Divide and Conquer
Randomized Algorithms

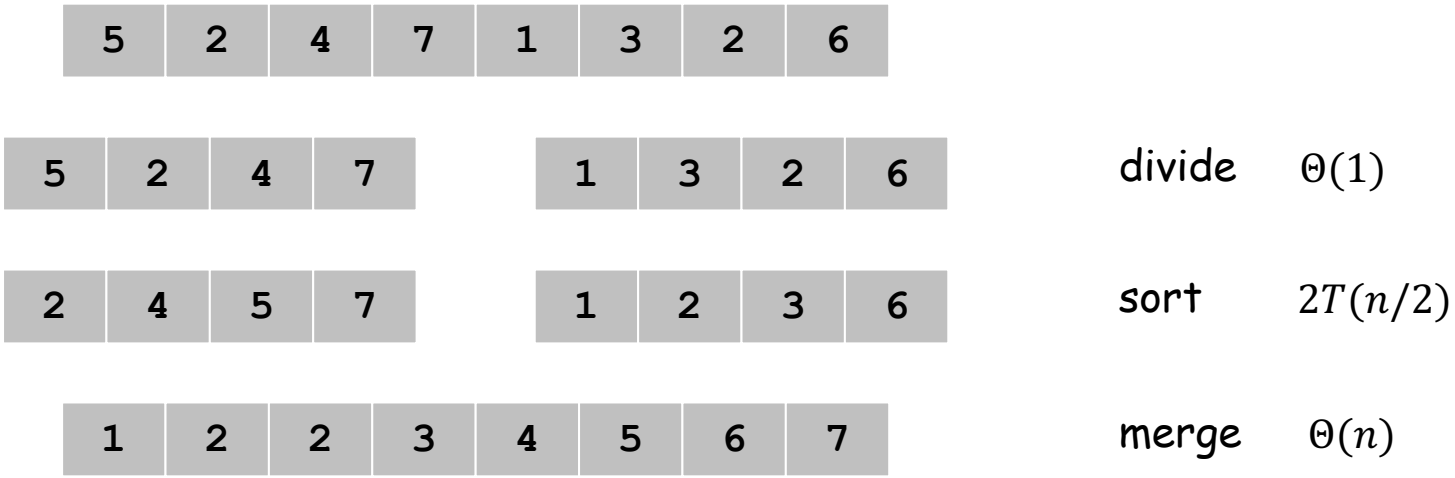
Outline

1. Quicksort & Partition
2. Randomization
3. Analysis of Randomized Quicksort
4. Randomized Selection

Quicksort: The "dual" of merge sort

Mergesort(A, p, r):
if $p = r$ **then return**
$q \leftarrow \lfloor (p + r)/2 \rfloor$
Mergesort(A, p, q)
Mergesort($A, q + 1, r$)
Merge(A, p, q, r)

First call: **Mergesort**($A, 1, n$)



Quicksort: The "dual" of merge sort [II]

Mergesort(A, p, r)**:**

if $p = r$ **then return**

$q \leftarrow \lfloor (p + r)/2 \rfloor$

Mergesort(A, p, q)

Mergesort$(A, q + 1, r)$

Merge(A, p, q, r)

First call: **Mergesort**$(A, 1, n)$

Quicksort(A, p, r)**:**

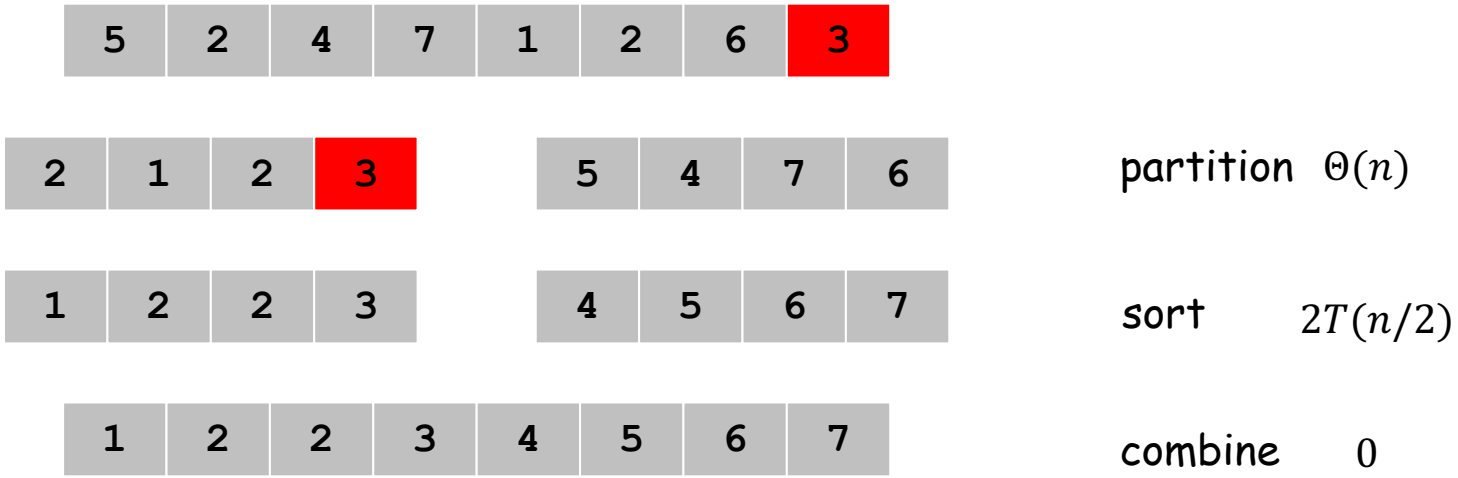
if $p \geq r$ **then return**

$q =$ **Partition**(A, p, r)

Quicksort$(A, p, q - 1)$

Quicksort$(A, q + 1, r)$

First call: **Quicksort**$(A, 1, n)$



Quicksort: The "dual" of merge sort [III]

Quicksort chooses an item as pivot.

It *partitions* array so that all items less than or equal to pivot are on the left and all items greater than pivot on the right.

It then recursively Quicksorts left and right sides.

Quicksort(A, p, r)**:**

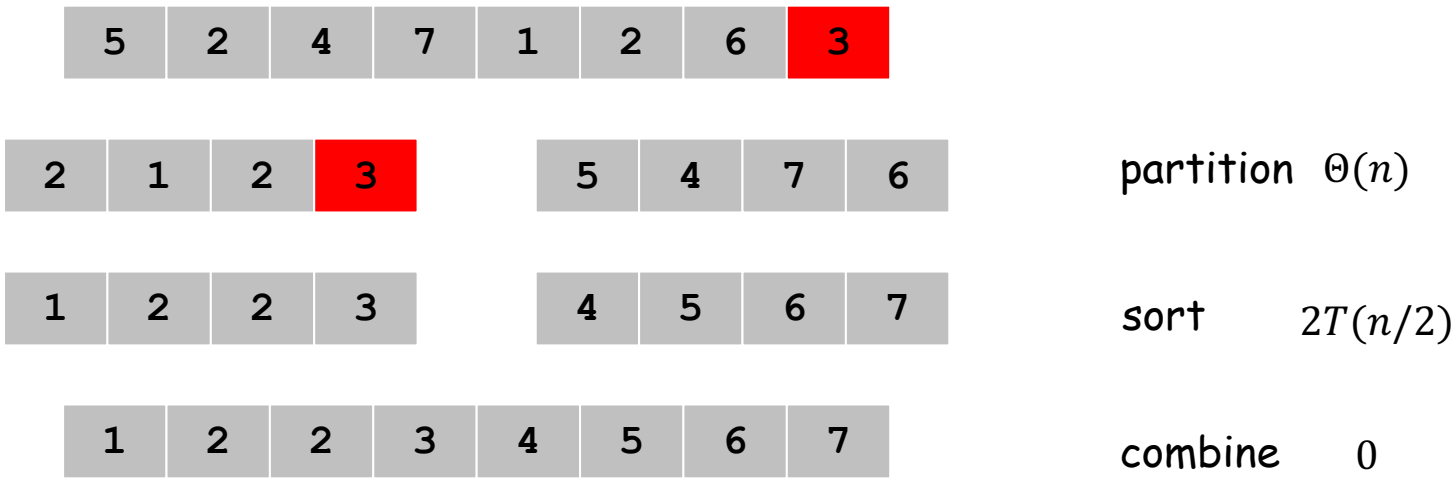
if $p \geq r$ **then return**

$q =$ **Partition**(A, p, r) **//new pivot position**

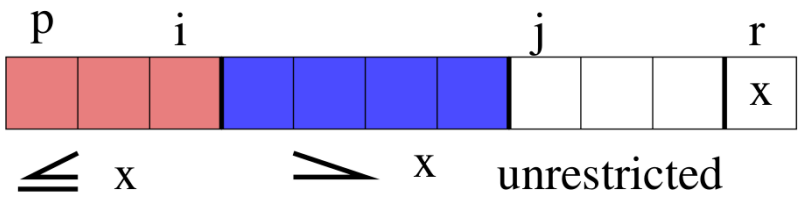
Quicksort$(A, p, q-1)$

Quicksort$(A, q+1, r)$

First call: **Quicksort**$(A, 1, n)$



Partition with the last element as the pivot



Partition(A, p, r)**:**

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$

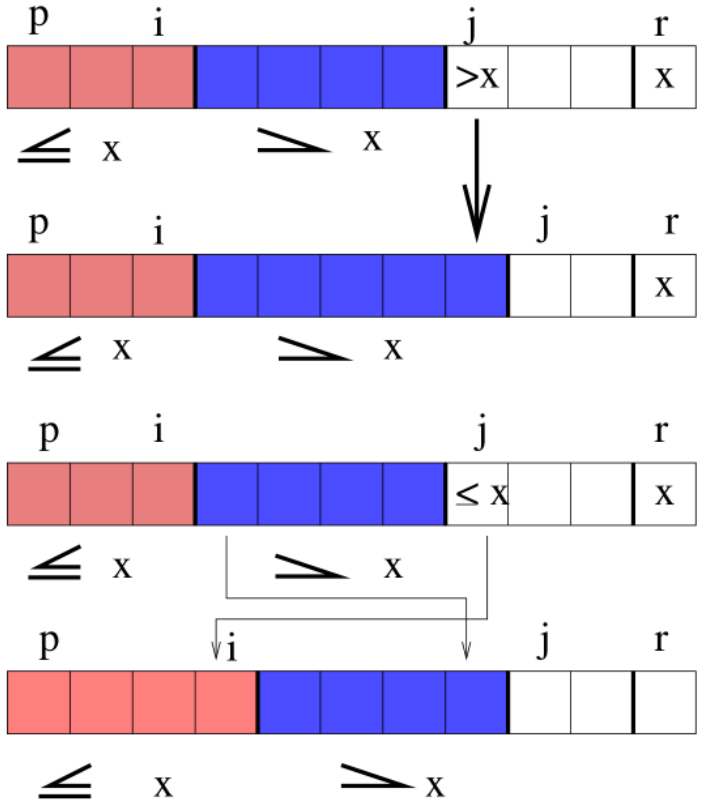
  if $A[j] \leq x$ **then**

    $i \leftarrow i + 1$

    swap $A[i]$ **and** $A[j]$

swap $A[i + 1]$ **and** $A[r]$

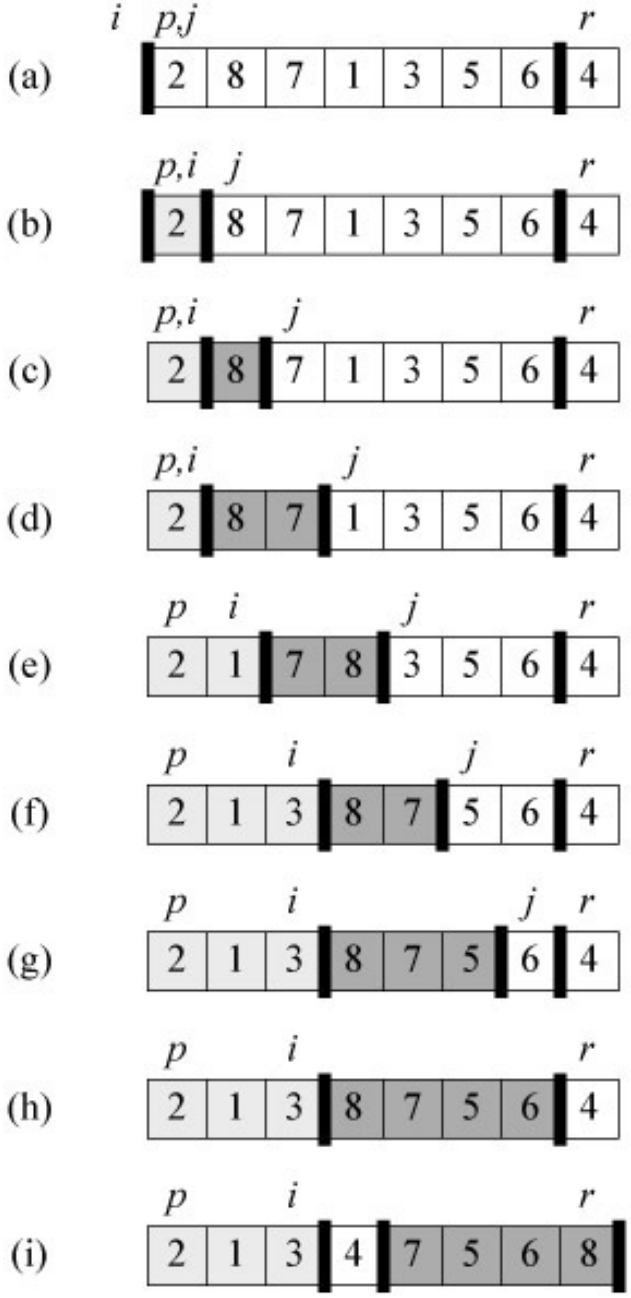
return $i + 1$ **//pivot position**



Time: $\Theta(n)$

Working space: $O(1)$ (in-place algorithm)

Partition: Example



Pivot selection is crucial

Running time.

- [Best case.] Select the median (middle value) element as the pivot: quicksort runs in $\Theta(n \log n)$ time *(problem is that we don't know the median).*
- [Worst case.] Select the smallest (or the largest) element as the pivot: quicksort runs in $\Theta(n^2)$ time.

Q: How to make running time **independent** of input

A: Randomly choose an element as the pivot
(swap it with last item in array and then run partition)

This is what's known as a ***randomized algorithm:***
algorithm makes a ***random*** choice each time it chooses a pivot.

It might help to think of the algorithm as trying to fool an evil adversary who is attempting to create a bad input by forcing bad pivots all the time. By making random pivot choices, the algorithm makes it impossible for the adversary to force a bad pivot, since the adversary can't know which key will be chosen as a pivot.

Analysis for Randomized Algorithms

Worst case almost never happens: Every pivot would have to be minimum or maximum; occurs with very low probability.

Expected running time of any input of size n.

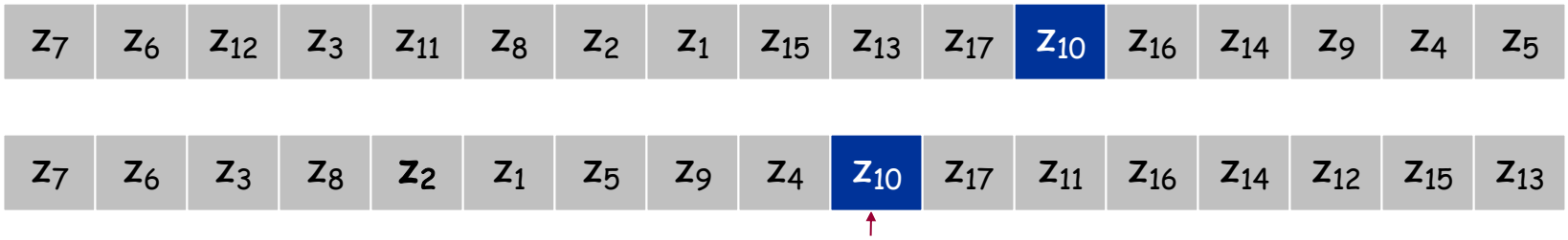
Average case analysis	Expected case analysis
Used for deterministic algorithms	Used for randomized algorithms
Assume the input is chosen randomly from some distribution	Need to work for any input
Depends on assumptions on the input, weaker	Randomization is inherent within the algorithm, stronger

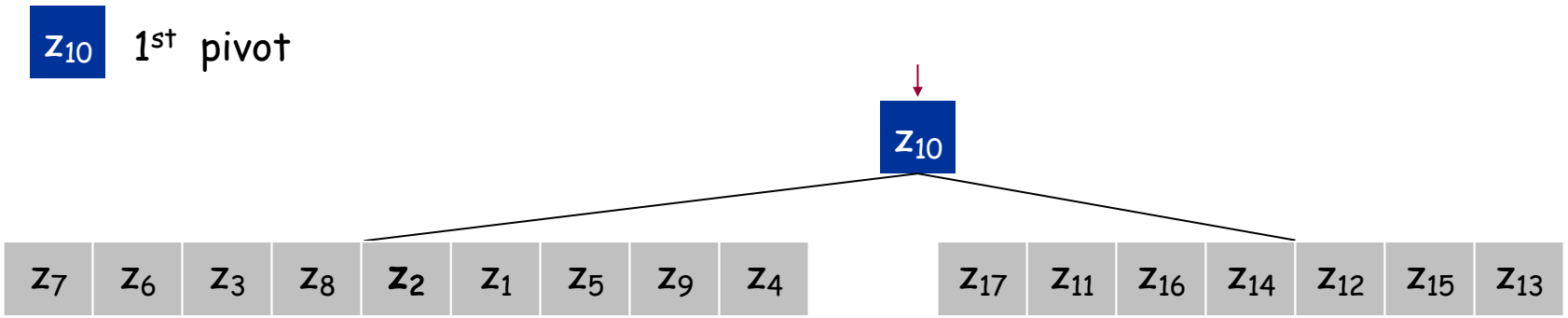
Our Randomized QuickSort Analysis

Analysis of Randomized Quicksort: The binary tree representation

Assumption: All elements are distinct - Relabel the elements from small to large as $z_1, z_2, \ldots, z_n$. We measure the running time based on the ***average number*** of comparisons performed.

Create a Binary tree corresponding to the Quicksort partitioning. The root of the tree will be the first pivot. All items to the left (right) of the root will be in its left (right) subtree.

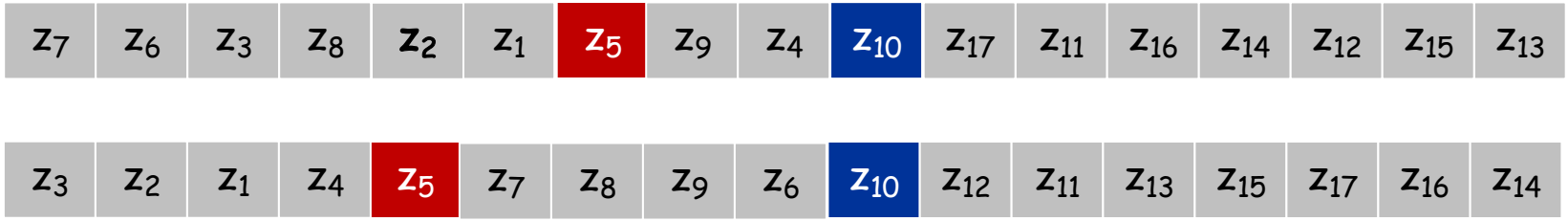


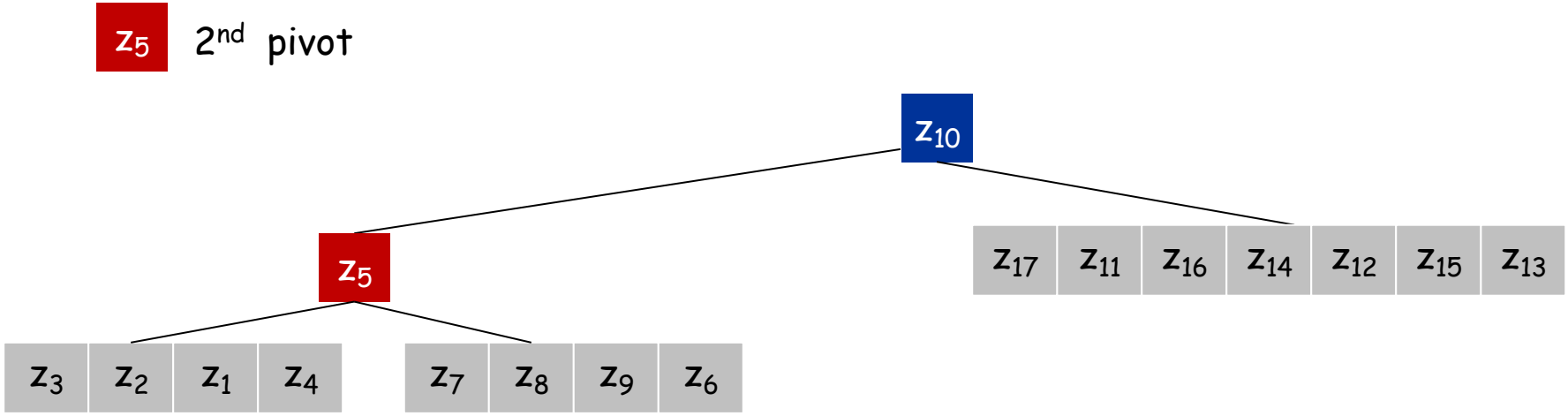


Analysis of Randomized Quicksort: The binary tree representation

Create a Binary tree corresponding to the Quicksort partitioning

Pivots will be nodes: items in subarray to left of pivot will be in left subtree; items to right of pivot, in right subtree

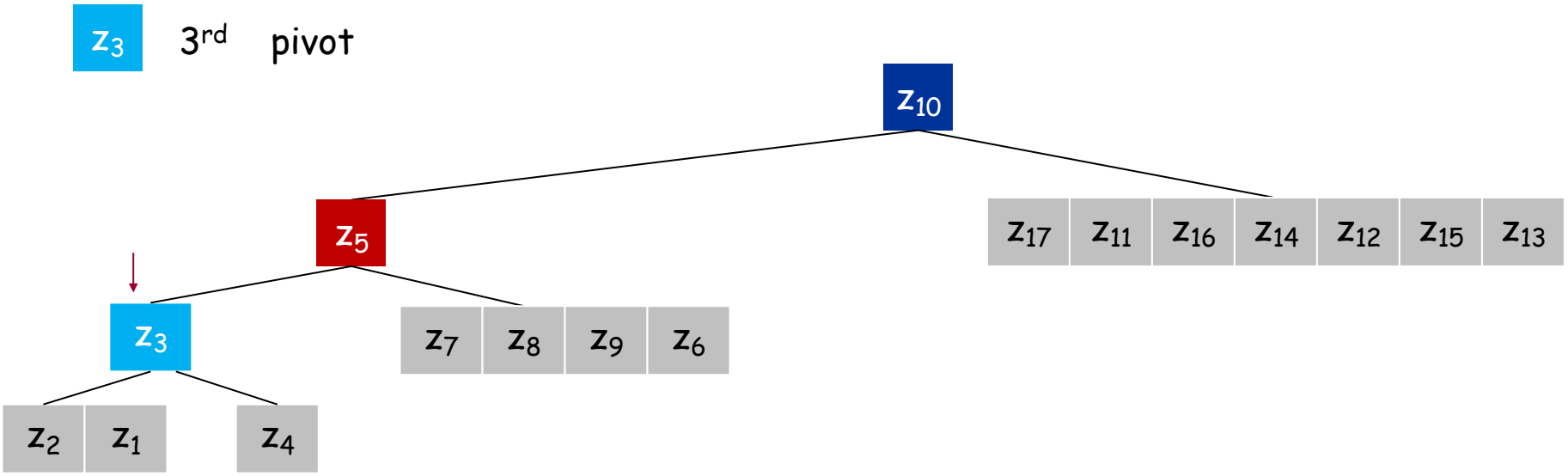




Analysis of Randomized Quicksort: The binary tree representation

z_3	z_1	z_4	z_2	z_5	z_7	z_6	z_8	z_9	z_{10}	z_{12}	z_{11}	z_{13}	z_{15}	z_{17}	z_{16}	z_{14}

z_2	z_1	z_3	z_4	z_5	z_7	z_8	z_9	z_6	z_{10}	z_{17}	z_{11}	z_{16}	z_{14}	z_{12}	z_{15}	z_{13}

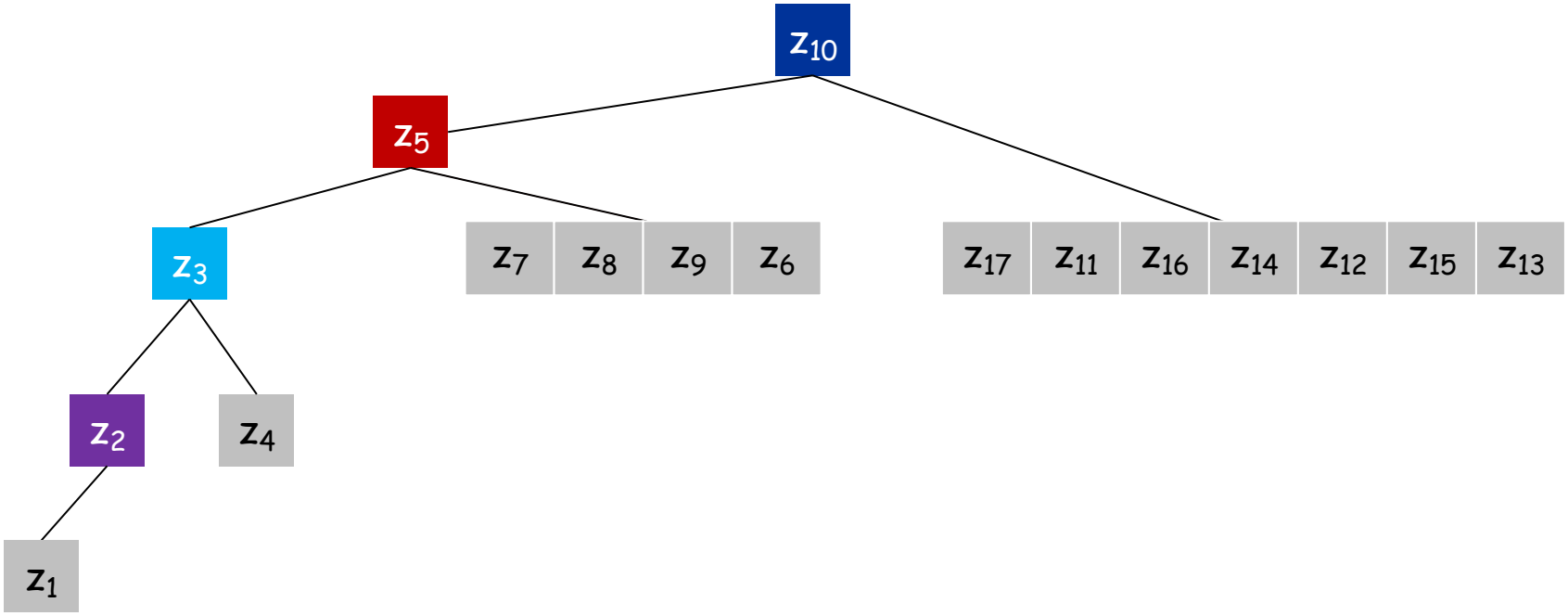


Analysis of Randomized Quicksort: The binary tree representation

z_3	z_1	z_4	z_2	z_5	z_7	z_6	z_8	z_9	z_{10}	z_{12}	z_{11}	z_{13}	z_{15}	z_{17}	z_{16}	z_{14}

z_2	z_1	z_3	z_4	z_5	z_7	z_8	z_9	z_6	z_{10}	z_{17}	z_{11}	z_{16}	z_{14}	z_{12}	z_{15}	z_{13}

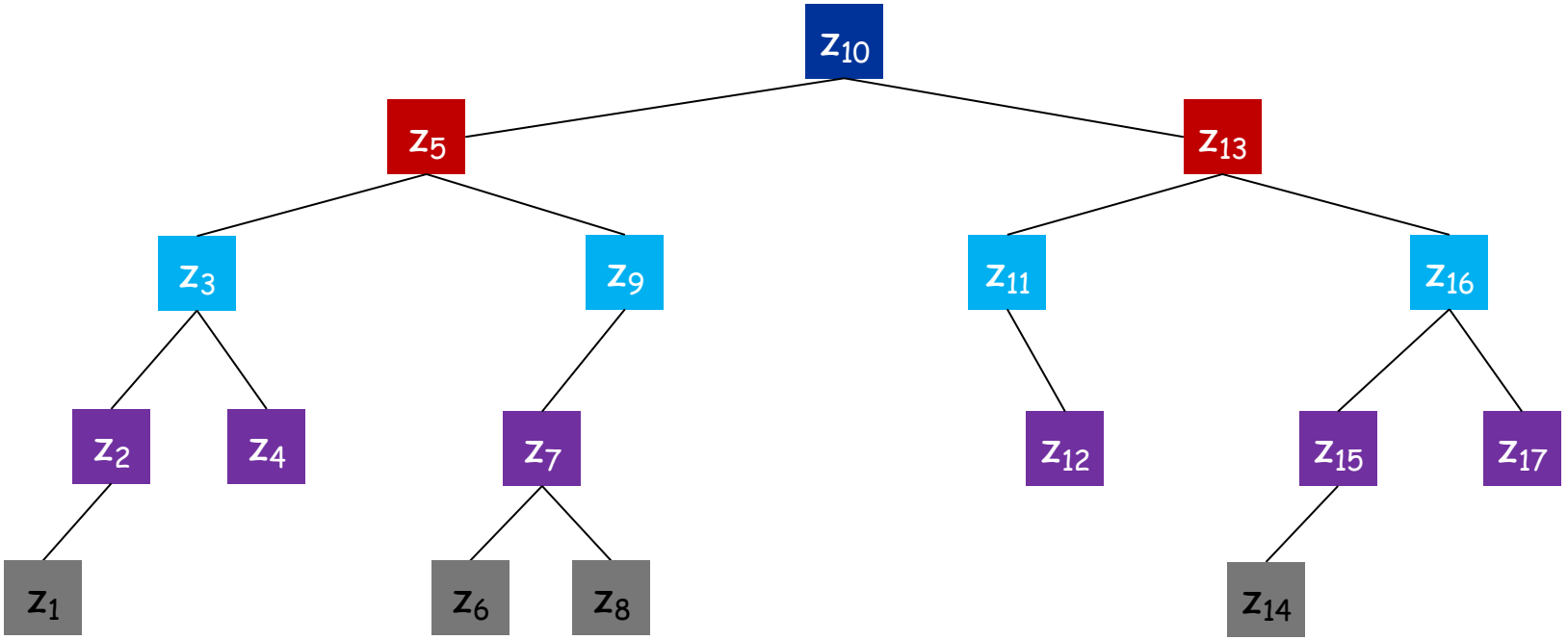
z_2 4th pivot



Analysis of quicksort: The binary tree representation

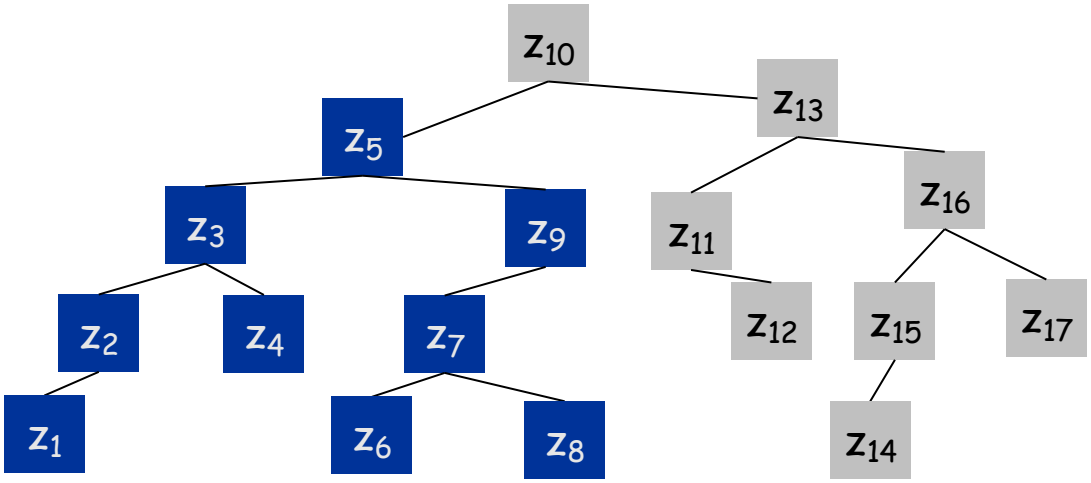
Keep going.

At the end we have a binary search tree that corresponds to the choice of pivots



Analysis of quicksort: The binary tree representation

z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8	z_9	z_{10}	z_{12}	z_{11}	z_{13}	z_{14}	z_{15}	z_{16}	z_{17}



The tree structure completely encodes the running of quicksort.

Let T_i be the subtree rooted at z_i

Ex: T_5 is the subtree rooted at z_5

This means that when z_i was a pivot, its subarray contained exactly the items in T_i.

Those items are then partitioned around z_i (corresponding to being placed in the left and right subtrees).

Fact:
(*) z_i is compared with z_j by Qsort
if and only if
(**) in the tree
z_i is an ancestor of z_j
or z_j is an ancestor of z_i

Summary of Observations

Elements z_i and z_j are compared once in Qsort iff in the binary tree:

1. z_i is an ancestor of z_j, i.e., z_i is the first pivot chosen among $\{z_i, z_{i+1}, \ldots, z_j\}$, or
2. z_j is an ancestor of z_i i.e., z_j is the first pivot chosen among $\{z_i, z_{i+1}, \ldots, z_j\}$

Otherwise, if another element $z_k \in \{z_i, z_{i+1}, \ldots, z_j\}$ is chosen as a pivot, z_i and z_j will go to different subtrees (subarrays) rooted at z_k and never be compared.

1. Define Indicator Random Variables: $X_{ij} = 1$ if z_i is compared with z_j
   Then # of comparisons is $X = \sum_{i<j} X_{ij}$
2. By Linearity of Expectation
   $E[\#\text{ of comparisons made by Qsort}] = \sum_{i<j} E[X_{ij}] = \sum_{i<j} \Pr[\, z_i, z_j \text{ compared}]$
3. Will show that
   $$\Pr[z_i, z_j \text{ compared}] = 2/(j - i + 1)$$
4. And also show
   $E[\#\text{ of comparisons made by Qsort}] = \sum_{I<k} 2/(j - i + 1) = O(n \log n).$

Analysis of quicksort

(*) z_i and z_j are compared iff z_i or z_j are chosen as the first pivot among $\{z_i, z_{i+1}, \ldots, z_j\}$

Consider the running of Quicksort.

1. For any pair i,j, consider first time that an item in $Z = \{z_i, z_{i+1}, \ldots, z_j\}$ is chosen as a pivot.
2. At each step of Qsort, every item in the current subarray is equally likely to be chosen as a pivot.
3. Because Qsort chooses pivots randomly, when the first pivot in Z is chosen it is equally likely to be any item in Z (which has size j-i+1).

$$\Rightarrow \Pr[z_i \text{ is first pivot chosen in } Z] = \frac{1}{j-i+1} = Pr[z_j \text{ is first pivot chosen in } Z]$$

$$\Rightarrow Pr[z_i \text{ and } z_j \text{ are compared}] = \Pr[\text{Either } z_i \text{ or } z_j \text{ are first pivot chosen in } Z] = \frac{2}{j-i+1}$$

4. E[# of comparisons made by Qsort] $= \sum_{i<j} E[X_{ij}] = \sum_{i<j} \Pr[z_i, z_j \text{ compared}] =$

$$\sum_{i<j} \frac{2}{j-i+1}$$

Analysis of quicksort (continued)

Theorem. Expected # of comparisons is $\Theta(n \log n)$.

Pf. Remains to show that $\sum_{i<j} \frac{2}{j-i+1} = \theta(n \log n)$

- Idea is to write out the summation as items in an upper triangular matrix and then show that each of the n rows has sum $O(\log n)$

	$j=2$	3	4	…	n	
$i=1$	$\frac{2}{2}$	$\frac{2}{3}$	$\frac{2}{4}$	…	$\frac{2}{n}$	$= O(\log n)$
2		$\frac{2}{2}$	$\frac{2}{3}$	…	$\frac{2}{n-1}$	$= O(\log n)$
3			$\frac{2}{2}$	…	$\frac{2}{n-2}$	$= O(\log n)$
…	…	…	…	…	…	…
$n-1$					$\frac{2}{2}$	$= O(\log n)$

2(1+1/2+…1/n)-1
1+1/2+…1/n is the harmonic series Θ(logn)

Since every row has sum $O(\log n)$ the entire matrix has sum $O(\text{n} \log n)$.

Quicksort in practice

Why does quicksort work very well in practice?

- $\Theta(n \log n)$ time in expectation on any input
  - Actually, it's $\Theta(n \log n)$ time with very high probability
- Small hidden constants and Cache-efficient
- Even though it has a $\Theta(n^2)$ worst-case running time it beats the $\Theta(n \log n)$ worst case Mergesort "on average"

Real Algorithmic Engineering

- Start with quicksort
- When recursion is too deep (say, $> 10 \log n$), switch to insertion sort or heap sort (discussed later)
- Use better ways of choosing "random" pivot
- Implemented in C++ Standard Template Library (STL)
- For details, see

  Engineering a Sort Function, J. L. Bentley & M.D. McIlroy

  SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 23(11), 1249–1265 (NOV 1993)

Randomized Selection: Main Idea

Selection. Given an unsorted array $A[1..n]$ and an integer i, return the i-th smallest element of $A[1..n]$.

- Trivial cases: If $i = 1$, return the minimum. If $i = n$ return the maximum.
- Special Case: If $i = n/2$, return the median.

1. Choose a Pivot x from $A[p..r]$
2. Partition A around x. (linear time)
3. After partitioning, pivot x will be at known location q.

- If $(i = q - p + 1)$
  Then x is the actual solution
- If $(i < q - p + 1)$
  Then the i-th element of $A[p..r]$ is the i-th element of $A[p..q-1]$.
  Solve recursively
- If $(i > q - p + 1)$
  Then the i-th element of $A[p..r]$ is the $j = (i - q + p - 1)$-th element of $A[q+1..n]$. Solve recursively

Example of Randomized Selection

Call Selection(A, 1, 15, 10)

A	1	2	3	4	9	6	7	8	11	10	12	5	15	14	13

p = 1, r = 15, i = 10, q = 13

return Selection(A, 1, 12, 10)

A	1	2	3	4	9	6	7	8	11	10	12	5	13	14	15

Goal is to find 10^{th} item in A[1..15].
Current problem is to find 10^{th} item in A[1..15]

Yellow item is current pivot; Grey items have been thrown away.
Search is only done in white (+yellow) subarray.

After Pivoting, problem is reduced to finding 10^{th} item in A[1..12]

Example (cont.)

Call Selection(A, 1, 12, 10)

A	1	2	3	4	9	6	7	8	11	10	12	5	13	14	15

p = 1, r = 12, i = 10, q = 5, j = 5

return Selection(A, 6, 12, 5)

A	1	2	3	4	5	6	7	8	11	10	12	9	13	14	15

Goal is to find 10^{th} item in A[1..15].
Current problem is to find 10^{th} item in A[1..12]

Yellow item is current pivot; Grey items have been thrown away.
Search is only done in white (+yellow) subarray.

After Pivoting, problem is reduced to finding 5^{th} item in A[6..12]

Example

Call Selection(A, 6, 12, 5)

A	1	2	3	4	5	6	7	8	11	10	12	9	13	14	15

p = 6, r = 12, i = 5, q = 9, j = 1

return Selection(A, 10, 12, 1)

A	1	2	3	4	5	6	7	8	9	10	12	11	13	14	15

Goal is to find 10^{th} item in A[1..15].
Current problem is to find 5^{th} item in A[6..12]

Yellow item is current pivot; Grey items have been thrown away.
Search is only done in white (+yellow) subarray.

After Pivoting, problem is reduced to finding 1^{st} item in A[10..12]

Example

Call Selection(A, 10, 12, 1)

A	1	2	3	4	5	6	7	8	9	10	12	11	13	14	15

p = 10, r = 12, i = 1, q = 11

return Selection(A, 10, 10, 1)

A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Goal is to find 10^{th} item in A[1..15].
Current problem is to find 1^{st} item in A[10..12]

Yellow item is current pivot; Grey items have been thrown away.
Search is only done in white (+yellow) subarray.

After Pivoting, problem is reduced to finding 1^{st} item in A[10..10]

Example

Call Selection(A, 10, 10, 1)

A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

p = 10, r = 10, i = 1

return A[10]

A	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Goal is to find 10^{th} item in A[1..15].
Current problem is to find 1^{st} item in A[10..10]

Yellow item is current pivot; Grey items have been thrown away.
Search is only done in white (+yellow) subarray.

After Pivoting, problem is solved!

Randomized Selection: Pseudocode

Selection. Given an array A of n distinct elements and an integer i, return the i-th smallest element of A. Examples:

Goal: Want to do better than sorting, i.e., linear time.

```
Select(A,p,r,i) : i-th smallest element of A[p..r]
if p = r then return A[p]
Randomly choose an element in A[p..r]
    as the pivot and swap it with A[r]
q ← Partition(A,p,r) //pivot position in A[p..r] after partition
k ← q − p + 1 //relative position of pivot in A[p..r]
if i = k then return A[q]
else if i < k then return Select(A,p,q − 1,i)
else return Select(A,q + 1,r,i − k)// //relative position of i in A[q + 1..r]

First call: Select(A,1,n,i)
```

Mathematical Revision

- Suppose you have a fair coin with probability $p = \frac{1}{2}$ of turning up Heads and $p = \frac{1}{2}$ of Tails.

- Start flipping. Let X be the number of flips until a Head is seen (we called it waiting time in previous lecture).
  We already saw $E[X] = 1/p = 2$.

- Keep flipping, marking every time new Head is seen.
- Let **stage *i*** be the time between the i-th Head and the (i+1)-st Head (not including i-th and including $(i+1)$st), $i = 0, 1, 2, \ldots$

- Set

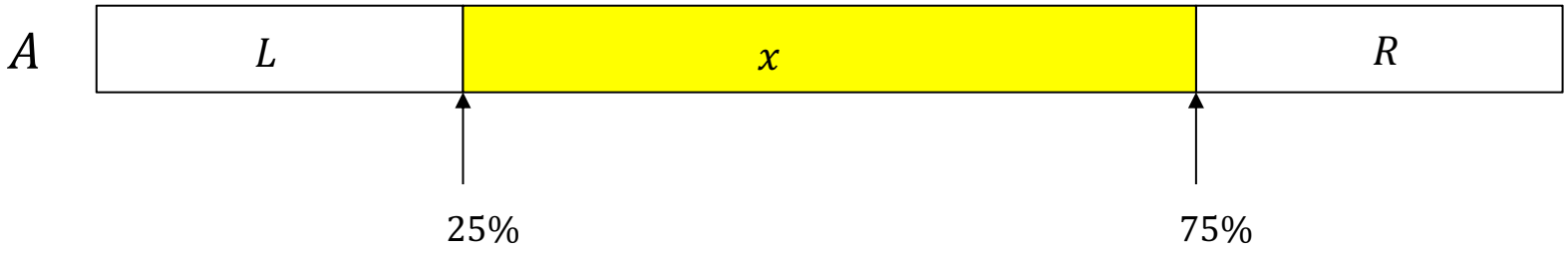
  X_0 = length of stage 0 = # of flips until first head is seen

  X_i = length of stage i

  = # of flips from after i-th Head is seen until $(i+1)^{st}$ Head is seen.

From above, for all i, $E[X_i] = 2$

Analysis of randomized selection



If pivot x falls within the (yellow) middle half
- either everything to the left of x, including box L is thrown away or
- everything to the right of x, including box R is thrown away.

So, if pivot x falls within the (yellow) middle half,
=> at least $1/4$ of the items are thrown away.

Call a pivot "good" if it's between the 25%- and 75%-percentile of A, otherwise "bad".

- The probability of a random pivot x being good is $1/2$.
- Each good pivot reduces size of array by at least $1/4$.

Analysis of randomized selection

Theorem: The expected running time of randomized selection is $\Theta(n)$.

Pf: Call a pivot "good" if it's between the 25%- and 75%-percentile of A

- The probability of a random pivot being good is 1/2.
- Each good pivot reduces n by at least $1/4$.

- After i good pivots, total # of items left to process is $\leq \left(\frac{3}{4}\right)^i n$
  - i.e., After throwing away $1/4$ of remaining items i times

- Let i-th stage be the time between the i-th good pivot (not including) and the $(i+1)$-st good pivot (including), $i = 0, 1, 2, \ldots i$
  - In i-th stage, # of items being processed in array is $\leq \left(\frac{3}{4}\right)^i n$

Analysis of randomized selection

Theorem: The expected running time of randomized selection is $\Theta(n)$.

Pf: A pivot is "good" if it is between the 25%- and 75%-percentile of A,

- The probability of a random pivot being good is $1/2$.
- After i good pivots, total # of items left to process is $\leq \left(\frac{3}{4}\right)^i n$

Let Y_i = the running time of i^{th} stage

X_i = the # of pivots (recursive calls) in i^{th} stage

$Y_i \leq X_i \left(\frac{3}{4}\right)^i n$

From the mathematical revision

$E[X_i] = 2$ (waiting time) so $E[Y_i] \leq E\left[X_i \left(\frac{3}{4}\right)^i n\right] = 2\left(\frac{3}{4}\right)^i n$

=> Expected total running time $\leq E[\sum_i Y_i] = \sum_i E[Y_i] \leq \sum_i 2\left(\frac{3}{4}\right)^i n = O(n)$.

Remark: A deterministic linear-time selection algorithm exists but it is complicated

Exercise on Sorting and Selection

You are given two sorted arrays A1 and A2 of sizes m and n. Design an algorithm to find the k-th smallest element in the union of the elements in A1 and A2 ($k \leq \min(m, n)$). Analyze the time complexity of your algorithm.

1] Very Naïve method

Merge A1 and A2 into a sorted array of size m+n. Return the kth element in the merged array. Cost $\Theta(m + n)$.

2] Naïve method

Start merging A1 and A2 (using the merge function of merge sort). Stop when you reach the kth element in the merged array. Cost $\Theta(k)$.

Exercise on Sorting and Selection (cont)

D&C Method

Lets call the method: Search(arrays A1, A2, integers start1, end1, start2, end2, k)

First call: Search(A1,A2,$1, k, 1, k, k$)

Main Idea: Compare elements A1[start1+$k/2-1$] and A2[start2 + $k/2-1$]
If A1[start1+$k/2-1$] $<$ A2[start2 + $k/2-1$]:
Eliminate the first $k/2$ elements of A1
Return Search(A1, A2, start1 + $k/2$, end1, start2, end2, $k/2$)

Else (A1[start1+$k/2-1$] $\geq$ A2[start2 + $k/2-1$]):
Eliminate the first $k/2$ elements of A2
Return Search(A1, A2, start1, end1, start2+ $k/2$, end2, $k/2$)

Base case: If $k = 1$, return $\min(A1[1], A2[1])$

Cost: $\Theta(\log k)$