CE100 Algorithms and Programming II

Matrix Multiplication / Quick Sort

Author: Asst. Prof. Dr. Uğur CORUH

## CE100 Algorithms and Programming II

## Week-3 (Matrix Multiplication/ Quick Sort)

#### Spring Semester, 2021-2022

Download [DOC-PDF](ce100-week-3-matrix.en.md_doc.pdf), [DOC-DOCX](ce100-week-3-matrix.en.md_word.docx), [SLIDE](ce100-week-3-matrix.en.md_slide.pdf), [PPTX](ce100-week-3-matrix.en.md_slide.pptx)

## Matrix Multiplication / Quick Sort

## Outline (1)

* Matrix Multiplication
  + Traditional
  + Recursive
  + Strassen

## Outline (2)

* Quicksort
  + Hoare Partitioning
  + Lomuto Partitioning
  + Recursive Sorting

## Outline (3)

* Quicksort Analysis
  + Randomized Quicksort
  + Randomized Selection
    - Recursive
    - Medians

## Matrix Multiplication (1)

* **Input:**
* **Output:**

## Matrix Multiplication (2)



alt:“alt” height:450px center

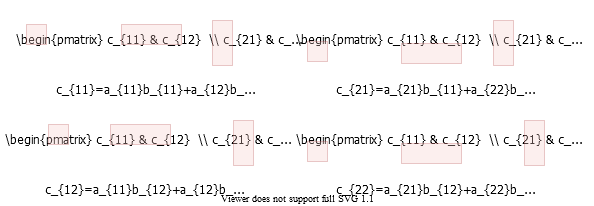
## Matrix Multiplication: Standard Algorithm

Running Time:

for i=1 to n do  
 for j=1 to n do  
 C[i,j] = 0  
 for k=1 to n do  
 C[i,j] = C[i,j] + A[i,k] + B[k,j]  
 endfor  
 endfor  
endfor

## Matrix Multiplication: Divide & Conquer (1)

**IDEA:** Divide the matrix into matrix of submatrices.



alt:“alt” height:450px center

## Matrix Multiplication: Divide & Conquer (2)

## Matrix Multiplication: Divide & Conquer (3)

MATRIX-MULTIPLY(A, B)  
 // Assuming that both A and B are nxn matrices  
 if n == 1 then   
 return A \* B  
 else   
 //partition A, B, and C as shown before  
 C[1,1] = MATRIX-MULTIPLY (A[1,1], B[1,1]) +   
 MATRIX-MULTIPLY (A[1,2], B[2,1]);   
  
 C[1,2] = MATRIX-MULTIPLY (A[1,1], B[1,2]) +   
 MATRIX-MULTIPLY (A[1,2], B[2,2]);   
  
 C[2,1] = MATRIX-MULTIPLY (A[2,1], B[1,1]) +   
 MATRIX-MULTIPLY (A[2,2], B[2,1]);  
  
 C[2,2] = MATRIX-MULTIPLY (A[2,1], B[1,2]) +   
 MATRIX-MULTIPLY (A[2,2], B[2,2]);  
 endif   
  
 return C

## Matrix Multiplication: Divide & Conquer Analysis

* recursive calls
* each problem has size
* Submatrix addition

## Matrix Multiplication: Solving the Recurrence

* + ,
* Case 1:

Similar with ordinary (iterative) algorithm.

## Matrix Multiplication: Strassen’s Idea (1)

Compute using recursive multiplications.

In normal case we need as below.

## Matrix Multiplication: Strassen’s Idea (2)

* **Reminder:**
  + Each submatrix is of size
  + Each add/sub operation takes time
* Compute using recursive calls to matrix-multiply

## Matrix Multiplication: Strassen’s Idea (3)

* How to compute using ?

## Matrix Multiplication: Strassen’s Idea (4)

* recursive multiply calls
* add/sub operations

## Matrix Multiplication: Strassen’s Idea (5)

e.g. Show that :

## Strassen’s Algorithm

* **Divide:** Partition and into submatrices. Form terms to be multiplied using and .
* **Conquer:** Perform multiplications of submatrices recursively.
* **Combine:** Form using and on submatrices.

**Recurrence:**

## Strassen’s Algorithm: Solving the Recurrence (1)

* + ,
* Case 1:

so

or use https://www.omnicalculator.com/math/log

## Strassen’s Algorithm: Solving the Recurrence (2)

* The number may not seem much smaller than
* But, it is significant because the difference is in the exponent.
* Strassen’s algorithm beats the ordinary algorithm on today’s machines for or so.
* Best to date: (of theoretical interest only)

## Matrix Multiplication Solution Faster Than Strassen’s Algorithm

* In 5 Oct. 2022 new paper published
  + [Discovering faster matrix multiplication algorithms with reinforcement learning | Nature](https://www.nature.com/articles/s41586-022-05172-4)
  + [GitHub - deepmind/alphatensor](https://github.com/deepmind/alphatensor)
  + [Article](files/s41586-022-05172-4.pdf)
  + [Discovering novel algorithms with AlphaTensor](https://www.deepmind.com/blog/discovering-novel-algorithms-with-alphatensor)

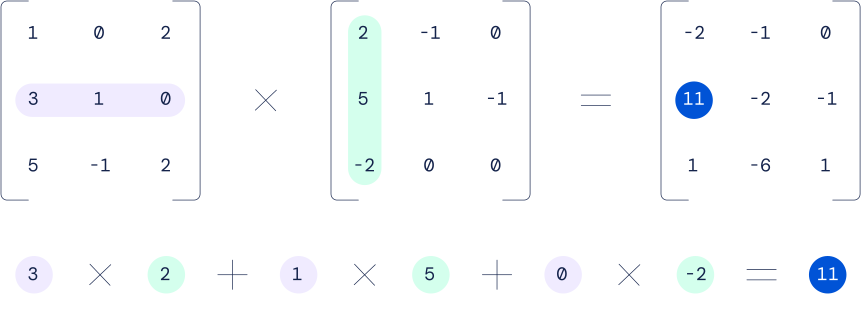
## Matrix Multiplication Solution Faster Than Strassen’s Algorithm

For example, if the traditional algorithm taught in school multiplies a 4x5 by 5x5 matrix using 100 multiplications, and this number was reduced to 80 with human ingenuity, AlphaTensor has found algorithms that do the same operation using just 76 multiplications.

## Matrix Multiplication Solution Faster Than Strassen’s Algorithm

[center h:450px]

## Standard Multiplication



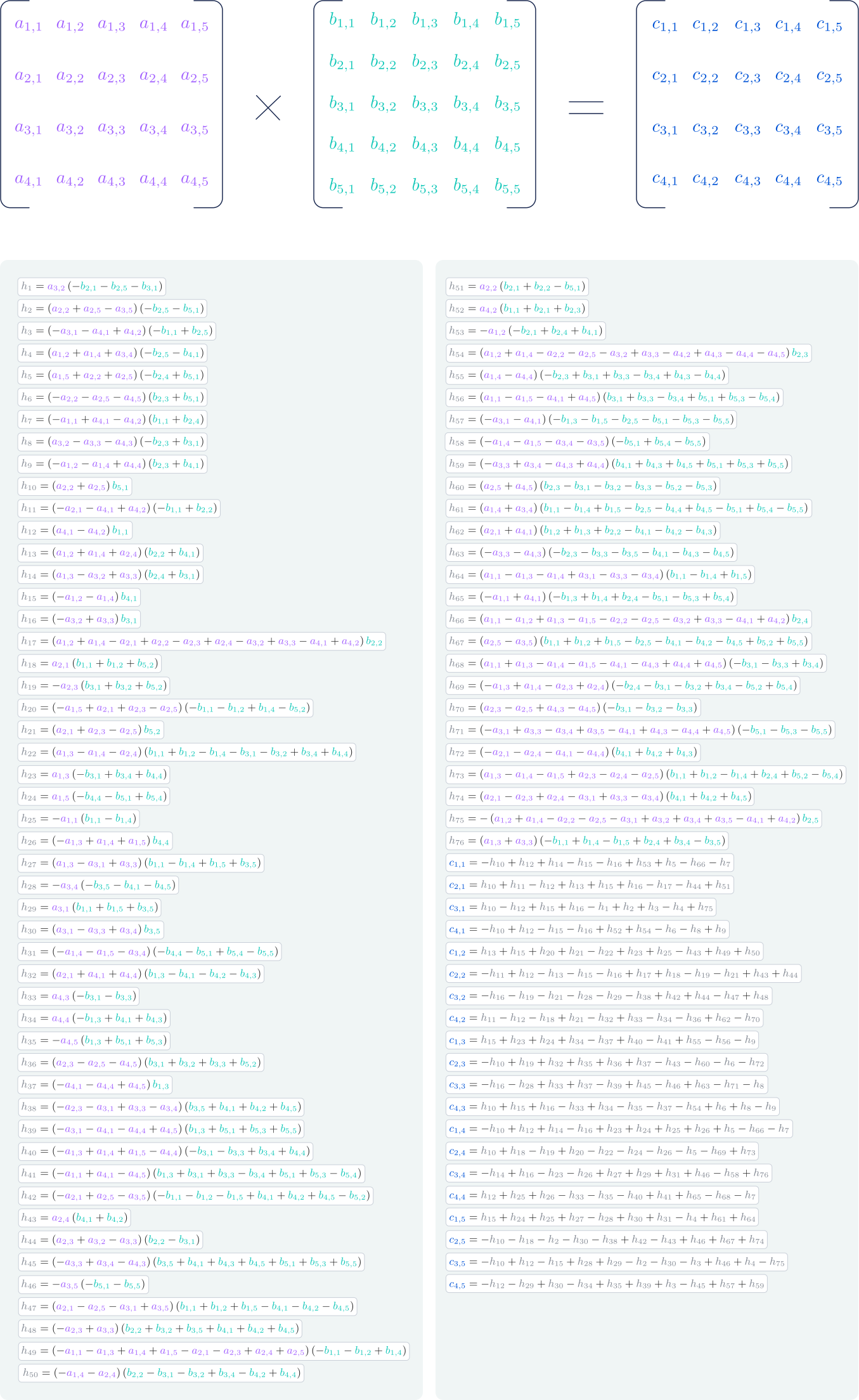
center h:400px

## Standard and Strassen Comparison



bg right h:650px

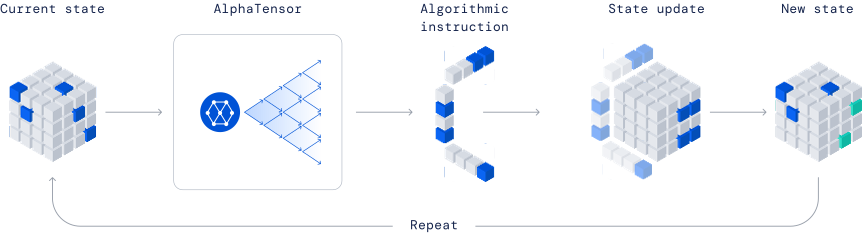
## Improved Solution



bg right h:660px

## How it’s done

“Single-player game played by AlphaTensor, where the goal is to find a correct matrix multiplication algorithm. The state of the game is a cubic array of numbers (shown as grey for 0, blue for 1, and green for -1), representing the remaining work to be done.”



center h:350px

## Maximum Subarray Problem

**Input:** An array of values

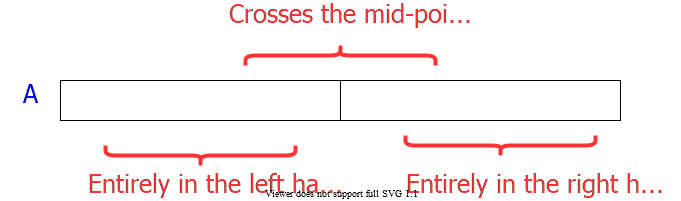
**Output:** The contiguous subarray that has the largest sum of elements

* Input array:

## Maximum Subarray Problem: Divide & Conquer (1)

* **Basic idea:**
  + **Divide** the input array into 2 from the middle
  + Pick the **best** solution among the following:
    - The max subarray of the **left half**
    - The max subarray of the **right half**
    - The max subarray **crossing the mid-point**

## Maximum Subarray Problem: Divide & Conquer (2)



alt:“alt” height:450px center

## Maximum Subarray Problem: Divide & Conquer (3)

* **Divide:** Trivial (divide the array from the middle)
* **Conquer:** Recursively compute the max subarrays of the left and right halves
* **Combine:** Compute the max-subarray crossing the
  + (can be done in time).
  + Return the max among the following:
    - the max subarray of the
    - the max subarray of the
    - the max subarray crossing the

TODO : detailed solution in textbook…

## Conclusion : Divide & Conquer

* Divide and conquer is just one of several powerful techniques for algorithm design.
* Divide-and-conquer algorithms can be analyzed using recurrences and the master method (so practice this math).
* Can lead to more efficient algorithms

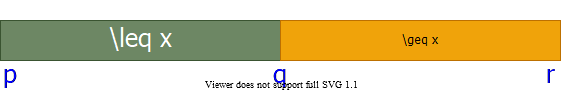
## Quicksort (1)

* One of the most-used algorithms in practice
* Proposed by **C.A.R.** *Hoare* in 1962.
* Divide-and-conquer algorithm
* In-place algorithm
  + The additional space needed is O(1)
  + The sorted array is returned in the input array
  + *Reminder: Insertion-sort is also an in-place algorithm, but Merge-Sort is not in-place.*
* Very practical

## Quicksort (2)

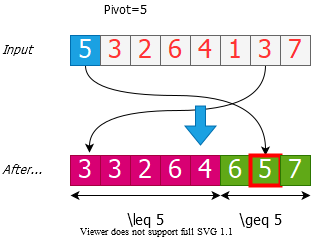
* **Divide:** Partition the array into 2 subarrays such that elements in the lower part elements in the higher part
* **Conquer:** Recursively sort 2 subarrays
* **Combine:** Trivial (because in-place)

**Key:** Linear-time partitioning algorithm



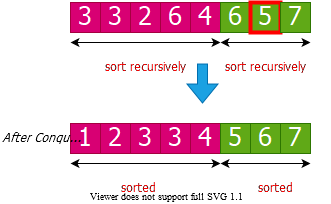
alt:“alt” height:200px center

## Divide: Partition the array around a pivot element

* Choose a pivot element
* Rearrange the array such that:
  + Left subarray: All elements
  + Right subarray: All elements
* 
* alt:“alt” height:350px center

## Conquer: Recursively Sort the Subarrays

Note: Everything in the left subarray ≤ everything in the right subarray

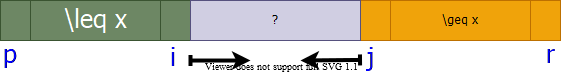


alt:“alt” height:350px center

Note: Combine is trivial after conquer. Array already sorted.

## Two partitioning algorithms

* **Hoare’s algorithm:** Partitions around the first element of subarray



alt:“alt” height:100px center

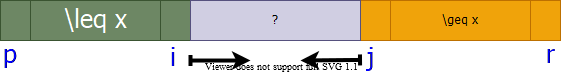
* **Lomuto’s algorithm:** Partitions around the last element of subarray



alt:“alt” height:100px center

## Hoare’s Partitioning Algorithm (1)

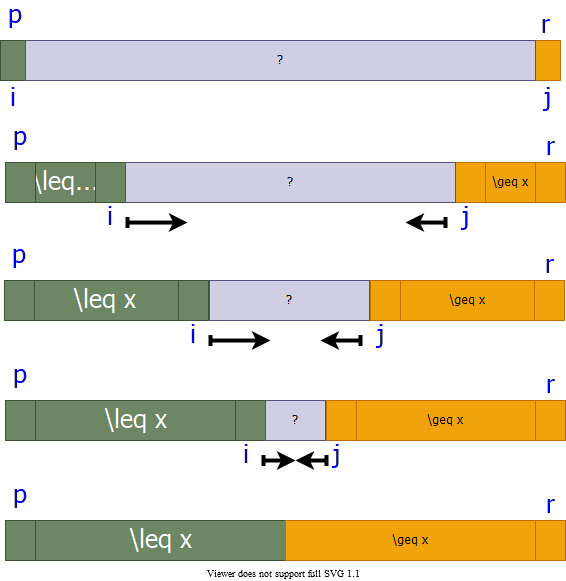
* Choose a pivot element:



alt:“alt” height:100px center

* Grow two regions:
  + from left to right:
  + from right to left:
    - such that:
  + every element in pivot
  + every element in pivot

## Hoare’s Partitioning Algorithm (2)



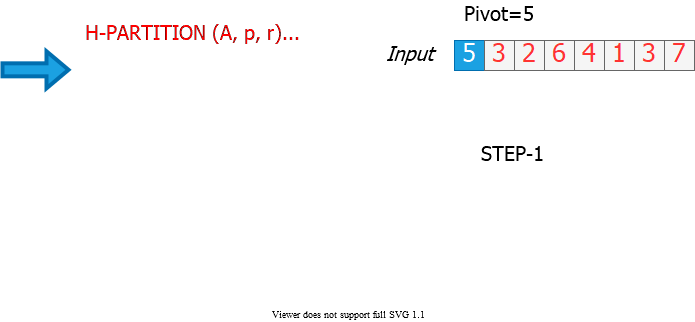
alt:“alt” height:550px center

## Hoare’s Partitioning Algorithm (3)

* Elements are exchanged when
  + is **too large** to belong to the **left** region
  + is **too small** to belong to the **right** region
    - assuming that the inequality is strict
* The two regions and grow until

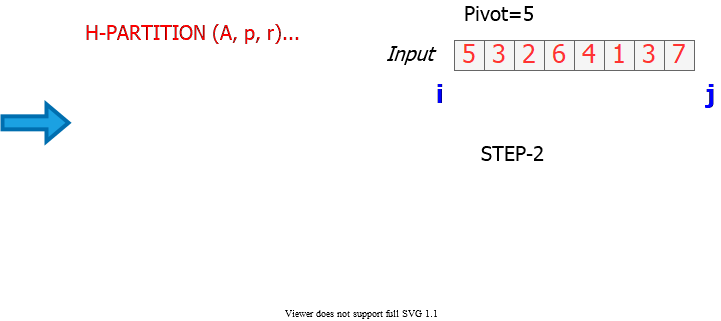
H-PARTITION(A, p, r)  
 pivot = A[p]  
 i = p - 1  
 j = r - 1  
 while true do  
 repeat j = j - 1 until A[j] <= pivot  
 repeat i = i - 1 until A[i] <= pivot  
 if i < j then   
 exchange A[i] with A[j]  
 else   
 return j

## Hoare’s Partitioning Algorithm Example (Step-1)



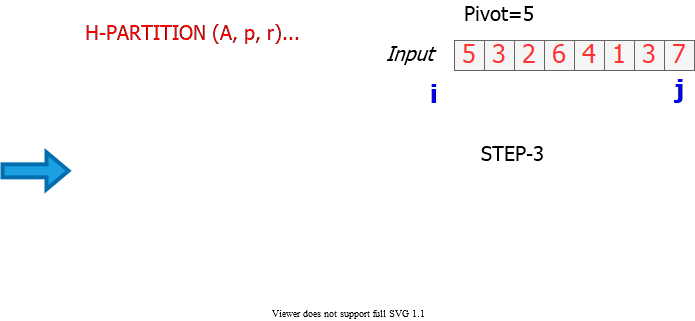
alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-2)



alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-3)



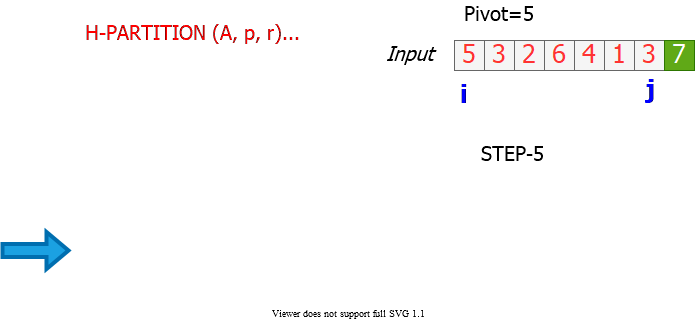
alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-4)



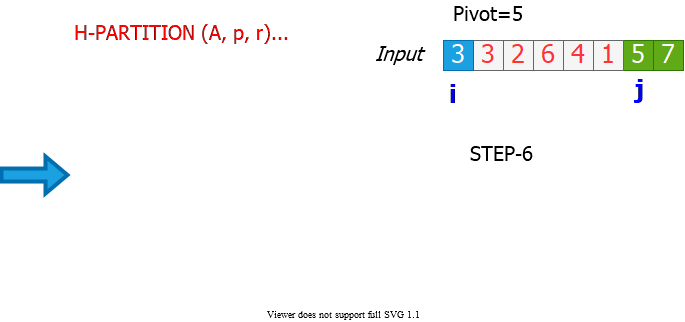
alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-5)



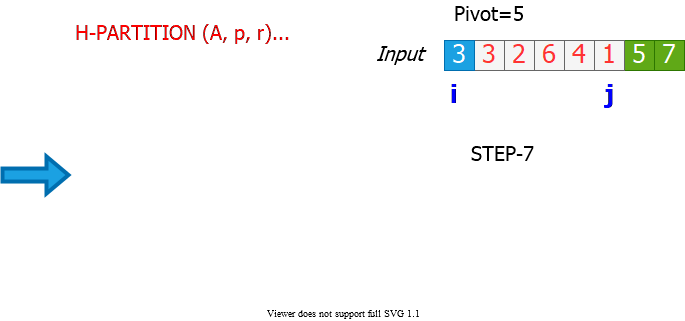
alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-6)



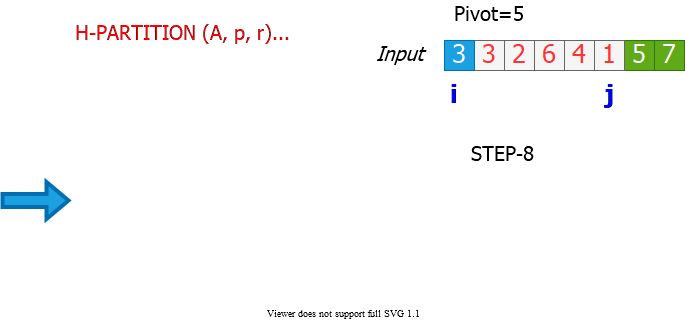
alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-7)



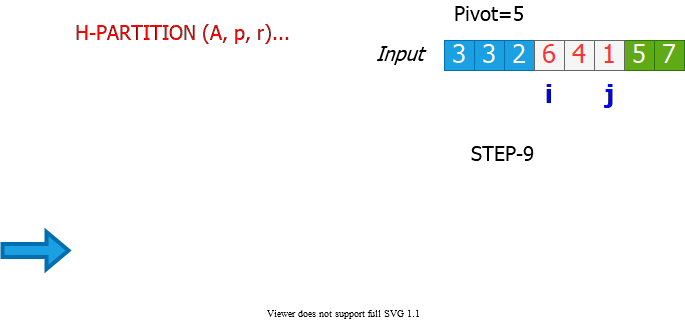
alt:“alt” height:550px center

## Hoare’s Partitioning Algorithm Example (Step-8)



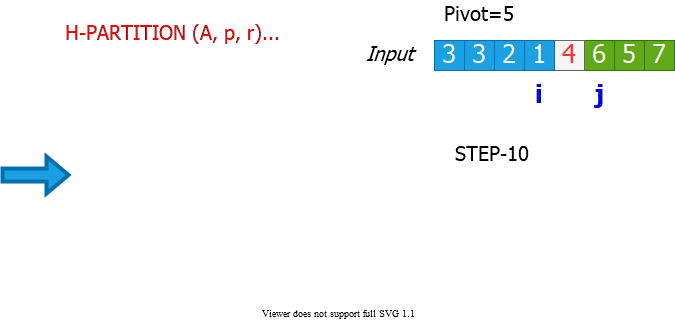
alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-9)



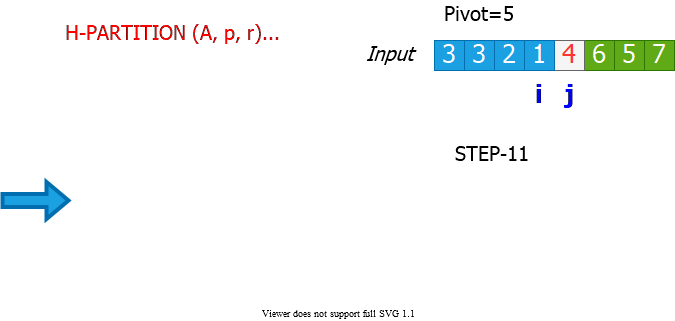
alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-10)



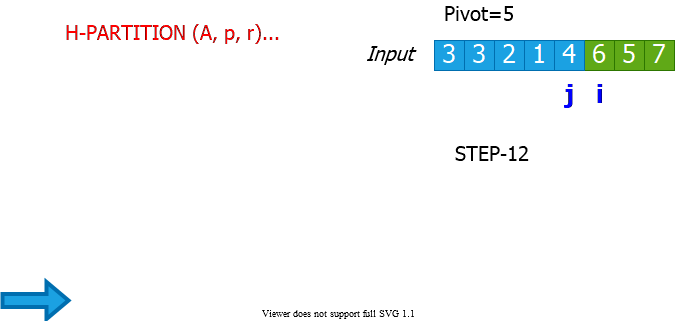
alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-11)



alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm Example (Step-12)



alt:“alt” height:500px center

## Hoare’s Partitioning Algorithm - Notes

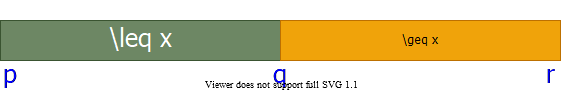
* Elements are exchanged when
  + is **too large** to belong to the **left** region
  + is **too small** to belong to the **right** region
    - assuming that the inequality is strict
* The two regions and grow until
* The asymptotic runtime of Hoare’s partitioning algorithm

H-PARTITION(A, p, r)  
 pivot = A[p]  
 i = p - 1  
 j = r - 1  
 while true do  
 repeat j = j - 1 until A[j] <= pivot  
 repeat i = i - 1 until A[i] <= pivot  
 if i < j then exchange A[i] with A[j]  
 else return j

## Quicksort with Hoare’s Partitioning Algorithm

QUICKSORT (A, p, r)  
 if p < r then  
 q = H-PARTITION(A, p, r)  
 QUICKSORT(A, p, q)  
 QUICKSORT(A, q + 1, r)  
 endif

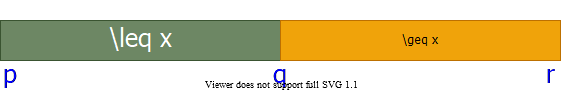
Initial invocation: QUICKSORT(A,1,n)



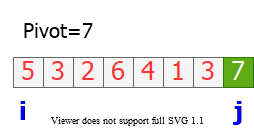
alt:“alt” height:150px center

## Hoare’s Partitioning Algorithm: Pivot Selection

* if we select pivot to be instead of in **H-PARTITION**



alt:“alt” height:90px center



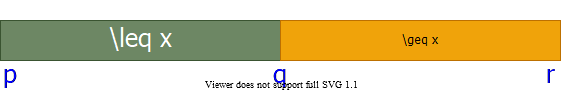
alt:“alt” height:140px center

* Consider the example where is the largest element in the array:
  + End of H-PARTITION:
  + In QUICKSORT:
    - So, recursive call to:
      * QUICKSORT(A, p, q=r)
        + **infinite loop**

## Correctness of Hoare’s Algorithm (1)

We need to prove claims to show correctness:

* Indices and never reference outside the interval
* Split is always non-trivial; i.e., at termination
* Every element in every element in at termination



alt:“alt” height:150px center

## Correctness of Hoare’s Algorithm (2)

* Notations:
  + : of times the while-loop iterates until termination
  + : the value of index i at the end of iteration
  + : the value of index j at the end of iteration
  + : the value of the pivot element
* **Note**: We always have and because

## Correctness of Hoare’s Algorithm (3)

**Lemma 1:** Either or at termination

**Proof of Lemma 1:**

* The algorithm terminates when (the else condition).
* So, it is sufficient to prove that
* There are cases to consider:
  + Case 1: , i.e. the algorithm terminates in a single iteration
  + Case 2: , i.e. the alg. does not terminate in a single iter.
* **By contradiction**, assume there is a run with

## Correctness of Hoare’s Algorithm (4)

**Original correctness claims:**

* Indices and never reference A outside the interval
* Split is always non-trivial; i.e., at termination

**Proof:**

* **For :**
  + Trivial because (*see Case 1 in proof of Lemma 2*)
* **For :**
  + and (*due to the repeat-until loops moving indices*)
  + and (*due to Lemma 1 and the statement above*)

**The proof of claims (a) and (b) complete**

## Correctness of Hoare’s Algorithm (5)

**Lemma 2:** At the end of iteration , where (*i.e. m is not the last iteration*), we must have: and

**Proof of Lemma 2:**

* **Base case:** and (*i.e. the alg. does not terminate in the first iter.*)

**Ind. Hyp.:** At the end of iteration , where (*i.e. m is not the last iteration*), we must have: and

**General case:** The lemma holds for , where

**Proof of base case complete!**

## Correctness of Hoare’s Algorithm (6)

**Original correctness claim:**

* 1. Every element in every element in at termination

**Proof of claim (c)**

* There are cases to consider:
  + **Case 1:** , i.e. the algorithm terminates in a single iteration
  + **Case 2:** and
  + **Case 3:** and

## Lomuto’s Partitioning Algorithm (1)

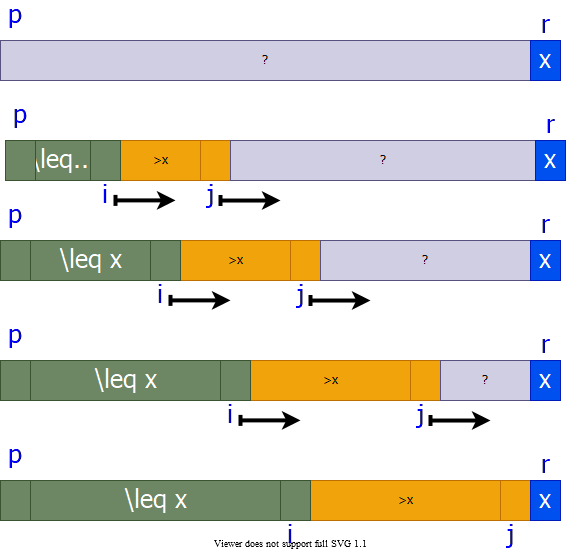
* Choose a pivot element:



alt:“alt” height:100px center

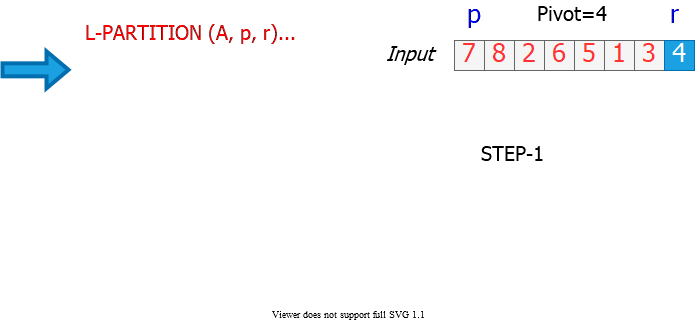
* Grow two regions:
  + from left to right:
  + from left to right:
    - such that:
      * every element in
      * every element in

## Lomuto’s Partitioning Algorithm (2)



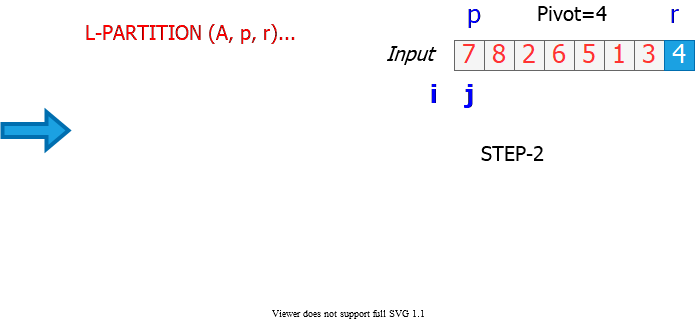
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-1)



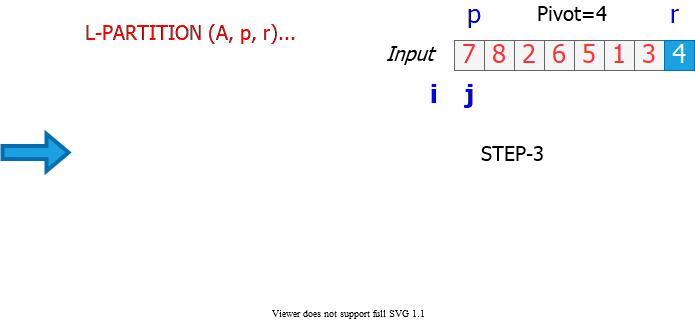
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-2)



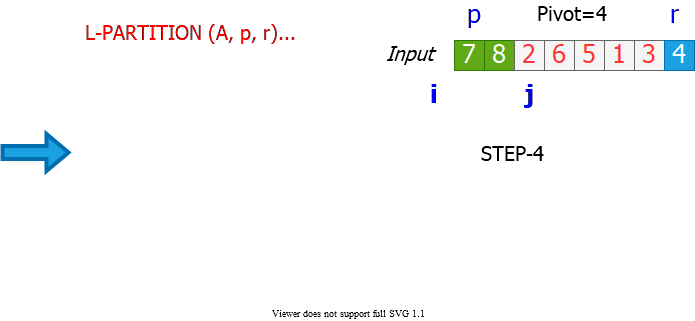
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-3)



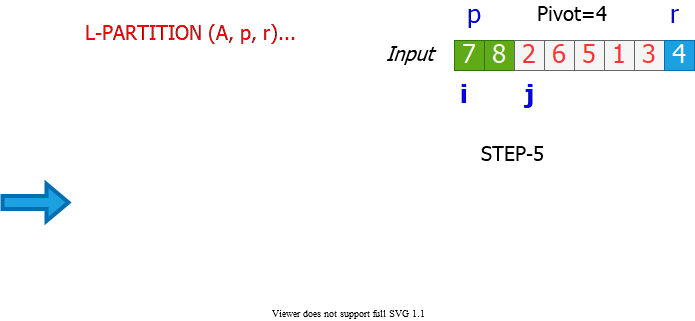
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-4)



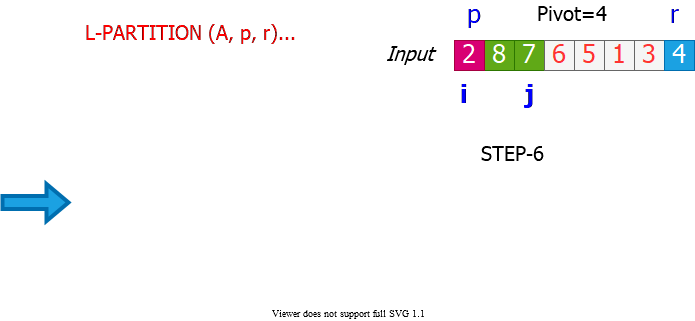
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-5)



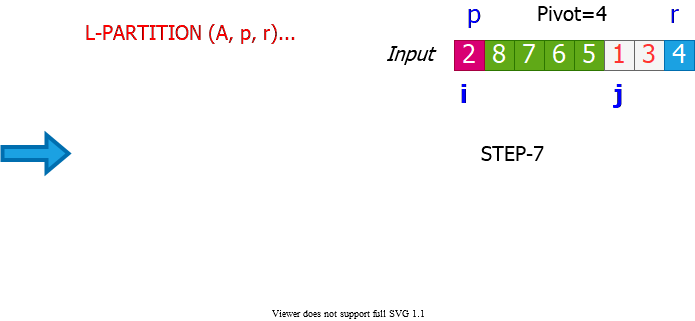
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-6)



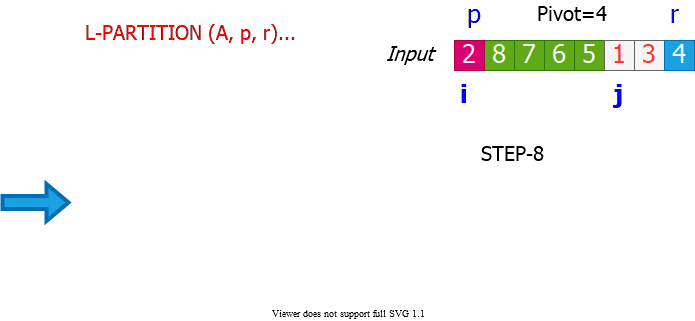
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-7)



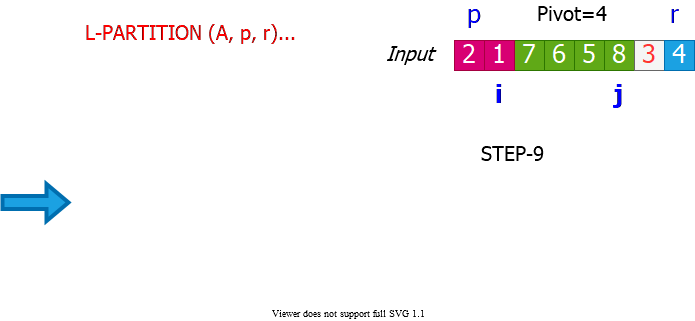
alt:“alt” height:550px center

## Lomuto’s Partitioning Algorithm Ex. (Step-8)



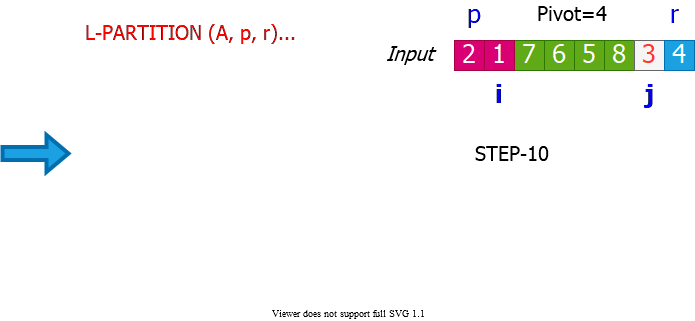
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-9)



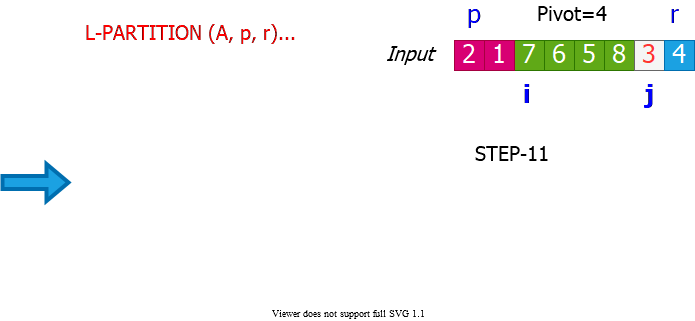
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-10)



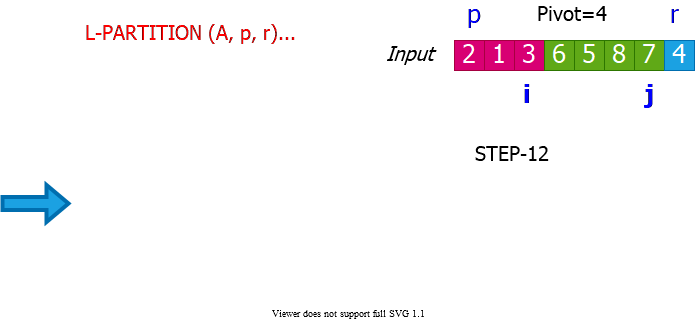
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-11)



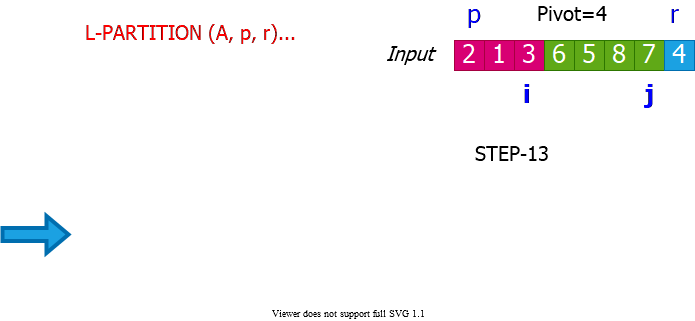
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-12)



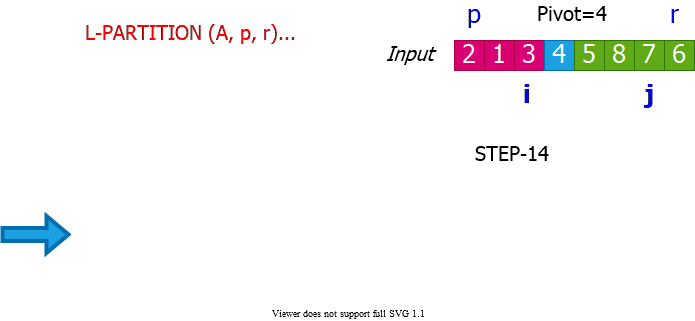
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-13)



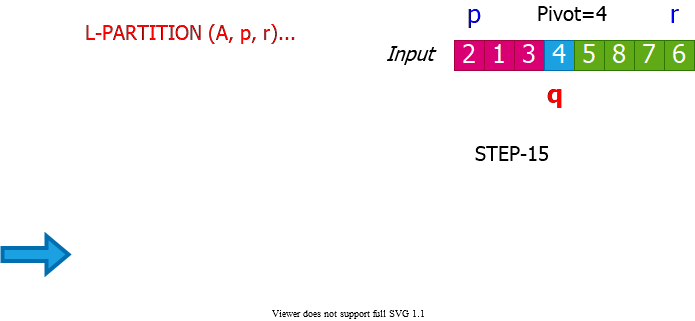
alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-14)



alt:“alt” height:500px center

## Lomuto’s Partitioning Algorithm Ex. (Step-15)

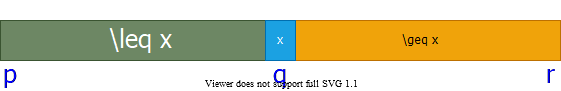


alt:“alt” height:500px center

## Quicksort with Lomuto’s Partitioning Algorithm

QUICKSORT (A, p, r)  
 if p < r then  
 q = L-PARTITION(A, p, r)  
 QUICKSORT(A, p, q - 1)  
 QUICKSORT(A, q + 1, r)  
 endif

Initial invocation: QUICKSORT(A,1,n)



alt:“alt” height:150px center

## Comparison of Hoare’s & Lomuto’s Algorithms (1)

* Notation:
  + (*Hoare*)
  + (*Lomuto*)
* **of element exchanges:** 
  + Hoare:
    - **Best**: with (i.e., )
    - **Worst**:
  + Lomuto :
    - **Best**:
    - **Worst**:

## Comparison of Hoare’s & Lomuto’s Algorithms (2)

* **of element comparisons:** 
  + **Hoare**:
    - **Best**:
    - **Worst**:
  + **Lomuto**:
* **of index comparisons:** 
  + **Hoare**:
  + **Lomuto**:

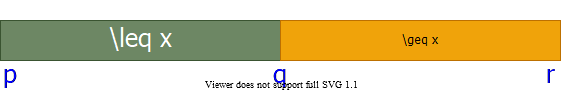
## Comparison of Hoare’s & Lomuto’s Algorithms (3)

* **of index increment/decrement operations:** 
  + **Hoare**:
  + **Lomuto**:
* Hoare’s algorithm is in general faster
* Hoare behaves better when pivot is repeated in
  + **Hoare**: Evenly distributes them between left & right regions
  + **Lomuto**: Puts all of them to the left region

## Analysis of Quicksort (1)

QUICKSORT (A, p, r)  
 if p < r then  
 q = H-PARTITION(A, p, r)  
 QUICKSORT(A, p, q)  
 QUICKSORT(A, q + 1, r)  
 endif

Initial invocation: QUICKSORT(A,1,n)



alt:“alt” height:150px center

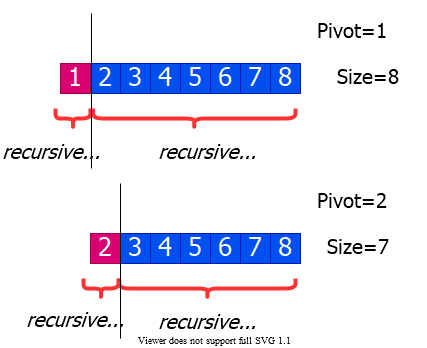
Assume **all elements are distinct** in the following analysis

## Analysis of Quicksort (2)

* **H-PARTITION** always chooses (the first element) as the pivot.
* The runtime of **QUICKSORT** on an already-sorted array is

## Example: An Already Sorted Array

Partitioning always leads to parts of size and

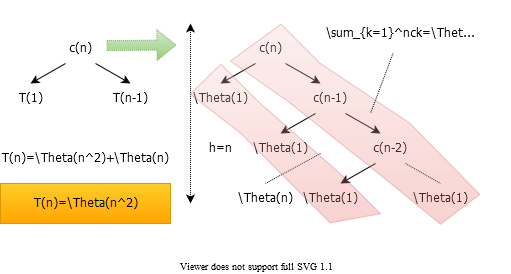


alt:“alt” height:450px center

## Worst Case Analysis of Quicksort

* **Worst case** is when the **PARTITION** algorithm always returns **imbalanced partitions** (of size and ) in every recursive call.
  + This happens when the pivot is selected to be either the min or **max** element.
  + This happens for **H-PARTITION** when the input array is already sorted or reverse sorted

## Worst Case Recursion Tree



alt:“alt” height:450px center

## Best Case Analysis (for intuition only)

* If we’re extremely lucky, **H-PARTITION** splits the array evenly at every recursive call

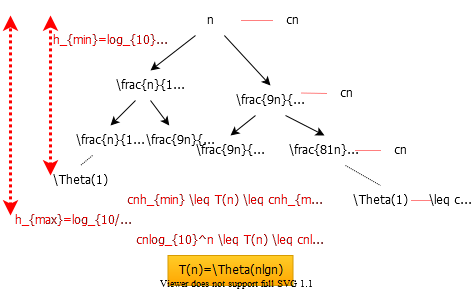
*(same as merge sort)*

* Instead of splitting , if we split then we need solve following equation.

$$

$$

## “Almost-Best” Case Analysis



alt:“alt” height:500px center

## Balanced Partitioning (1)

* We have seen that if **H-PARTITION** always splits the array with ratio, the runtime will be .
* Same is true with a split ratio of , etc.
* Possible to show that if the split has always constant proportionality, then the runtime will be .
* In other words, for a **constant** :
  + proportional split yields total runtime

## Balanced Partitioning (2)

* In the rest of the analysis, assume that all input permutations are equally likely.
  + This is only to gain some intuition
  + We cannot make this assumption for average case analysis
  + We will revisit this assumption later
* Also, assume that all input elements are distinct.

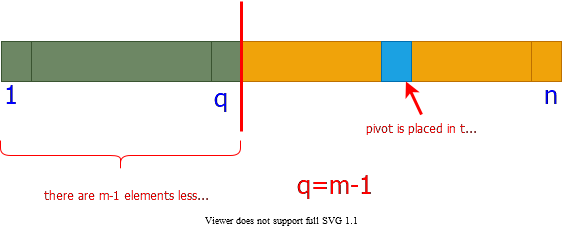
## Balanced Partitioning (3)

* **Question:** What is the probability that H-PARTITION returns a split that is more balanced than ?

## Balanced Partitioning (4)

**Reminder:** *H-PARTITION* will place the pivot in the right partition unless the pivot is the smallest element in the arrays.

**Question:** If the pivot selected is the mth smallest value in the input array, what is the size of the left region after partitioning?



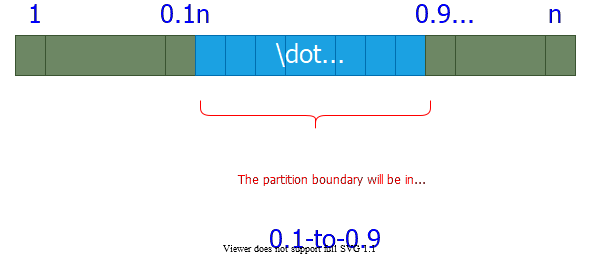
alt:“alt” height:330px center

## Balanced Partitioning (5)

* **Question:** What is the probability that the **pivot** selected is the smallest value in the array of size ?
  + (*since all input permutations are equally likely*)
* **Question:** What is the probability that the left partition returned by **H-PARTITION** has size , where ?
  + (*due to the answers to the previous 2 questions*)

## Balanced Partitioning (6)

* **Question:** What is the probability that H-PARTITION returns a split that is more balanced than ?



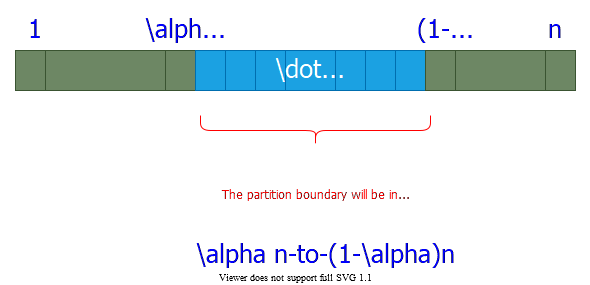
bg right:50% w:500px

## Balanced Partitioning (7)

* The probability that **H-PARTITION** yields a split that is more balanced than is on a random array.
* Let be the probability that **H-PARTITION** yields a split more balanced than , where
* Repeat the analysis to generalize the previous result

## Balanced Partitioning (8)

* **Question:** What is the probability that H-PARTITION returns a split that is more balanced than ?



bg right:40% w:500px

## Balanced Partitioning (9)

* We found
  + Ex: and
* Hence, **H-PARTITION** produces a split
  + **more balanced than a**
    - split of the time
    - split of the time
  + **less balanced than a**
    - split of the time
    - split of the time

## Intuition for the Average Case (1)

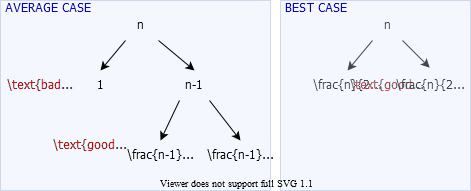
* **Assumption:** All permutations are equally likely
  + Only for intuition; we’ll revisit this assumption later
* **Unlikely:** Splits always the same way at every level
* **Expectation:**
  + Some splits will be reasonably balanced
  + Some splits will be fairly unbalanced
* **Average case:** A mix of good and bad splits
  + **Good** and **bad** splits distributed randomly thru the tree

## Intuition for the Average Case (2)

* **Assume for intuition:** Good and bad splits occur in the alternate levels of the tree
  + **Good split:** Best case split
  + **Bad split:** Worst case split

## Intuition for the Average Case (3)

Compare 2-successive levels of avg case vs. 1 level of best case



alt:“alt” h:450px center

## Intuition for the Average Case (4)

* In terms of the remaining subproblems, **two levels of avg case** is slightly better than the **single level of the best case**
* The avg case has **extra divide cost of**  at alternate levels
* The extra divide cost of bad splits absorbed into the of good splits.
* Running time is still
  + But, slightly larger hidden constants, because the height of the recursion tree is about twice of that of best case.

## Intuition for the Average Case (5)

* Another way of looking at it:
  + Suppose we alternate lucky, unlucky, lucky, unlucky,
  + We can write the recurrence as:
    - lucky split (best)
    - unlucky split (worst)
  + Solving:
* How can we make sure we are usually lucky for all inputs?

## Summary: Quicksort Runtime Analysis (1)

* **Worst case:** Unbalanced split at every recursive call
* **Best case:** Balanced split at every recursive call (*extremely lucky*)

## Summary: Quicksort Runtime Analysis (2)

* **Almost-best case:** Almost-balanced split at every recursive call

for any constant

## Summary: Quicksort Runtime Analysis (3)

* For a random input array, the probability of having a split
  + more balanced than
  + more balanced than
  + more balanced than
* for any constant

## Summary: Quicksort Runtime Analysis (4)

* **Avg case intuition:** Different splits expected at different levels
  + some balanced (good), some unbalanced (bad)
* **Avg case intuition:** Assume the good and bad splits alternate
  + i.e. good split -> bad split -> good split -> …
    - (informal analysis for intuition)

## Randomized Quicksort

* In the avg-case analysis, we assumed that **all permutations** of the input array are **equally likely.**
  + But, this assumption **does not always hold**
  + e.g. What if **all** the input arrays are **reverse sorted**?
    - **Always worst-case behavior**
* Ideally, the avg-case runtime should be **independent of the input permutation**.
* **Randomness should be within the algorithm**, not based on the distribution of the inputs.
  + i.e. The avg case should hold for all possible inputs

## Randomized Algorithms (1)

* Alternative to assuming a uniform distribution:
  + **Impose a uniform distribution**
  + e.g. Choose a random pivot rather than the first element
* Typically useful when:
  + there are many ways that an algorithm can proceed
  + but, it’s **difficult** to determine a way that is **always guaranteed to be good**.
  + If there are **many good alternatives**; simply **choose one randomly**.

## Randomized Algorithms (1)

* Ideally:
  + Runtime should be **independent of the specific inputs**
  + No specific input should cause worst-case behavior
  + Worst-case should be determined only by output of a random number generator.

## Randomized Quicksort (1)

* Using Hoare’s partitioning algorithm:

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

R-PARTITION(A, p, r)  
 s = RANDOM(p, r)  
 exchange A[p] with A[s]  
 return H-PARTITION(A, p, r)

* Alternatively, permuting the whole array would also work
  + but, would be more difficult to analyze

## Randomized Quicksort (2)

* Using Lomuto’s partitioning algorithm:

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

R-PARTITION(A, p, r)  
 s = RANDOM(p, r)  
 exchange A[r] with A[s]  
 return L-PARTITION(A, p, r)

* Alternatively, permuting the whole array would also work
  + but, would be more difficult to analyze

## Notations for Formal Analysis

* Assume all elements in are distinct
  + Let
* Let
* i.e.  is the number of array elements with value less than or equal to
  + - i.e. it is the smallest element in the array

## Formal Analysis for Average Case

* The following analysis will be for **Quicksort** using **Hoare’s** partitioning algorithm.
* **Reminder:** The **pivot** is selected **randomly** and exchanged with before calling **H-PARTITION**
* Let be the **random pivot** chosen.
* What is the probability that for ?

## Various Outcomes of H-PARTITION (1)

* Assume that
  + i.e. the **random pivot** chosen is the **smallest** element
  + What will be the **size of the left partition** ?
  + **Reminder:** Only the elements less than or equal to will be in the left partition.

**TODO: convert to image…S6\_P9**

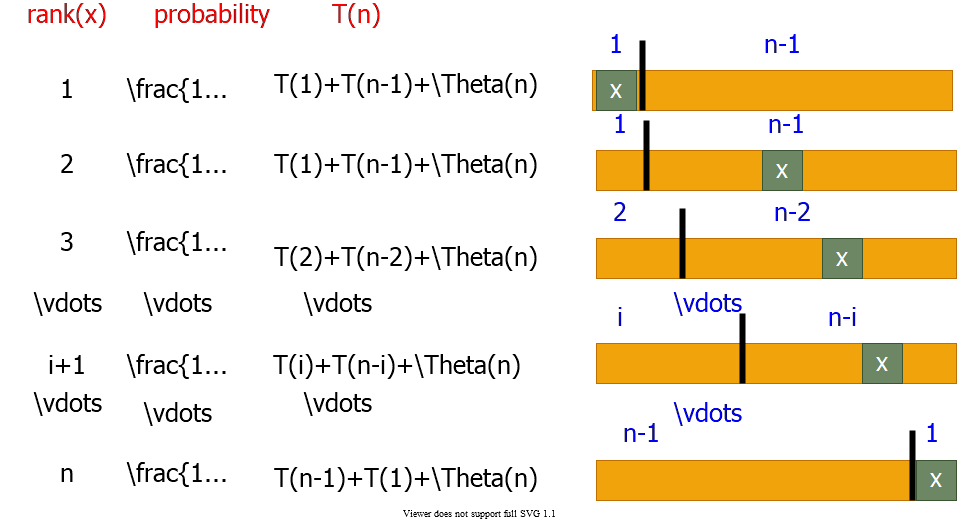
## Various Outcomes of H-PARTITION (2)

* Assume that
  + i.e. the random pivot chosen is not the smallest element
  + What will be the size of the left partition ?
  + **Reminder:** Only the elements less than or equal to will be in the left partition.
  + **Reminder:** The pivot will stay in the right region after **H-PARTITION** if

**TODO: convert to image…S6\_P10**

## Various Outcomes of H-PARTITION - Summary (1)

## Various Outcomes of H-PARTITION - Summary (2)



alt:“alt” h:450px center

## Average - Case Analysis: Recurrence (1)

## Average - Case Analysis: Recurrence (2)

for each term appears twice once for and once for

## Average - Case Analysis -Solving Recurrence: Substitution

* Guess:
* for , for some constant
* Need a tight bound for

## Tight bound for (1)

* Bounding the terms
  + This bound **is not strong** enough because
  + couldn’t prove

## Tight bound for (2)

* **Splitting summations:** ignore ceilings for simplicity
  + **First summation**:
  + **Second summation**:

## Splitting: (3)

## Substituting: - (4)

* We can choose a large enough so that

Q.E.D.

## Medians and Order Statistics

* **ith order statistic**: smallest element of a set of elements
* **minimum:** *first* order statistic
* **maximum:** order statistic
* **median:** “halfway point” of the set

## Selection Problem

* **Selection problem:** Select the smallest of elements
* **Naïve algorithm:** Sort the input array ; then return
  + - *using e.g. merge sort (but not quicksort)*
* Can we do any better?

## Selection in Expected Linear Time

* Randomized algorithm using divide and conquer
* Similar to randomized quicksort
  + **Like quicksort:** Partitions input array recursively
  + **Unlike quicksort:** Makes a single recursive call
    - **Reminder:** *Quicksort makes two recursive calls*
* Expected runtime:
  + **Reminder:** *Expected runtime of quicksort:*

## Selection in Expected Linear Time: Example 1

* Select the smallest element:
* Partition the input array:
* make a recursive call to select the  **smallest** element in **left subarray**

## Selection in Expected Linear Time: Example 2

* Select the smallest element:
* Partition the input array:
* make a recursive call to select the  **smallest** element in **right subarray**

## Selection in Expected Linear Time (1)

R-SELECT(A,p,r,i)  
 if p == r then   
 return A[p];  
 q = R-PARTITION(A, p, r)  
 k = q–p+1;  
 if i <= k then   
 return R-SELECT(A, p, q, i);  
 else  
 return R-SELECT(A, q+1, r, i-k);

## Selection in Expected Linear Time (2)

## Selection in Expected Linear Time (2)

* All elements in all elements in
* contains:
  + k smallest elements of
  + if then
    - **search** recursively for its smallest element
  + else
    - **search** recursively for its smallest element

## Runtime Analysis (1)

* **Worst case:**
  + Imbalanced partitioning at every level and the recursive call always to the larger partition

## Runtime Analysis (2)

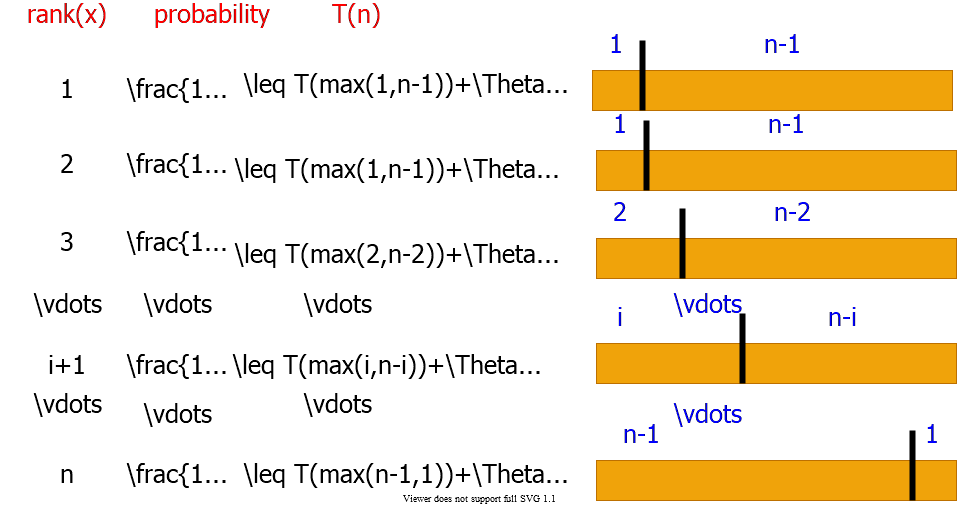
* **Worst case:** Worse than the naïve method (based on sorting)
* **Best case:** Balanced partitioning at every recursive level
* **Avg case:** Expected runtime – need analysis T.B.D.

## Reminder: Various Outcomes of H-PARTITION

## Average Case Analysis of Randomized Select

* To compute the **upper bound** for the **avg case**, assume that the element always falls into the **larger partition**.
* We will analyze the case where the recursive call is always made to the larger partition
  + *This will give us an upper bound for the avg case*

## Various Outcomes of H-PARTITION



alt:“alt” h:450px center

## Average-Case Analysis of Randomized Select (1)

**Upper bound:** Assume element always falls into the larger part.

## Average-Case Analysis of Randomized Select (2)

* is odd: appears twice for
* is even: appears once appears twice for

## Average-Case Analysis of Randomized Select (3)

* Hence, in both cases:

## Average-Case Analysis of Randomized Select (4)

* By substitution guess
* Inductive hypothesis:

## Average-Case Analysis of Randomized Select (5)

* since we can choose c large enough so that dominates

## Summary of Randomized Order-Statistic Selection

* Works fast: linear expected time
* Excellent algorithm in practise
* But, the worst case is very bad:
* **Blum, Floyd, Pratt, Rivest & Tarjan[1973]** algorithms are runs in **linear time** in the **worst case**.
* Generate a **good pivot** recursively

## Selection in Worst Case Linear Time

//return i-th element in set S with n elements  
SELECT(S, n, i)   
  
 if n <= 5 then  
  
 SORT S and return the i-th element  
  
 DIVIDE S into ceil(n/5) groups  
 //first ceil(n/5) groups are of size 5, last group is of size n mod 5  
  
 FIND median set M={m , …, m\_ceil(n/5)}  
 // m\_j : median of j-th group  
  
 x = SELECT(M,ceil(n/5),floor((ceil(n/5)+1)/2))  
  
 PARTITION set S around the pivot x into L and R  
  
 if i <= |L| then  
 return SELECT(L, |L|, i)  
 else  
 return SELECT(R, n–|L|, i–|L|)

## Selection in Worst Case Linear Time - Example (1)

* **Input:** Array and index
* **Output:** The smallest value

## Selection in Worst Case Linear Time - Example (2)

**Step 1:** Divide the input array into groups of size

## Selection in Worst Case Linear Time - Example (3)

**Step 2:** Compute the median of each group ()

* Let be the set of the medians computed:

## Selection in Worst Case Linear Time - Example (4)

**Step 3:** Compute the median of the median group

where

* Let be the set of the medians computed:
* The runtime of the recursive call:

## Selection in Worst Case Linear Time - Example (5)

**Step 4:** Partition the input array around the median-of-medians

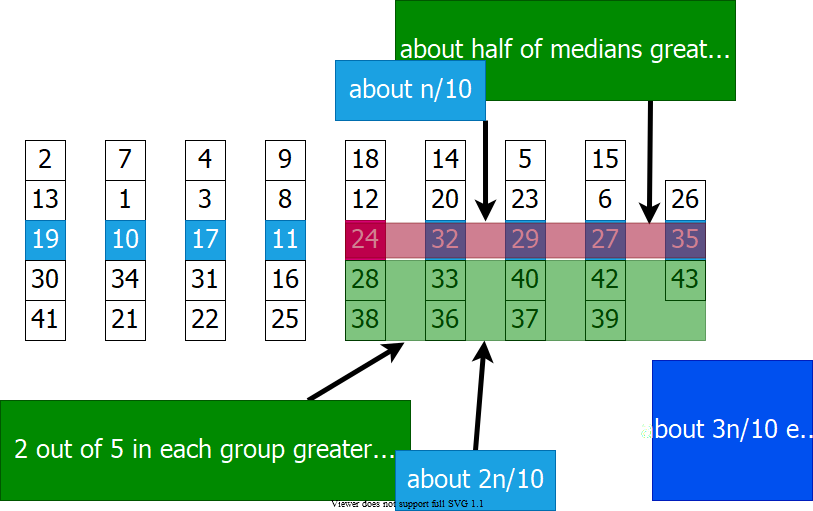
Partition around

**Claim:** Partitioning around x is guaranteed to be **well-balanced.**

## Selection in Worst Case Linear Time - Example (6)

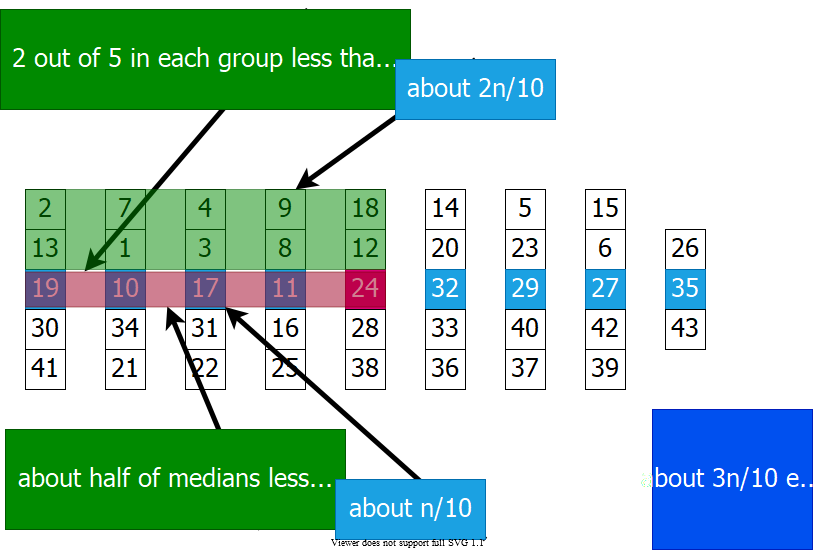
* : Median, : Median of Medians
* About half of the medians greater than (about )

## Selection in Worst Case Linear Time - Example (7)



alt:“alt” h:500px center

## Selection in Worst Case Linear Time - Example (8)



alt:“alt” h:500px center

## Selection in Worst Case Linear Time - Example (9)

* Partitioning around will lead to partitions of sizes and in the **worst case**.

**Step 5:** Make a recursive call to one of the partitions

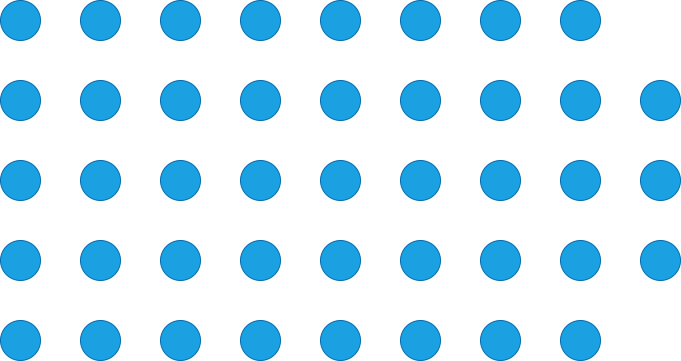
if i <= |L| then   
 return SELECT(L,|L|,i)  
else   
 return SELECT(R,n-|L|,i-|L|)

## Selection in Worst Case Linear Time

//return i-th element in set S with n elements  
SELECT(S, n, i)   
  
 if n <= 5 then  
  
 SORT S and return the i-th element  
  
 DIVIDE S into ceil(n/5) groups  
 //first ceil(n/5) groups are of size 5, last group is of size n mod 5  
  
 FIND median set M={m , …, m\_ceil(n/5)}  
 // m\_j : median of j-th group  
  
 x = SELECT(M,ceil(n/5),floor((ceil(n/5)+1)/2))  
  
 PARTITION set S around the pivot x into L and R  
  
 if i <= |L| then  
 return SELECT(L, |L|, i)  
 else  
 return SELECT(R, n–|L|, i–|L|)

## Choosing the Pivot (1)

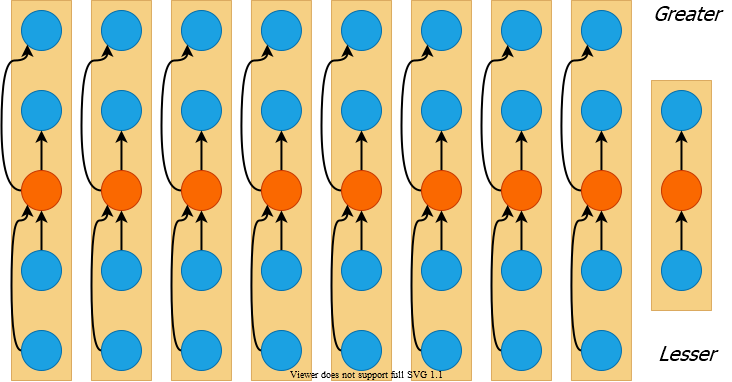
1. Divide S into groups of size 5



alt:“alt” h:450px center

## Choosing the Pivot (2)

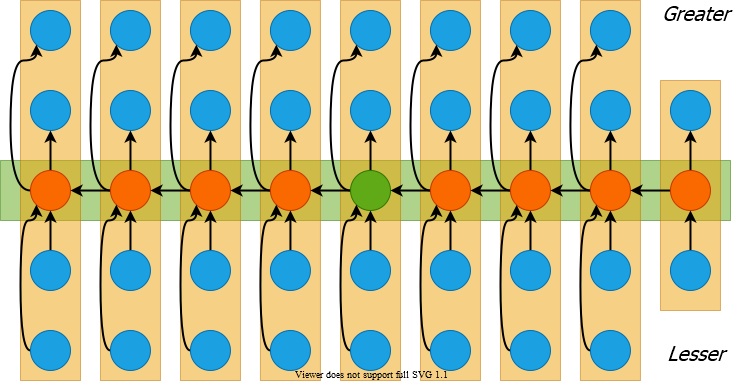
* Divide S into groups of size 5
* Find the median of each group



alt:“alt” h:450px center

## Choosing the Pivot (3)

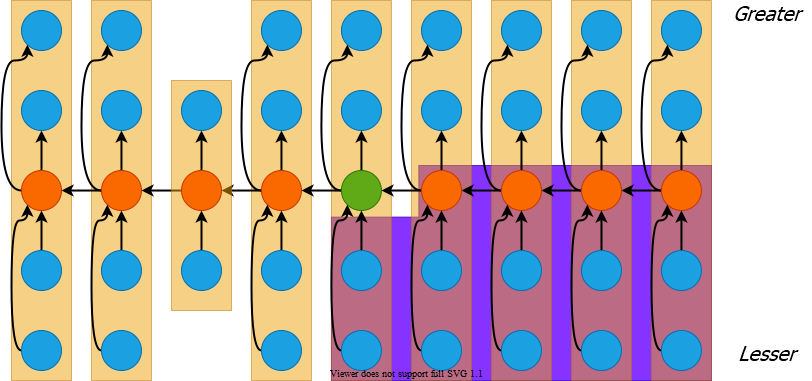
* Divide S into groups of size 5
* Find the median of each group
* Recursively select the median x of the medians



alt:“alt” h:450px center

## Choosing the Pivot (4)

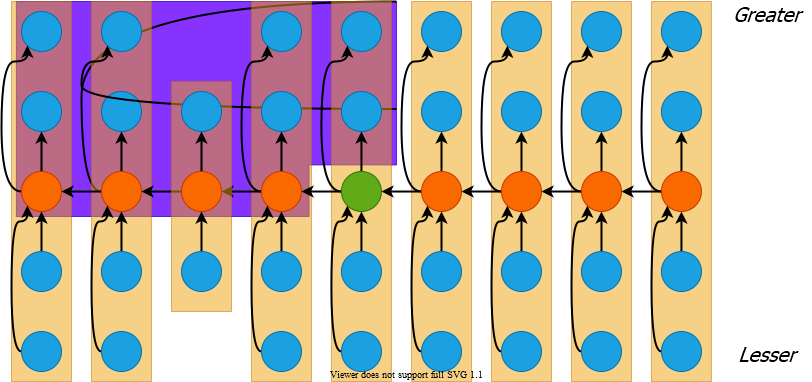
* At least half of the medians
* Thus groups contribute 3 elements to R except possibly the last group and the group that contains ,



alt:“alt” h:400px center

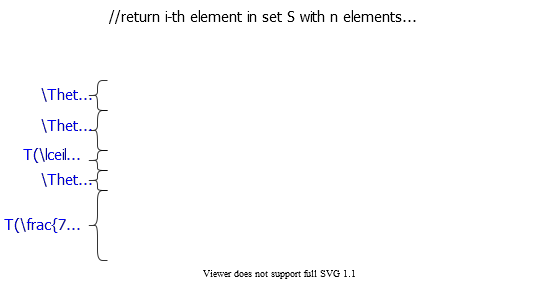
## Choosing the Pivot (5)

* Similarly
* Therefore, **SELECT** is recursively called on at most elements



alt:“alt” h:400px center

## Selection in Worst Case Linear Time (1)



alt:“alt” h:500px center

## Selection in Worst Case Linear Time (2)

* Thus recurrence becomes
* Guess and prove by induction
* Inductive step:
* Work at each level of recursion is a constant factor smaller

## References

* [Introduction to Algorithms, Third Edition | The MIT Press](https://mitpress.mit.edu/books/introduction-algorithms-third-edition)
* [Bilkent CS473 Course Notes (new)](http://nabil.abubaker.bilkent.edu.tr/473/)
* [Bilkent CS473 Course Notes (old)](http://cs.bilkent.edu.tr/~ugur/teaching/cs473/)
* [Insertion Sort - GeeksforGeeks](https://www.geeksforgeeks.org/insertion-sort/)
* [NIST Dictionary of Algorithms and Data Structures](https://xlinux.nist.gov/dads/)
* [NIST - Dictionary of Algorithms and Data Structures](https://xlinux.nist.gov/dads/)
* [NIST - big-O notation](https://xlinux.nist.gov/dads/HTML/bigOnotation.html)
* [NIST - big-Omega notation](https://xlinux.nist.gov/dads/HTML/omegaCapital.html)
* [Discovering novel algorithms with AlphaTensor](https://www.deepmind.com/blog/discovering-novel-algorithms-with-alphatensor)