CS630 Graduate Algorithms

November 12, 2024

by Dora Erdos and Jeffrey Considine

Randomized quicksort and median finding CLRS ch 7 and ch 9

slides by Dora Erdos based on C. Tsourakakis, S. Thankachan

Sorting

Today:

- how does the input influence the runtime of some algorithms?
- worst case vs average case runtime
- randomized variant of algorithm
  - expected runtime analysis

Comparison-based sorting

input: unsorted array $A = [a_1, a_2, \ldots, a_n]$

output: permutation $A' = [a'_1, a'_2, \ldots, a'_n]$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Is the output the same for any order of the input?

Some sorting algorithms:

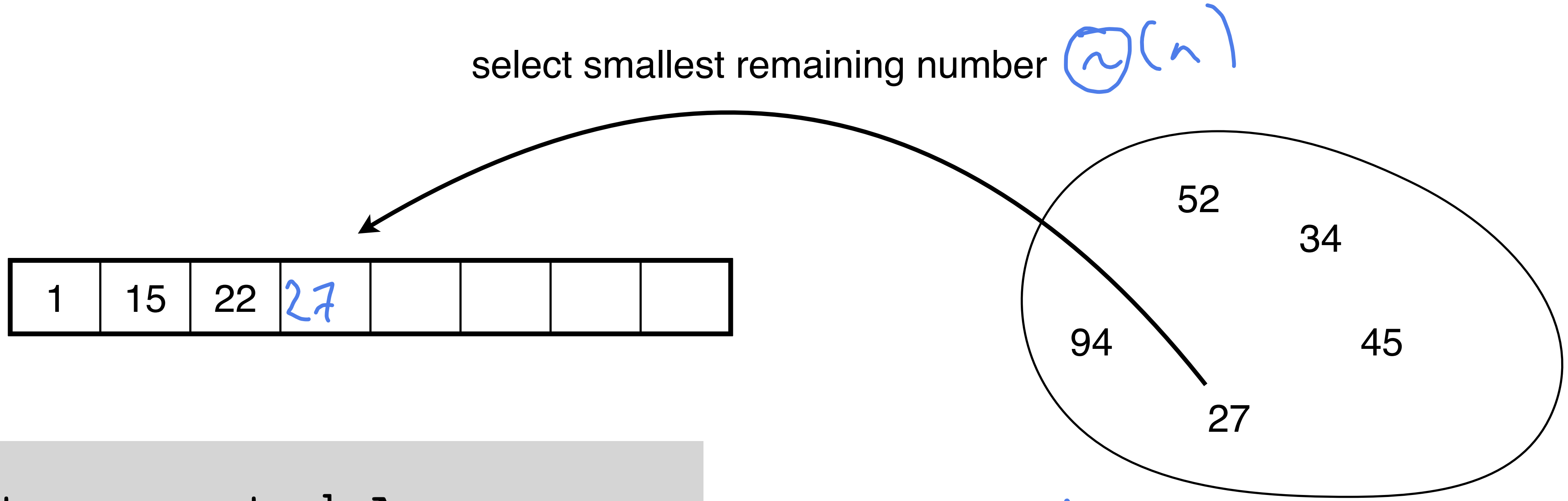
Selection Sort

Insertion Sort

QuickSort

Linear Selection

Selection Sort




```
input: unsorted A
For i = 0 to n-1
  • j = argmin(A[i,:])
  • swap A[i] and A[j]
output: sorted A
```

return index where A[j] is min

A[p, r] = subarray starting at index p up to r (inclusive)

A[p, :] = subarray from index p to the end

runtime: $n + (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$

first lowest

Does the runtime depend on the input?

in each iteration to find the next min we have to iterate over all remaining items, regardless of order

Recurrences

Function to express the running time of an algorithm on an input of size n

T(n) = the (asymptotic) number of computational steps that the algorithm performs on an input of size n

goal: find a simple arithmetic formula for T, e.g. $T(n) = \Theta(n), \Theta(n \log n), \Theta(n^3)$

examples:

find min & swap

Selection Sort: $T(n) = n + T(n-1)$ $\quad T(n) = \Theta(n) + \underbrace{T(n-1)}$

repeat same algo recursive on input of n-1

MergeSort : $T(n) = 2T\left(\frac{n}{2}\right) + c \cdot n$

Write recurrence

```
foo(array A):
//A has length n
a = 1 + foo(A[0,n/2])
b = 1 + foo(A[n/2+1:n-1])
return a+b
```

n == 1 : return 1

total # of recursive calls if we always divide array into two equal parts and call on two halves

number of calls

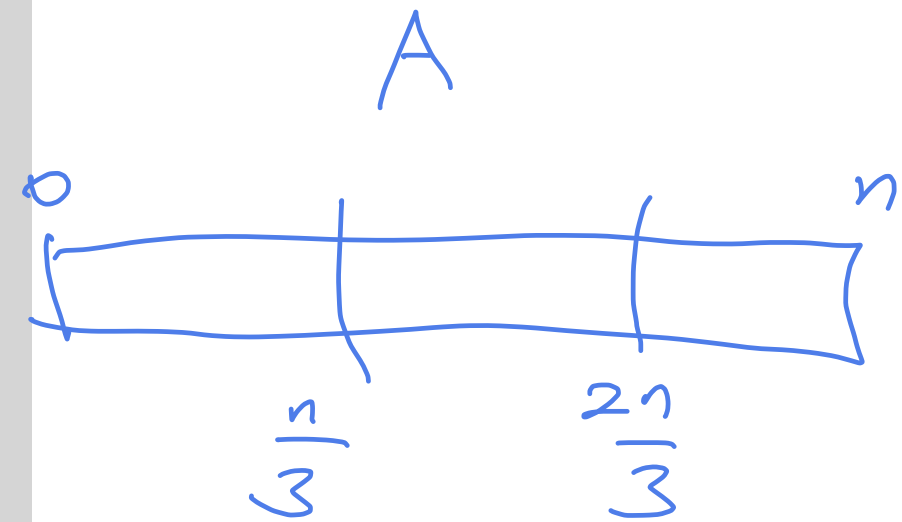
input of size $\frac{n}{2}$

$$T(n) = \underbrace{2\,T\left(\frac{n}{2}\right)}_{\text{corresponding to recursive calls}} + \underbrace{\text{const}}_{\text{happening in place}}$$

runtime on input of n

Write recurrence

```
fun(array A):
//A has length n
val = 0
for i = 0 to n/3
   val += A[i]
val += fun(A[n/3+1; 2n/3])
val += fun(A[2n/3+1; n-1])
return val
```



T(n) = $\underbrace{2T\left(\frac{n}{3}\right)} + \Theta(n)$

Write recurrence

Some algorithm A divides the size-n input into *b* equal parts. It calls itself recursively on *a* of those parts. Finally it spends $\Theta(f(n))$ combining the results and performing other local (non-recursive) operations

how many recursive calls

$$T(n) = aT\left(\frac{n}{b}\right) + \Theta(f(n))$$

operations outside of the rec calls

size of subproblem

~~TopHat - write recurrence~~

```
foo(array A):
//A has length n
if n = 1:
   return A[0]
a = foo(A[0:n/4])
b = foo(A[3n/4+1: n-1])
c = 0
for i = n/4+1 to 3n/4
   c+= A[i]
return a+b+c
```

practice at home

Select the correct expression for T(n):

A. $T(n) = 2T\left(\frac{n}{2}\right) + n$

B. $T(n) = 4T\left(\frac{n}{4}\right) + \Theta(1)$

C. $T(n) = 2T\left(\frac{n}{4}\right) + \Theta(n)$

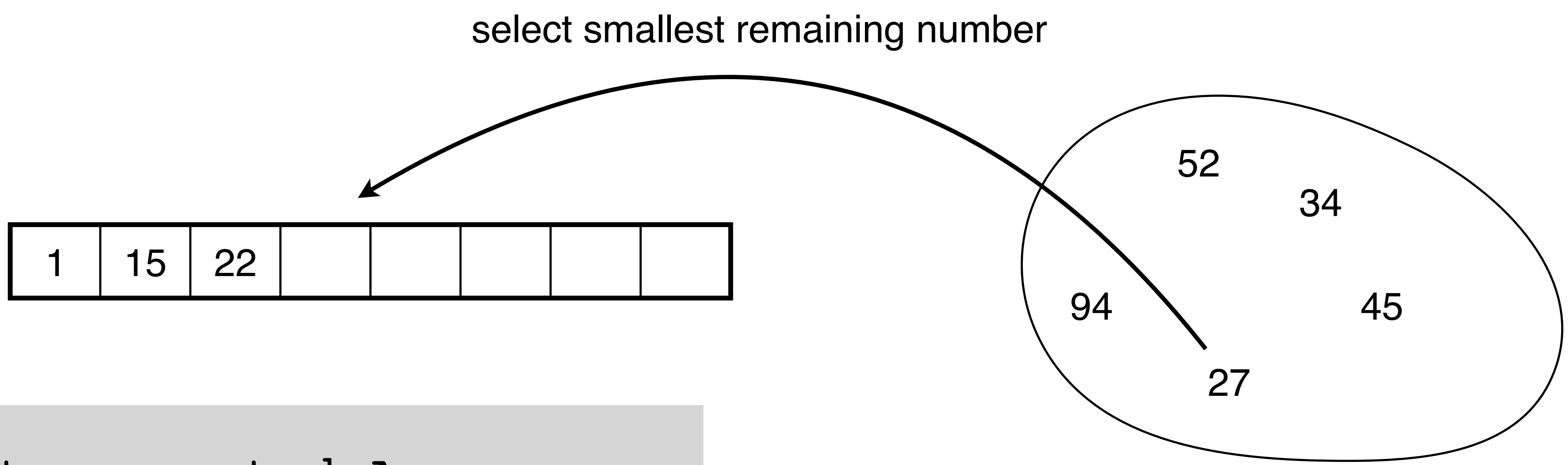
D. $T(n) = 2T(n-1) + n$

Master Method

Theorem: if $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^d)$ where $a \geq 1, b > 1, d \geq 0$, then

- if $d > \log_b a \Rightarrow T(n) = \Theta(n^d)$
- if $d < \log_b a \Rightarrow T(n) = \Theta(n^{\log_b a})$
- if $d = \log_b a \Rightarrow T(n) = \Theta(n^d \log n)$

Selection Sort



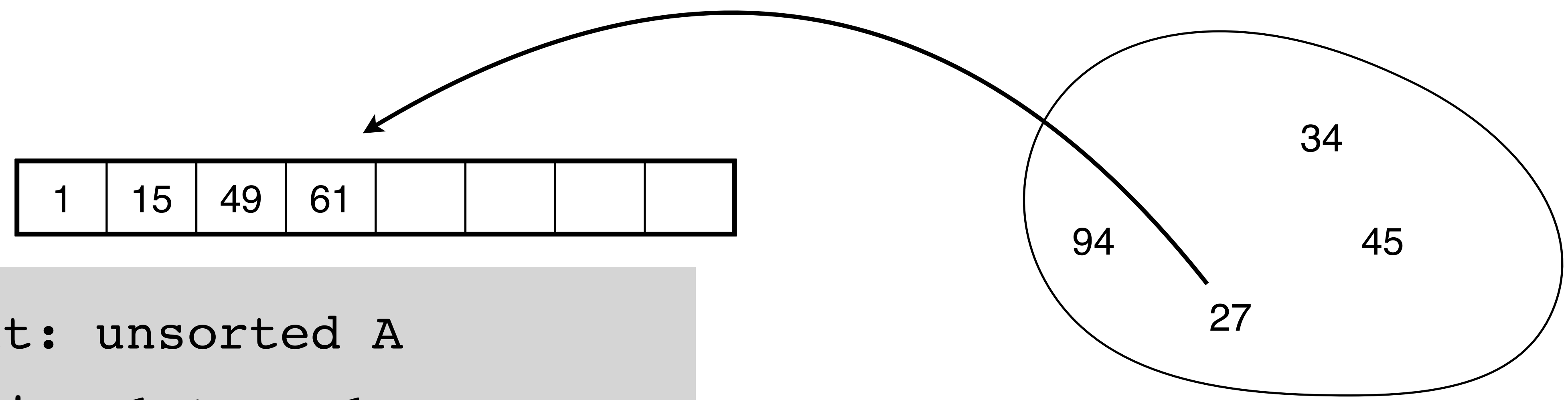

```
input: unsorted A
For i = 0 to n-1
  • j = argmin(A[i,:])
  • swap A[i] and A[j]
output: sorted A
```

runtime:

T(n) =

Insertion Sort

select any value and move to its position



input: unsorted A

For i = 1 to n-1

- j = i
- While A[j] ~~>~~ < A[j-1]:
  - swap A[j] and A[j-1]
  - j = j-1

output: sorted A

worst-case and best-case input and runtime:

worst case $\Theta(n^2)$: sorted in reverse

best : $\Theta(n)$: already sorted

Insertion Sort Correctness

Insertion Sort returns an ordered array.

proof:

Loop invariant: at the start of iteration i the subarray A[0,i-1] consists of the values originally in A[0,i-1] but in sorted order.

Initialization: true for i=0

Maintenance: in iteration i A[j] gets moved down until it reaches a position such that A[j] > A[j-1]. We know that indices prior to j-1 have lower values as A[j-1] due to the loop invariant. Thus A[j] is larger than anything before it in A. We also know that A[j] is smaller than anything positioned above it due to the swaps. Hence, once iteration i terminates A[0,i] is sorted.

```
input: unsorted A
For i = 1 to n-1
  • j = i
  • While A[j]>A[j-1]:
    - swap A[j] and A[j-1]
    - j = j-1
output: sorted A
```

Hiring problem

We want to hire a new office assistant

- n candidates, we know how they compare to each other
- goal: hire the best candidate
- cost: the number of times we fire a person

```
Hiring(c1, c2, …, cn):
current_best = c1
For i= 2…n
  • If ci better than current_best
    - fire current_best
    - current_best = ci
return current_best
```

best and worst-case input and number of people fired:

0 firings: first person
n-1 : ordered worst to best

Hiring problem - expected cost

suppose candidates come in random order

compute expectation: define corresponding random variable

$E[X]$

$X =$ (# of times we fire someone); # of times we hire someone

$X = X_1 + X_2 + \dots + X_n$

$X_i = I\{c_i \text{ is hired}\} = \begin{cases} 1 & \text{if } c_i \text{ is hired} \\ 0 & \text{not hired} \end{cases}$

← indicator

$E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$

by def: $E[X_i] = 1 \cdot pr(c_i \text{ is hired}) + 0 \cdot pr(\text{not})$

prob that i gets hired:

- when c_i arrives → have seen $i-1$ candidates
- $pr(c_i \text{ is the best so far}) = \frac{1}{i}$

Hiring problem - expected cost

If the candidates arrive in a random order, we can computed an expected cost which is less than the worst case

X = random variable indicating the number of times we hire a new assistant

$X_i = I\{\text{candidate } i \text{ is hired}\}$ (indicator, takes on value 0 or 1)

We know $X = X_1 + X_2 + \ldots + X_n$

by linearity of expectation we have $E[X] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$

How much is $E[X_i]$?

- X_i is 1 if candidate i is better than all i-1 candidates before
- if input is random, then this is true with probability $\frac{1}{i}$

$E[X] = \sum_{i=1}^{n} \frac{1}{i} = \ln n + O(1)$ (harmonic series)

hiring problem

If the input is random then we can compute the expected cost.

What if the input is not random? What can we do to get the expected cost with any input?

Make input random ourselves

$\rightarrow$ shuffle the list of n candidates

Exercise: randomly permute an array in place in time O(n).

MergeSort - Divide-and-Conquer algorithm

Algorithm:

1. Divide the unsorted list into n sublists (of length 1)

2. Repeatedly merge sublists to produce new sorted sublists until there is only one list remaining. This will be the sorted list.

runtime:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$$

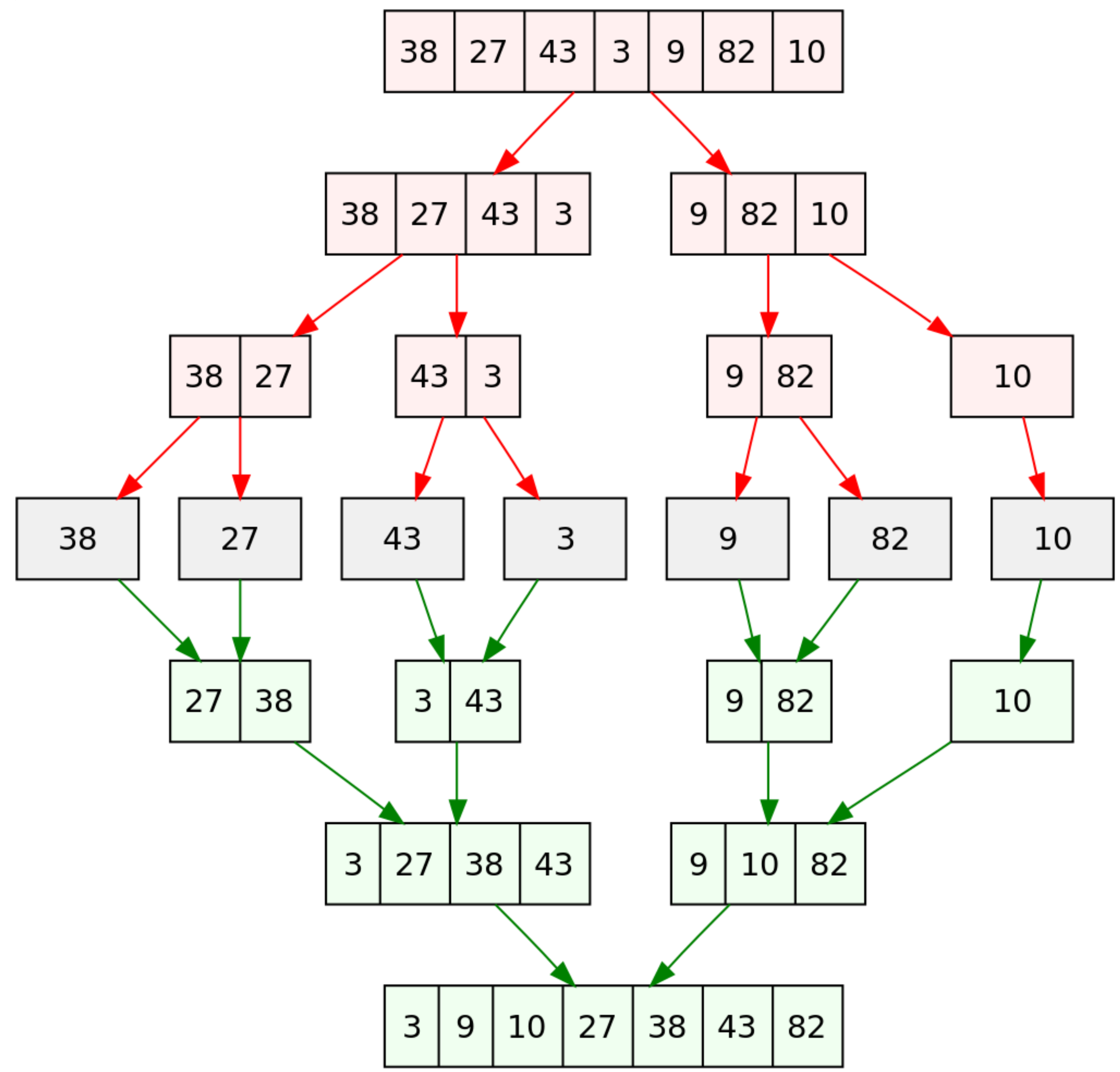
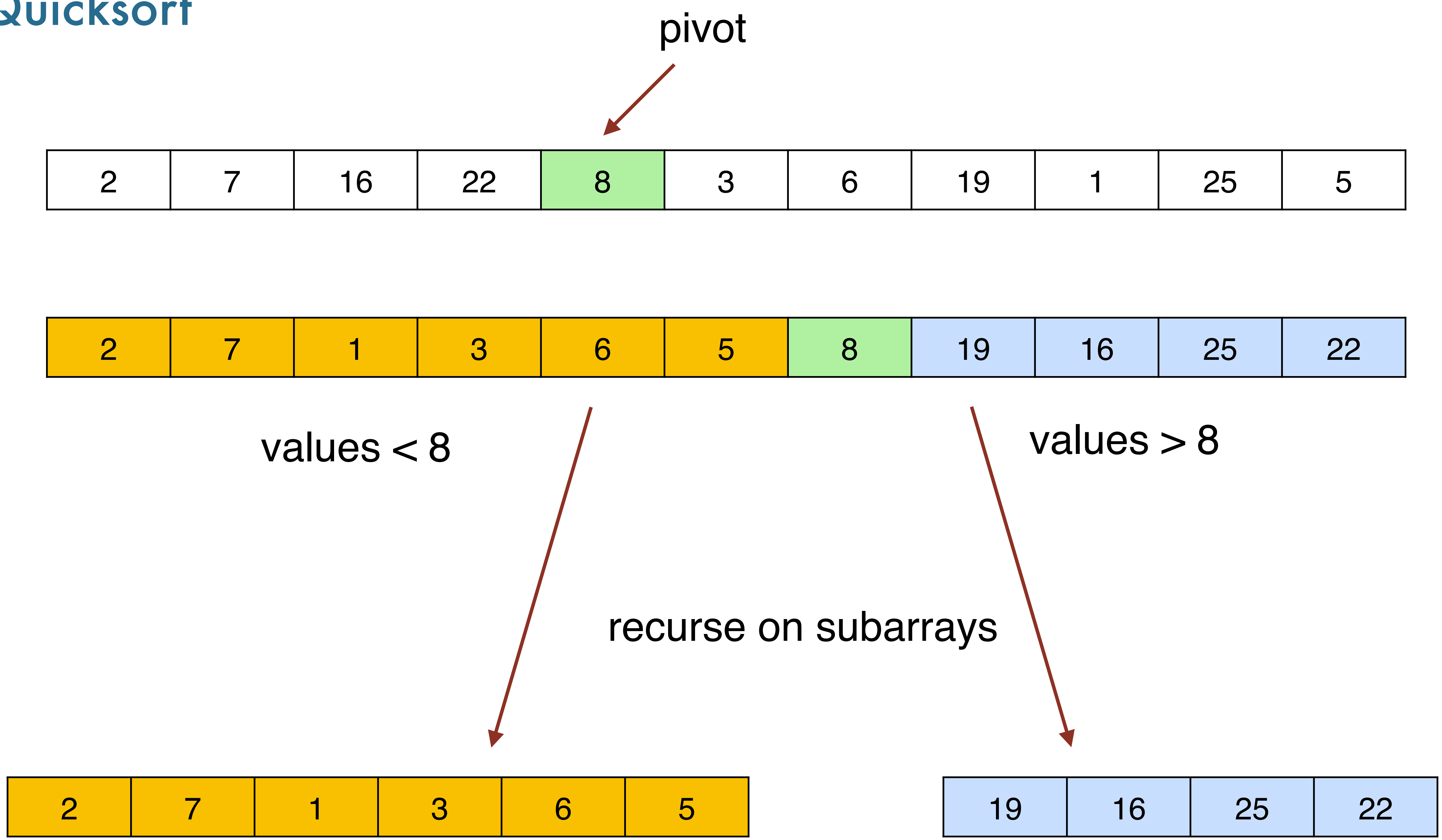


image from Wikipedia

Quicksort



Exercise. Write the proof that Quicksort indeed results in a sorted array using loop invariants.

Loop invariant: for a pivot A[p], the values in A[0,p-1] are all < A[p], values in A[p+1,n-1] are all greater than A[p]

Quicksort

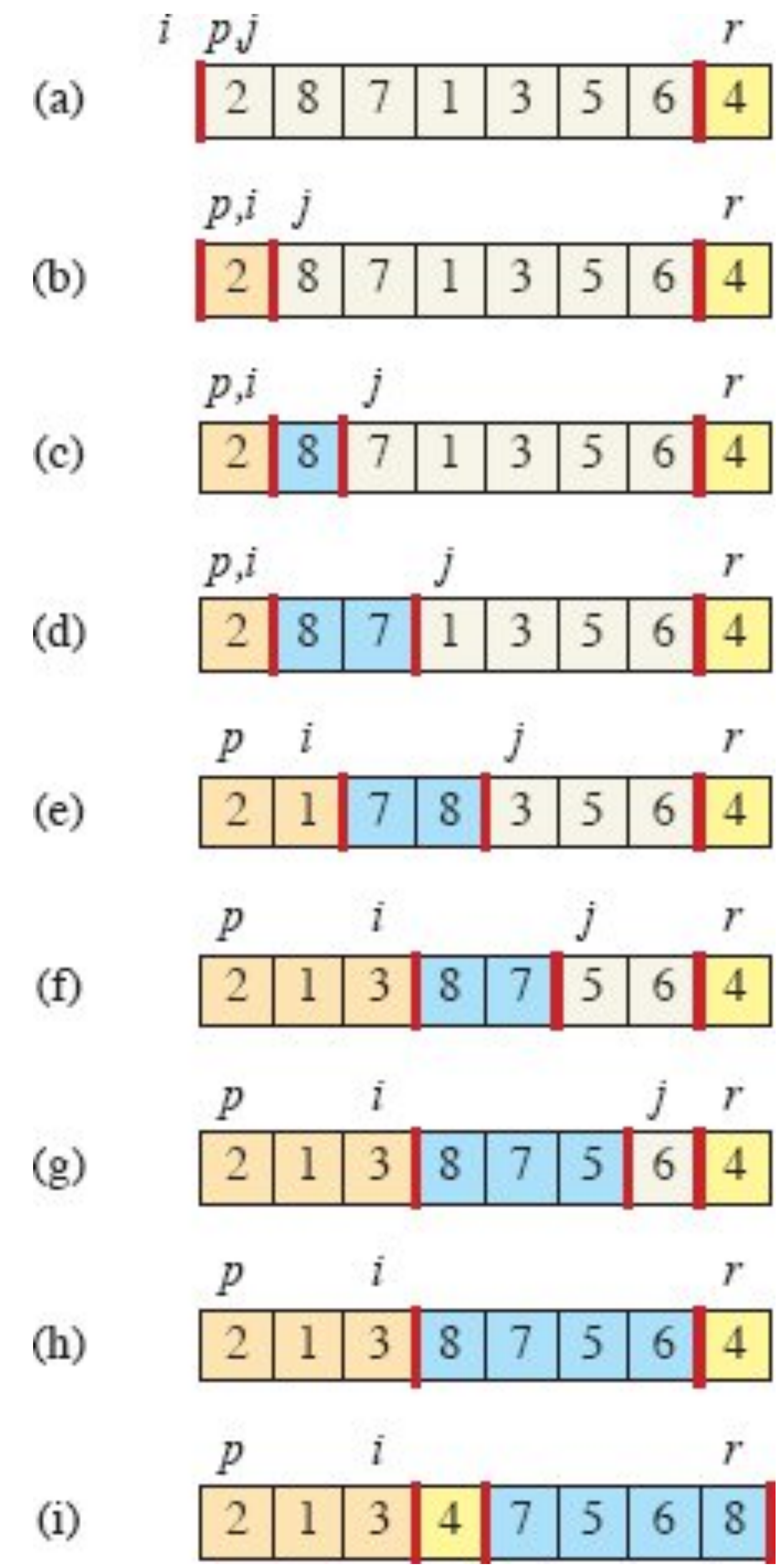
final location of the pivot

move the values of A[p,r] before and after the pivot

```
QUICKSORT(A,p,r):
  if (p < r)
    q = PARTITION(A,p,r) //pivot goes
    to index A[q]
    QUICKSORT(A,p,q-1)
    QUICKSORT(A,q+1,r)
```

subarray left to pivot

subarray to the right.



Source: CLRS ch. 7

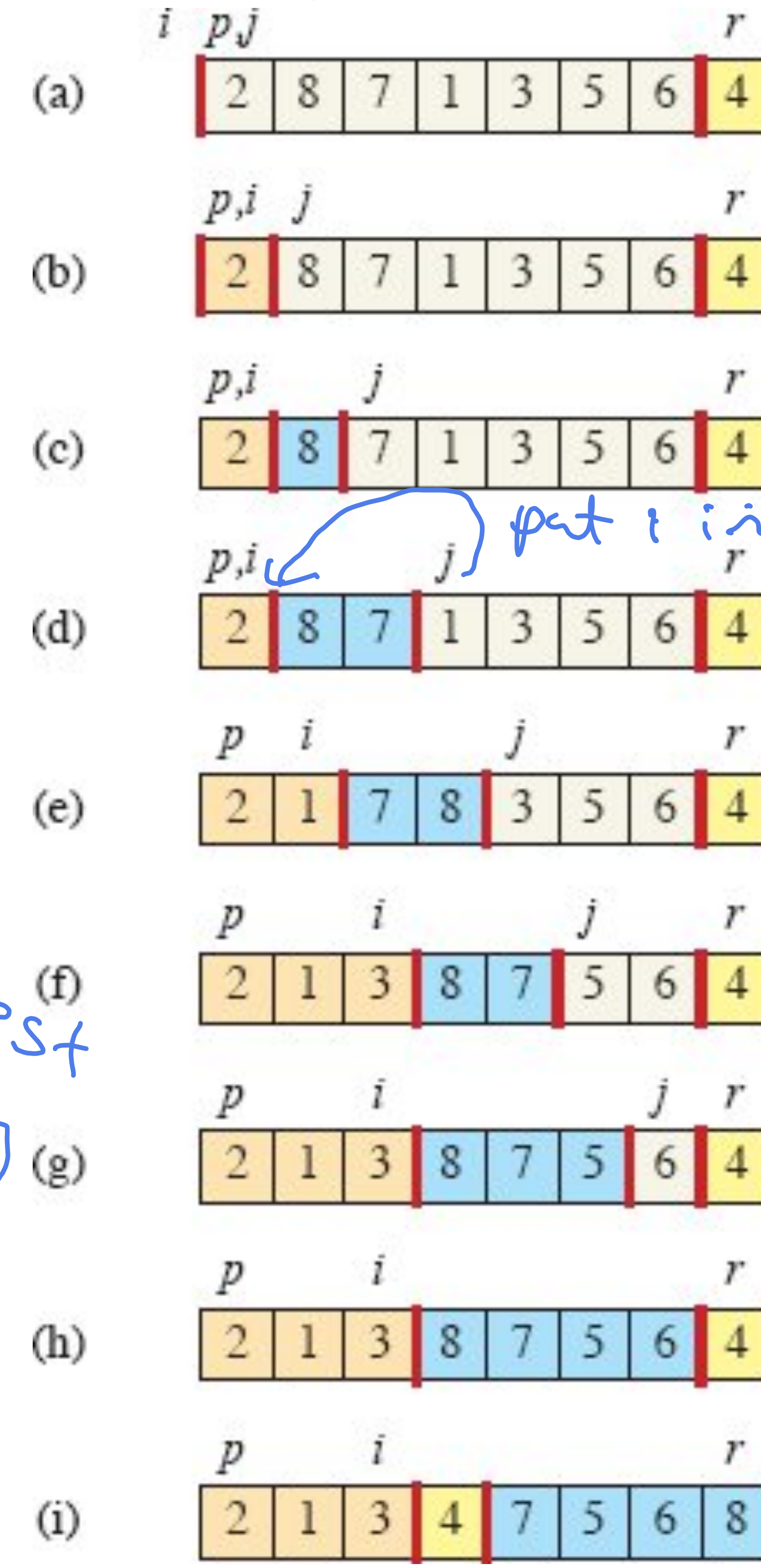
Quicksort

yellow: pivot   white numbers: not processed yet
orange: less than the pivot
blue: greater than pivot

```
QUICKSORT(A,p,r):
  if (p < r)
    q  =  PARTITION(A,p,r)  //pivot goes
    to index A[q]
    QUICKSORT(A,p,q-1)
    QUICKSORT(A,q+1,r)
```

```
PARTITION(A,p,r):
x = A[r]
i = p -1
for j = p to r-1
    if A [j ] ≤ x
        i = i+1
        exchange A[i] with A[j]
exchange A[i+1] with A[r]
return i+1
```

A[j] should go to orange part
swap left-most blue with A[j]



put i in the orange part

Source: CLRS ch. 7

Quicksort running time

Worst case running time:

if the pivot is at either end

Best case running time:

pivot is in the middle

Quicksort running time

Worst case running time:

$$T(n) = T(n-1) + T(0) + \Theta(n) = \Theta(n^2)$$

when all elements go on the same side of the pivot

Best case running time:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = O(n \log n)$$

when the two subarrays are balanced, thus the selected pivot is a median

Quicksort running time

Worst case running time:

$$T(n) = T(n-1) + T(0) + \Theta(n) = \Theta(n^2)$$

when all elements go on the same side of the pivot

Best case running time:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = O(n \log n)$$

when the two subarrays are balanced, thus the selected pivot is a median

don't need perfect balance: if the pivot can always produce at least 9-1 split:

$$T(n) = T\left(\frac{9n}{10}\right) T\left(\frac{n}{10}\right) + \Theta(n) \leq 2T\left(\frac{9n}{10}\right) + \Theta(n) = \Theta(n \log n)$$

conclusion: any split of constant proportionality produces $\Theta(n \log n)$

Randomized Quicksort

option 1: randomly permute the input array (we don't use this)

option 2: instead of using right most element as pivot, pick a pivot at random

```
RND-QUICKSORT(A,p,r):
  if (p < r)
    q  =  RND-PARTITION(A,p,r)  //pivot
    goes to index A[q]
    RND-QUICKSORT(A,p,q-1)
    RND-QUICKSORT(A,q+1,r)
```

```
RND-PARTITION(A,p,r):
  i = random(p,r)
  exchange A[r] with A[i]
  PARTITION(A,p,r)
```

Randomized Quicksort analysis → expected run time

dominant part of the algorithm is PARTITION

- the pivot is removed from further consideration → called at most n times
- work in PARTITION: constant + number of comparisons
- X = total number of comparisons in PARTITION through all of QUICKSORT
- The total work done is O(n+X)

iterate over array

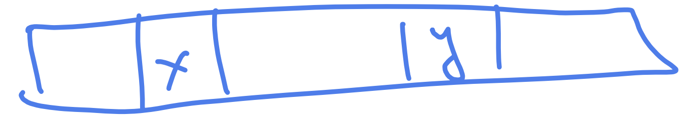
$\rightsquigarrow \Theta(n)$

each step : const

3x comparison

- 1 comparison
- advance pointer
- swap

TopHat - pivots and comparisons

A = 

Select all true statements for what happens in Quicksort to any pair of elements x and y in A:

During the whole algorithm: items get compared to pivot.

A. in worse case x and y get compared $\Theta(n)$ times

→ each element is a pivot at most once

→ only time they get compared

B. If x and y have never been compared then we can't know which is larger

see case 1

✓ C. x and y get compared at most once

if one is the pivot & the other is in that array

D. if x is selected as a pivot then y is always compared to x

see case 1

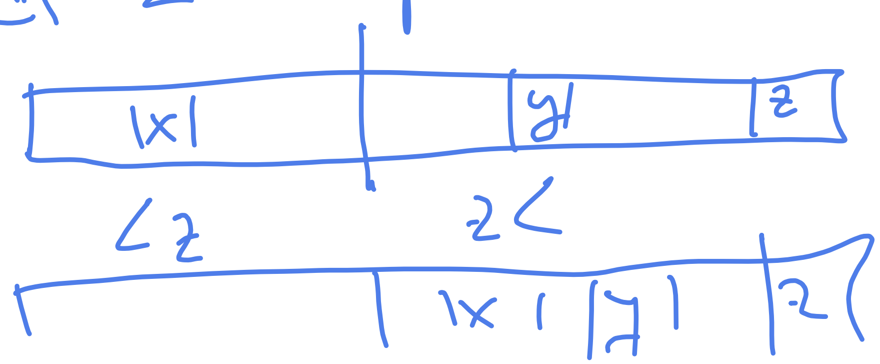
✓ E. if x and y have not been compared so far and we chose a pivot $x < z < y$ then they will never be compared in the future

case 1: x and y end up on different sides

case 2: same side of z

select z as pivot:

case 1



Randomized Quicksort analysis

Total work is O(n+X) where X is the number of comparisons across quicksort

take away: x and y are only ever compared iff they are in the same subarray and one gets selected

Randomized Quicksort analysis cont'd

Total work is O(n+X) where X is the number of comparisons across quicksort

goal: compute expected X

Let $z_1, z_2, \ldots z_n$ the elements in A in the order, such that z_i is the ith smallest number.

$Z_{ij} = \{z_i, z_{i+1} \ldots z_j\}$ is the set of elements between z_i and z_j inclusively

observation: any pair of elements is compared at most once

- an element is only compared to the pivot, which is never used in later iterations

$X_{ij} = I\{z_i \text{ is compared to } z_j\}$ (indicator whether i and j are ever compared) ← 1 if z_i and z_j are compared

$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$ total number of comparisons = sum of pairwise comparisons

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(z_i \text{ is compared to } z_j)$$

linearity of expectation

Randomized Quicksort analysis (extra space)

Total work is O(n+X) where X is the number of comparisons across quicksort

compute $P(z_i \text{ is compared to } z_j)$:

Randomized Quicksort analysis cont'd

compute $P(z_i \text{ is compared to } z_j)$:

- numbers in separate partitions are *not* compared
- if we ever chose a pivot $z_i < x < z_j$ they will never be compared
- if either z_i or z_j is chosen as pivot before any other element in Z_{ij} then they will be compared
- This probability is $\dfrac{2}{j - i + 1}$

- $E[X] = = \displaystyle\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(z_i \text{ is compared to } z_j) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=}^{n-i} \frac{2}{k+1} <$
- $< \displaystyle\sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} = \sum_{i=1}^{n} \Theta(\log n) = \Theta(n \log n)$

Improve odds for Quicksort

We just concluded that randomized Quicksort takes $\Theta(n \log n)$ in expectation

Ideas to improve odds for close-to-average runtime?

Why Quicksort?

We just concluded that randomized Quicksort takes $\Theta(n \log n)$ in expectation - which is not better than some of our deterministic algorithms.

Why use it?

- constants better than mergeSort $\sim 30\%$ faster
- can be in place
- we can stop early $\rightarrow$ have "buckets" that are sorted relative to each other.

k-median and order statistics

- kth order statistics/ k-median is the *k*th smallest of n elements
- the minimum elements is the *1*st order statistics
- the maximum is the *n*th
- the median is the middle element $\lceil n/2 \rceil$th

Find smallest element:

Find largest element:

Find smallest and largest at the same time with 3n/2 comparisons:

k-median and order statistics

- kth order statistics/ k-median is the *k*th smallest of n elements
- the minimum elements is the *1*st order statistics
- the maximum is the *n*th
- the median is the middle element $\lceil n/2 \rceil$th

Find the kth element:

- what is the best running time you can get?

randomized k-median

Can we use a Quicksort-style algorithm?

It turns out that this is $\Theta(n)$ in expectation

deterministic linear time k-median

goal: select a good pivot.

trick:

- divide array into subarrays of 5: n/5 subarrays.
- compute median of each part
- compute median of medians recursively
- use the resulting value as pivot

random permutation in place

Given a length-n array A, permute its elements uniformly.

Quicksort proof of correctness