Problem Solving

2-9 Sorting and Selection

MA Jun

Institute of Computer Software

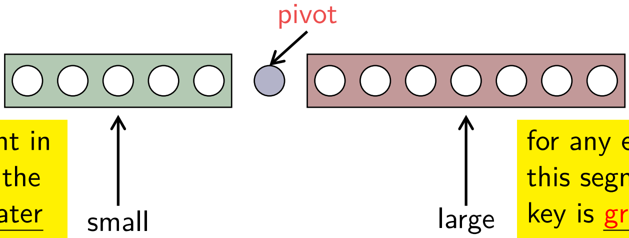
April 25, 2020

Contents

Contents

Quicksort

Question : What is the **KEY** idea of Quicksort?



for any element in this segment, the key is not greater than pivot.

for any element in this segment, the key is greater than pivot.

To Be Sorted Recursively

Quicksort

Question : What are the **SIMILARITIES** and **DIFFERENCES** between Quicksort and Mergesort?

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q − 1)
    QUICKSORT(A, q + 1, r)
```

VS

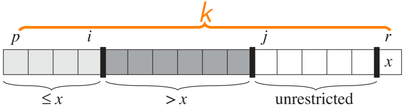
```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Similarity: both are **divide-and-conquer** strategies.

Difference: the process

	QuickSort	**MergeSort**
Partition	hard	easy
Combination	easy	hard

Quicksort: PARTITION



PARTITION(A, p, r)

```
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Question : How to prove the correctness of PARTITION?

At the beginning of each iteration of the loop of lines 3-6, for any array index k, we have:

1. If $p \le k \le i$, then $A[k] \le x$.
2. If $i + 1 \le k \le j - 1$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

Quicksort: Time Complexity

Question : What is the time complexity of QUICKSORT?

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

The recurrence: $T(n) = T(n_1) + T(n_2) + cn$

where:

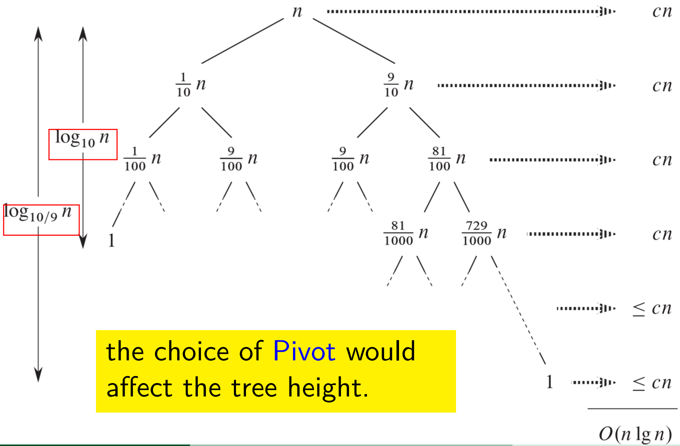
$$n_1 = q - 1 - p + 1 = q - p$$
$$n_2 = r - (q + 1) + 1 = r - q$$
$$n_1 + n_2 = r - p$$
$$\textit{initially}, p = 1, r = n$$

n_1, n_2 vary and depend on $q = \text{PARTITION}(A, p, r)$

Quicksort: Time Complexity

Question : Which factor would affect the efficiency of QUICKSORT?

always produces a 9-to-1 split

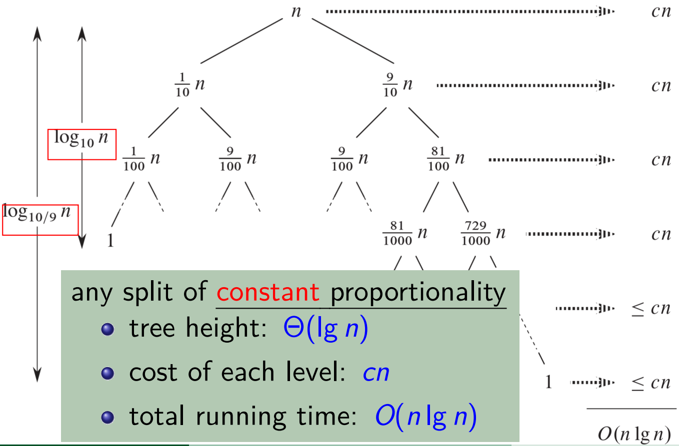


the choice of Pivot would affect the tree height.

Quicksort: Time Complexity

Question : Which factor would affect the efficiency of QUICKSORT?

always produces a 9-to-1 split

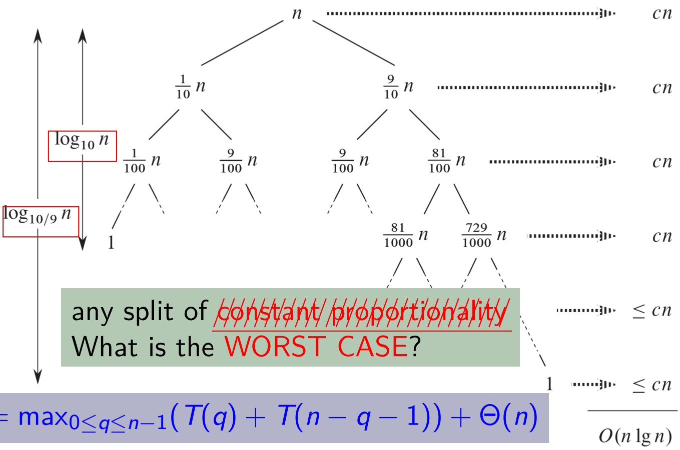


any split of <u>constant proportionality</u>

- tree height: $\Theta(\lg n)$
- cost of each level: cn
- total running time: $O(n \lg n)$

Quicksort: Time Complexity

Question : Which factor would affect the efficiency of QUICKSORT?



any split of ~~constant proportionality~~
What is the WORST CASE?

$$T(n) = \max_{0 \leq q \leq n-1}(T(q) + T(n-q-1)) + \Theta(n)$$

Quicksort: Time Complexity

Worst Case:

$$T(n) = \max_{0 \leq q \leq n-1}(T(q) + T(n-q-1)) + \Theta(n)$$

Question : When would the worst case happen?

The pivot is always the greatest or smallest element for each recursion.

Unlucky: $T(n) = O(n^2)$ for the worst case!

Lucky: worst case seldom happens!

Quicksort: Time Complexity

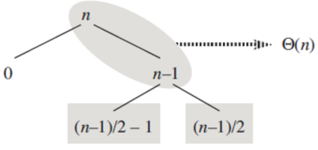
Impression & Intuition:

Quick sort performs quite well in practice.

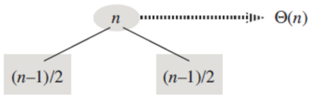
We usually obtain an $O(n \lg n)$ execution in most cases, rather than the worst case.

WHY?

PARTITION produces a mix of “good” and “bad” splits.



(a)



(b)

$$T(n) = O(n \lg n)$$

Quicksort: Time Complexity

Critical operation?

- The key cost of Quicksort comes from PARTITION
- The key cost of PARTITION comes from line 4.

```
QUICKSORT(A, p, r)
if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p - 1
3  for j = p to r - 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Quicksort

Lemma (7.1)

Let X be the number of comparisons performed in line 4 of PARTITION *over the entire execution of* QUICKSORT *on an n-element array. Then the running time of* QUICKSORT *is $O(n + X)$.*

Proof.

By the discussion above, the algorithm makes at most n calls to PARTITION, each of which does a constant amount of work and then executes the **for loop** some number of times. Each iteration of the **for loop** executes line 4. □

Randomized Quicksort

Randomized Quicksort

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Goal:

To compute X, the **TOTAL** number of comparisons performed in **all** calls to PARTITION.

We will **NOT** attempt to analyze how many comparisons are made in **EACH** call to PARTITION.

Randomized Partition

```
RANDOMIZED-PARTITION(A, p, r)
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Randomized Quicksort: Expected Running Time

Question : How to compute the expected value of X?

X: the **TOTAL** number of comparisons performed in **all** calls to PARTITION.

- We must understand **when the algorithm compares two elements of the array and when it does not**.
- For ease of analysis, we rename the elements of the array A as $\{z_1, z_2, ..., z_n\}$, with z_i being the ith smallest element.
- $Z_{ij} = \{z_i, z_{i+1}, ..., z_j\}$: the set of elements between z_i and z_j, inclusive.

Randomized Quicksort: Expected Running Time

Question : When does the algorithm compare z_i and z_j?

- Each pair of elements is compared **at most once**
- Elements are compared **only to the pivot element**
- After a particular call of PARTITION finishes, **the pivot element** used in that call is **never again** compared to any other elements.

X_{ij}: indicator random variables

$$\boxed{X_{ij} = I\{z_i \text{ is compared to } z_j\}}$$

Then, we have:

$$\boxed{X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}}$$

Randomized Quicksort: Expected Running Time

Question : How to compute the expected value of X?

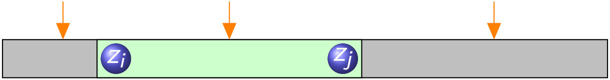
$$\boxed{E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right]}$$

$$\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n)
\end{aligned}$$

Randomized Quicksort: Expected Running Time

Question : What is $Pr\{z_i \text{ is compared to } z_j\}$?

$$
\begin{aligned}
\Pr\{z_i \text{ is compared to } z_j\} &= \Pr\{z_i \text{ or } z_j \text{ is } \boxed{\text{first pivot}} \text{ chosen from } Z_{ij}\} \\
&= \Pr\{z_i \text{ is } \boxed{\text{first pivot}} \text{ chosen from } Z_{ij}\} \\
&\quad + \Pr\{z_j \text{ is first pivot chosen from } Z_{ij}\} \\
&= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&= \frac{2}{j-i+1} .
\end{aligned}
$$



Top 10 Algorithms

The 10 Algorithms with the Greatest Influence on the Development and Practice of Science and Engineering in the 20th Century

- Metropolis Algorithm for Monte Carlo
- Simplex Method for Linear Programming
- Krylov Subspace Iteration Methods
- The Decompositional Approach to Matrix Computations
- The Fortran Optimizing Compiler
- QR Algorithm for Computing Eigenvalues
- **Quicksort Algorithm for Sorting**
- Fast Fourier Transform
- Integer Relation Detection
- Fast Multipole Method

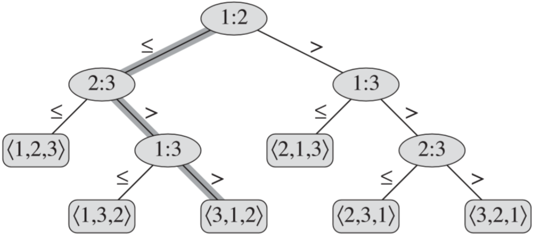
Sorting is a central problem in many areas of computing so it is no surprise to see an approach to solving the problem as one of the top 10. Joseph JaJa describes Quicksort as one of the best practical sorting algorithm for general inputs. In addition, its complexity analysis and its structure have been a rich source of inspiration for developing general algorithm techniques for various applications.

Jack Dongarra and Francis Sullivan. 2000. Guest Editors' Introduction: The Top 10 Algorithms. Computing in Science and Engg. 2, 1 (January 2000), 22–23. DOI:https://doi.org/10.1109/MCISE.2000.814652

Comparison-based Sort Algorithm

Theorem (8.1)

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



- $n!$ reachable leaves, each of which corresponds to a possible permutation
- h: the height of the decision (binary) tree
- $n! \leq 2^h \Longrightarrow h \geq \lg n! = \Omega(n \lg n)$

Sorting in Linear Time

- Counting Sort
- Radix Sort
- Bucket Sort

Sorting in Linear Time: Counting Sort

Assumption

Each of the input elements is an integer in the range 0 to k.

$$T(n) = \Theta(n + k), \text{ and if } k = O(n),\ T(n) = \Theta(n).$$

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i − 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] − 1
```

Stable

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

stable: numbers with the **same value** appear in the output array **in the same order** as they do in the input array.

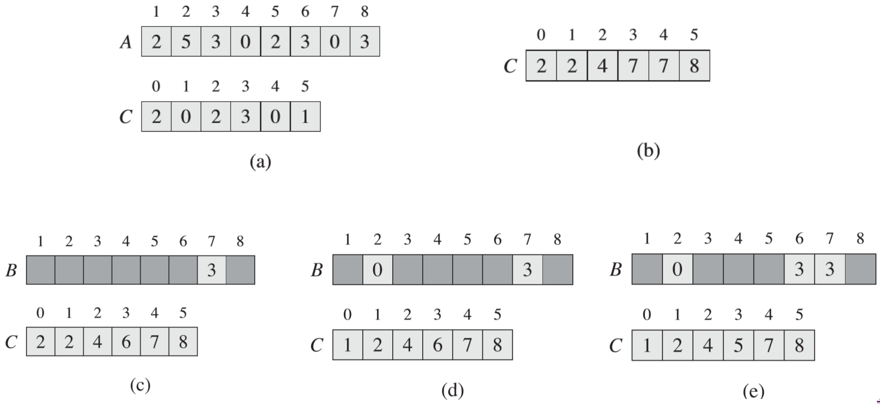
(a)

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

(b)

Sorting in Linear Time: Counting Sort

```
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```



Sorting in Linear Time: Radix Sort

Assumption

- Each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit.
- Each digit can take on up to k possible values

```
RADIX-SORT(A, d)
for i = 1 to d
    use a stable sort to sort array A on digit i
```

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

Sorting in Linear Time: Radix Sort

Lemma (8.3)

Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT *correctly sorts these numbers in* $\Theta(d(n+k))$ *time if the* ***stable sort*** *it uses takes* $\Theta(n+k)$ *time.*

Lemma (8.4)

Given n b-bit numbers and any positive integer $r \leq b$*,* RADIX-SORT *correctly sorts these numbers in* $\Theta((b/r)(n+2^r))$ *time if the* ***stable sort*** *it uses takes* $\Theta(n+k)$ *time for inputs in the range* 0 *to k.*

Proof For a value $r \leq b$, we view each key as having $d = \lceil b/r \rceil$ digits of r bits each. Each digit is an integer in the range 0 to $2^r - 1$, so that we can use counting sort with $k = 2^r - 1$. (For example, we can view a 32-bit word as having four 8-bit digits, so that $b = 32$, $r = 8$, $k = 2^r - 1 = 255$, and $d = b/r = 4$.) Each pass of counting sort takes time $\Theta(n+k) = \Theta(n+2^r)$ and there are d passes, for a total running time of $\Theta(d(n+2^r)) = \Theta((b/r)(n+2^r))$. ■

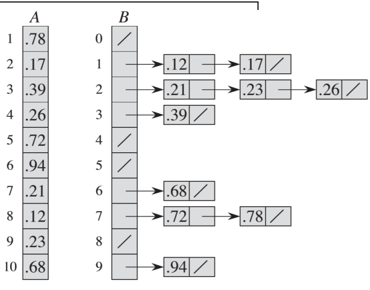
Sorting in Linear Time: Bucket Sort

Assumption

The input is drawn from a **uniform distribution**

BUCKET-SORT(A)

```
let B[0..n-1] be a new array
n = A.length
for i = 0 to n-1
    make B[i] an empty list
for i = 1 to n
    insert A[i] into list B[⌊n A[i]⌋]
for i = 0 to n-1
    sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n-1] together in order
```



(a) (b)

Sorting in Linear Time: Bucket Sort

BUCKET-SORT(A)

1 let $B[0\,..\,n-1]$ be a new array
2 $n = A.length$
3 **for** $i = 0$ **to** $n - 1$
4   make $B[i]$ an empty list
5 **for** $i = 1$ **to** n
6   insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
7 **for** $i = 0$ **to** $n - 1$
8   sort list $B[i]$ with $\boxed{\text{insertion sort}}$ $O(n_i^2)$
9 concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

- All lines except line 8 take $O(n)$ time in the worst case.
- n_i: the number of elements placed in bucket $B[i]$.

Sorting in Linear Time: Bucket Sort

$$
\begin{aligned}
E\left[T(n)\right] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} E\left[O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} O(E\left[n_i^2\right]) \\
&= \Theta(n)
\end{aligned}
$$

$X_{ij} = \mathrm{I}\{A[j] \text{ falls in bucket } i\}$

for $i = 0, 1, \ldots, n-1$ and $j = 1, 2, \ldots, n$. Thus,

$$n_i = \sum_{j=1}^{n} X_{ij} .$$

$\rightarrow$ $\boxed{\sum_{i=0}^{n-1} O(E\left[n_i^2\right]) = 2 - \frac{1}{n}}$

Contents

Minimum and Maximum

Problem (Minimum or Maximum)

Given a subset of a total-order set, find the maximum ***or*** *minimum element of the subset.*

- *requires* ***at least*** $n-1$ *comparisons*

```
MINIMUM(A)
min = A[1]
for i = 2 to A.length
    if min > A[i]
        min = A[i]
return min
```

Minimum and Maximum

Problem (Maximum & minimum)

*Given a subset of a total-order set, find both the maximum **and** minimum elements of the subset.*

- *does not require $2n - 2$ comparisons*

A possible way for finding both maximum & minimum.

- compare pairs of elements from the input first with each other
- then compare the smaller with the current minimum and the larger to the current maximum
- at most $3\lfloor n/2 \rfloor$ comparisons

□

General Selection Problem

Problem (General Selection)

Given a subset of a total-order set, find the i-th smallest element of the subset.

Selection in Expected Linear Time : RANDOMIZED-SELECT

```
RANDOMIZED-QUICKSORT(A, p, r)
if p < r
    q = RANDOMIZED-PARTITION(A, p, r)
    RANDOMIZED-QUICKSORT(A, p, q − 1)
    RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q − p + 1
if i == k          // the pivot value is the answer
    return A[q]
elseif i < k
    return RANDOMIZED-SELECT(A, p, q − 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Similar to RANDOMIZED-QUICKSORT, but only have to handle exact one sub-problem in each step of the recursion.

RANDOMIZED-SELECT: Expected Running Time

Question : What is the expected running time of RANDOMIZED-SELECT?

indicator random variable X_k :

- $X_k =$ $I\{\text{the subarray } A[p..q] \text{ has exactly } k \text{ elements}\}$
- assuming the elements are distinct, we have $E[X_k] = 1/n$

```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]
q = RANDOMIZED-PARTITION(A, p, r)
k = q - p + 1
if i == k            // the pivot value is the answer
    return A[q]
elseif i < k
    return RANDOMIZED-SELECT(A, p, q - 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```

$T(n)$: the running time on an input array of size n

$$\begin{aligned} T(n) &\boxed{\le} \sum_{k=1}^{n} X_k \cdot (\boxed{T(\max(k-1, n-k))} + O(n)) \\ &= \sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n) . \end{aligned}$$

RANDOMIZED-SELECT: Expected Running Time

$E[T(n)]$: the expected running time on an input array of size n

$$
\begin{aligned}
& \mathrm{E}[T(n)] \\
& \leq \mathrm{E}\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] \\
& = \sum_{k=1}^{n} \mathrm{E}[X_k \cdot T(\max(k-1, n-k))] + O(n) \qquad \text{(by linearity of expectation)} \\
& = \sum_{k=1}^{n} \mathrm{E}[X_k] \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) \qquad \text{(by equation (C.24))} \\
& = \sum_{k=1}^{n} \frac{1}{n} \cdot \mathrm{E}[T(\max(k-1, n-k))] + O(n) \qquad \text{(by equation (9.1))} \ .
\end{aligned}
$$

Then, we could prove $E[T(n)] = O(n)$ by substitution. Assuming:

$$E[T(n)] \leq cn$$

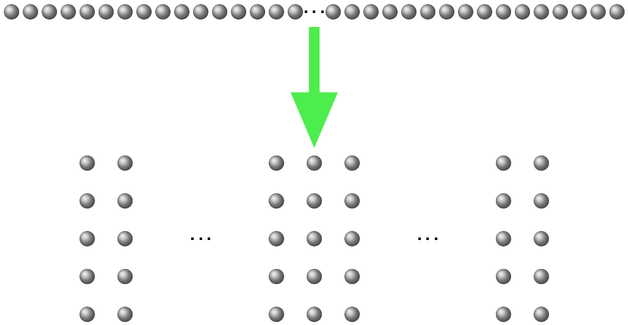
Selection in Expected Linear Time: SELECT

SELECT

1. Divide the input array into $\lceil n/5 \rceil$ groups of 5 elements each
   - at most one group made up of the remaining n mod 5 elements.
2. Find the **median** of each of the $\lceil n/5 \rceil$ groups with **insertion-sort**.
3. Use SELECT recursively to find the **median** m^* of the medians found in step 2.
4. **Partition** the input array around the **median-of-medians** m^*.
5. Assume that m^* is the kth smallest element. If $i = k$, then return m^*. Otherwise, use SELECT recursively:
   - if $i < k$, find the ith smallest element on the low side
   - if $i > k$, find the $(i - k)$th smallest element on the high side

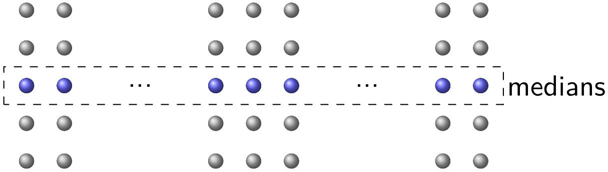
Select

Step 1: Divide the input array into $\lceil n/5 \rceil$ groups of 5 elements each



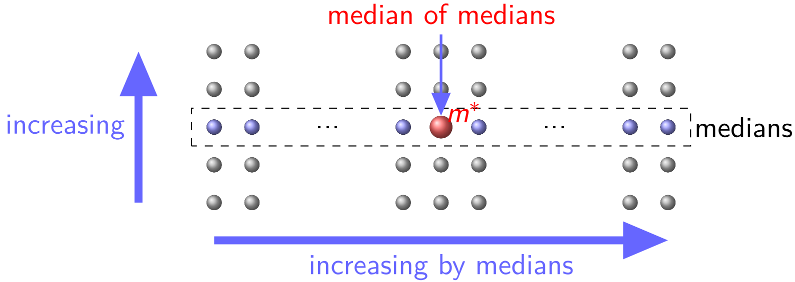
SELECT

Step 2: Find the **median** of each of the $\lceil n/5 \rceil$ groups with INSERTION-SORT.



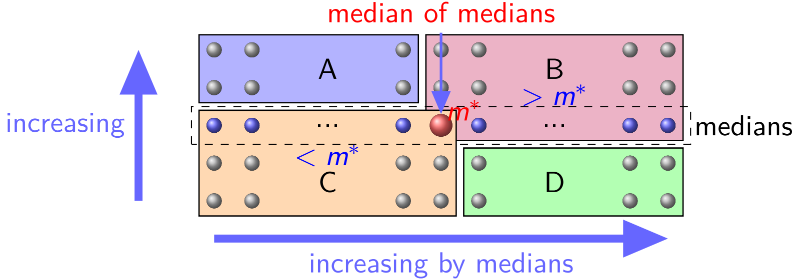
Select

Step 3: Use SELECT recursively to find the **median** m^* of the medians found in step 2.



Select

Step 4: Partition the input array around m^*.

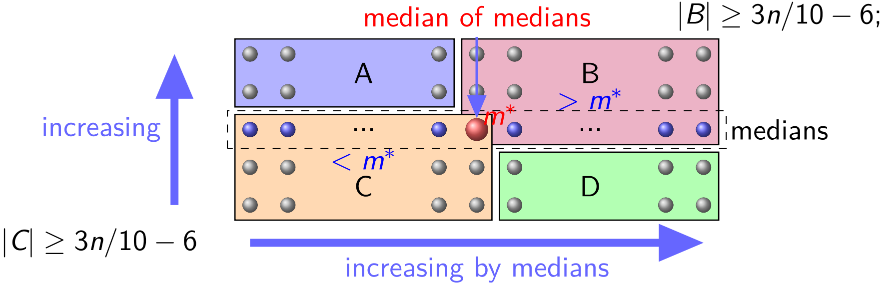


$> m^*$ or $< m^*$ are unknown only for elements in A and D

SELECT

Step 5: Assume that m^* is the kth smallest element.

- If $i = k$, then return m^*.
- Otherwise, use SELECT recursively:
  - if $i < k$, find the ith smallest element on the low side
  - if $i > k$, find the $(i - k)$th smallest element on the high side



calls SELECT recursively on at most $7n/10 + 6$ elements.

The SELECT algorithm: Running Time in Worst-case

Counting the total number of comparisons

$$T(n) \leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + O(n)$$

- $T(\lceil n/5 \rceil)$: find the median of the medians
- $T(7n/10 + 6)$: maximum cost for calling SELECT recursively.
- $O(n)$:
  - divide the input array into 5-elements groups
  - find medians of all 5-elements groups, about $6 * \lceil n/5 \rceil$
  - PARTITION with the pivot m^*

We could show that the running time $T(n) = O(n)$ by substitution

Thank You!
Questions?

Office 819
majun@nju.edu.cn