# UMC 201: Data Structures & Algorithms

## Lecture 6: Medians and Order Statistics

Indian Institute of Science

# Recap

- In the last lecture, we showed that any *comparison-based sorting* algorithm has running time $\Omega(n \log n)$.
- However, in some cases, *additional information about the input* can be leveraged to sort in $O(n)$ time.

# Recap

- In the last lecture, we showed that any *comparison-based sorting* algorithm has running time $\Omega(n \log n)$.
- However, in some cases, *additional information about the input* can be leveraged to sort in $O(n)$ time.

- Question. Can we find the $i$-th smallest element in an array of $n$ elements in time better than $\Theta(n \log n)$?

# Recap

- In the last lecture, we showed that any *comparison-based sorting* algorithm has running time $\Omega(n \log n)$.
- However, in some cases, *additional information about the input* can be leveraged to sort in $O(n)$ time.

- Question. Can we find the $i$-th smallest element in an array of $n$ elements in time better than $\Theta(n \log n)$?
- In today's lecture, we'll discuss **two algorithms** to solve this problem – one has *expected running time* $\Theta(n)$, the other has *worst-case running time* $\Theta(n)$.

# Order Statistics

- The $i$-th order statistic of a set of $n$ elements is the $i$-th smallest element in the set.
- The minimum is the first order statistic, and the maximum is the $n$-th order statistic.

# Order Statistics

- The $i$-th order statistic of a set of $n$ elements is the $i$-th smallest element in the set.
- The minimum is the first order statistic, and the maximum is the $n$-th order statistic.
- The median is (roughly) the $n/2$-th order statistic.
- If $n$ is *odd*, median occurs at $i = (n + 1)/2$.
- If $n$ is *even*, the lower median occurs at $i = n/2$ and the upper median at $i = n/2 + 1$.

# Order Statistics

- The $i$-th order statistic of a set of $n$ elements is the $i$-th smallest element in the set.
- The minimum is the first order statistic, and the maximum is the $n$-th order statistic.
- The median is (roughly) the $n/2$-th order statistic.
- Thus, regardless of the *parity of* $n$, medians occur at $i = \left\lfloor \frac{n+1}{2} \right\rfloor$ and $i = \left\lceil \frac{n+1}{2} \right\rceil$.
- We'll use the phrase "the median" to refer to the *lower median*.

# The Selection Problem

- Selection problem. Given a set $A$ of $n$ *distinct numbers* and an integer $i \in [1, n]$, output the element $x \in A$ that is larger than exactly $i - 1$ other elements of $A$.
- Note: It is only for convenience that we assume that the set contains distinct elements. All *the arguments extend to the general situation* with minor alterations.

# The Selection Problem

- Selection problem. Given a set $A$ of $n$ *distinct numbers* and an integer $i \in [1, n]$, output the element $x \in A$ that is larger than exactly $i - 1$ other elements of $A$.
- Note: It is only for convenience that we assume that the set contains distinct elements. All *the arguments extend to the general situation* with minor alterations.

- By sorting the elements in $A$, we can solve the above problem in $O(n \log n)$ time. We will see how to solve the problem in $O(n)$ time *by avoiding sorting*.

# Minimum and Maximum

# Finding minimum

- Finding *minimum* $(i = 1)$ and *maximum* $(i = n)$ are special cases of the selection problem.

```
MINIMUM(A, n)
1  min = A[1]
2  for i = 2 to n
3       if min > A[i]
4            min = A[i]
5  return min
```

# Finding minimum

- Finding *minimum* $(i = 1)$ and *maximum* $(i = n)$ are special cases of the selection problem.

```
MINIMUM(A, n)
1  min = A[1]
2  for i = 2 to n
3      if min > A[i]
4          min = A[i]
5  return min
```

- Note. The algorithm finds the minimum with $n - 1$ comparisons. This is the *best* we can do. Why?

# Finding minimum

- Finding *minimum* $(i = 1)$ and *maximum* $(i = n)$ are special cases of the selection problem.

```
MINIMUM(A, n)
1  min = A[1]
2  for i = 2 to n
3      if min > A[i]
4          min = A[i]
5  return min
```

- Note. The algorithm finds the minimum with $n - 1$ comparisons. This is the *best* we can do. Why?
- Hint. Think of any algorithm that determines the minimum as a *tournament among the elements*.

# Finding minimum

- Finding *minimum* $(i = 1)$ and *maximum* $(i = n)$ are special cases of the selection problem.

```
MINIMUM(A, n)
1  min = A[1]
2  for i = 2 to n
3      if min > A[i]
4          min = A[i]
5  return min
```

- Note. The algorithm finds the minimum with $n - 1$ comparisons. This is the *best* we can do. Why?

- Similarly, we can find the *maximum* using $n - 1$ comparisons.

# Simultaneous min and max

- Question. How many comparisons are required to find minimum and maximum *simultaneously*?
- Easy to see that $2n - 2$ comparisons suffices.

# Simultaneous min and max

- Question. How many comparisons are required to find minimum and maximum *simultaneously*?
- Easy to see that $2n - 2$ comparisons suffices.
- We can do better. We can find both the min and max using *at most* $3\lfloor n/2 \rfloor$ comparisons.
- Idea. Compare *pairs* of elements from the input *first with each other*, and then compare the smaller with the current min and the larger with the current max, at a cost of $3$ *comparisons for every* $2$ *elements*.

# Simultaneous min and max

- Question. How many comparisons are required to find minimum and maximum *simultaneously*?
- Easy to see that $2n - 2$ comparisons suffices.
- We can do better. We can find both the min and max using *at most* $3\lfloor n/2 \rfloor$ comparisons.

- Lower bound. Turns out that ~$3n/2$ comparisons is the best we can do. (see *Exercise 9.1-4* in Cormen et. al.'s book).

# Selection in expected linear time

# Selection in linear time

- The *general selection problem* – finding the $i$-th order statistic for any $i$ – appears more difficult than the problem of finding minimum or maximum.
- Yet, surprisingly, the asymptotic running time for both problems is $\Theta(n)$.

# Selection in linear time

- The *general selection problem* – finding the $i$-th order statistic for any $i$ – appears more difficult than the problem of finding minimum or maximum.
- Yet, surprisingly, the asymptotic running time for both problems is $\Theta(n)$.

- We'll first present a *divide-and-conquer* ***randomized*** *algorithm* for the selection problem. The algorithm has *expected running time* $\Theta(n)$.

# Selection in linear time

- The *general selection problem* – finding the $i$-th order statistic for any $i$ – appears more difficult than the problem of finding minimum or maximum.
- Yet, surprisingly, the asymptotic running time for both problems is $\Theta(n)$.

- We'll first present a *divide-and-conquer* ***randomized*** *algorithm* for the selection problem. The algorithm has *expected running time* $\Theta(n)$. Later, we'll see an algorithm with *worst-case running time* $\Theta(n)$.

# Randomized Select: The idea

- The randomized select algorithm is modelled after the *randomized quicksort* algorithm.
- Like quicksort it *partitions the input array, recursively*.

# Randomized Select: The idea

- The randomized select algorithm is modelled after the *randomized quicksort* algorithm.
- Like quicksort it *partitions the input array, recursively*.
- But unlike quicksort, which recursively processes both sides of the partition, randomized select *works on only one side of the partition* (depending on which side of the partition the $i$-th order statistic lies).

# Randomized Select: The idea

- The randomized select algorithm is modelled after the *randomized quicksort* algorithm.
- Like quicksort it *partitions the input array, recursively*.
- But unlike quicksort, which recursively processes both sides of the partition, randomized select *works on only one side of the partition* (depending on which side of the partition the $i$-th order statistic lies).
- This difference shows up in the analysis – we get a running time of $\Theta(n)$ instead of $\Theta(n \log n)$.

# Randomized Select: Pseudocode

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6       return A[q]        // the pivot value is the answer
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

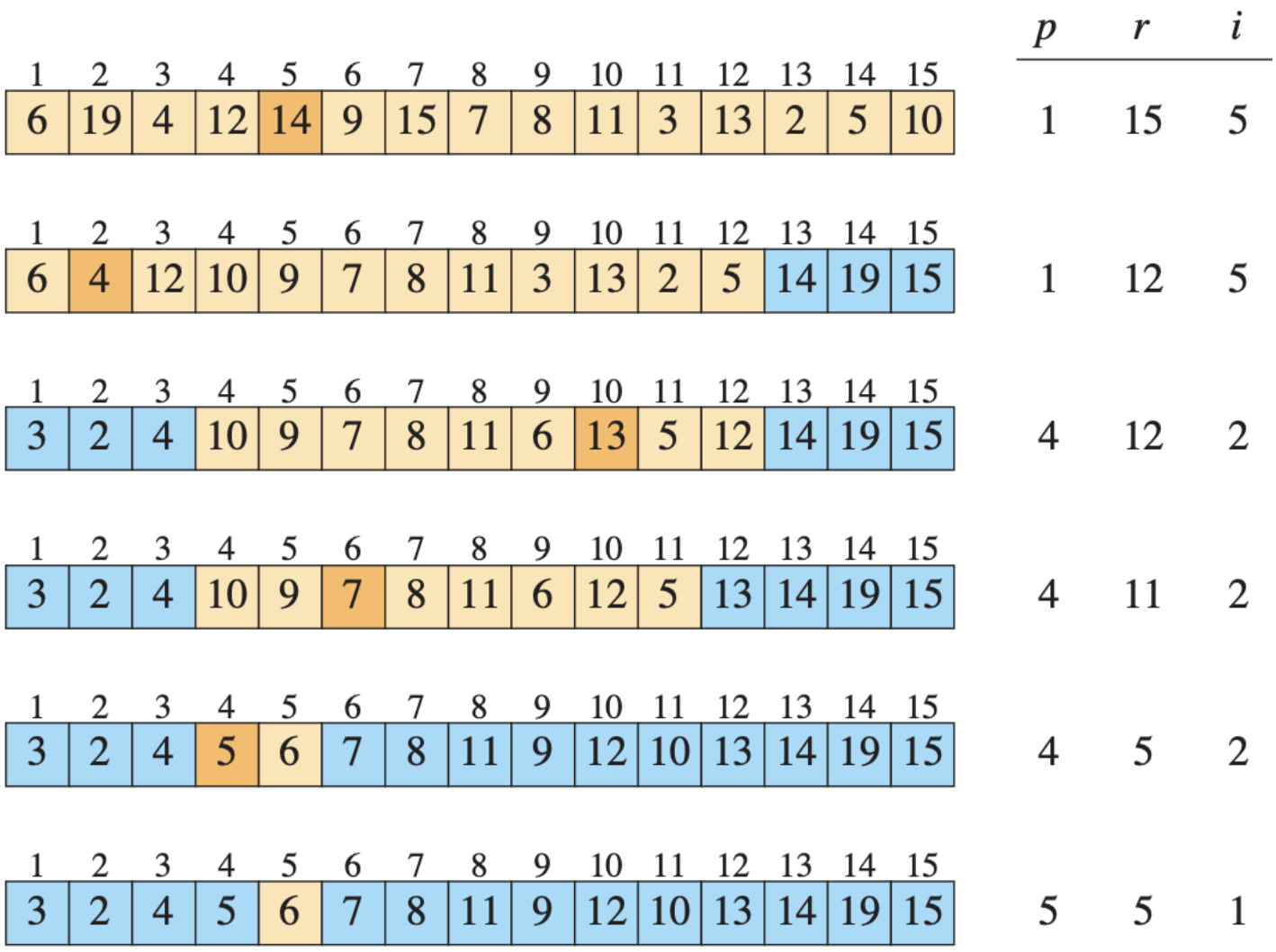
- We invoke RANDOMIZED-SELECT with $p = 1$ and $r = n$.

# Randomized Select: Pseudocode

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6       return A[q]        // the pivot value is the answer
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

- Recall from Lecture 4 that RANDOMIZED-PARTITION partitions the array $A[p:r]$ around a *random pivot* and returns the index of the pivot in the sorted array. This index is stored in $q$.

# Randomized Select: An example




tan → still in play; dark tan → pivot
blue → no longer in play

# Randomized Select: Time complexity

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6       return A[q]        // the pivot value is the answer
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

- Also recall that the time taken by RANDOMIZED-PARTITION is $\Theta(n)$, where $n = |A[p:r]|$. So, the time taken by the above algorithm before the recursive call in Step 8 or 9 is $\Theta(n)$.

# Randomized Select: Time complexity

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6       return A[q]        // the pivot value is the answer
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

- The *worst-case running time* is $\Theta(n^2)$ as the algorithm could be extremely unlucky and always partition around the largest remaining elements. This gives the recurrence $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$.

# Randomized Select: Expected time

- Let $A^{(j)}$ be the set (subarray) of elements of $A$ that are *still in play* after $j$ partitioning. $A^{(0)} = A$.
- Since each partitioning removes the pivot, the sequence $|A^{(0)}|, |A^{(1)}|, |A^{(2)}|, \ldots$ *strictly decreases*.

# Randomized Select: Expected time

- Let $A^{(j)}$ be the set (subarray) of elements of $A$ that are *still in play* after $j$ partitioning. $A^{(0)} = A$.
- Since each partitioning removes the pivot, the sequence $|A^{(0)}|, |A^{(1)}|, |A^{(2)}|, \ldots$ *strictly decreases*.
- Definition. We call the $j$-th partitioning helpful if $|A^{(j)}| \leq (3/4)|A^{(j-1)}|$.

# Randomized Select: Expected time

- Let $A^{(j)}$ be the set (subarray) of elements of $A$ that are *still in play* after $j$ partitioning. $A^{(0)} = A$.
- Since each partitioning removes the pivot, the sequence $|A^{(0)}|, |A^{(1)}|, |A^{(2)}|, \ldots$ *strictly decreases*.
- Definition. We call the $j$-th partitioning helpful if $|A^{(j)}| \leq (3/4)|A^{(j-1)}|$.

- Lemma. A partitioning is helpful with probability at least ½.

# Proof of the lemma

- Define the middle half of an $m$-element subarray as all but the smallest $\lceil m/4 \rceil - 1$ and the greatest $\lceil m/4 \rceil - 1$ elements.

# Proof of the lemma

- Define the middle half of an $m$-element subarray as all but the smallest $\lceil m/4 \rceil - 1$ and the greatest $\lceil m/4 \rceil - 1$ elements.
- Obs. If the *pivot falls in the middle half*, then at least $\lceil m/4 \rceil$ elements (including the pivot) will no longer be in play. Hence, the number of elements remaining in play will be at most $m - \lceil m/4 \rceil \leq 3m/4$.

# Proof of the lemma

- Define the middle half of an $m$-element subarray as all but the smallest $\lceil m/4 \rceil - 1$ and the greatest $\lceil m/4 \rceil - 1$ elements.
- Obs. If the *pivot falls in the middle half*, then the partitioning is *helpful*.

# Proof of the lemma

- Define the middle half of an $m$-element subarray as all but the smallest $\lceil m/4 \rceil - 1$ and the greatest $\lceil m/4 \rceil - 1$ elements.
- Obs. If the *pivot falls in the middle half*, then the partitioning is *helpful*.
- The probability that a randomly chosen pivot is **not** in the middle half is at most:

$$\frac{2(\lceil m/4 \rceil - 1)}{m} \leq 1/2.$$

- So, a partitioning is helpful with probability $\geq 1/2$.

# Randomized Select: Expected time

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]        // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k
6       return A[q]        // the pivot value is the answer
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

- As mentioned before, the time taken by the above algorithm before the recursive call in Step 8 or 9 is $\Theta(n)$. Let $c$ be the constant hidden in the $\Theta$ notation.

# Randomized Select: Expected time

- Let $y_j$ be the number of partitioning between the $(j-1)$-th helpful partitioning and the $j$-th helpful partitioning (*excluding* the $(j-1)$ -th helpful partitioning but *including* the $j$-th helpful partitioning).

# Randomized Select: Expected time

- Let $y_j$ be the number of partitioning between the $(j-1)$-th helpful partitioning and the $j$-th helpful partitioning (*excluding* the $(j-1)$ -th helpful partitioning but *including* the $j$-th helpful partitioning).
- Then, the time T taken by RANDOMIZED-SELECT is at most $cn(y_1 + (3/4)y_2 + (3/4)^2 y_3 + \cdots)$. Why?

# Randomized Select: Expected time

- Let $y_j$ be the number of partitioning between the $(j-1)$-th helpful partitioning and the $j$-th helpful partitioning (*excluding* the $(j-1)$ -th helpful partitioning but *including* the $j$-th helpful partitioning).
- Then, the time T taken by RANDOMIZED-SELECT is at most $cn(y_1 + (3/4)y_2 + (3/4)^2 y_3 + \cdots)$. Why?
- Hence, by linearity of expectation,

$$E[T] \leq cn(E[y_1] + (3/4)E[y_2] + (3/4)^2 E[y_3] + \cdots)$$

# Randomized Select: Expected time

- Let $y_j$ be the number of partitioning between the $(j-1)$-th helpful partitioning and the $j$-th helpful partitioning (*excluding* the $(j-1)$ -th helpful partitioning but *including* the $j$-th helpful partitioning).
- Then, the time T taken by RANDOMIZED-SELECT is at most $cn(y_1 + (3/4)y_2 + (3/4)^2 y_3 + \cdots)$. Why?
- Hence, by linearity of expectation,

  $$E[T] \leq cn(E[y_1] + (3/4)E[y_2] + (3/4)^2 E[y_3] + \cdots)$$
- Obs. $E[y_i] \leq 2$. (By the previous lemma)
- Hence, $E[T] = \Theta(n)$.

# Selection in worst-case linear time

# Deterministic Select

- Algorithms that do not use randomization are called **deterministic algorithms**.
- We'll now discuss a (deterministic) select algorithm whose running time is $\Theta(n)$ *in the worst-case*.

# Deterministic Select

- Algorithms that do not use randomization are called **deterministic algorithms**.
- We'll now discuss a (deterministic) select algorithm whose running time is $\Theta(n)$ *in the worst-case*.
- The running time of RANDOMIZED-SELECT was *quadratic* in the worst-case.

# Deterministic Select

- Algorithms that do not use randomization are called **deterministic algorithms**.
- We'll now discuss a (deterministic) select algorithm whose running time is $\Theta(n)$ *in the worst-case*.
- The running time of RANDOMIZED-SELECT was *quadratic* in the worst-case.
- Although deterministic select has $\Theta(n)$ run-time in the worst-case, it is *not nearly as practical* as RANDOMIZED-SELECT. It is mostly of theoretical interest, but the *idea behind it is remarkable*.

# Deterministic Select: The idea

- Like RANDOMIZED-SELECT, the deterministic SELECT finds the desired order statistic by *recursively partitioning the input array*.

# Deterministic Select: The idea

- Like RANDOMIZED-SELECT, the deterministic SELECT finds the desired order statistic by *recursively partitioning the input array*.
- Unlike RANDOMIZED-SELECT, however, SELECT *guarantees a good split* by choosing a *provably good pivot* when partitioning the array.

# Deterministic Select: The idea

- Like RANDOMIZED-SELECT, the deterministic SELECT finds the desired order statistic by *recursively partitioning the input array*.
- Unlike RANDOMIZED-SELECT, however, SELECT *guarantees a good split* by choosing a *provably good pivot* when partitioning the array.
- The cleverness in the algorithm is that it *finds the pivot recursively*, using SELECT!

# Deterministic Select: The idea

- Like RANDOMIZED-SELECT, the deterministic SELECT finds the desired order statistic by *recursively partitioning the input array*.
- Unlike RANDOMIZED-SELECT, however, SELECT *guarantees a good split* by choosing a *provably good pivot* when partitioning the array.
- The cleverness in the algorithm is that it *finds the pivot recursively*, using SELECT!
- Thus, there're *two invocations* of SELECT: one to find a good pivot, and another to recursively find the desired order statistic.

# Deterministic Select: Pseudocode

```
SELECT(A, p, r, i)
 1  while (r − p + 1) mod 5 ≠ 0
 2      for j = p + 1 to r                          // put the minimum into A[p]
 3          if A[p] > A[j]
 4              exchange A[p] with A[j]
 5      // If we want the minimum of A[p : r], we're done.
 6      if i == 1
 7          return A[p]
 8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
 9      p = p + 1
10      i = i − 1
11  g = (r − p + 1)/5                               // number of 5-element groups
12  for j = p to p + g − 1                          // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)                // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                                 // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

We invoke SELECT with $p = 1$ and $r = n$.

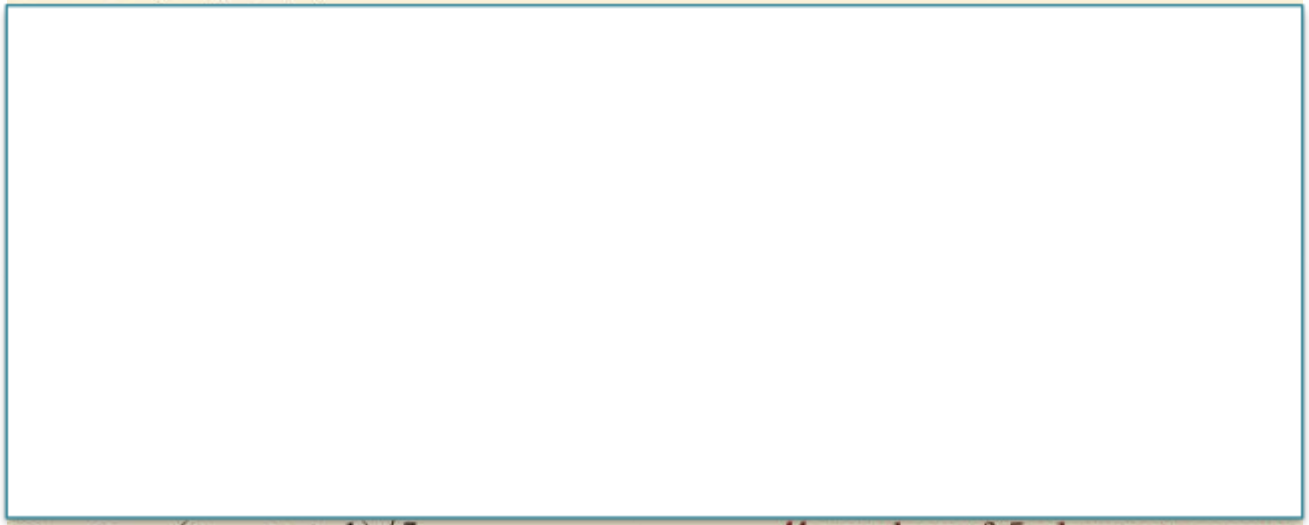
# Deterministic Select: Pseudocode

```
SELECT(A, p, r, i)
1  while (r − p + 1) mod 5 ≠ 0
2      for j = p + 1 to r                          // put the minimum into A[p]
3          if A[p] > A[j]
4              exchange A[p] with A[j]
5      // If we want the minimum of A[p : r], we're done.
6      if i == 1
7          return A[p]
8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
9      p = p + 1
10     i = i − 1
11 g = (r − p + 1)/5                               // number of 5-element groups
12 for j = p to p + g − 1                          // sort each group
13     sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14 // All group medians now lie in the middle fifth of A[p : r].
15 // Find the pivot x recursively as the median of the group medians.
16 x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17 q = PARTITION-AROUND(A, p, r, x)                 // partition around the pivot
18 // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19 k = q − p + 1
20 if i == k
21     return A[q]                                 // the pivot value is the answer
22 elseif i < k
23     return SELECT(A, p, q − 1, i)
24 else return SELECT(A, q + 1, r, i − k)
```

The **while** loop (*Steps 1-10*) executes at most 4 times to ensure $n = r - p + 1$ is a multiple of $5$.

# Deterministic Select: Pseudocode

```
SELECT(A, p, r, i)
```



```
11  g = (r − p + 1)/5                          // number of 5-element groups
12  for j = p to p + g − 1                     // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)           // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                            // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

$g = n/5$

# Deterministic Select: Pseudocode

```
SELECT(A, p, r, i)
```

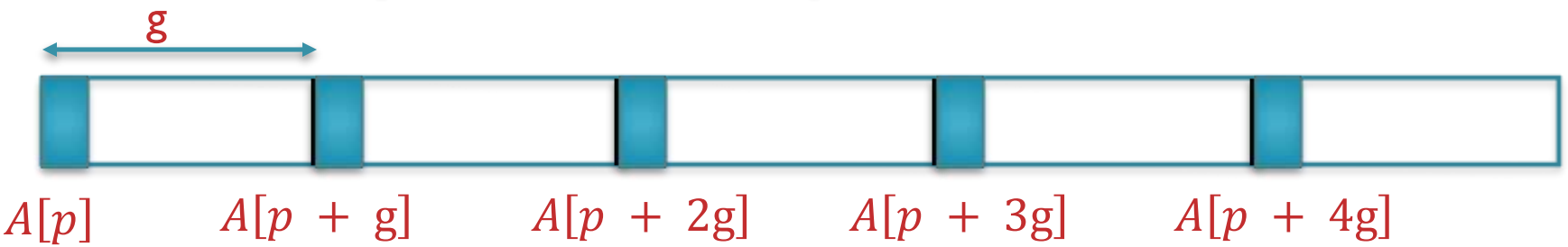
```
11  g = (r − p + 1)/5                          // number of 5-element groups
12  for j = p to p + g − 1                     // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)           // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                            // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

*Steps 12-13* divide $A[p:r]$ into $g = n/5$ groups of $5$ elements each and sort each of the $5$-element groups.

# The array after Steps 12-13


- Think of dividing the array $A$ into $5$ *equal parts*, each of length $g = (r - p + 1)/5$.

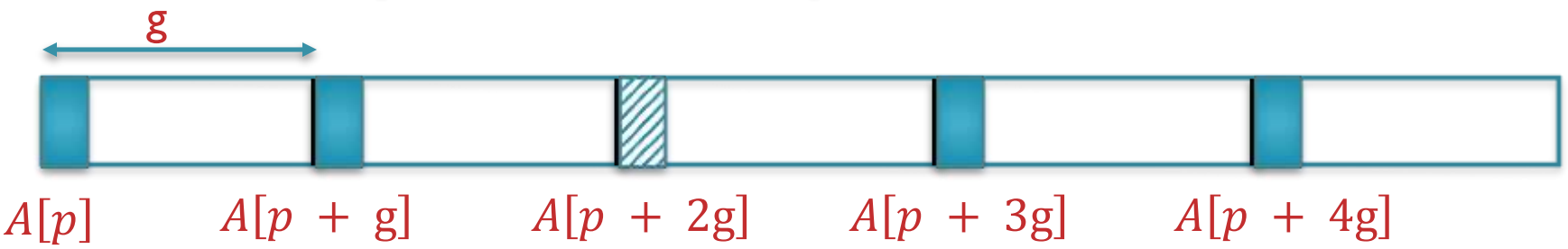
# The array after Steps 12-13




- The first 5-element group is

$$A[p], A[p + g], A[p + 2g], A[p + 3g], A[p + 4g].$$

# The array after Steps 12-13

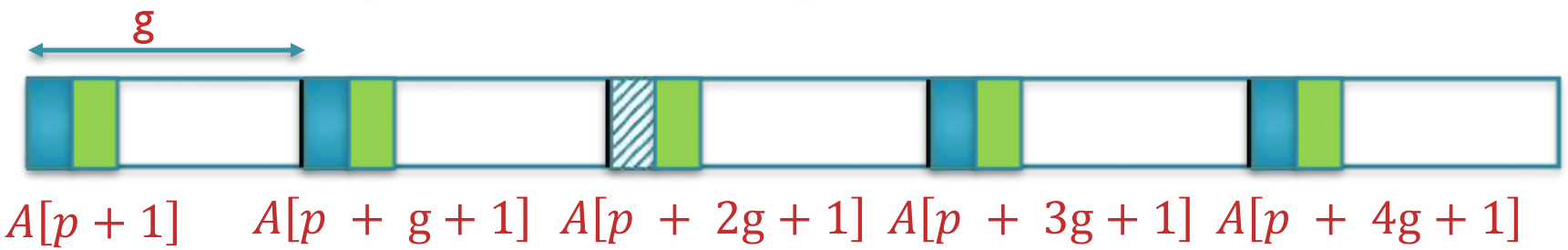



- The first 5-element group is

$$A[p], A[p+\mathrm{g}], A[p+2\mathrm{g}], A[p+3\mathrm{g}], A[p+4\mathrm{g}].$$

- If the group is sorted, the *median* would lie in $A[p+2\mathrm{g}]$.

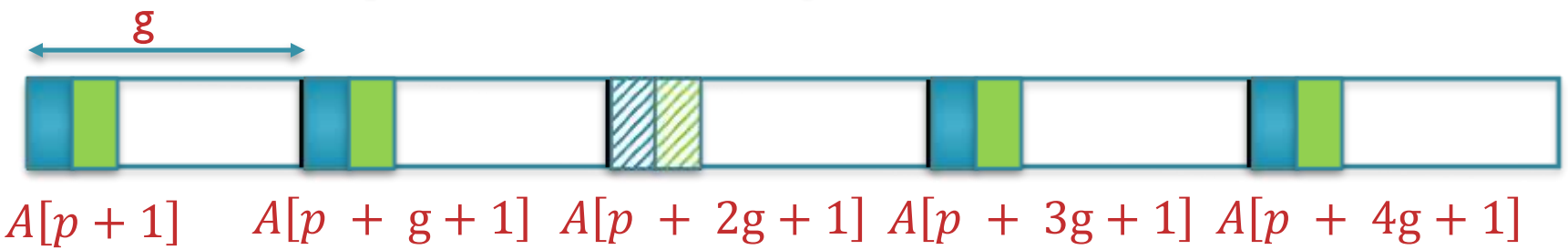
# The array after Steps 12-13




- The second 5-element group is

$$A[p+1], A[p+g+1], A[p+2g+1], A[p+3g+1], A[p+4g+1].$$

# The array after Steps 12-13

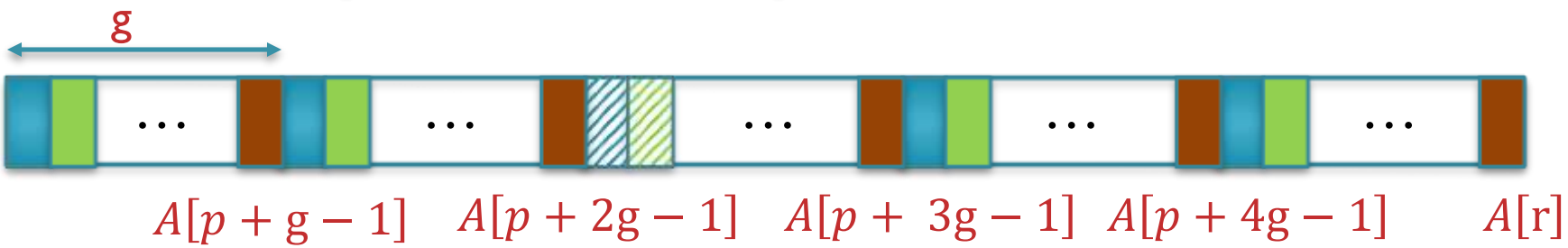



- The second 5-element group is

$$A[p+1], A[p+g+1], A[p+2g+1], A[p+3g+1], A[p+4g+1].$$

- If the group is sorted, the *median* would lie in $A[p+2g+1]$.

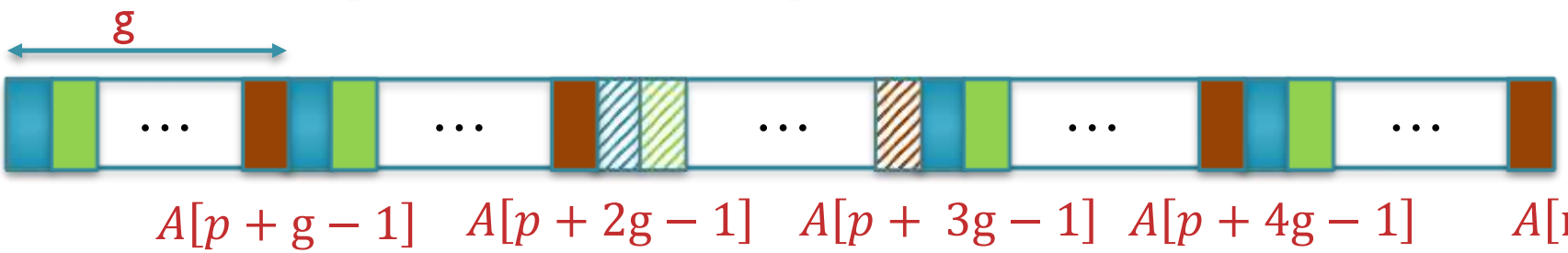
# The array after Steps 12-13




- The g-th 5-element group is

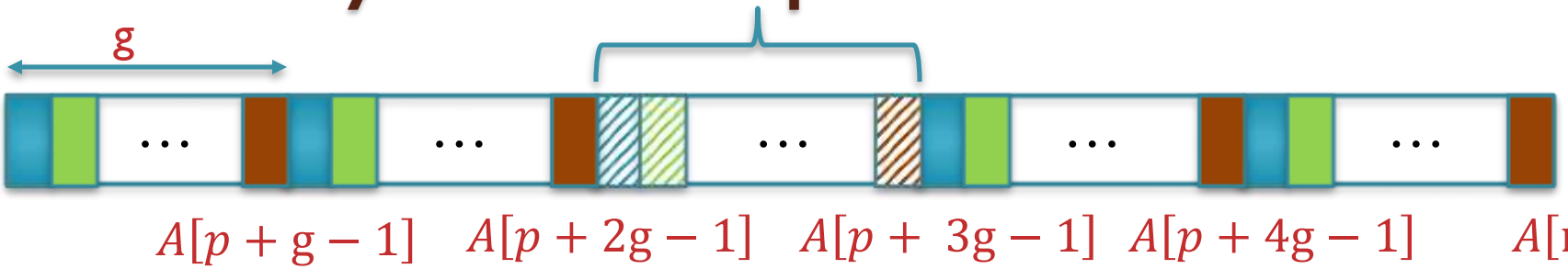
$$A[p + g - 1], A[p + 2g - 1], A[p + 3g - 1], A[p + 4g - 1], A[r].$$

# The array after Steps 12-13




- The g-th 5-element group is
$$A[p+g-1], A[p+2g-1], A[p+3g-1], A[p+4g-1], A[r].$$
- If the group is sorted, the *median* would lie in $A[p+3g-1]$.

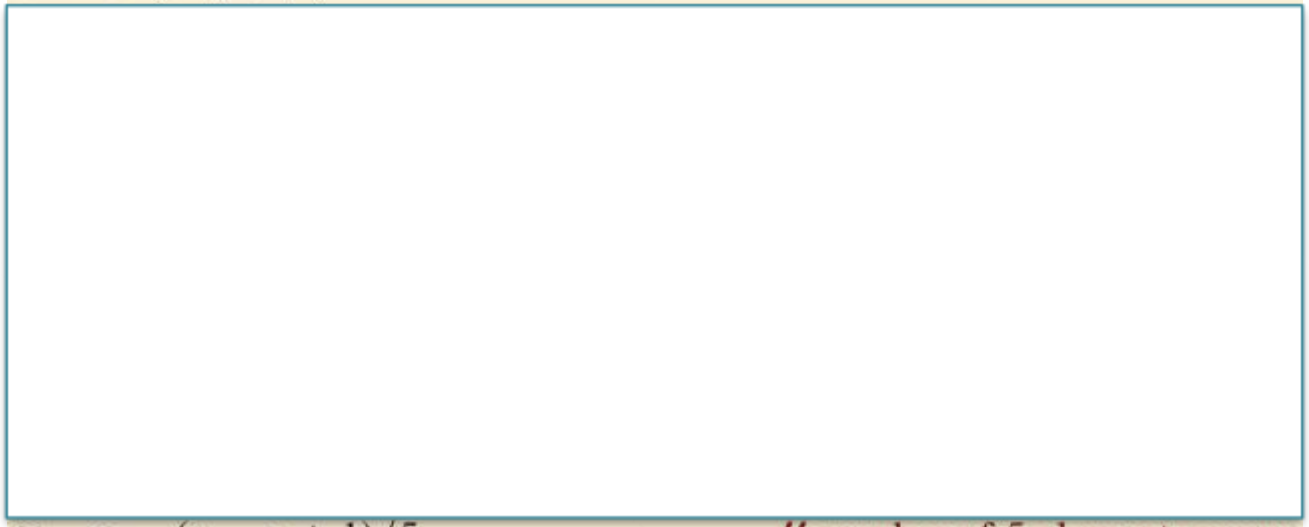
# The array after Steps 12-13




- The $g$-th 5-element group is
$$A[p + g - 1], A[p + 2g - 1], A[p + 3g - 1], A[p + 4g - 1], A[r].$$
- If the group is sorted, the *median* would lie in $A[p + 3g - 1]$.
- Therefore, the median of each of the $g$, 5-element groups lies in the range $A[p + 2g : p + 3g - 1]$, i.e., the *middle fifth* of $A[p : r]$.

# Deterministic Select: Pseudocode

```
SELECT(A, p, r, i)
```

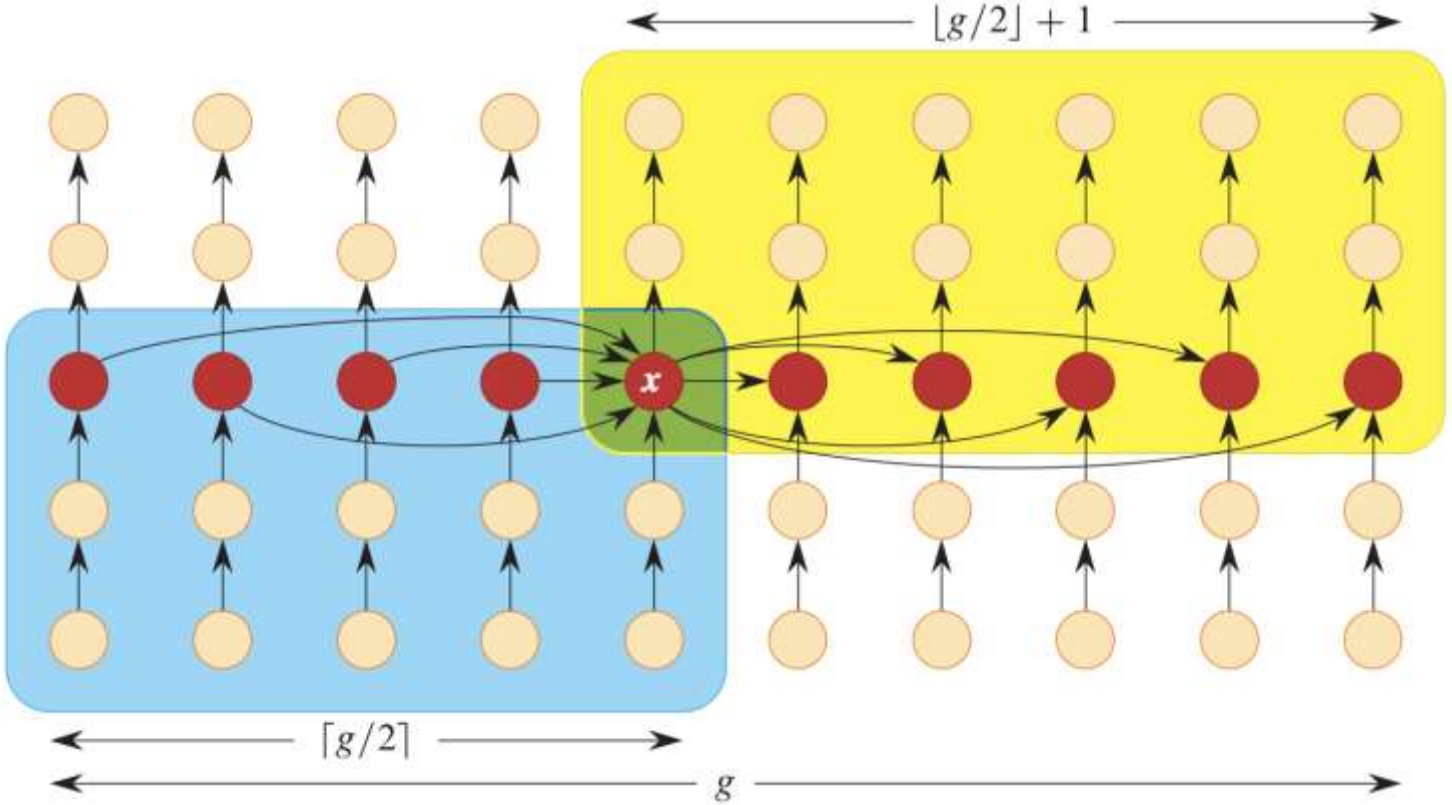


```
11  g = (r − p + 1)/5                          // number of 5-element groups
12  for j = p to p + g − 1                     // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)           // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                            // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

*Steps 16* finds the *median of the g group medians*. This element is chosen as the **pivot** $x$.

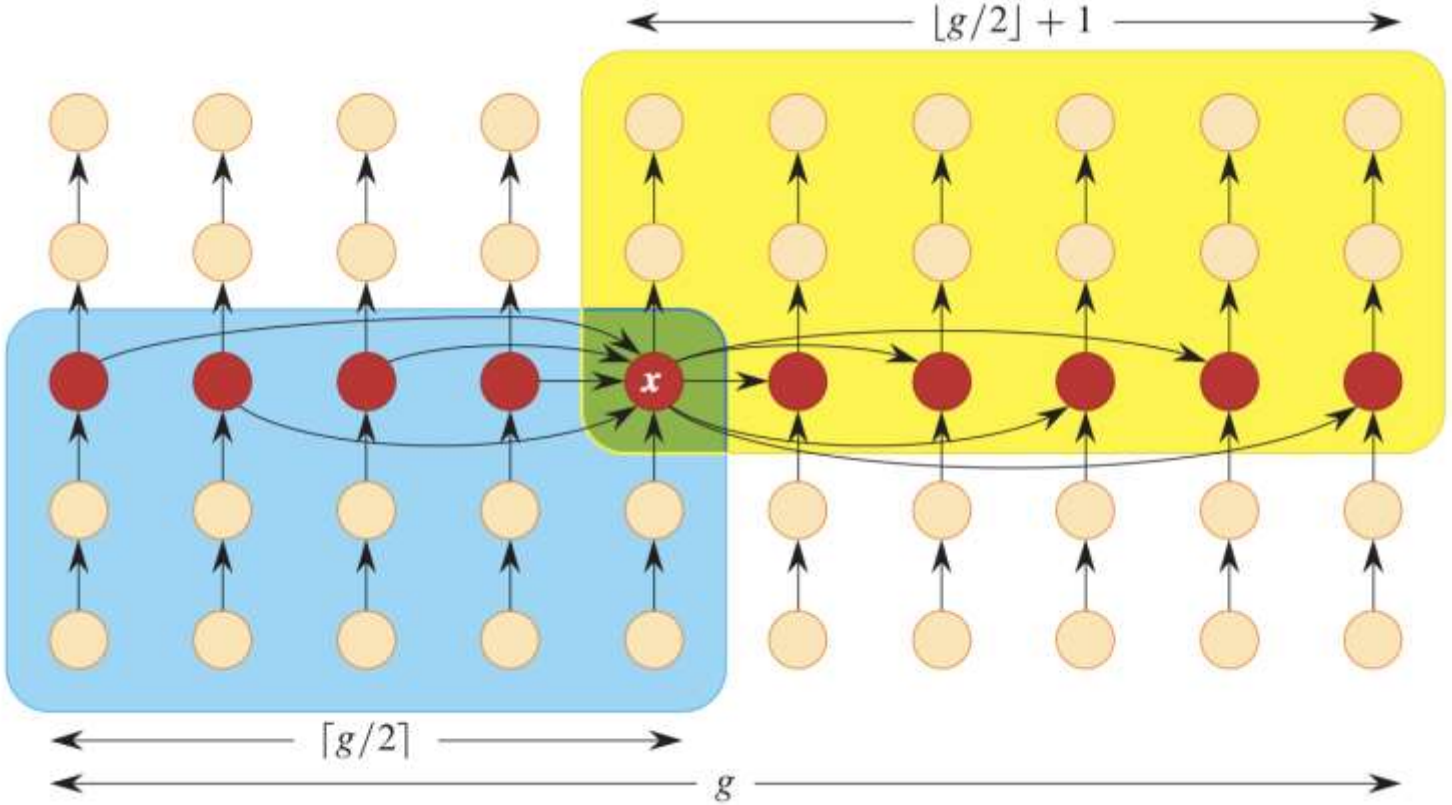
But why?

# Median of medians



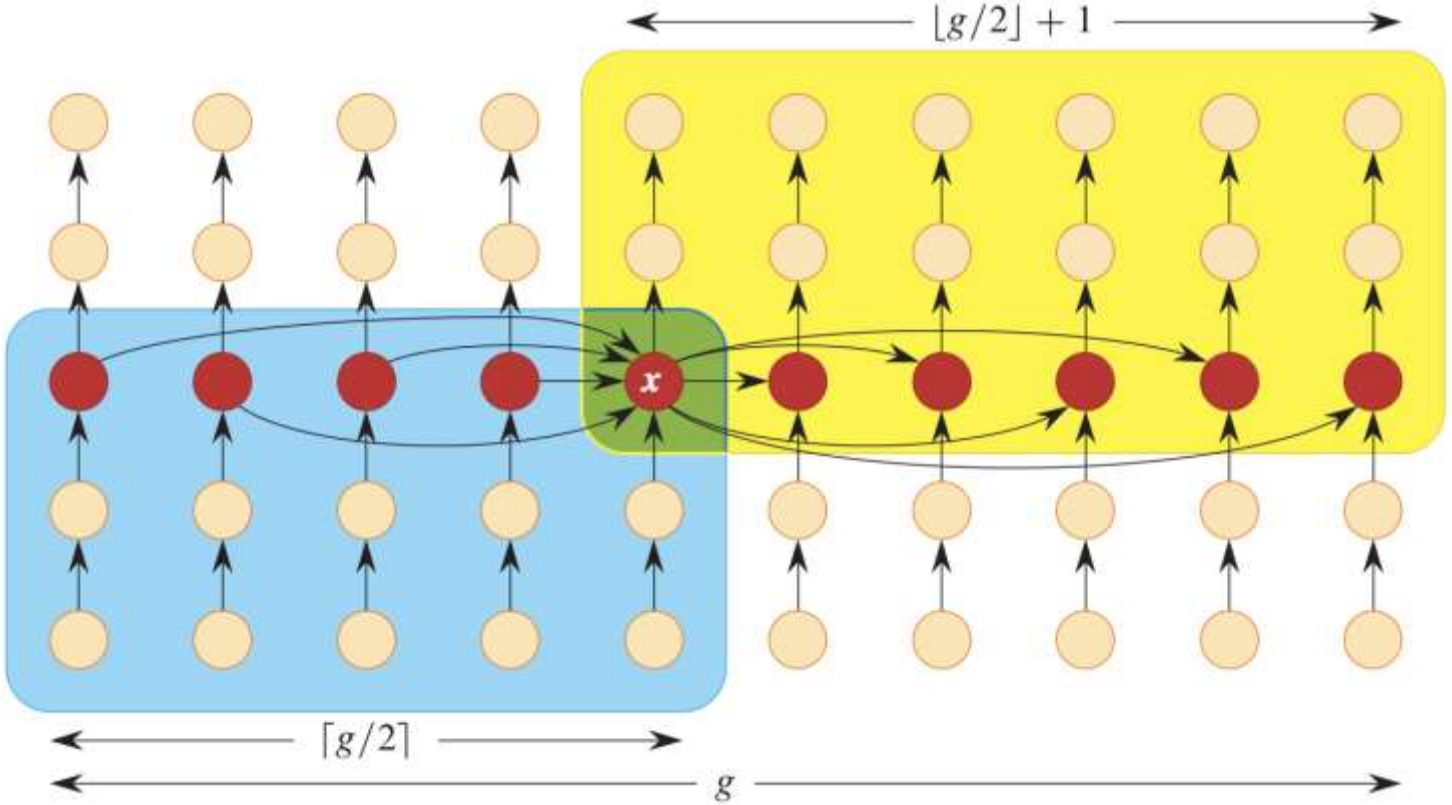

- Each *vertical column* depicts a sorted group of 5 elements.
- Obs. The yellow region contains elements that we know are *greater than or equal to* $x$; the blue region contains elements that we know are *less than or equal to* $x$.

# Median of medians




- Each of the yellow and the blue regions contains *at least* $3g/2$ elements.

# Median of medians




- Each of the yellow and the blue regions contains *at least* $3g/2$ elements. Thus, each of the upper and lower sides of the partition of $A[p:r]$ around $x$ contains *at most* $5g-3g/2 = 7g/2 = 7n/10$ elements. $(g = n/5)$

# Deterministic Select: Pseudocode

```
SELECT(A, p, r, i)
```

```
11  g = (r − p + 1)/5                          // number of 5-element groups
12  for j = p to p + g − 1                     // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)           // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                            // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

If we partition around $x$, then *each* of the two partitions has size at most $7n/10$.

# Deterministic Select: Time complexity

```
SELECT(A, p, r, i)
1  while (r − p + 1) mod 5 ≠ 0
2      for j = p + 1 to r                          // put the minimum into A[p]
3          if A[p] > A[j]
4              exchange A[p] with A[j]
5      // If we want the minimum of A[p : r], we're done.
6      if i == 1
7          return A[p]
8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
9      p = p + 1
10     i = i − 1
11 g = (r − p + 1)/5                               // number of 5-element groups
12 for j = p to p + g − 1                          // sort each group
13     sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14 // All group medians now lie in the middle fifth of A[p : r].
15 // Find the pivot x recursively as the median of the group medians.
16 x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17 q = PARTITION-AROUND(A, p, r, x)                 // partition around the pivot
18 // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19 k = q − p + 1
20 if i == k
21     return A[q]                                 // the pivot value is the answer
22 elseif i < k
23     return SELECT(A, p, q − 1, i)
24 else return SELECT(A, q + 1, r, i − k)
```

Suppose $T(n)$ is the time complexity of SELECT.

Takes $\Theta(1)$ time

# Deterministic Select: Time complexity

```
SELECT(A, p, r, i)
1  while (r − p + 1) mod 5 ≠ 0
2      for j = p + 1 to r                          // put the minimum into A[p]
3          if A[p] > A[j]
4              exchange A[p] with A[j]
5      // If we want the minimum of A[p : r], we're done.
6      if i == 1
7          return A[p]
8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
9      p = p + 1
10     i = i − 1
11 g = (r − p + 1)/5                               // number of 5-element groups
12 for j = p to p + g − 1                          // sort each group
13     sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14 // All group medians now lie in the middle fifth of A[p : r].
15 // Find the pivot x recursively as the median of the group medians.
16 x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17 q = PARTITION-AROUND(A, p, r, x)                 // partition around the pivot
18 // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19 k = q − p + 1
20 if i == k
21     return A[q]                                 // the pivot value is the answer
22 elseif i < k
23     return SELECT(A, p, q − 1, i)
24 else return SELECT(A, q + 1, r, i − k)
```

Suppose $T(n)$ is the time complexity of SELECT.

Takes $\Theta(n)$ time

# Deterministic Select: Time complexity

```
SELECT(A, p, r, i)
 1  while (r − p + 1) mod 5 ≠ 0
 2      for j = p + 1 to r                    // put the minimum into A[p]
 3          if A[p] > A[j]
 4              exchange A[p] with A[j]
 5      // If we want the minimum of A[p : r], we're done.
 6      if i == 1
 7          return A[p]
 8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
 9      p = p + 1
10      i = i − 1
11  g = (r − p + 1)/5                         // number of 5-element groups
12  for j = p to p + g − 1                    // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)          // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                           // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

Suppose $T(n)$ is the time complexity of SELECT.

Takes $T(n/5)$ time

# Deterministic Select: Time complexity

```
SELECT(A, p, r, i)
 1  while (r − p + 1) mod 5 ≠ 0
 2      for j = p + 1 to r                    // put the minimum into A[p]
 3          if A[p] > A[j]
 4              exchange A[p] with A[j]
 5      // If we want the minimum of A[p : r], we're done.
 6      if i == 1
 7          return A[p]
 8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
 9      p = p + 1
10      i = i − 1
11  g = (r − p + 1)/5                         // number of 5-element groups
12  for j = p to p + g − 1                    // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)          // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                           // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

Suppose $T(n)$ is the time complexity of SELECT.

Takes $T(7n/10)$ time

# Deterministic Select: Time complexity

```
SELECT(A, p, r, i)
 1  while (r − p + 1) mod 5 ≠ 0
 2      for j = p + 1 to r                          // put the minimum into A[p]
 3          if A[p] > A[j]
 4              exchange A[p] with A[j]
 5      // If we want the minimum of A[p : r], we're done.
 6      if i == 1
 7          return A[p]
 8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
 9      p = p + 1
10      i = i − 1
11  g = (r − p + 1)/5                               // number of 5-element groups
12  for j = p to p + g − 1                          // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)                // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                                 // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

Suppose $T(n)$ is the time complexity of SELECT.

Then, $T(n)$ is $\leq T(n/5) + T(7n/10) + \Theta(n)$.

# Solving the recurrence

$$T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$$

- We can use the recursion tree method to solve the above recurrence and show that $T(n) = \Theta(n)$.

# Solving the recurrence

$$T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$$

- We can use the recursion tree method to solve the above recurrence and show that $T(n) = \Theta(n)$.
- It is easy to verify using the *substitution method* that $T(n) = \Theta(n)$ is indeed the correct solution:
  - Suppose $T(m) \leq cm$ for some suitable *large constant* $c > 0$ and for all $1 \leq m < n$.

# Solving the recurrence

$$T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$$

- We can use the recursion tree method to solve the above recurrence and show that $T(n) = \Theta(n)$.
- It is easy to verify using the *substitution method* that $T(n) = \Theta(n)$ is indeed the correct solution:
  - Suppose $T(m) \leq cm$ for some suitable *large constant* $c > 0$ and for all $1 \leq m < n$.
  - Then, $T(n) \leq cn/5 + 7cn/10 + \Theta(n) = 9cn/10 + \Theta(n)$.

# Solving the recurrence

$$T(n) \leq T(n/5) + T(7n/10) + \Theta(n)$$

- We can use the recursion tree method to solve the above recurrence and show that $T(n) = \Theta(n)$.
- It is easy to verify using the *substitution method* that $T(n) = \Theta(n)$ is indeed the correct solution:
  - Suppose $T(m) \leq cm$ for some suitable *large constant* $c > 0$ and for all $1 \leq m < n$.
  - Then, $T(n) \leq cn/5 + 7cn/10 + \Theta(n) = 9cn/10 + \Theta(n)$.
  - If we choose $c$ large enough such that $c/10$ dominates the constant hidden in $\Theta(n)$, then $T(n) \leq cn$.

# Deterministic Select: Time complexity

```
SELECT(A, p, r, i)
 1  while (r − p + 1) mod 5 ≠ 0
 2      for j = p + 1 to r                    // put the minimum into A[p]
 3          if A[p] > A[j]
 4              exchange A[p] with A[j]
 5      // If we want the minimum of A[p : r], we're done.
 6      if i == 1
 7          return A[p]
 8      // Otherwise, we want the (i − 1)st element of A[p + 1 : r].
 9      p = p + 1
10      i = i − 1
11  g = (r − p + 1)/5                         // number of 5-element groups
12  for j = p to p + g − 1                    // sort each group
13      sort ⟨A[j], A[j + g], A[j + 2g], A[j + 3g], A[j + 4g]⟩ in place
14  // All group medians now lie in the middle fifth of A[p : r].
15  // Find the pivot x recursively as the median of the group medians.
16  x = SELECT(A, p + 2g, p + 3g − 1, ⌈g/2⌉)
17  q = PARTITION-AROUND(A, p, r, x)          // partition around the pivot
18  // The rest is just like lines 3–9 of RANDOMIZED-SELECT.
19  k = q − p + 1
20  if i == k
21      return A[q]                           // the pivot value is the answer
22  elseif i < k
23      return SELECT(A, p, q − 1, i)
24  else return SELECT(A, q + 1, r, i − k)
```

Suppose $T(n)$ is the time complexity of SELECT.

Then, $T(n)$ is $\leq T(n/5) + T(7n/10) + \Theta(n)$.

Hence, $T(n) = \Theta(n)$.

# Practice problems

# Practice Problem 1

Show how to use SELECT as a subroutine to make quicksort run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

# Practice Problem 2

You have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

# Practice Problem 3

The $k$th ***quantiles*** of an $n$-element set are the $k - 1$ order statistics that divide the sorted set into $k$ equal-sized sets (to within 1). Give an $O(n \lg k)$-time algorithm to list the $k$th quantiles of a set.

# Practice Problem 4

Describe an $O(n)$-time 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 $S$.

# Practice Problem 5

Let $X[1:n]$ and $Y[1:n]$ be two arrays, each containing $n$ numbers already in sorted order. Give an $O(\lg n)$-time algorithm to find the median of all $2n$ elements in arrays $X$ and $Y$. Assume that all $2n$ numbers are distinct.

# Practice Problem 6

You are given a set of $n$ numbers, and you wish to find the $i$ largest in sorted order using a comparison-based algorithm. Describe the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of $n$ and $i$.

***a.*** Sort the numbers, and list the $i$ largest.

***b.*** Build a max-priority queue from the numbers, and call EXTRACT-MAX $i$ times.

***c.*** Use an order-statistic algorithm to find the $i$th largest number, partition around that number, and sort the $i$ largest numbers.

# Practice Problem 7

Professor Mendel has proposed simplifying RANDOMIZED-SELECT by eliminating the check for whether $i$ and $k$ are equal. The simplified procedure is SIMPLER-RANDOMIZED-SELECT.

```
SIMPLER-RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]        // 1 ≤ i ≤ r − p + 1 means that i = 1
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i ≤ k
6       return SIMPLER-RANDOMIZED-SELECT(A, p, q, i)
7  else return SIMPLER-RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

***a.*** Argue that in the worst case, SIMPLER-RANDOMIZED-SELECT never terminates.

***b.*** Prove that the expected running time of SIMPLER-RANDOMIZED-SELECT is still $O(n)$.