2IL50 Data Structures

2023-24 Q3

Lecture 5: QuickSort & Selection

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

QuickSort

One more sorting algorithm ...

Sorting algorithms

Input: a sequence of n numbers $A = \langle a_1, a_2, \dots, a_n \rangle$
Output: a permutation of the input such that $\langle a_{i1} \leq a_{i2} \leq \cdots \leq a_{in} \rangle$

Important properties of sorting algorithms:

running time: how fast is the algorithm in the worst case

in place: only a constant number of input elements are ever stored outside the input array

	worst case running time	in place
InsertionSort	$\Theta(n^2)$	yes
MergeSort	$\Theta(n \log n)$	no
HeapSort	$\Theta(n \log n)$	yes
QuickSort	$\Theta(n^2)$	yes

QuickSort

	worst case running time	in place
InsertionSort	$\Theta(n^2)$	yes
MergeSort	$\Theta(n \log n)$	no
HeapSort	$\Theta(n \log n)$	yes
QuickSort	$\Theta(n^2)$	yes

Why QuickSort?

1. Expected running time: $\Theta(n \log n)$ *(randomized QuickSort)*
2. Constants hidden in $\Theta(n \log n)$ are small
3. *Using linear time median finding to guarantee good pivot gives worst case* $\Theta(n \log n)$

QuickSort

QuickSort is a divide-and-conquer algorithm

To sort the subarray $A[p:r]$:

Divide

Partition $A[p:r]$ into two subarrays $A[p:q-1]$ and $A[q+1:r]$, such that each element in $A[p:q-1]$ is $\leq$ $A[q]$ and $A[q]$ is $<$ each element in $A[q+1:r]$.

Conquer

Sort the two subarrays by recursive calls to QuickSort.

Combine

No work is needed to combine the subarrays, since they are sorted in place.

Divide using a procedure Partition which returns q.

QuickSort

QuickSort(A, p, r)

if $p < r$
 $q = $ Partition(A, p, r)
 QuickSort$(A, p, q - 1)$
 QuickSort$(A, q + 1, r)$

Initial call: QuickSort$(A, 1, n)$

Partition(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
 if $A[j] \leq x$
  $i = i + 1$
  exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

Partition always selects $A[r]$ as the pivot
(the element around which to partition)

Partition

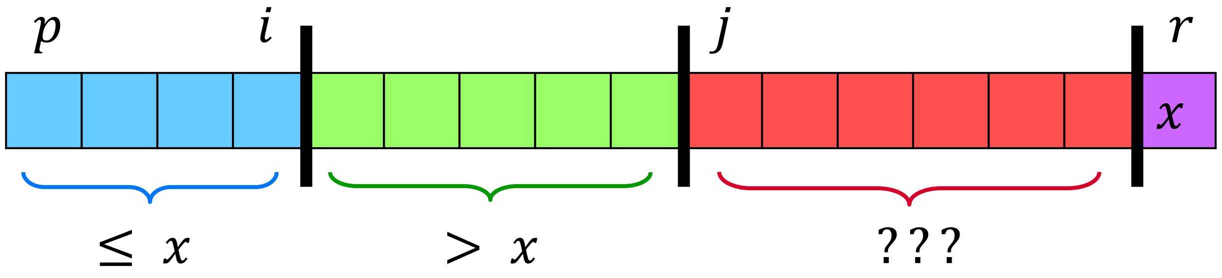
As Partition executes, the array is partitioned into four regions (some may be empty)

Partition(A, p, r)

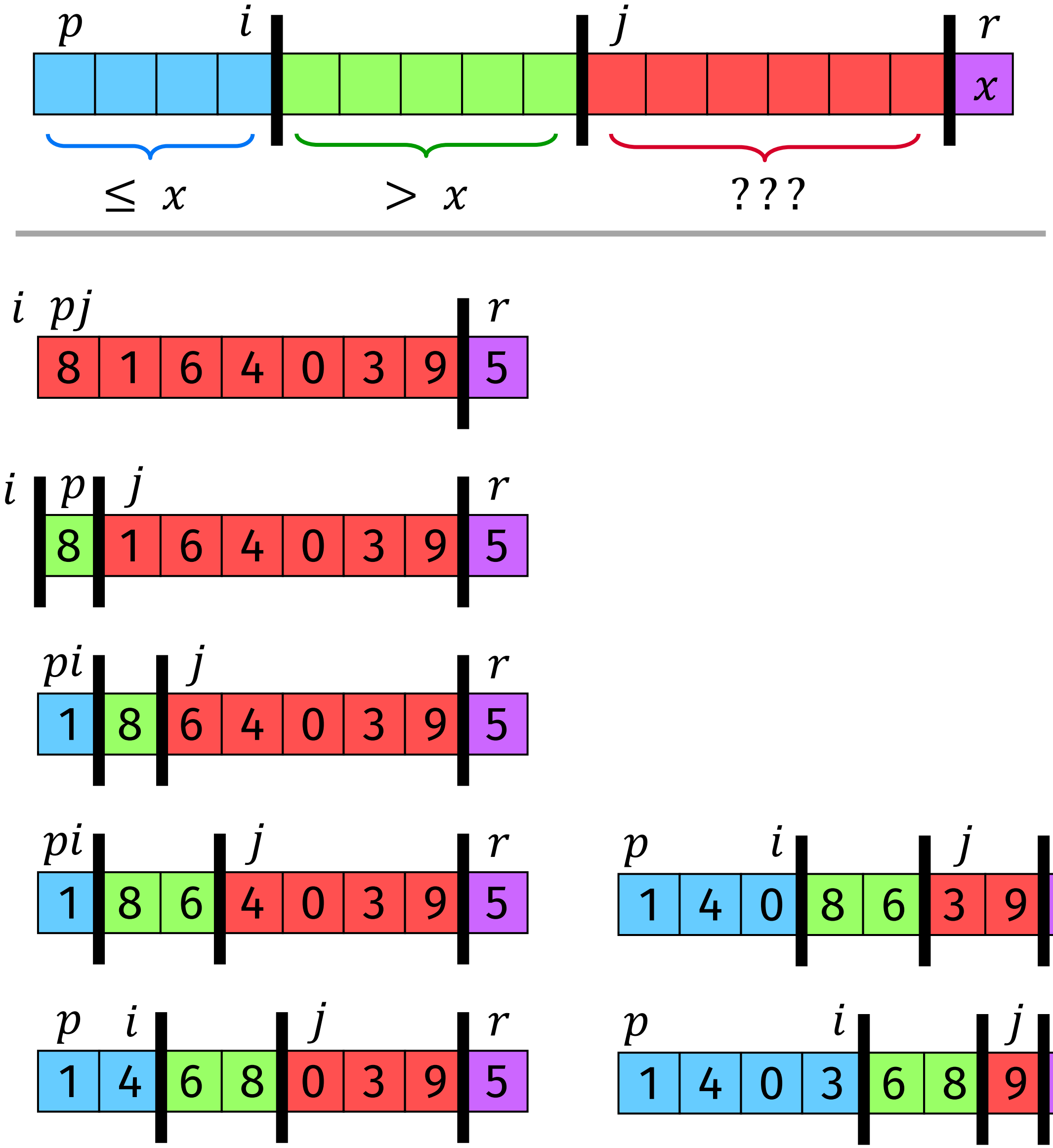
$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

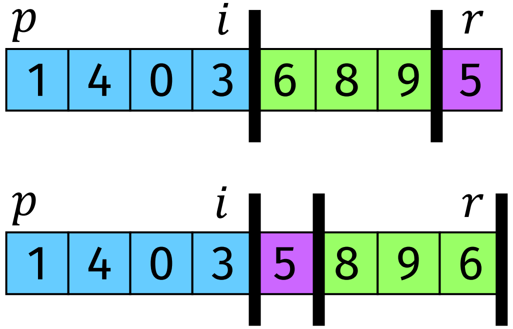
Loop invariant

1. all entries in $A[p\!:i]$ are $\leq$ pivot
2. all entries in $A[i + 1\!:j - 1]$ are $>$ pivot
3. $A[r]\ =$ pivot



Partition





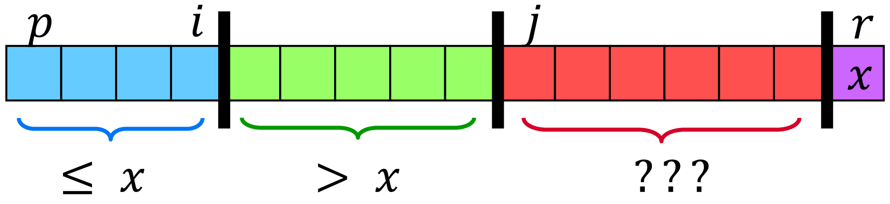
Partition(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
    if $A[j] \le x$
        $i = i + 1$
        exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

Partition - Correctness

Loop invariant

1. all entries in $A[p:i]$ are $\leq$ pivot
2. all entries in $A[i+1:j-1]$ are $>$ pivot
3. $A[r] = \text{pivot}$

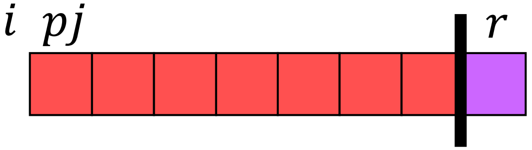


Partition(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i] \leftrightarrow A[j]$
exchange $A[i+1] \leftrightarrow A[r]$
return $i + 1$

Initialization

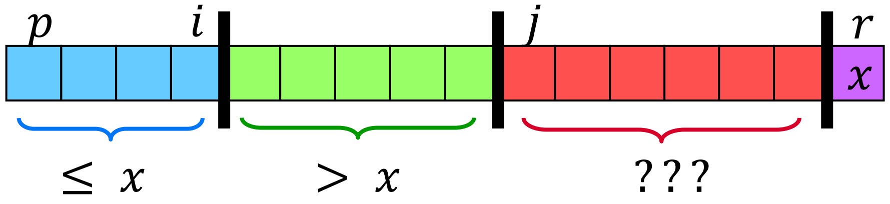
before the loop starts, all conditions are satisfied, since r is the pivot and the two subarrays $A[p:i]$ and $A[i+1:j-1]$ are empty



Partition - Correctness

Loop invariant

1. all entries in $A[p\colon i]$ are $\leq \mathrm{pivot}$
2. all entries in $A[i+1\colon j-1]$ are $> \mathrm{pivot}$
3. $A[r] = \mathrm{pivot}$

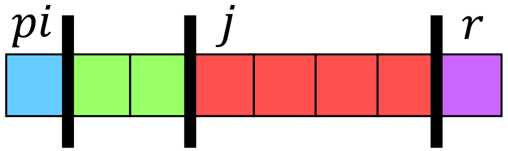



```
Partition(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
        if A[j] ≤ x
              i = i + 1
              exchange A[i] ↔ A[j]
  exchange A[i + 1] ↔ A[r]
  return i + 1
```

Maintenance

while the loop is running, if $A[j] \leq \mathrm{pivot}$, then $A[j]$ and $A[i+1]$ are swapped and then i and j are incremented ➡ 1. and 2. hold.

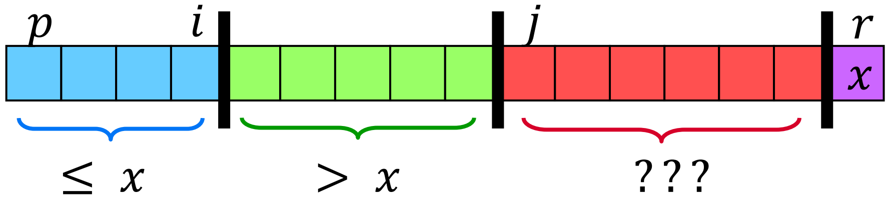
If $A[j] > \mathrm{pivot}$, then increment only j ➡ 1. and 2. hold.



Partition - Correctness

Loop invariant

1. all entries in $A[p\!:\!i]$ are $\leq \text{pivot}$
2. all entries in $A[i+1\!:\!j-1]$ are $> \text{pivot}$
3. $A[r] = \text{pivot}$



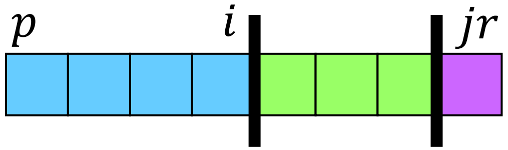

```
Partition(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

Termination

When the loop terminates, $j = r$, by LI all elements in A are partitioned into one of three cases:

$A[p\!:\!i] \leq \text{pivot}$, $A[i+1\!:\!r-1] > \text{pivot}$, and $A[r] = \text{pivot}$

Lines 7 and 8 move the pivot between the two subarrays



Running time: $\Theta(n)$ for an n-element subarray

QuickSort: running time

```
QuickSort(A, p, r)
if p < r
        q = Partition(A, p, r)
        QuickSort(A, p, q − 1)
        QuickSort(A, q + 1, r)
```

Running time depends on partitioning of subarrays:

- if they are balanced, then QuickSort is as fast as MergeSort
- if they are unbalanced, then QuickSort can be as slow as InsertionSort

Worst case

- subarrays completely unbalanced: 0 elements in one, $n-1$ in the other
- $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n) = \Theta(n^2)$
- input: sorted array

QuickSort: running time

```
QuickSort(A, p, r)
if p < r
    q = Partition(A, p, r)
    QuickSort(A, p, q − 1)
    QuickSort(A, q + 1, r)
```

Running time depends on partitioning of subarrays:

- if they are balanced, then QuickSort is as fast as MergeSort
- if they are unbalanced, then QuickSort can be as slow as InsertionSort

Best case

- subarrays completely balanced: each has $\leq n/2$ elements
- $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$

Average?

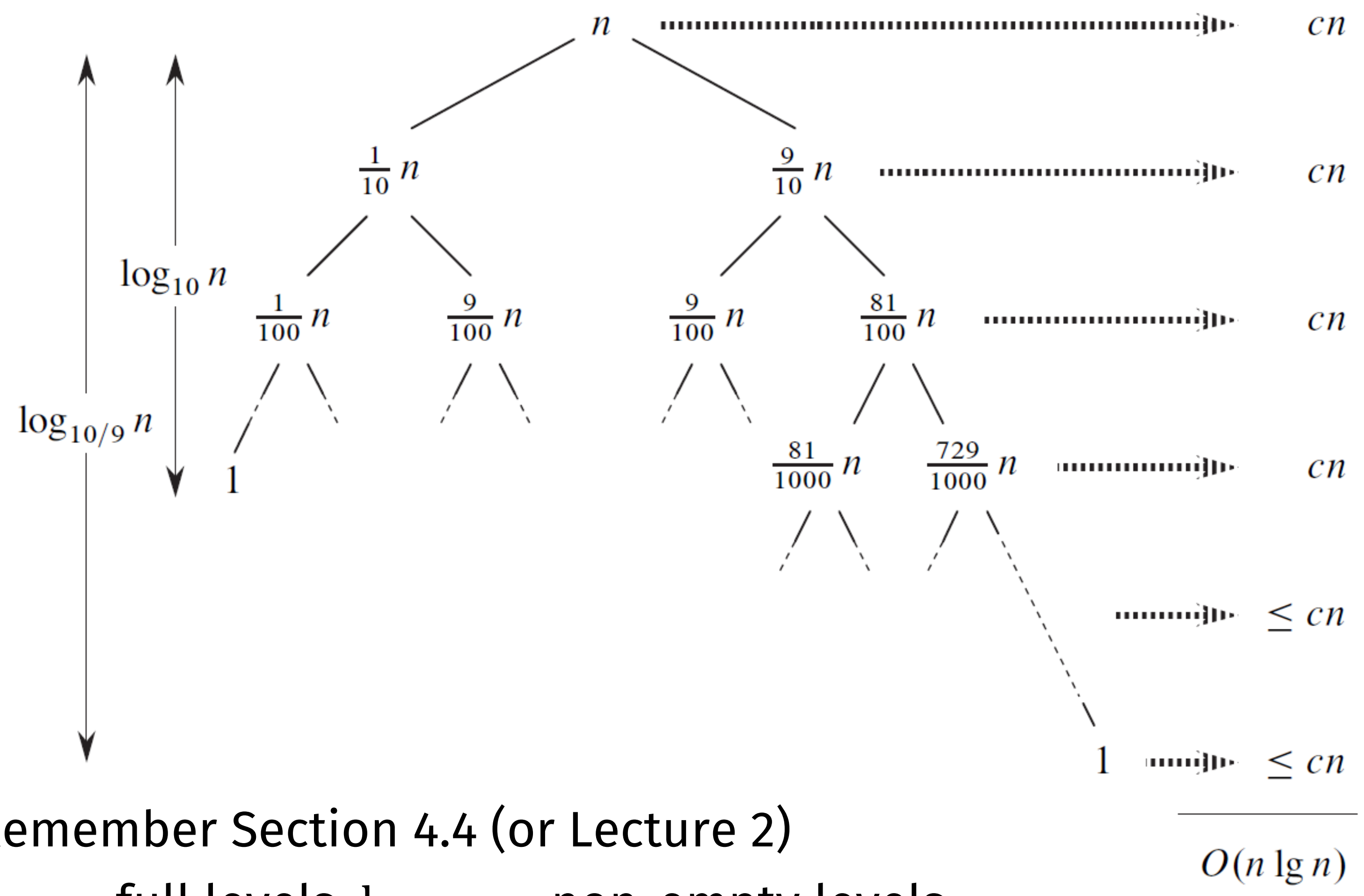
QuickSort: running time

Average running time is much closer to best case than to worst case.

Intuition

- imagine that Partition always produces a 9-to-1 split
- $T(n) = T(9n/10) + T(n/10) + \Theta(n)$

T(n) = T(9n/10) + T(n/10) + Θ(n)



Remember Section 4.4 (or Lecture 2)

$\log_{10} n$ full levels, $\log_{10/9} n$ non-empty levels

base of log does not matter in asymptotic notation
(as long as it is constant)

QuickSort: running time

Average running time is much closer to best case than to worst case.

Intuition

- imagine that Partition always produces a 9-to-1 split
- $T(n) = T(9n/10) + T(n/10) + \Theta(n)$
  $= \Theta(n \log n)$

Any split of constant proportionality yields a recursion tree of depth $\Theta(\log n)$

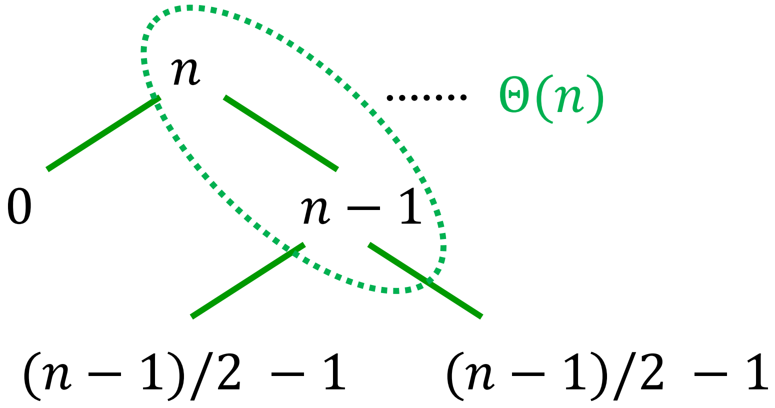
But splits will not always be constant,
there will be a mix of good and bad splits ...

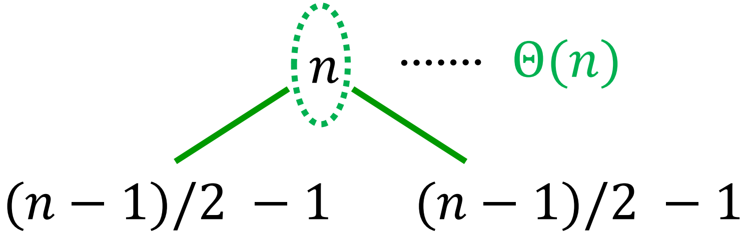
QuickSort: running time

Average running time is much closer to best case than to worst case.

More intuition …

- mixing good and bad splits does not affect the asymptotic running time
- assume levels alternate between best-case and worst-case splits





- extra levels add only to hidden constant, in both cases $O(n \log n)$

Randomized QuickSort

pick pivot at random

RandomizedPartition(A, p, r)

```
i = Random(p, r)
exchange A[r] ↔ A[i]
return Partition(A, p, r)
```

random pivot results in reasonably balanced split on average
expected running time $\Theta(n \log n)$

see book for detailed analysis
BCS 1st year: this analysis will be covered in Probability & Statistics in March

alternative: use linear time median finding to find a good pivot
worst case running time $\Theta(n \log n)$

price to pay: added complexity

Selection

Medians and Order Statistics

Definitions

i^{th} order statistic: i^{th} smallest of a set of n elements

minimum: 1^{st} order statistic

maximum: n^{th} order statistic

median: “halfway point”

- n odd unique median at $i = (n+1)/2$
- n even lower median at $i = n/2$, upper median at $i = n/2 + 1$

here: median means lower median

The selection problem

Input: a set A of n distinct numbers and a number i, with $1 \leq i \leq n$.
Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements in A.
(The i^{th} smallest element of A.)

Easy solution:

1. sort the input in $\Theta(n \log n)$ time
2. return the i^{th} element in the sorted array

This can be done faster … start with minimum and maximum

Minimum and maximum

Find the minimum with $n - 1$ comparisons:
examine each element in turn and keep track of the smallest one

Is this the best we can do? yes

Each element (except the minimum) must be compared to a smaller element at least once ...

Minimum(A, n)

$min = A[1]$
for $i = 2$ **to** n
    if $min > A[i]$
        $min = A[i]$
return min

Find maximum by replacing $>$ with $<$

Simultaneous minimum and maximum

Assume we need to find both the minimum and the maximum

Easy solution: find both separately

$2n - 2$ comparisons $\Theta(n)$ time

But only $3\lfloor n/2 \rfloor$ are needed …

- maintain the minimum and maximum seen so far
- Do not compare elements to the minimum and maximum separately, process them in pairs
- compare the elements of each pair to each other, then compare the largest to the maximum and the smallest to the minimum

3 comparisons for every 2 elements

The selection problem

Input: a set A of n distinct numbers and a number i, with $1 \leq i \leq n$.
Output: The element $x \in A$ that is larger than exactly $i - 1$ other elements in A.
(The i^{th} smallest element of A.)

Theorem
The i^{th} smallest element of A can be found in $O(n)$ time in the worst case.

Idea:

- partition the input array, recurse on one side of the split
- guarantee a good split
- use Partition with a designated pivot element

Selection in worst-case linear time

20	3	10	8	14	6	12	9	11	18	7	4	5	17	15	1	2	13
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$i = 12$

1. Divide the n elements into groups of 5 ➡ $\lceil n/5 \rceil$ groups

Selection in worst-case linear time

										x							
20	3	10	8	14	6	12	9	11	18	7	4	5	17	15	1	2	13
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$i = 12$

1. Divide the n elements into groups of 5 ➡ $\lceil n/5 \rceil$ groups
2. Find the median of each of the $\lceil n/5 \rceil$ groups
   (sort each group of 5 elements in constant time and simply pick the median)
3. Find the median x of the $\lceil n/5 \rceil$ medians recursively
4. Partition the array around x

Selection in worst-case linear time

x

20	3	10	8	14	6	12	9	11	18	7	4	5	17	15	1	2	13
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$i = 12$

1. Divide the n elements into groups of 5 ➡ $\lceil n/5 \rceil$ groups
2. Find the median of each of the $\lceil n/5 \rceil$ groups
   (sort each group of 5 elements in constant time and simply pick the median)
3. Find the median x of the $\lceil n/5 \rceil$ medians recursively
4. Partition the array around x ➡ x is the k^{th} element after partitioning

x

3	6	4	5	1	2	7	12	9	11	18	13	10	8	17	15	14	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$\underbrace{\qquad}_{k-1}$ $\underbrace{\qquad}_{n-k}$

Selection in worst-case linear time

										x							
20	3	10	8	14	6	12	9	11	18	7	4	5	17	15	1	2	13
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$i = 12$

1. Divide the n elements into groups of 5 ➡ $\lceil n/5 \rceil$ groups
2. Find the median of each of the $\lceil n/5 \rceil$ groups
   (sort each group of 5 elements in constant time and simply pick the median)
3. Find the median x of the $\lceil n/5 \rceil$ medians recursively
4. Partition the array around x ➡ x is the k^{th} element after partitioning

						x											
3	6	4	5	1	2	7	12	9	11	18	13	10	8	17	15	14	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$k-1$ $\qquad\qquad$ $n-k$

5. If $i = k$, return x. If $i < k$, recursively find the i^{th} smallest element on the low side. If $i > k$, recursively find the $(i-k)^{\text{th}}$ smallest element on the high side.

Selection in worst-case linear time

										x							
20	3	10	8	14	6	12	9	11	18	7	4	5	17	15	1	2	13
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$i = 12$

1. Divide the n elements into groups of 5 ➡ $\lceil n/5 \rceil$ groups
2. Find the median of each of the $\lceil n/5 \rceil$ groups
   (sort each group of 5 *elements in constant time and simply pick the median)*
3. Find the median x of the $\lceil n/5 \rceil$ medians recursively
4. Partition the array around x ➡ x is the k^{th} element after partitioning

3	6	4	5	1	2	7	12	9	11	18	13	10	8	17	15	14	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

$i = 5$

5. If $i = k$, return x. If $i < k$, recursively find the i^{th} smallest element on the low side. If $i > k$, recursively find the $(i - k)^{\text{th}}$ smallest element on the high side.

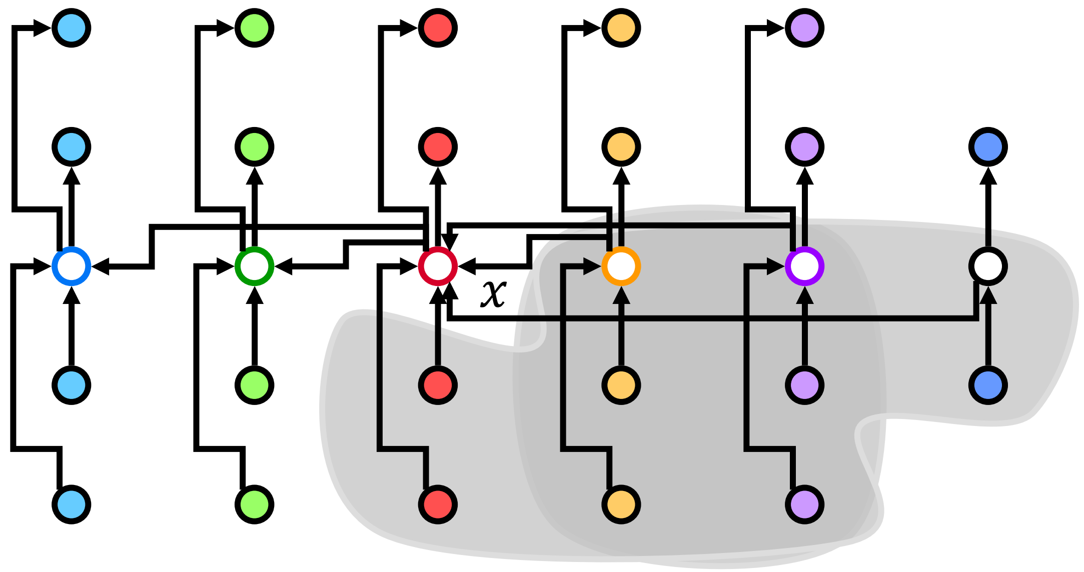
Analysis

3	6	4	5	1	2	7	12	9	11	18	13	10	8	17	15	14	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

How many elements are larger than x?

Analysis

How many elements are larger than x?



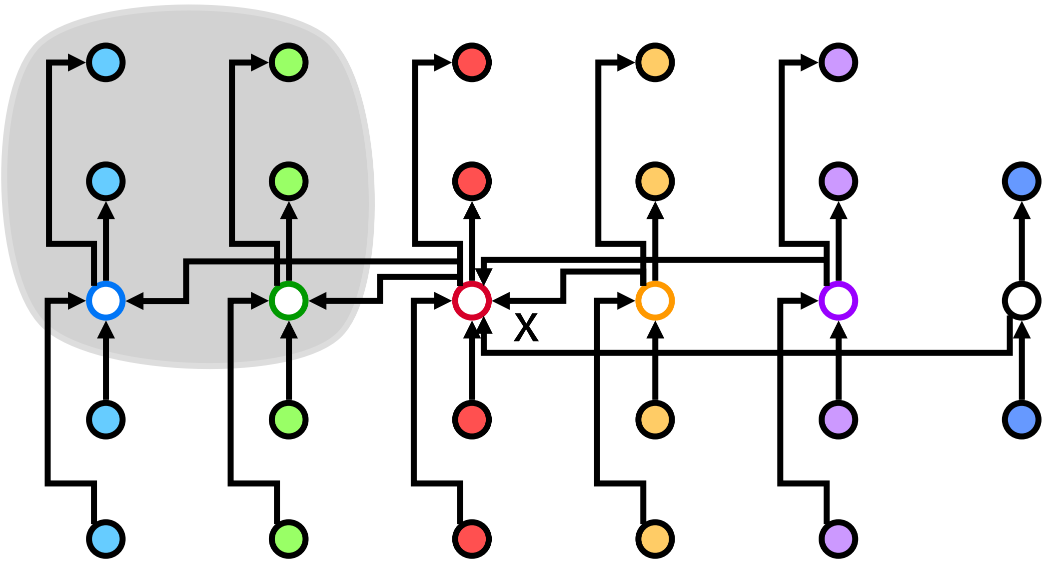
Half of the medians found in step 2 are $\geq x$

The groups of these medians contain 3 elements each which are bigger than x. (discounting x's group and the last group)

At least $3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$ elements are bigger than x.

Analysis

Symmetrically, at least $3n/10 - 6$ elements are smaller than x



the algorithm recurses on at most $7n/10 + 6$ elements.

Analysis

1. Divide the n elements into groups of 5 — $O(n)$
   $\lceil n/5 \rceil$ groups

2. Find the median of each of the $\lceil n/5 \rceil$ groups — $O(n)$
   (sort each grou
   simply pick the

3. Find the media

$$T(n) = \begin{cases} O(1) & \text{if } n < 140 \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 6\right) + O(n) & \text{if } n \geq 140 \end{cases}$$

4. Partition the array around x — $O(n)$

5. If $i = k$, return x. — $\leq T(7n/10 + 6)$
   If $i < k$, recursively find the i^{th} smallest element on the low side.
   If $i > k$, recursively find the $(i - k)^{\text{th}}$ smallest element on the high side.

Solving the recurrence

$$T(n) = \begin{cases} O(1) & \text{if } n < 140 \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 6\right) + O(n) & \text{if } n \geq 140 \end{cases}$$

Solve by substitution

Inductive hypothesis: $T(n) \leq cn$ for some constant c and all $n > 0$

- Assume that c is large enough such that $T(n) \leq cn$ for all $n < 140$
- Pick constant a such that the $O(n)$ term is $\leq an$ for all $n > 0$

$$\begin{aligned} T(n) &\leq T(\lceil n/5 \rceil) + T(7n/10 + 6) + an \\ &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \quad \text{(by IH)} \\ &\leq c(n/5 + 1) + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

Remains to show: $-cn/10 + 7c + an \leq 0$.

Solving the recurrence

$$T(n) = \begin{cases} O(1) & \text{if } n < 140 \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 6\right) + O(n) & \text{if } n \geq 140 \end{cases}$$

Remains to show: $-cn/10 + 7c + an \leq 0.$

$$\begin{aligned} -cn/10 + 7c + an &\leq 0 \\ cn/10 - 7c &\geq an \\ cn - 70c &\geq 10an \\ c(n-70) &\geq 10an \\ c &\geq 10a(n/(n-70)) \end{aligned}$$

for $n \geq 140$ we have $n/(n-70) \leq 2$,
particularly $20a \geq 10a(n/(n-70))$.

choose $c \geq 20a$

Why 140?
Any integer > 70 would have worked …

Selection

Theorem

The i^{th} smallest element of A can be found in $O(n)$ time in the worst case.

Does not require any assumptions on the input

Is not in conflict with the $\Omega(n \log n)$ lower bound for sorting, as it does not use sorting.

Randomized Selection: pick a pivot at random

Theorem

The i^{th} smallest element of A can be found in $O(n)$ expected time.

Using median finding

Median can be used to make efficient algorithms (worst case)

Divide and Conquer

- Use median to divide in two equal halves
- Partition with median as pivot   avoids sorting
- Running time: $T(n) = 2T(n/2) + \Theta(n)$ ➡ $T(n) = \Theta(n \log n)$
- Quicksort ➡ $T(n) = \Theta(n \log n)$

Pruning

- Compute median
- Partition with median as pivot
- Determine if answer is in left or right half
- Running time: $T(n) = T(n/2) + \Theta(n)$ ➡ $T(n) = \Theta(n)$