Lecture 6: Quicksort and Linear-Time Selection

Version of March 14, 2019

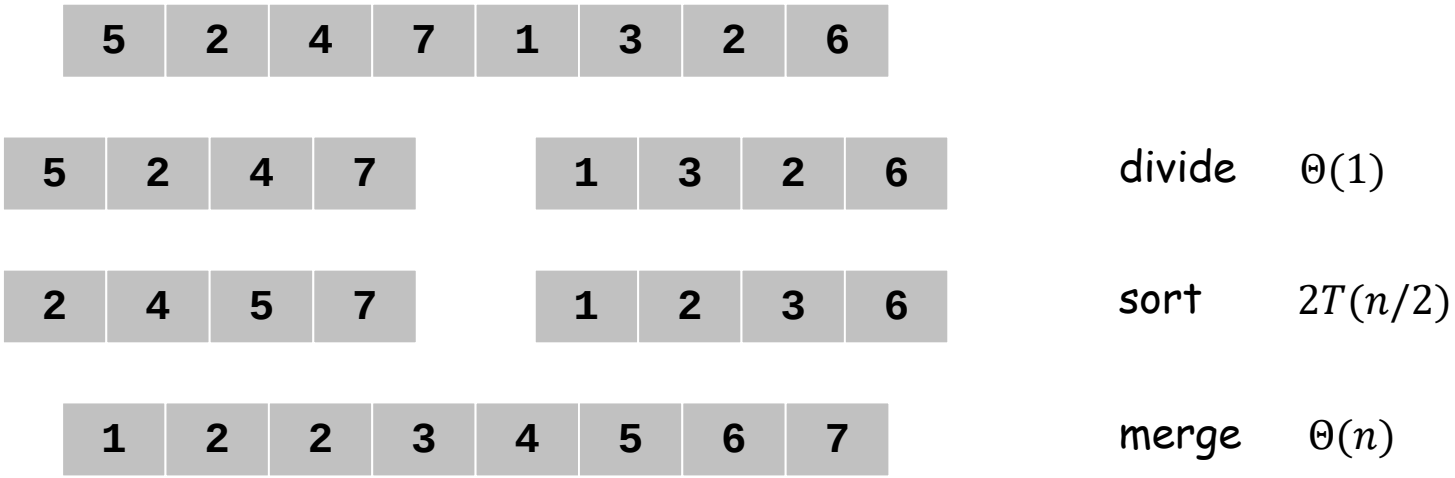
Outline

1. Quicksort & Partition
2. Run Time & Randomization
3. Analysis of Randomized Quicksort
4. Quicksort in Practice
5. Randomized Selection

Quicksort: The "dual" of merge sort

```
Mergesort(A,p,r):
if p = r then return
q ← ⌊(p + r)/2⌋
Mergesort(A,p,q)
Mergesort(A,q + 1,r)
Merge(A,p,q,r)

First call: Mergesort(A,1,n)
```



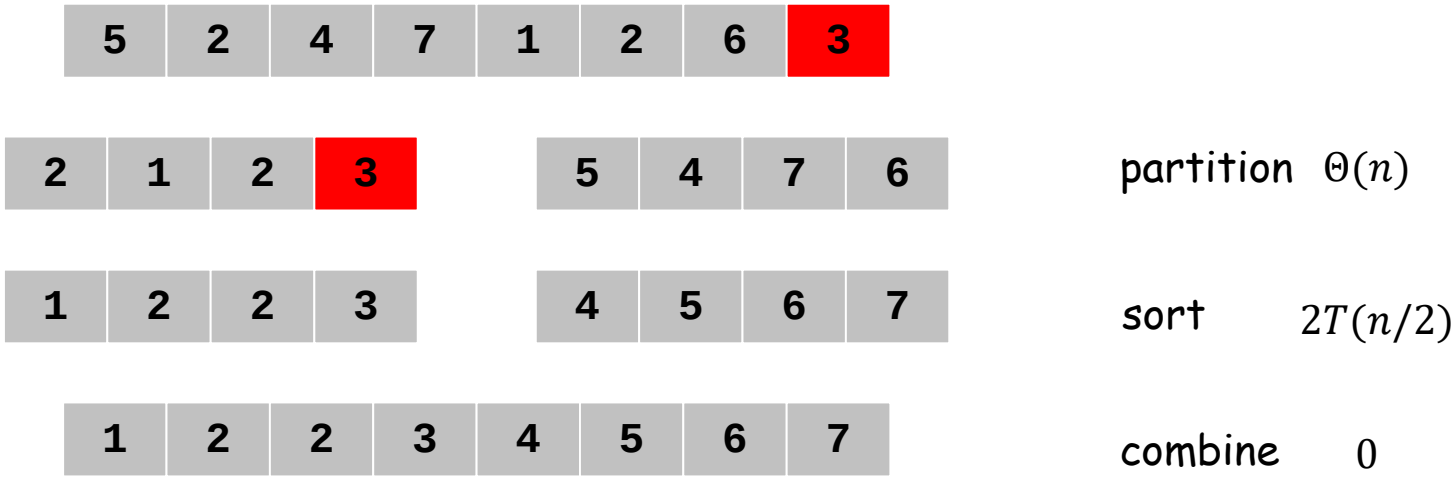
Quicksort: The "dual" of merge sort [II]

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]

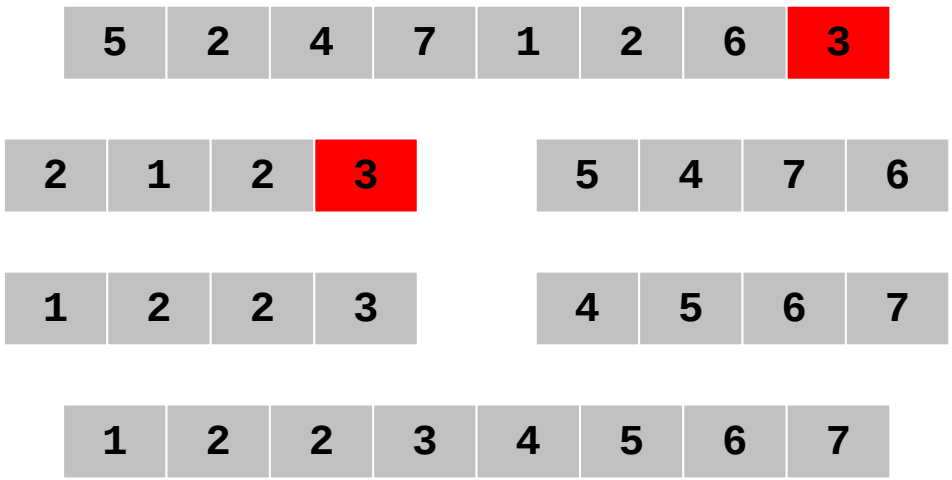
Quicksort chooses item as pivot.

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.

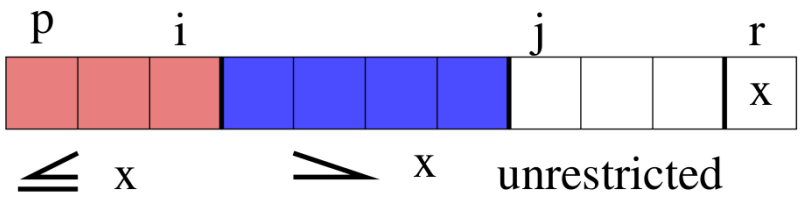
It then recursively Quicksorts left and right sides.

Quicksort(A, p, r):
if $p \geq r$ **then return**
$q =$ **Partition**(A, p, r)
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** $r - 1$

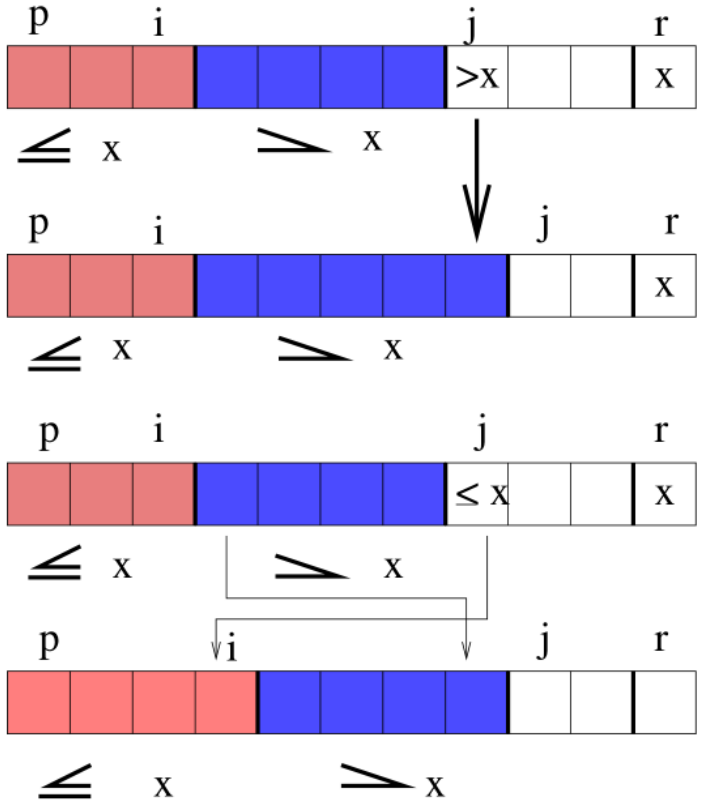
  if $A[j] \leq x$ **then**

    $i \leftarrow i + 1$

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

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

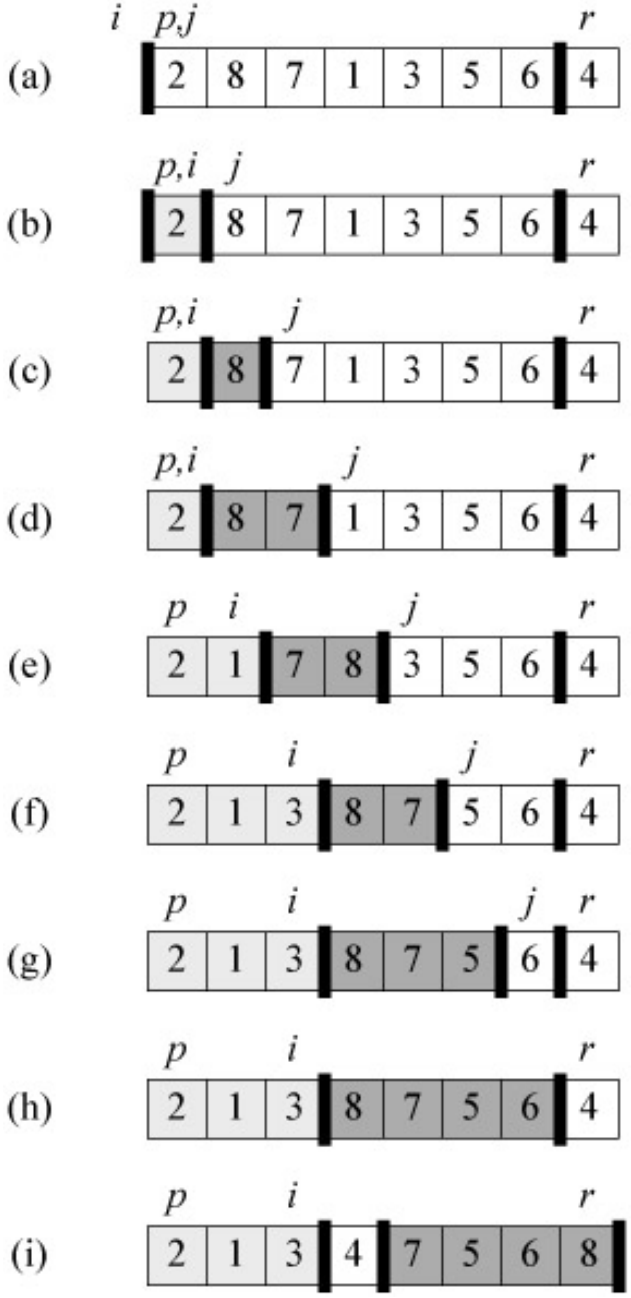
return $i + 1$



Time: $\Theta(n)$

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

Partition: Example



Outline

1. Quicksort & Partition
2. Run Time & Randomization
3. Analysis of Randomized Quicksort
4. Quicksort in Practice
5. Randomized Selection

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 is making 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: $\max_I E_R[T(I,R)]$,

I is any input of size n, R is the set of random numbers used internally, i.e., this is **expected** running time (expectation over the random numbers) on the worst possible input.

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

Outline

1. Quicksort & Partition
2. Run Time & Randomization
3. Analysis of Randomized Quicksort
4. Quicksort in Practice
5. Randomized Selection

Analysis of Randomized Quicksort:

Assumption: All elements are distinct

Note: Running time = Θ(# comparisons)

Relabel the elements from small to large as $z_1, z_2, \ldots, z_n$

Want to analyze ***average number*** of comparisons performed.

Will say that *z_i and z_j are compared by Quicksort* if
z_i is pivot and z_j is in subarray that is compared to z_i or vice-versa

---

1. Define Indicator RVs: $X_{ij} = 1$ if z_i is compared with z_j
   Then # of comparisons is $X = \sum_{i<j} X_{ij}$

2. By Linearity of Expectation
   E[# 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[# of comparisons made by Qsort] $= \sum_{i<j} 2 / (j - i + 1) = O(n \log n)$.

Analysis of Randomized Quicksort: The binary tree representation

Create a Binary tree corresponding to the Quicksort partitioning

The root of the tree will be the first pivot.

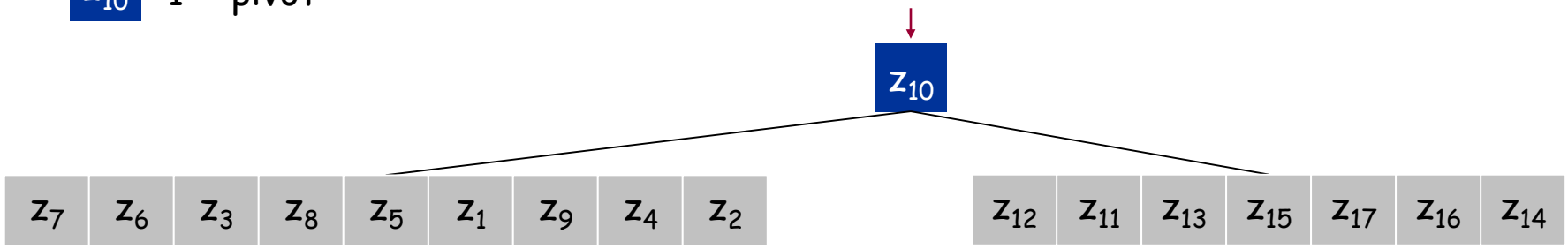
All items to the left (right) of the root will be in its left (right) subtree.

This process recurses in each subtree, until all items are nodes.

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

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

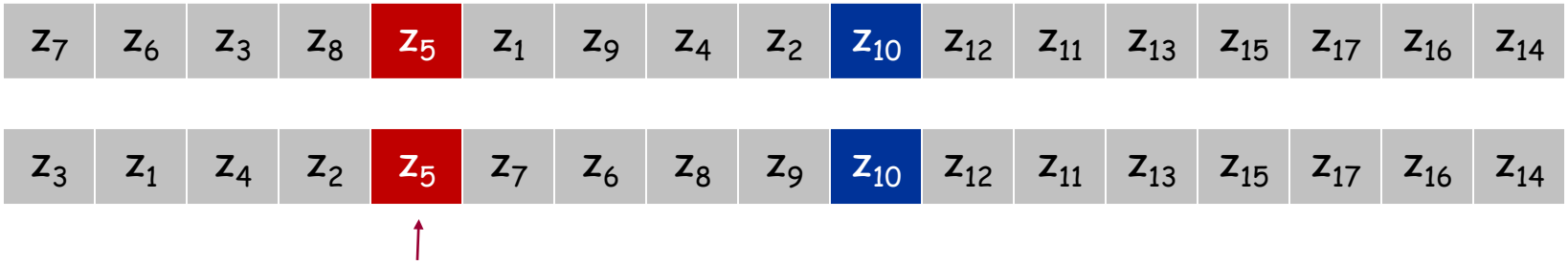
z_{10} 1st pivot

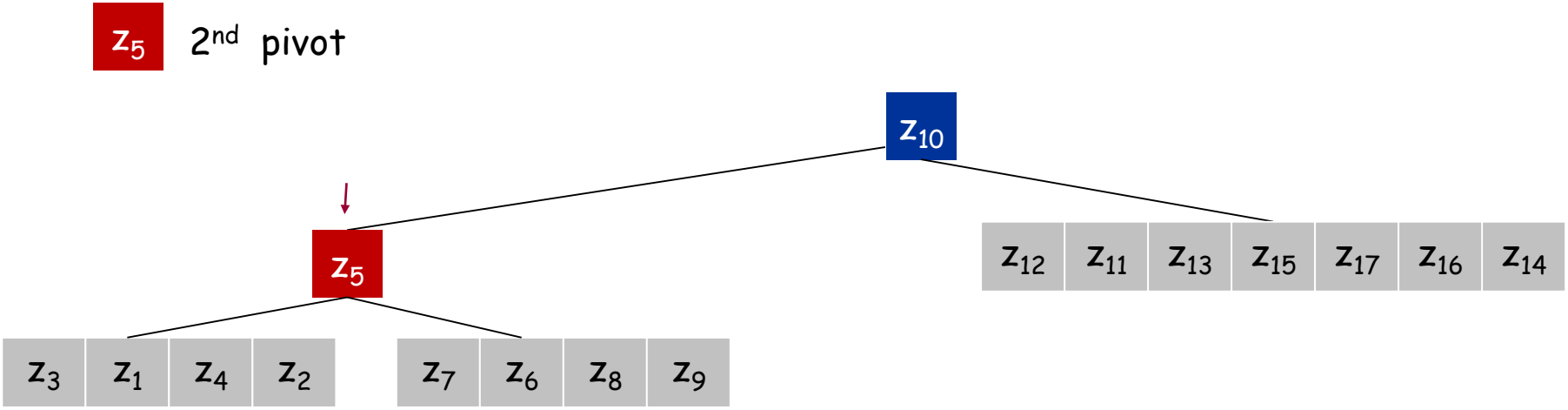


Analysis of Randomized Quicksort: The binary tree representation

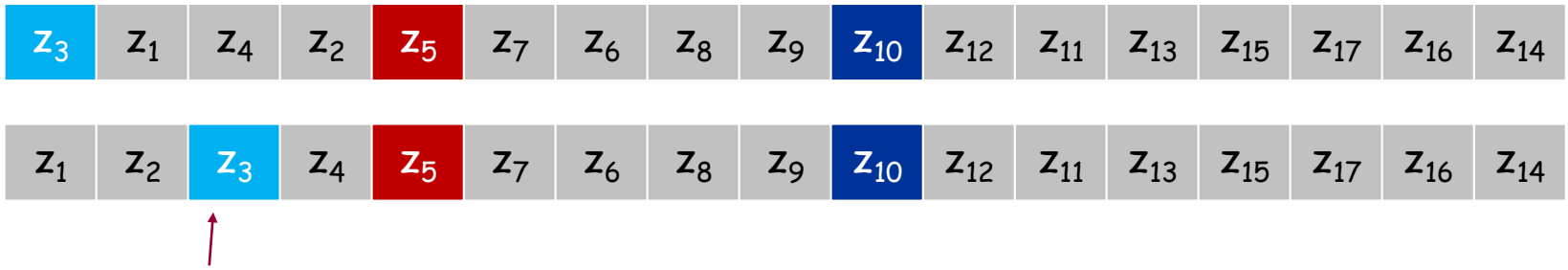
Create a Binary tree corresponding to the Quicksort partitioning

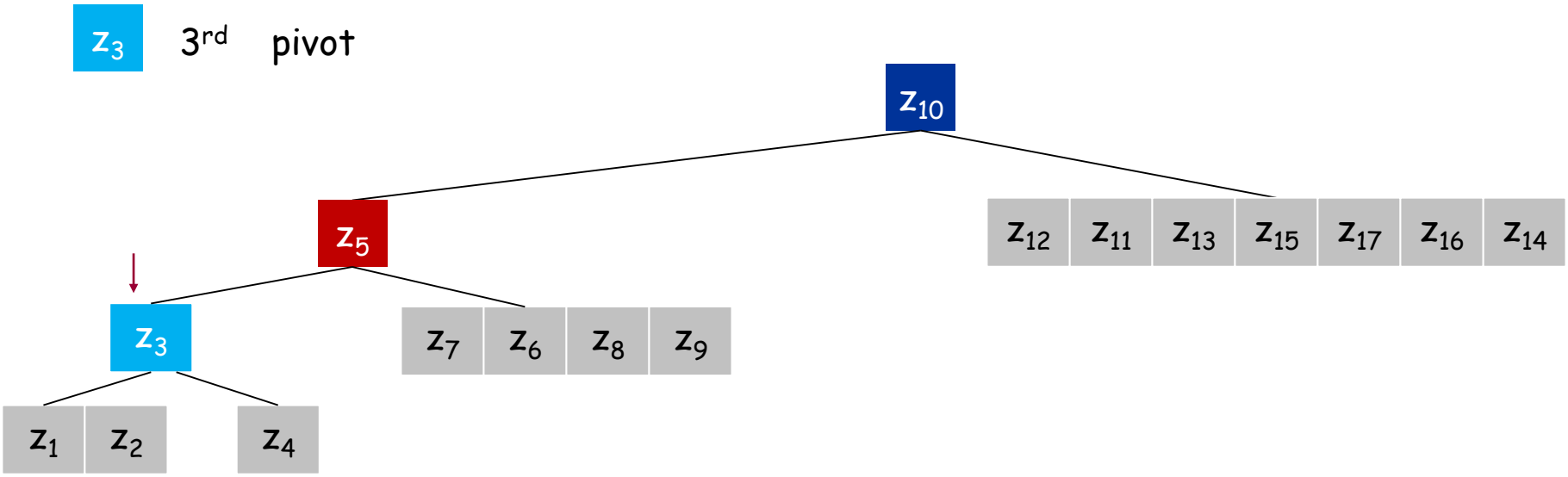
Pivots will be nodes: items in subarray to left of pivot will be in left subtree; items to right of pivot, in right subtree



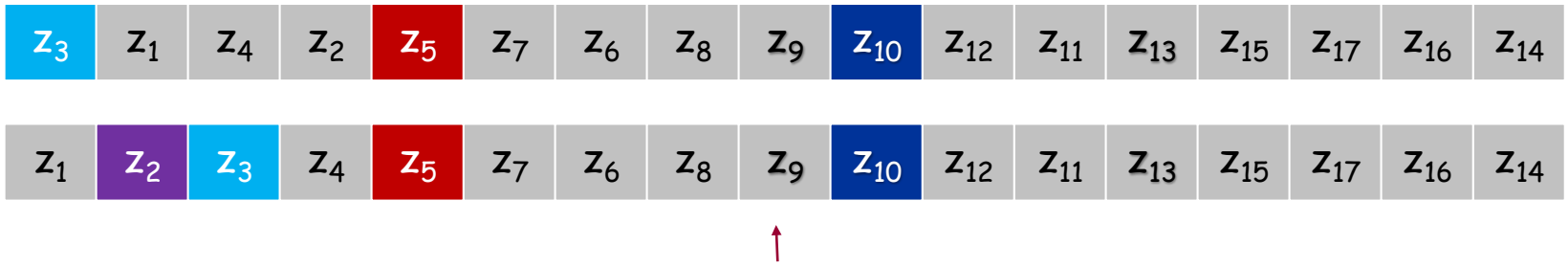


Analysis of Randomized Quicksort: The binary tree representation

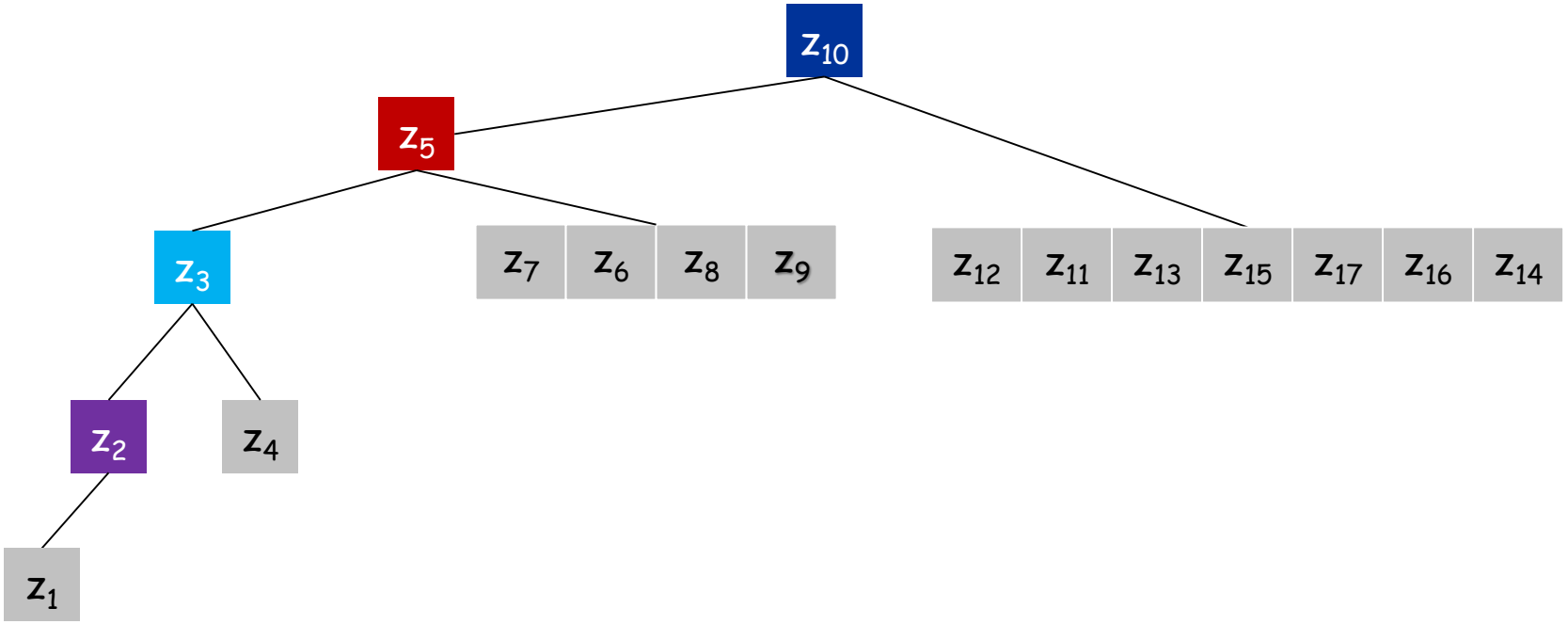




Analysis of Randomized Quicksort: The binary tree representation



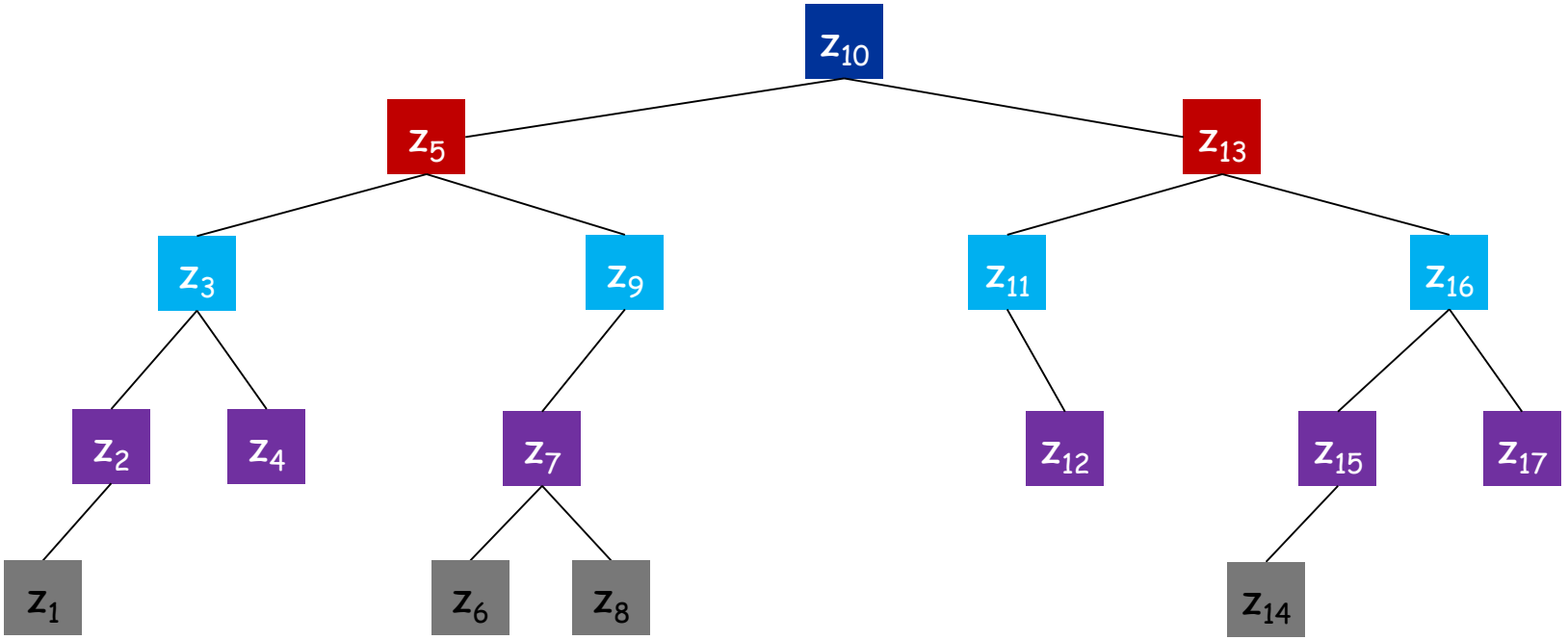
z_2 4th pivot



Analysis of quicksort: The binary tree representation

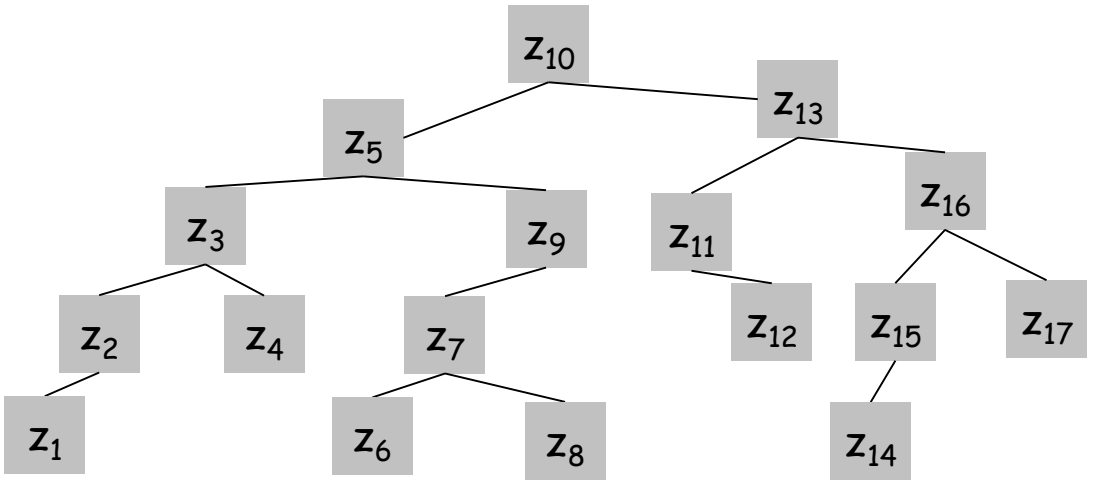
Keep going.

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}

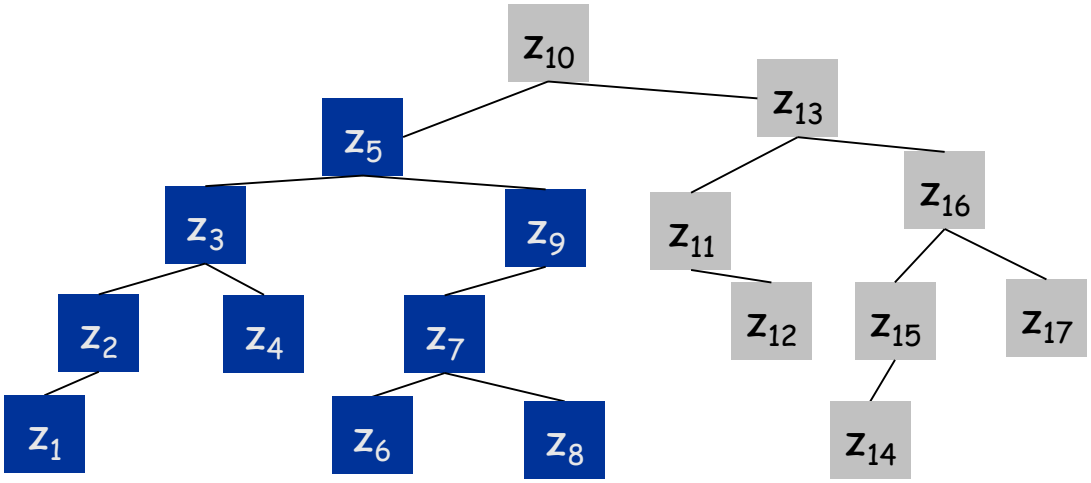


The tree structure completely encodes the running of quicksort.

Let T_i be the subtree rooted at z_i

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 subarry 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

Next slides derive that (**) occurs with Probability $\frac{2}{j-i+1}$

Analysis of quicksort

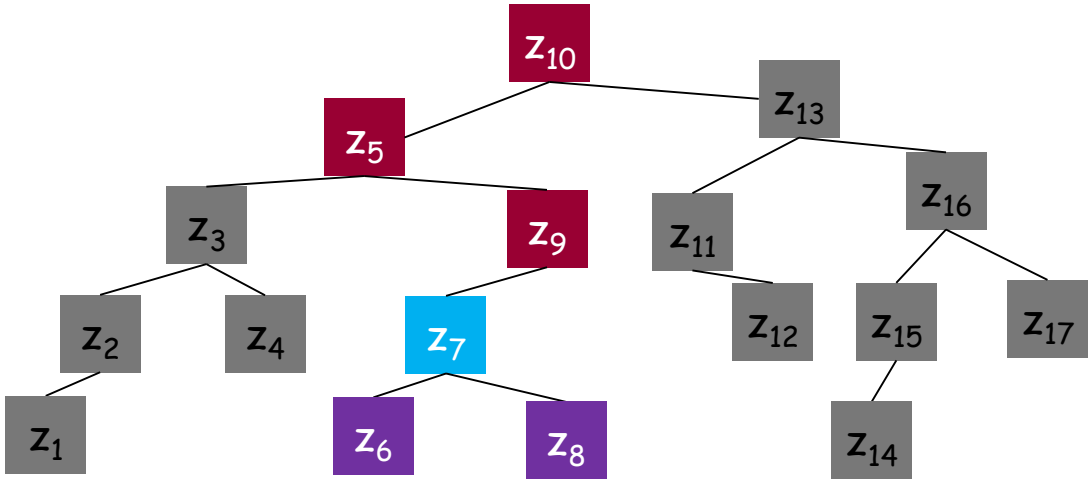
Observation 1:

Element **is only compared in Quicksort**

with its ancestors *(pivots for subarrays it is in)* in tree

and descendants *(items in subarray for which it is pivot)* in tree

- Example: z_7 is compared to ancestors z_9, z_5, z_{10} and descendants z_6, z_8



Analysis of quicksort

Observation 1:

Element **is only compared in Quicksort**

with its ancestors *(pivots for subarrays it is in)* in tree

and descendants *(items in subarray for which it is pivot)* in tree

- Example: z_7 is compared to ancestors z_9, z_5, z_{10} and descendants z_6, z_8

Observation 2: Consider walking down from root, searching for some item in set $Z=\{z_i, z_{i+1}, \ldots, z_j\}$.

The search path is the same until the first time that an item $z_k \in Z$ is seen at a node (why?)

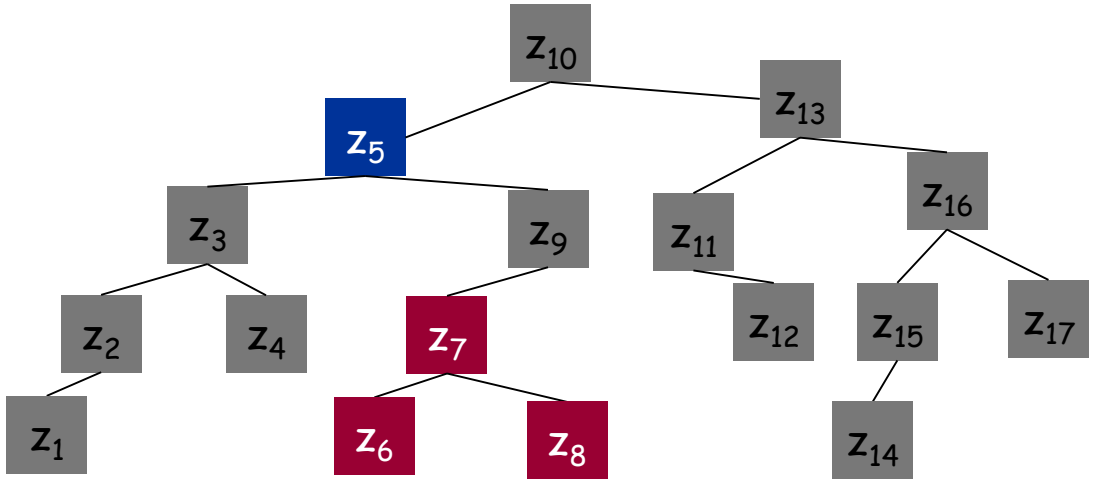
z_k denotes first pivot in Z

All other items in Z are compared with z_k

3 Possibilities

a) $z_i = z_k$

=> in particular, z_j compared to z_i



$Z=\{z_5, z_6, z_7, z_8\}$

Analysis of quicksort (when is z_i compared to z_j?)

Observation 1:

Element **is only compared in Quicksort**

with its ancestors *(pivots for subarrays it is in)* in tree

and descendants *(items in subarray for which it is pivot)* in tree

- Example: z_7 is compared to ancestors z_9, z_5, z_{10} and descendants z_6, z_8

Observation 2: Consider walking down from root, searching for some item in set $Z=\{z_i, z_{i+1}, \ldots, z_j\}$.

The search path is the same until the first time that an item $z_k \in Z$ is seen at a node (why?)

z_k denotes first pivot in Z

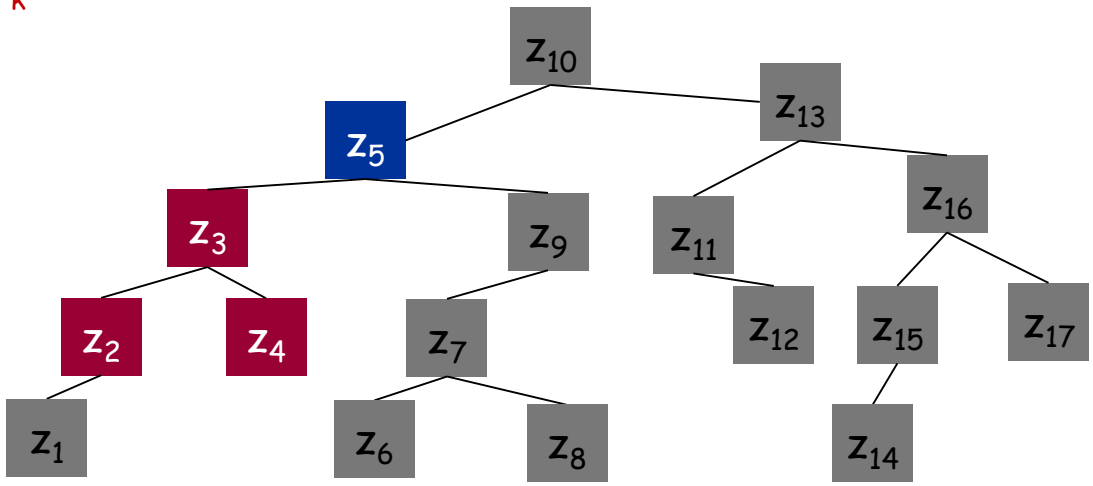
All other items in Z are compared with z_k

3 Possibilities

a) $z_i = z_k$

b) $z_j = z_k$

=> in particular, z_i compared to z_j



$Z=\{z_2, z_3, z_4, z_5\}$

Analysis of quicksort

Observation 1:

Element **is only compared in Quicksort**

with its ancestors *(pivots for subarrays it is in)* in tree

and descendants *(items in subarray for which it is pivot)* in tree

- Example: z_7 is compared to ancestors z_9, z_5, z_{10} and descendants z_6, z_8

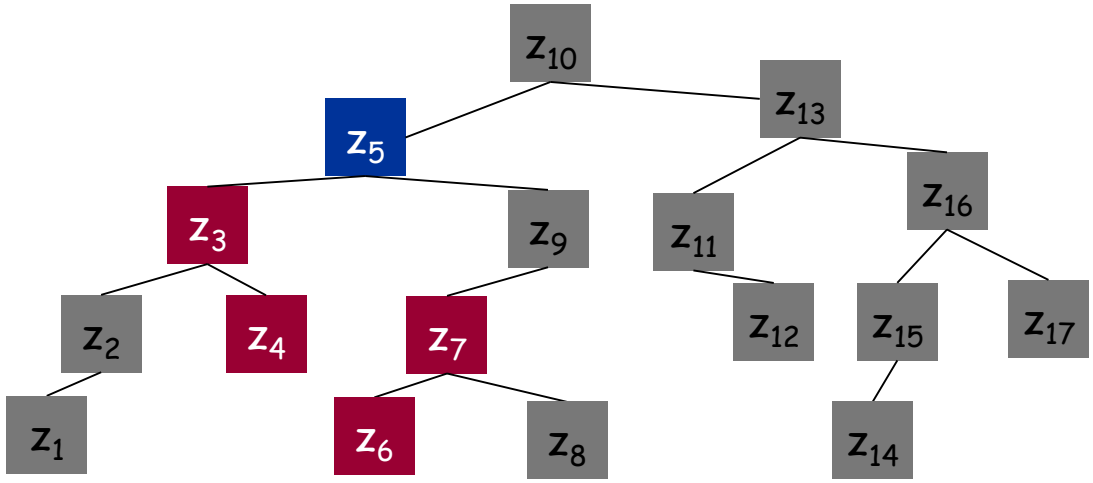
Observation 2: Consider walking down from root, searching for some item in set $Z=\{z_i, z_{i+1}, \ldots, z_j\}$.

The search path is the same until the first time that an item $z_k \in Z$ is seen at a node (why?)

z_k denotes first pivot in Z
All other items in Z are compared with z_k
3 Possibilities

a) $z_i = z_k$
b) $z_j = z_k$

c) $z_i \neq z_k$ and $z_j \neq z_k$
The first item seen is neither z_i, z_j
In this case z_i, z_j are put in different subarrays and never compared to each other



$Z=\{z_3, z_4, z_5, z_6, z_7\}$

Analysis of quicksort

From previous Observations, elements z_i and z_j are only compared in Qsort if, in the binary tree representation,
(1) z_i is an ancestor of z_j, or (2) z_j is an ancestor of z_i

1) z_i is an ancestor of z_j iff z_i is the first pivot chosen among $\{z_i, z_{i+1}, \ldots, z_j\}$
2) z_j is an ancestor of z_i iff z_j is the first pivot chosen among $\{z_i, z_{i+1}, \ldots, z_j\}$
3) 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 again.

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

Note: The tree representation was only introduced to derive statement (*).
Now that we know (*), we totally forget about the tree.

Analysis of quicksort

(*) z_i and z_j are compared iff z_i or z_i 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).

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

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

Analysis of quicksort (continued)

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

Proof.

1) Let $X_{ij} = 1$ if z_i is compared with z_j by Quicksort. Then (prev slides)

$$E(X_{ij}) = \Pr(X_{ij} = 1) = \frac{\mathbf{2}}{\boldsymbol{j - i + 1}}$$

2) # of comparisons made by Quicksort is $X = \sum_{i<j} X_{ij}$

3) Expected # of Comparisons made by Quicksort is then

$$\mathrm{E}(X) = E\left(\sum_{i<j} X_{ij}\right) = \sum_{i<j} E(X_{ij}) = \sum_{i<j} \frac{2}{j - i + 1}$$

$= \theta(n \log n)$ ←

Next Page

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			$\frac{2}{2}$	…	$\frac{2}{n-2}$	$= O(\log n)$
…	…	…	…	…	…	…
$n-1$					$\frac{2}{2}$	$= O(\log n)$

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

How can this be improved to $\theta(n \log n)$?

Outline

1. Quicksort & Partition
2. Run Time & Randomization
3. Analysis of Randomized Quicksort
4. Quicksort in Practice
5. Randomized Selection

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)

Outline

1. Quicksort & Partition
2. Run Time & Randomization
3. Analysis of Randomized Quicksort
4. Quicksort in Practice
5. Randomized Selection

Randomized Selection

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

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)
k ← q − p + 1
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)
```

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

Intuition is on next slide

Analysis: Textbook method very complicated (involving a lot of math)
In tutorial: Use indicator random variables, similar to quicksort
Here: A simple and clever Analysis method

Randomized Selection

Selection. Given a subarray $A[p..r]$ and an integer i, return the i-th smallest element of $A[p..r]$.

1. Choose a Pivot x AT RANDOM from $A[p..r]$
2. Partition $A[p..r]$ around x. (linear time)
3. After partitioning, item x will be at known location q.
   - Let $k = q - p + 1$.
   - x is the k'th smallest item in $A[p..r]$

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

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 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 good Head (not including i-th and including (i+1)st), i=0, 1, 2
- 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

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, otherwise "bad".

- The probability of a random pivot being good is $1/2$.
  - **This is probability $\frac{1}{2}$ of falling between the 25% and 75%**

- **Each good pivot reduces n by at least $1/4$.**
  - After a pivoting step, either all items to left of pivot are thrown away or all items to right are thrown away.
    In both cases, this is at least $\frac{1}{4}$ of the items

- **After i good pivots, total # of items left to process is $\leq \left(\frac{3}{4}\right)^i n$**
  - i.e., After throwing away $\frac{1}{4}$ 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, …
  - **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 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 (Sec 9.3 in textbook) but it is VERY complicated

Space analysis (advanced)

Observation: The selection algorithm can be written recursively, but it is tail-recursive.
When written without using recursion, it uses $O(1)$ working memory.

Note: Good compilers often do this automatically!

Working space of quicksort:

- Quicksort is not tail-recursive.
- Each level of recursion needs $O(1)$ working space (on system stack).
- In worst case, there can be $\Theta(n)$ levels of recursion => $\Theta(n)$ space.

- Can reduce to $O(\log n)$ space by the following trick:
  - Recurse into the smaller half first
  - Handle the second recursive call by tail recursion.