Durham University

Algorithms & Data Structures

Part – 3

Anish Jindal
anish.jindal@durham.ac.uk

Module Content

Part 1:

- Introduction and Pseudocode
- Arrays and Lists
- Stacks and Queues
- Hash Tables
- Recursion and Backtracking

Part 2:

- Asymptotic notations
- Sorting: InsertionSort, SelectionSort, MergeSort, QuickSort
- Recurrences
- Randomised QuickSort

Part 3:

- Algorithms for sorting, searching and selection
- Binary Search Trees
- AVL trees
- Heaps
- Lower bounds for sorting and selection

Part 4:

- Basic graph theory and Graph Isomorphism
- Computing shortest paths, Breadth First Search
- Graph Traversing through Depth First Search
- Minimum Spanning Trees

Durham University

Overview

- Topic 1- Algorithms for sorting, searching and selection
- Topic 2 - Binary Search Trees (BSTs)
- Topic 3 - AVL trees
- Topic 4 - Heaps
- Topic 5 - Lower bounds for sorting and selection

- Lecture material will be available on blackboard.
- Lectures will be recorded, so you have the option to listen again.

- Labs – Practical sessions to reinforce the lecture material (weeks 12-16).

Durham University

Further information

- **Assessment – Exam only.**
  - 2 hours - over summer
- **How to get a good grade?**
  - Attend all the lectures. The lecture notes are not a substitute to attendance.
  - If you missed lecture: (i) Read lecture notes (ii) ask for additional notes from peers (iii) watch recorded lectures.
  - Good to have a pen and paper (or a substitute) handy.
  - Follow the material and ask questions if not understood.
  - Attend revision lecture.
- **Office hours:** My office hour is Wednesday 2pm (in MCS 2065) or by appointment over email.

Let’s start

Have seen a good few sorting algorithms so far:

- Selection sort
- Insertion sort
- Merge sort
- Quicksort

Saw that their worst-case running times are somewhere between (asymptotically) $n \log n$ and n^2.

Can you tell?

Can, in fact prove that that's no coincidence ("that" being that apparently none beats n log n):

Theorem

For any **comparison-based** sorting algorithm **A** and any $n \in \mathbf{N}$ large enough there exists an input of length n that requires **A** to perform $\Omega(n \log n)$ comparisons.

This is an example of a general lower bound: no such algorithm, past, present or future, can consistently beat the stated bound.

We will see a proof of this theorem later.

Two observations:

1. Theorem talks about “comparison-based” algorithms. Vast majority of sorting algorithms are in that class (definitely all that we’ve seen so far).

2. Theorem says “there exists an input”. That means that for most inputs (possibly all but one!) of length n it may actually beat the bound, but that there must be at least one for which it does not.

Constraints on the problem

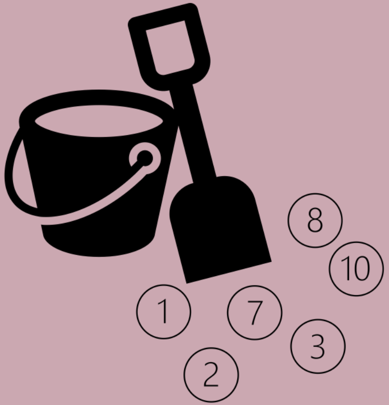
Suppose the values in the list to be sorted can repeat but the values have a limit (e.g., values are digits from 0 to 9)

Sorting, in this case, appears easier

Is it possible to come up with an algorithm better than O(n log n)?

- Yes
- Strategy will not involve comparisons

BucketSort



Bucket Sort

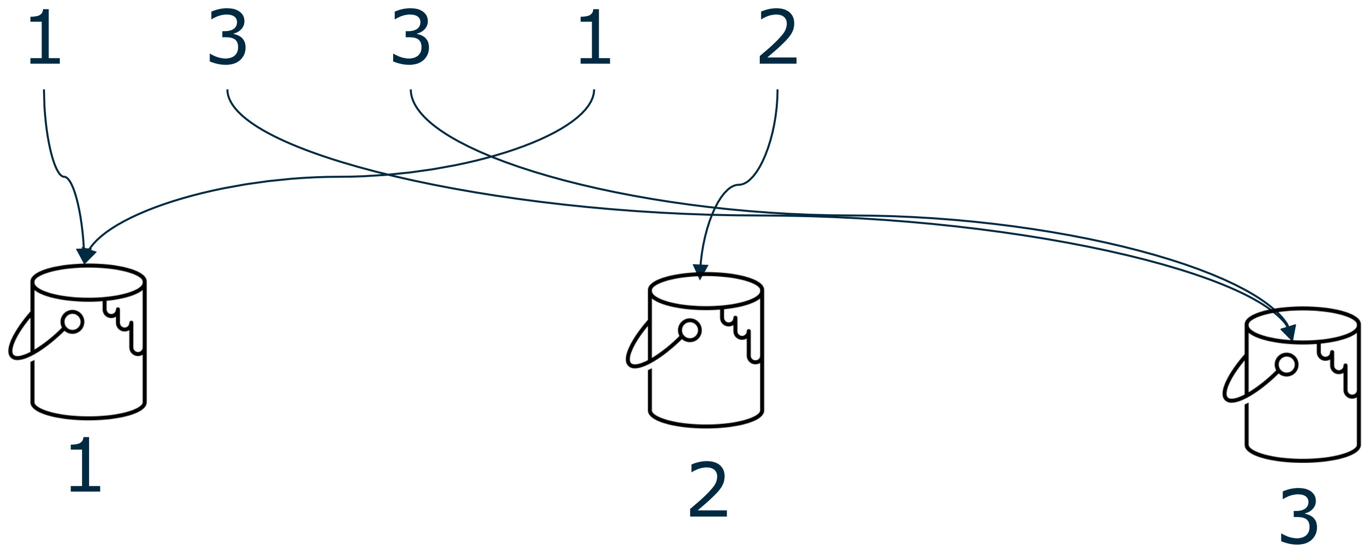
Consider the following set-up.

You are to sort n numbers, each taking a value in {0, . . . , k − 1} (the “range” of possible values is k).

If you were to use one of the known algorithms then that would give you a worst-case running time at least n log n (again, whatever algorithm you choose, at least one input of length n will make it take that many steps).

Specifically, they don’t care much about the range of the input items.

BucketSort



BucketSort

Idea: suppose the values are in the range 0..k-1; start with k empty *buckets* numbered 0 to k-1, scan the list and place element s[i] in bucket s[i], and then output the buckets in order.

Will need an array of buckets, and the values in the **list to be sorted** will be the indexes to the buckets

- No comparisons will be necessary

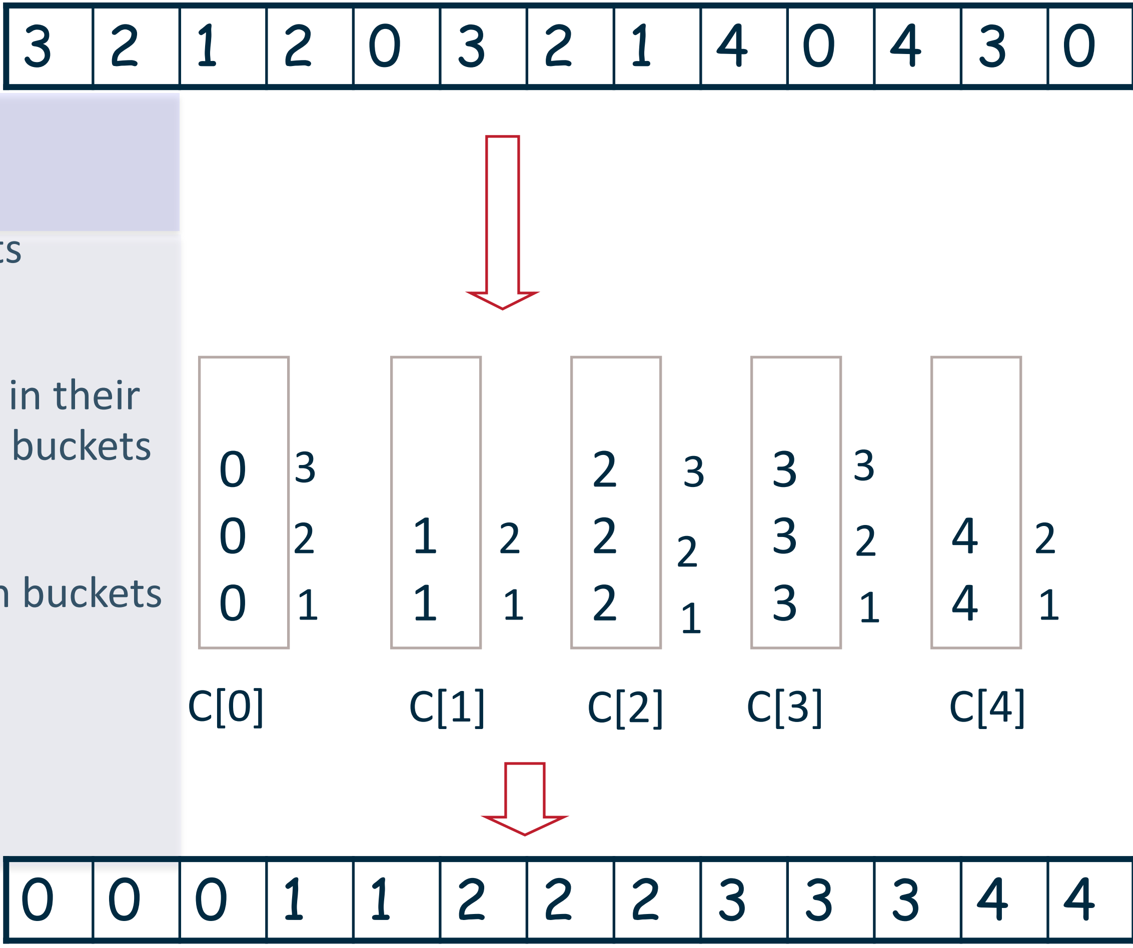
Algorithm

```
Algorithm BucketSort( S )
( values in S are between 0 and k-1 )
for j = 0 to k-1 do        // initialize k buckets
    b[j] = 0
end for
for i = 0 to n-1 do              // place elements in their
    b[S[i]] = b[S[i]]  + 1       // appropriate buckets
end for
i = 0
for j = 0 to k-1 do        // place elements in buckets
    for r = 1 to b[j] do         // back in S
      S[i] = j
      i = i + 1
    end for
end for
```

Durham University

Example

```
Algorithm BucketSort( S )
( values in S are between 0 and k-1 )
for j = 0 to k-1 do        // initialize k buckets
    b[j] = 0
end for
for i = 0 to n-1 do          // place elements in their
    b[S[i]] = b[S[i]]  + 1       // appropriate buckets
end for
i = 0
for j = 0 to k-1 do        // place elements in buckets
    for r = 1 to b[j] do        // back in S
      S[i] = j
      i = i + 1
    end for
end for
```



Durham University

Values/Entries

If we were sorting values, each bucket is just a counter that we increment whenever a value matching the bucket’s number is encountered

If we were sorting entries according to keys, then each bucket is a queue

- Entries are enqueued into a matching bucket
- Entries will be dequeued back into the array after the scan

Time complexity

Algorithm BucketSort(S)
(values in S are between 0 and k-1)

```
for j = 0 to k-1 do        // initialize k buckets          ] k
    b[j] = 0
end for
for i = 0 to n-1 do              // place elements in their ] n
    b[S[i]] = b[S[i]]  + 1       // appropriate buckets
end for
i = 0
for j = 0 to k-1 do        // place elements in buckets     ] n
    for r = 1 to b[j] do         // back in S
      S[i] = j
      i = i + 1
    end for
end for
```

Time complexity..

Bucket initialization: O(k)

From array to buckets: O(n)

From buckets to array: O(n)

- Even though this stage is a nested loop, notice that all we do is dequeue from each bucket until they are all empty –> n dequeue operations in all

- Time complexity is O (n + k) – worst case.
  - Since k will likely be small compared to n, bucket sort is O(n) in average case.

Durham
University

BucketSort..

It’s sometimes referred to as “sorting by counting” but more usually as “bucket sort” (or bin sort) because it simply chucks elements with key *i* into the *i*-th bucket, and then empties one bucket after another.

Its running time is O(n + k).

This means that if *k* is small, say o(n log n), the overall running time of bucket sort is o(n log n)!!!

Apparently beating our lower bound (Remember the theorem) !!! Why?

BucketSort…

The reason, of course, is that the theorem simply doesn’t apply – bucket sort doesn’t do any comparisons!

Also it’s a bit cheating: if k gets really large then the running time is dominated by O(k).

If $k = \omega(n \log n)$, then it’s worse that that of MergeSort, and

if $k = \omega(n^2)$, it’s worse than that of any other sorting algorithm that we’ve seen.

BucketSort…

Finally, what if we don't sort numbers but arbitrary things?

Easy: Those things normally come with a numerical key (obtain any way, perhaps by hashing), and you use that for sorting.

Problem is, when we dump the buckets in phase 2, what are we to do? Simple counters don't seem to work any more???

Again, easy: each bucket will be made a queue, and whenever an item's key is, say, k, then in phase 1 the item will be appended to the k-th queue, and in phase 2 each queue will be dequeued until empty.

Test your knowledge!

Solve the following using bucket sort:

1,3,4,1,5,6,8,4,7,3,6,8

Durham University

Can we perform bucket sort on any array of (non-negative) integers?

- Yes, but note that the number of buckets will depend on the maximum integer value

If you are sorting 1000 integers and the maximum value is 999999, you will need 1 million buckets!

- Time complexity is not really O(n) because k is much > than n. Actual time complexity is O(k)

Can we do better?

Durham
University

RadixSort

RadixSort

Obvious drawback: if range of items is large, then we need a large number of buckets (counters or queues).

Improvement: don’t look at the item values but “one level below”

Consider again subset of non-negative integers as range, and furthermore consider decimal representation (e.g., decimal digits).

Use this strategy when the keys are integers, and there is a reasonable limit on their values

- Number of passes (bucket sort stages) will depend on the number of digits in the maximum value

RadixSort

Idea of RadixSort:

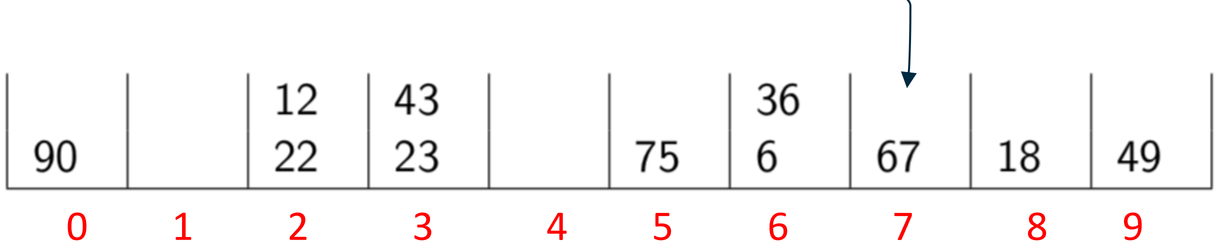
- have as many buckets as you’ve got different digits, that is, for base-10 you’ll have 10 of them (max)
    - If maximum value is 999999, only ten buckets (not 1 million) will be necessary
- repeatedly bucket-sort by given digit
- number of rounds will depend on values (the longer the base-10 representations, the more rounds), but number of buckets only depends on number of different digits

RadixSort: example

Consider this input:

67, 23, 90, 6, 43, 22, 18, 75, 49, 12, 36

First pass (right-most digit) gives the following buckets:



. . . which, in turn, gives the “new” array

90, 22, 12, 23, 43, 75, 6, 36, 67, 18, 49

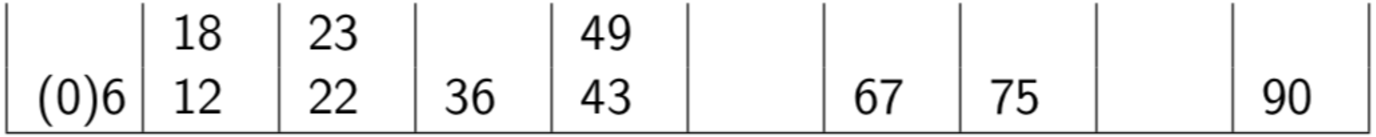
Why start from right-most digit?

RadixSort: example

Second round now works on this new array

90, 22, 12, 23, 43, 75, 6, 36, 67, 18, 49

but now considers the next digit from the right (here: left-most):



	18	23		49					
(0)6	12	22	36	43		67	75		90

Dump this table and get

6, 12, 18, 22, 23, 36, 43, 49, 67, 75, 90

Voila!

RadixSort: example together

67	23	90	6	43	22	18	75	49	12	36

sort by rightmost digit (first round)

90	22	12	23	43	75	6	36	67	18	49

sort by leftmost digit (second round)

6	12	18	22	23	36	43	49	67	75	90

Durham University

Example: Three digits

0 3 2	0 3 1	0 1 5	0 1 5
2 2 4	0 3 2	0 1 6	0 1 6
0 1 6	2 5 2	1 2 3	0 3 1
0 1 5	1 2 3	2 2 4	0 3 2
0 3 1	2 2 4	0 3 1	1 2 3
1 6 9	0 1 5	0 3 2	1 6 9
1 2 3	0 1 6	2 5 2	2 2 4
2 5 2	1 6 9	1 6 9	2 5 2

Durham University

RadixSort..

. . . works if the BucketSort phases are stable

. . . means, work done in previous rounds isn't being destroyed

- Suppose there are two elements whose first digit is the same; for example, 43 & 49
- If 43 occurs before 49 in the array prior to the sorting stage, 43 should occur before 49 in the resulting array

• If there is a fixed number **d** of bucket sort stages (six stages in the case where the maximum value is 999999), then radix sort is O(d n)

- There are **d** bucket sort stages, each taking O(n) time

• Overall, time complexity is O(d n), where **d** is the number of digits.

RadixSort: Time complexity

Compare to plain BucketSort for range {0,…,k-1}

- RadixSort $d=\log_{10}k$, giving O(n log k).
- BucketSort O(n+K)

Not immediately clear which is better in terms of time (RadixSort clearly better w.r.t space whenever K non-trivial).

Examples:

- if K = n then O(n log n) vs O(n) → BucketSort wins
- if K = n^2 then O(n log n) vs O(n^2) → RadixSort wins
- if K = 2^n then O(n^2) vs O(2^n) → RadixSort wins

Anyway, if K is constant then both are linear time (Θ (n)).

Durham University

Test your knowledge!

Radix sort for alphabets:

bcad, cadb, dabc, bdac, acbd, abcd, badc

Durham
University

Searching and Selecting

Searching

Suppose you're given n numbers in array A[1 . . . n]

Further suppose numbers are in **sorted order**

To make things slightly simpler, assume the *n* numbers are pairwise distinct, that is, no two are the same

Finally, assume you're also given a number *x* that's equal to one of the n numbers above, that is,

$$\exists \text{ index } i \in \{1,\ldots,n\} \text{ such that } x = A[i]$$

For instance,

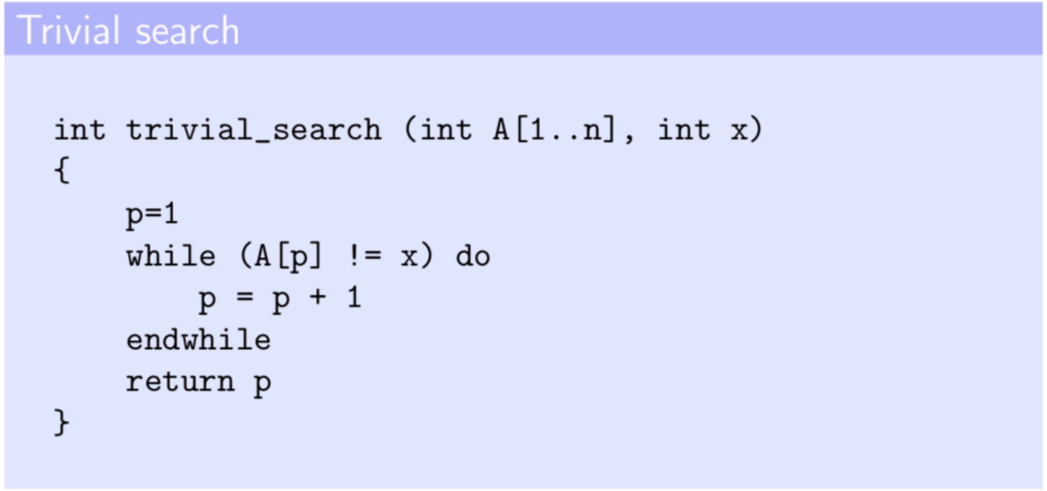
$$n = 8, \qquad A[1\ldots8] = (2,3,5,7,11,13,17,19), \qquad x = 17$$

Goal: devise algorithm that finds position *p* of *x* in A

(in example, p = 7 because A[7] = 17 = x)

Trivial solution

Scan array left to right, and stop as soon as value x found



Trivial search

```
int trivial_search (int A[1..n], int x)
{
    p=1
    while (A[p] != x) do
        p = p + 1
    endwhile
    return p
}
```

Notice: absolutely need fact that x is equal to one of the A[i].

Might also ask question: what if this isn't necessarily true, how would you find position *p* where *x* **would have to be inserted** in order to maintain overall sorted order?

Durham University

Binary Search

Clever(er) solution: (Recursive) Binary Search

Same assumptions as before.

1. Peek right into the middle of the given array, at position

$p = \lceil n/2 \rceil$

2. If **A[p] = x** then we're lucky & done, and can return p

3. Otherwise, **if x is greater than A[p]**, we may focus our search on stuff **to the right** of A[p] and may completely ignore anything to its left. E.g.,

n=8, [n/2]=4, x =17

$$A[1\ldots 8] = (\underbrace{2, 3, 5}_{\text{ignore}}, \boxed{7}, \underbrace{11, 13, \boxed{17}, 19}_{\text{search here}})$$

That's because x is already bigger than A[p] and hence must be **even** bigger than A[1...p−1]
Notice how the problem size got smaller!

Hence, if the "top-level" call was **search(A,1,n,x)** for "search, within array A, between the indices of 1 and n, for element x", we could now recursively call from within this function search, using the updated indices, like so:

search(A,p+1,n,x)

Durham
University

4. Likewise, **if x is smaller than A[p],** we may focus our search on stuff to the left of A[p] and may completely ignore anything to its right. E.g.,

n=8, [n/2]=4, x =3

$$A[1\ldots 8] = (\underbrace{2, \boxed{3}, 5}_{\text{search here}}, \boxed{7}, \underbrace{11, 13, 17, 19}_{\text{ignore}})$$

That's because x is already smaller than A[p] and hence must be even smaller than A[p+1...n] Notice how the problem size got smaller!

Hence, if the "top-level" call was **search(A,1,n,x)** for "search, within array A, between the indices of 1 and n, for element x", we could now recursively call from within this function search, using the updated indices, like so:

search(A,1,p-1,x)

Recursive binary search

```
int search (int A[1..n], int left, int right, int x)
{
  if (right == left and A[left] != x)
      handle error; leave function

  p = middle-index between left and right

  if (A[p] == x) then
      return p

  // here come the recursive calls (if x not yet found)
  if (x > A[p]) then
      return search(A,p+1,right,x)  // in right half
  else // x < A[p]
      return search(A,left,p-1,x)  // in left half
}
```

Initial call would be “search(A,1,n,x)”

This is called **binary search** because in each (unsuccessful) step it at least halves the search space (we end up with the remaining, interesting part of the array in which x is hiding)

It’s an example of the “problem solving paradigm” usually referred to as **divide & conquer**.

A note of interest: the “linear search” requires $O(n)$ comparisons in the worst case, whereas for binary search we have the recurrence

$$T(n) = T(n/2) + O(1) = O(\log n),$$

which, for large n, is an awful lot quicker

Example:

Find the value 33 from the sorted array as below:

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97

Changing the model: Selection

So far, had assumed that

- input is **sorted**, and
- we're looking for the **position** p of an element of given **value** x

Now we're changing the set-up:

- input is **unsorted**, and
- we're looking for **value** of the i-th smallest element in the input

  (clearly, if input were sorted it'd be trivial: return i-th from left)

This problem is called **Selection**. How can we go about solving it?

Easy if we're looking for smallest/largest (how?), 2nd-smallest/largest (how?), k-th smallest/largest for some constant k (how?).

What if i-th element is, say, median (n/2-nd smallest)? √n-th smallest? Ideas?

Yes, we can sort the input, say in time O(n log n), and **then** simply return the i-th element from the left.

However, is it **clear** that we can’t solve the problem quicker, in the worst case?

No, actually it used to be far from clear for many, many years.

Let’s see some clever solutions.

Durham University

QuickSelect

QuickSelect

QuickSelect is to selecting what QuickSort is to sorting. Much the same idea.

- recursive
- Partition() function for selecting pivot and partitioning into LOW and HIGH (those smaller/bigger than pivot)

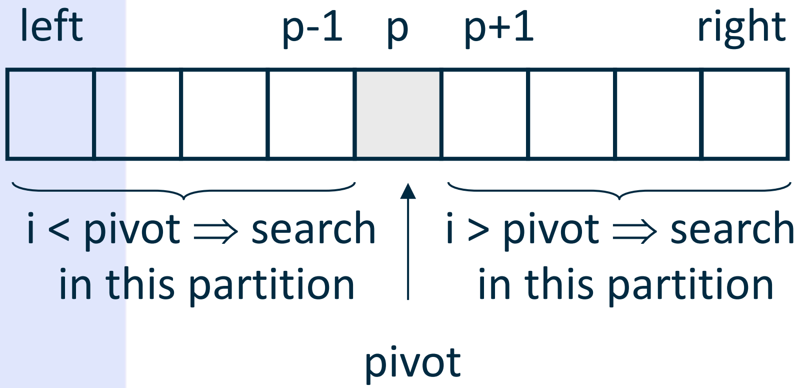
- not two recursive calls to sort but now only one:

- diving into the part (LOW or HIGH) where we know the *i*-th smallest element is to be found

- Know that after partitioning, pivot is in correct position w.r.t. overall sortedness, so can simply compare pivot position after partitioning with sought index *i*

QuickSelect (int A[1...n], int left, int right, int i)

```
1:  if (left == right) then
2:      return A[left]
3:  else
4:      // rearrange/partition in place
5:      // return value "pivot" is index of pivot element
6:      // in A[] after partitioning
7:      pivot = Partition (A, left, right)
8:      // Now:
9:      // everything in A[left...pivot-1] is smaller than pivot
10:     // everything in A[pivot+1...right] is bigger than pivot
11:     // the pivot is in correct position w.r.t. sortedness
12:     if ($i$ == $pivot$) then
13:         return $A[i]$
14:     else if ($i < pivot$) then
15:         return QuickSelect (A, left, pivot-1, i)
16:     else        // $i > pivot$
17:         return QuickSelect (A, pivot+1, right, i)
18:     end if
19: end if
```


left
p-1
p
p+1
right
i < pivot ⇒ search in this partition
i > pivot ⇒ search in this partition
pivot

Example:

Use QuickSelect to select 5th smallest element

5	8	1	3	7	9	2

What about performance?

Let's start with the bad. If things go wrong, i.e., input is in sorted order and you always pick the right-most element as pivot, then just as slow as QuickSort is when it's slow

→ the part (LOW/HIGH) that's being recursed into is just one element smaller than the current input is

→ $T(n) = T(n-1) + O(n)$, which means $T(n) = O(n^2)$

So, **actually** can be rather a lot slower than sorting with e.g. MergeSort

Durham University

What about performance?

Can show that if one chooses the pivot **at random** from the current sub-problem, between the indices left and right, then the **expected** running time of this **randomised QuickSelect** is indeed very good, namely O(n)

Clearly, can’t hope for better than linear! Why, btw?

But do note that this bound on the expectation is just that: with (certainly extremely small, but positive) probability it may still take $O(n^2)$ many steps

Durham University

Median-of-medians

Median-of-Medians

Was long unknown whether **worst case linear-time** deterministic selection in **unsorted** data is possible.

Recall: randomised QuickSelect will be fine almost all the time, but the O(n) is only in expectation

Answer: **yes** - the **Median-of-Medians** algorithm

- published by Blum, Floyd, Pratt, Rivest and Tarjan in 1973

  (some famous names in Computer Science!)
- can be thought of as QuickSelect with a good deterministic pivot-selection strategy
- is not singly but **doubly** recursive!!!
- has **guaranteed** linear time complexity for finding the (any) i-th smallest element of a sequence of *n* unsorted items

Durham University

Median-of-Medians: the algorithm

Finding pivot

```
SELECT($i$, $n$)
1. Divide the $n$ elements into groups of $5$. Find the
   median of each $5$-element group by rote.
2. Recursively SELECT the median $x$ of the $\lfloor n/5 \rfloor$ group
   medians to be the pivot.
3. Partition around the pivot $x$. Let $k$ = rank($x$).
4. if $i = k$ then return $x$
   elseif $i < k$
       then recursively SELECT the $i$th smallest
           element in the lower part
       else recursively SELECT the $(i-k)$th smallest
           element in the upper part
```

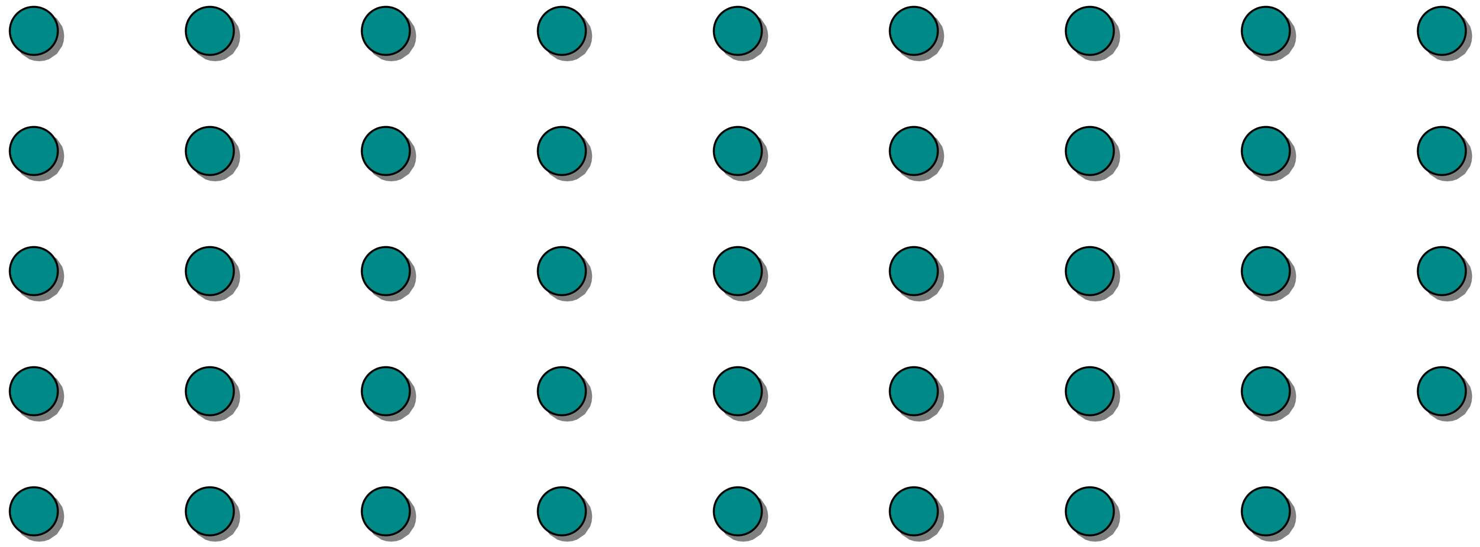
Same as Quick-Select

Example

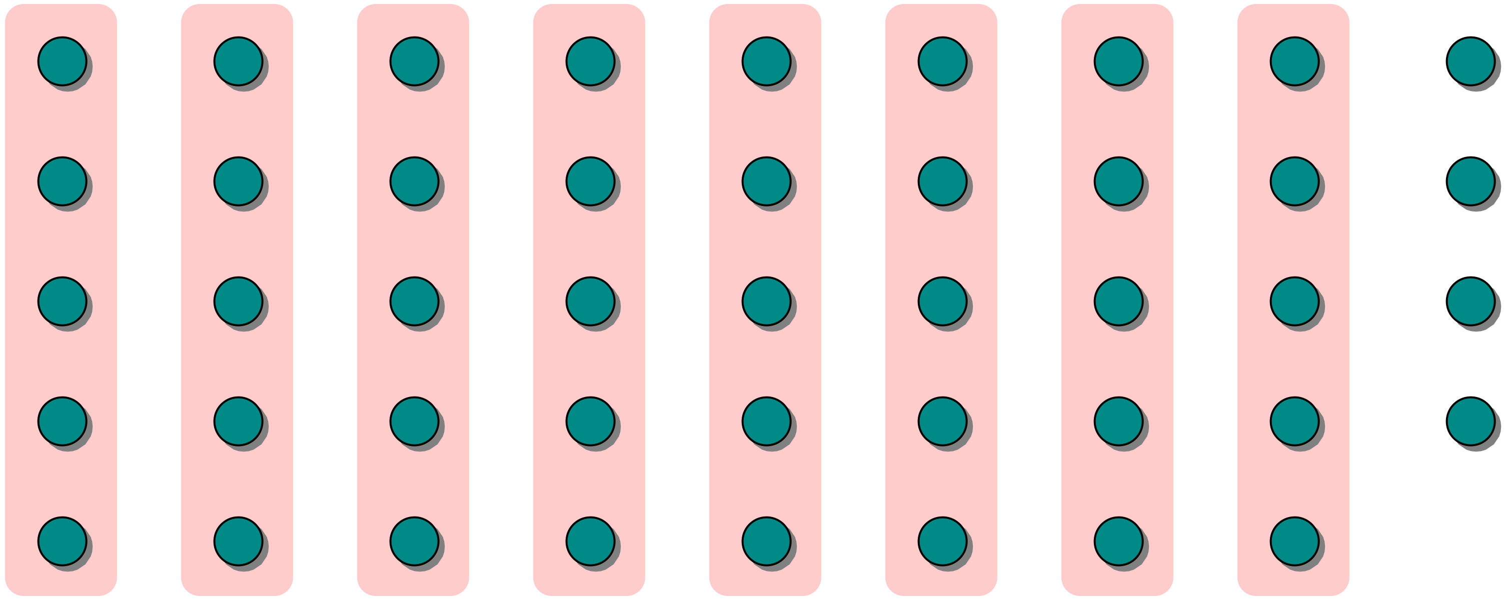
Find the 11th smallest element in array:

A = {12, 34, 0, 3, 22, 4, 17, 32, 3, 28, 43, 82, 25, 27,
34, 2 ,19 ,12 ,5 ,18 ,20 ,33, 16, 33, 21, 30, 3, 47}

Choosing the pivot

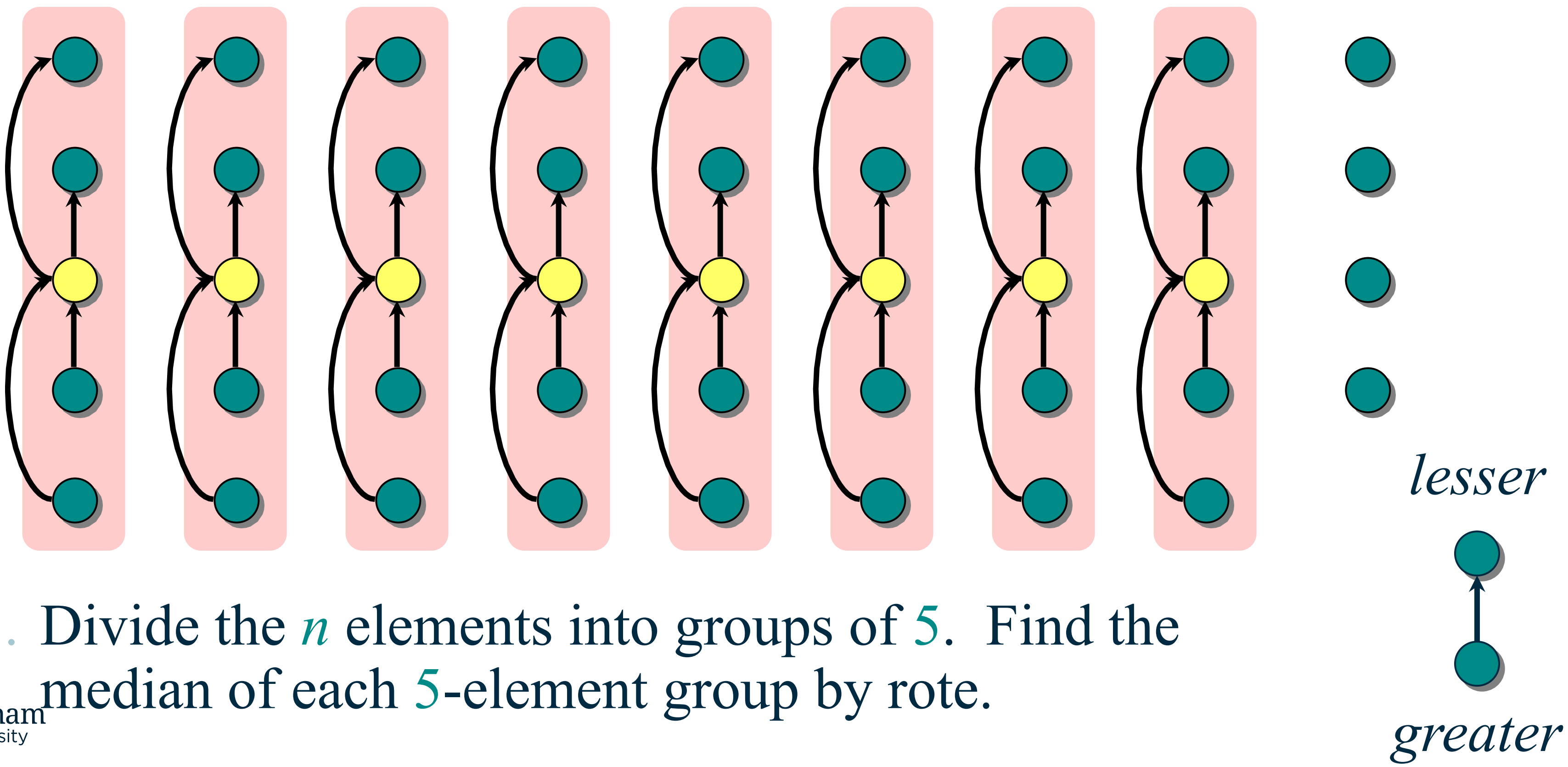


Choosing the pivot



1. Divide the n elements into groups of 5.

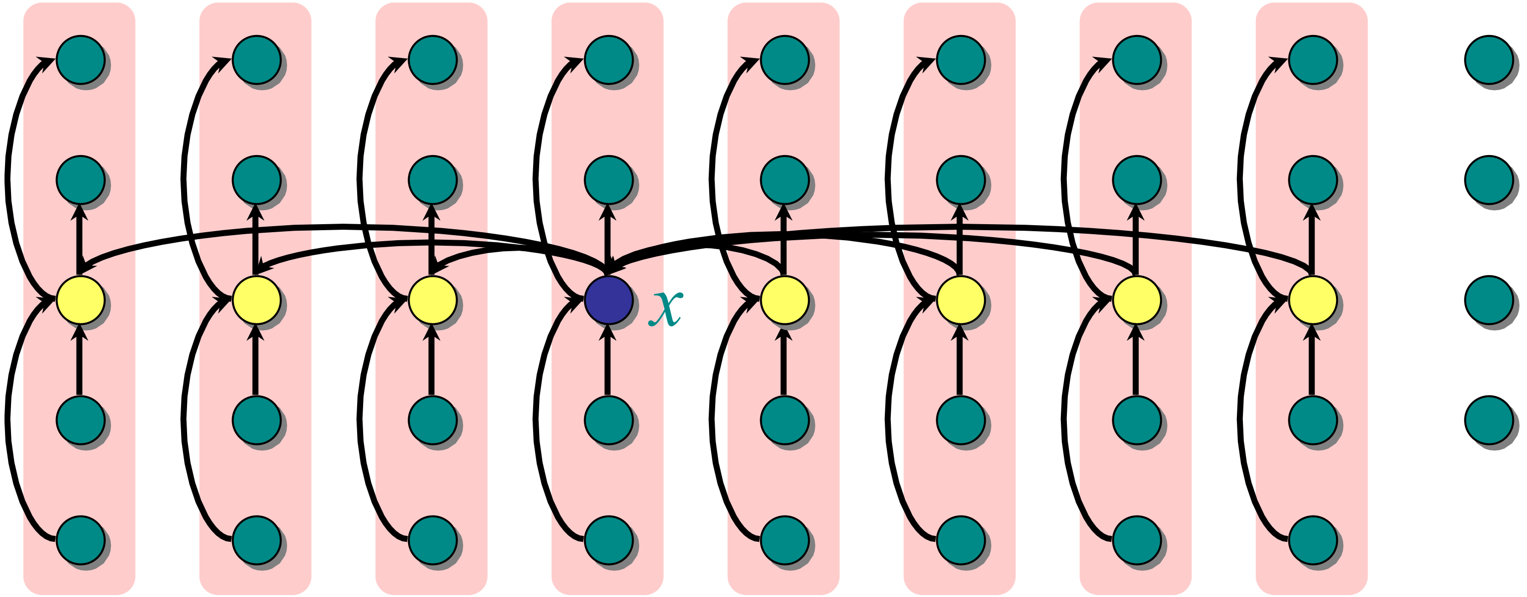
Choosing the pivot

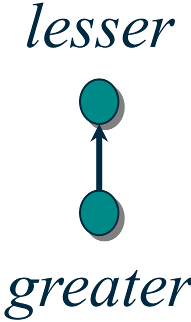


1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.

Durham University

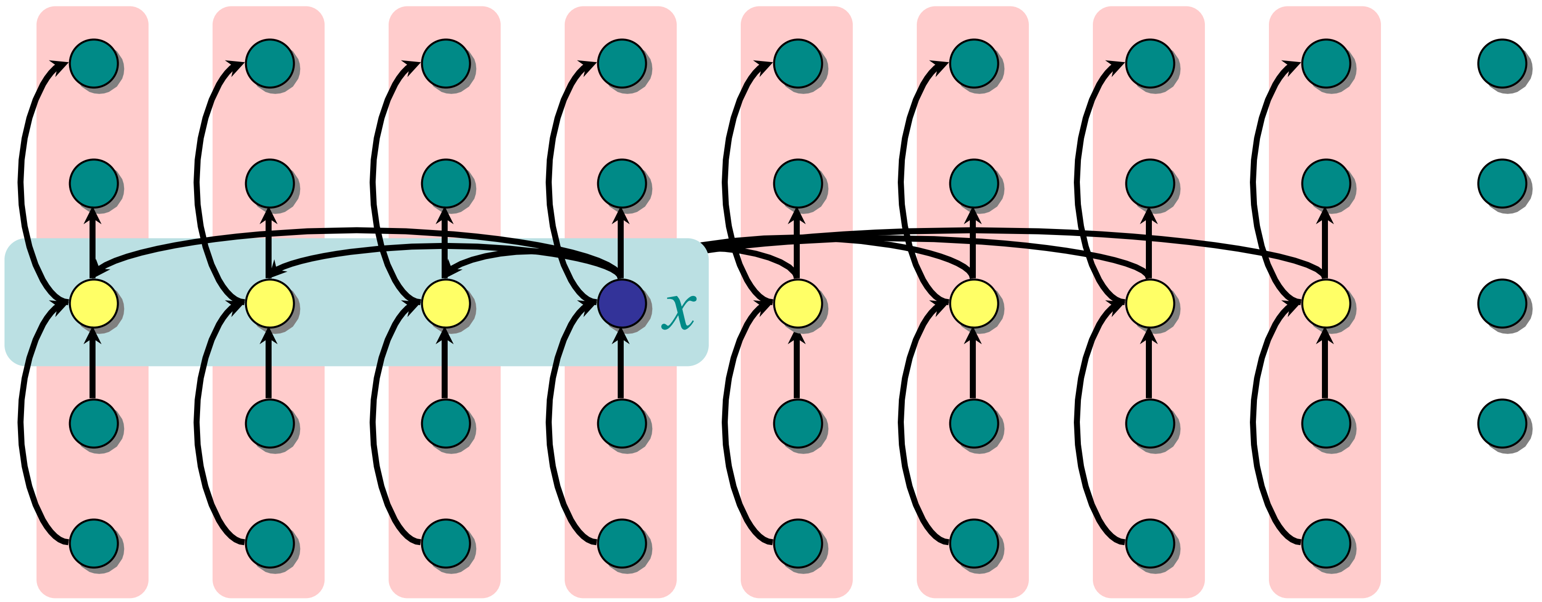
Choosing the pivot

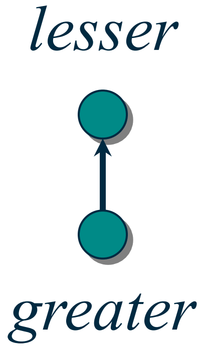




1. Divide the n elements into groups of 5. Find the median of each 5-element group by rote.
2. Recursively SELECT the median x of the $\lfloor n/5 \rfloor$ group medians to be the pivot.

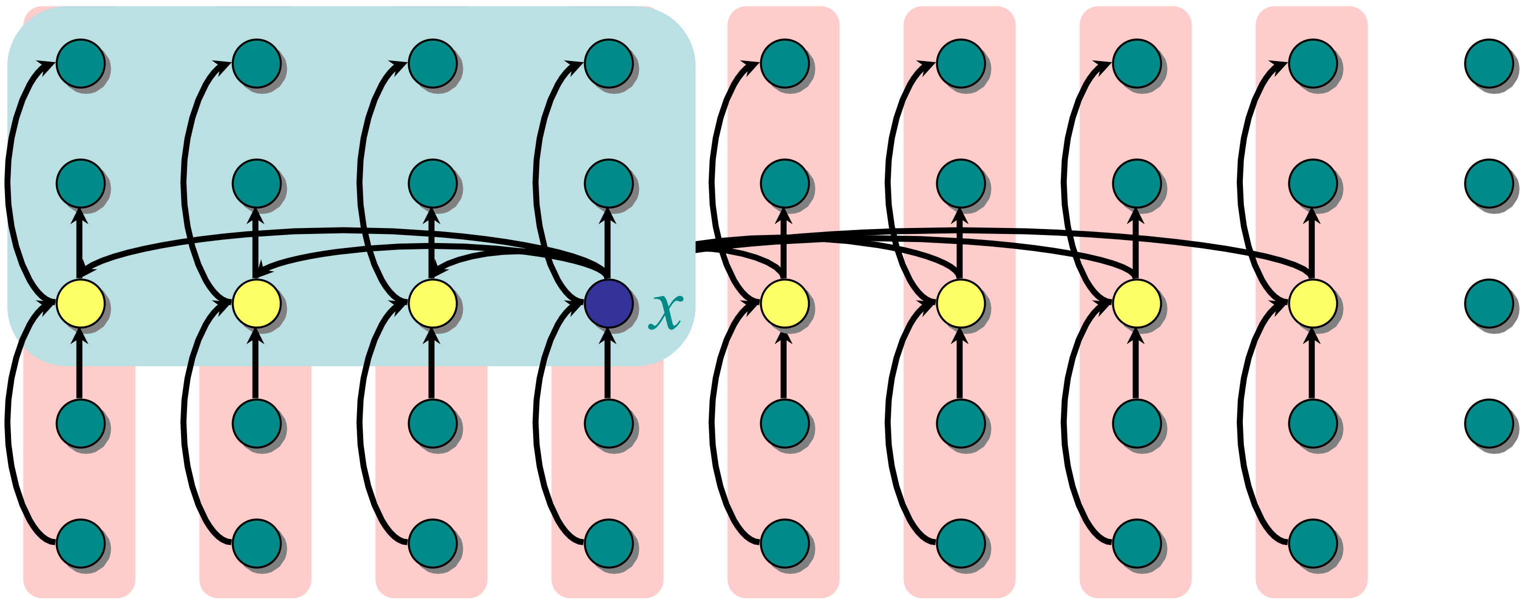
Analysis

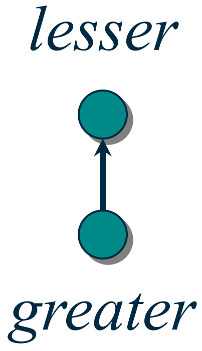




At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor$
$= \lfloor n/10 \rfloor$ group medians.

Analysis



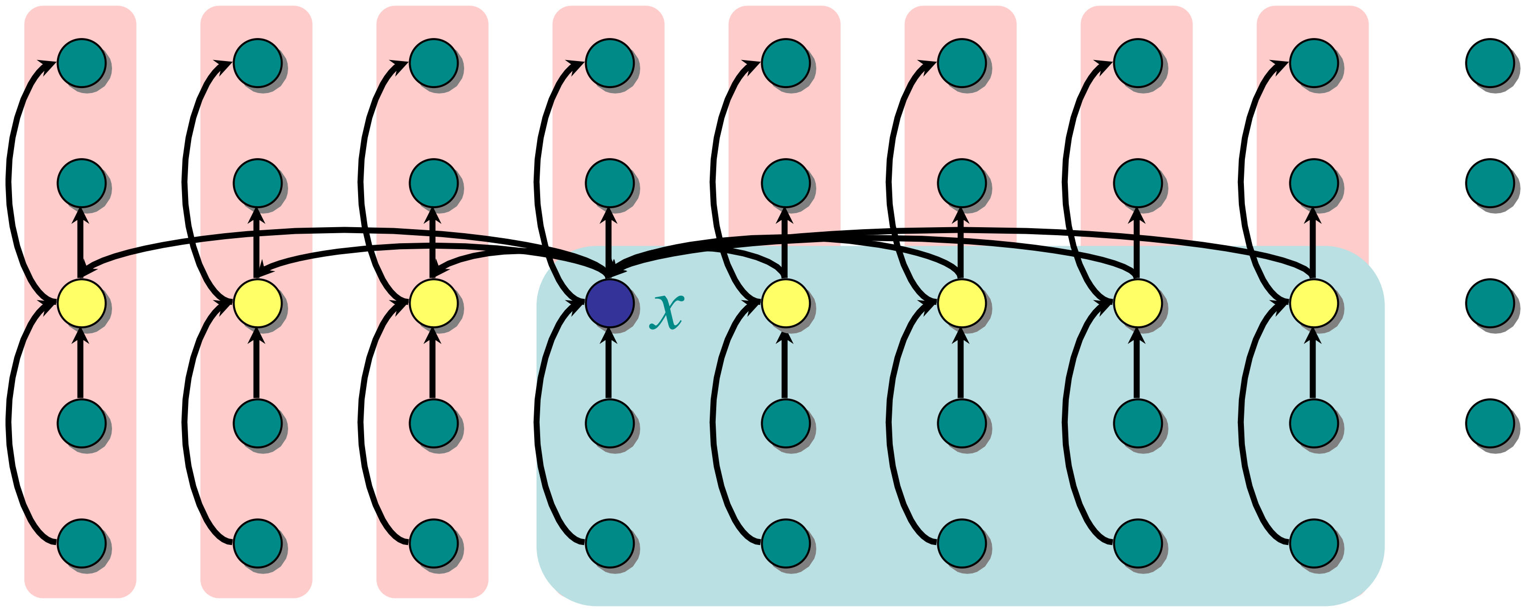


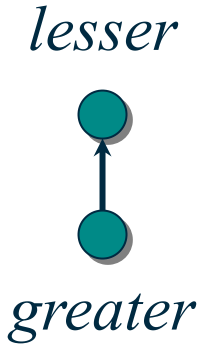
At least half the group medians are $\le x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\le x$.

(Assume all elements are distinct.)

Analysis





At least half the group medians are $\leq x$, which is at least $\lfloor \lfloor n/5 \rfloor /2 \rfloor = \lfloor n/10 \rfloor$ group medians.

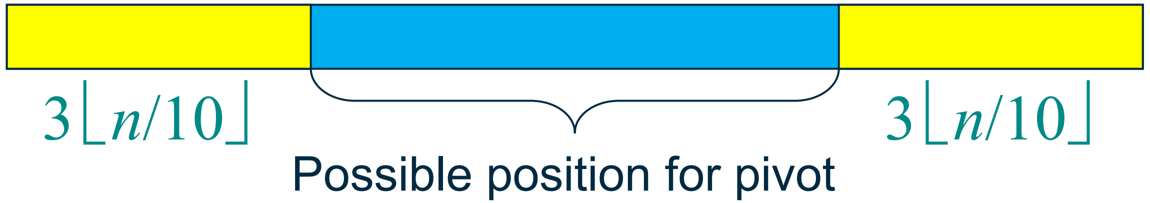
- Therefore, at least $3\lfloor n/10 \rfloor$ elements are $\leq x$.
- Similarly, at least $3\lfloor n/10 \rfloor$ elements are $\geq x$.

Analysis

Need "at most" for worst-case runtime

- At least $3\lfloor n/10 \rfloor$ elements are $\leq x$
  $\Rightarrow$ at most n-$3\lfloor n/10 \rfloor$ elements are $\geq x$
- At least $3\lfloor n/10 \rfloor$ elements are $\geq x$
  $\Rightarrow$ at most n-$3\lfloor n/10 \rfloor$ elements are $\leq x$

- The recursive call to SELECT in Step 4 is executed recursively on at most n-$3\lfloor n/10 \rfloor$ elements.



Analysis

- Use fact that $\lfloor a/b \rfloor > a/b-1$
- $n-3\lfloor n/10 \rfloor < n-3(n/10-1) \le 7n/10 + 3$

  $\le 3n/4 \text{ if } n \ge 60$

- The recursive call to SELECT in Step 4 is executed recursively on at most $7n/10+3$ elements.

60:

$7\text{x}60/10 + 3 \le 3\text{x}60/4$

$42 + 3 \le 180/4$

$45 \le 45$

70:

$7\text{x}70/10 + 3 \le 3\text{x}70/4$

$49 + 3 \le 210/4$

$52 \le 52.5$

80:

$7\text{x}80/10 + 3 \le 3\text{x}80/4$

$56 + 3 \le 240/4$

$59 \le 60$

```
$T(n)$          SELECT$(i, n)$
$O(n)$          1. Divide the $n$ elements into groups of 5. Find
                   the median of each 5-element group by rote.
$T(n/5)$        2. Recursively SELECT the median $x$ of the $\lfloor n/5 \rfloor$
                   group medians to be the pivot.
$O(n)$          3. Partition around the pivot $x$. Let $k = \text{rank}(x)$.
                4. if $i = k$ then return $x$
                   elseif $i < k$
                         then recursively SELECT the $i$th
$T(7n/10$                       smallest element in the lower
$+3)$                      part
                         else recursively SELECT the $(i–k)$th
                                smallest element in the upper
                           part
```

Durham University

Solving the relation

$$T(n) = T\left(\frac{1}{5}n\right) + T\left(\frac{7}{10}n + 3\right) + n$$

To show: *$T(k) \le ck$ for all $k < n$, $n > 0$*

Solving by substitution:

$$\begin{aligned} T(n) &\le c(n/5) + c(7n/10 + 3) + n \\ &\le cn/5 + 3cn/4 + n && \text{if } n \ge 60 \\ &= 19cn/20 + n \\ &\le cn - (cn/20 - n) \\ &\le cn && \text{if } c \ge 20 \text{ and } n \ge 60 \end{aligned}$$

What happens when dividing into groups of 3 or 7?

QuickSelect vs Median-of-medians

- QS is singly recursive – so less work in each iteration than MoM
- QS can have more iterations than MoM (if we are unlucky with pivot-selection)
- In practice, the choice of selection algorithm depends on circumstances.
- (Randomized) QuickSelect is often used.
    - Because its bad behaviour is rare, which is often tolerable.
- MoM is used when guaranteed good behaviour is absolutely needed.

Durham University

Durham University

Thank you!