) |
| 11. | $C[A[j]] \leftarrow C[A[j]] - 1$ | $\Theta(1)$ ($\Theta(1) \Theta(n) = \Theta(n)$) |

Total cost is $\Theta(k+n)$, suppose $k=O(n)$, then total cost is $\Theta(n)$.

So, it beats the $\Omega(n \log n)$ lower bound!

Stable sort

- Preserves order of elements with the same key.
- Counting sort is stable.

Crucial question: can counting sort be used to sort large integers efficiently?

Radix sort

Radix-Sort(A,d)

- **for** $i \leftarrow 1$ **to** d
- **do** use a stable sort to sort A on digit i

Analysis:

Given n d -digit numbers where each digit takes on up to k values, Radix-Sort sorts these numbers correctly in $\Theta(d(n+k))$ time.

Radix sort - example

1019	2231	1019	1019	1019
3075	3075	2225	3075	2225
2225	2225	2231	2225	2231
2231	1019	3075	2231	3075

Sorted!

1019	1019
3075	2231
2231	2225
2225	3075

**Not
sorted!**

Next: Medians and Order Statistics (Ch. 9)

Order statistics: The i^{th} order statistic of n elements $S = \{a_1, a_2, \dots, a_n\}$: i^{th} smallest elements

- Minimum and maximum, Median
- finding the k^{th} largest element in an unsorted array.

Already seen:

1. $k=1$: $\Theta(n)$ algorithm optimal.
2. Also, Heapify + Extract-max: $\Theta(n)$ algorithm.
Same bounds hold for any constant k .
3. Sorting solves it for any k . $\Theta(n \log n)$ algorithm.

What about $k=n/2$? Can we do better than $\Theta(n \log n)$ algorithm?

Medians and Order Statistics

To select the i^{th} smallest element of $S=\{a_1, a_2, \dots, a_n\}$

- Can we use PARTITION?
 - if we are very lucky, we will get it in the first try!
 - otherwise we should have a smaller set to recurse on.
- No guarantee of being lucky!
How can we guarantee a significantly smaller set?

The algorithm is the most complicated divide-and-conquer algorithm in this course!

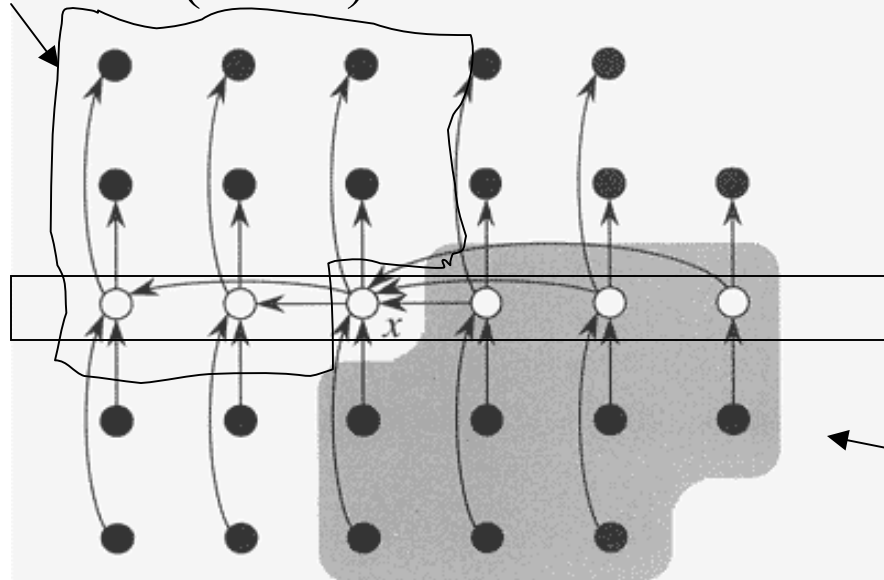
Order Statistics

1. Divide n elements into $\lceil n/5 \rceil$ groups of 5 elements.
2. Find the median of each group.
3. Use SELECT recursively to find the median x of the above $\lceil n/5 \rceil$ medians.
4. Partition using x as pivot, and find position k of x .
5. If $i=k$ return
else recurse on the appropriate subarray.

What kind of split does this produce?

The Way to Select x

At least $(3n/10)-6$ elements $< x$



Divide elements into $\lceil n/5 \rceil$ groups of 5 elements each.

Find the median of each group

Find the median of the medians

At least $(3n/10)-6$ elements $> x$

Figure 9.1 Analysis of the algorithm SELECT. The n elements are represented by small circles, and each group occupies a column. The medians of the groups are whitened, and the median-of-medians x is labeled. (When finding the median of an even number of elements, we use the lower median.) Arrows are drawn from larger elements to smaller, from which it can be seen that 3 out of every full group of 5 elements to the right of x are greater than x , and 3 out of every group of 5 elements to the left of x are less than x . The elements greater than x are shown on a shaded background.

Analysis of SELECT

- Steps 1,2,4 take $O(n)$,
- Step 3 takes $T(\lceil n/5 \rceil)$.
- Let us see step 5:
 - At least half of medians in step 2 are $\geq x$, thus at least $\lceil 1/2 \lceil n/5 \rceil \rceil - 2$ groups contribute 3 elements which are $\geq x$.
i.e, $3(\lceil 1/2 \lceil n/5 \rceil \rceil - 2) \geq (3n/10) - 6$.
 - Similarly, the number of elements $\leq x$ is also at least $(3n/10) - 6$.
 - Thus, $|S_1|$ is at most $(7n/10) + 6$, similarly for $|S_3|$.
 - Thus SELECT in step 5 is called recursively on at most $(7n/10) + 6$ elements.
- Recurrence is:
$$T(n) = \begin{cases} O(1) & \text{if } n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n) & \text{if } n \geq 140 \end{cases}$$

Solve recurrence by substitution

- Suppose $T(n) \leq cn$, for some c .
 - $$\begin{aligned} T(n) &\leq c \lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\leq cn/5 + c + 7/10cn + 6c + an \\ &= 9/10cn + an + 7c \\ &= cn + (-cn/10 + an + 7c) \end{aligned}$$
 - Which is at most cn if $-cn/10 + an + 7c < 0$.
 - i.e., $c \geq 10a(n/(n-70))$ when $n > 70$.
 - So select $n=140$, and then $c \geq 20a$.
- Note: n may not be 140, any integer > 70 is OK.

Implication for Quicksort

- Worst case improves to $O(n \log n)$
BUT...

Test your understanding

1. Problem 9.3-7: Describe an $O(n)$ algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S .
2. Problem 9.3-8: Let $X[1..n]$, $Y[1..n]$ be two sorted arrays. Give an $O(\lg n)$ algorithm to find the median of all $2n$ elements in arrays X, Y .

Next: dynamic programming

(no it's not a new programming language or paradigm!)