Sorting

Recurrence relationships

3. $F_i = F_{i-1} + F_{i-2}$, with $f_0=0$ and $f_1=1$

- $F_0=0$, $F_1=1$, $F_2=1$, $F_3=2$, $F_4=3$
- $F_0=5$, $F_1=8$, $F_2=13$, $F_3=21$, $F_4=34$

- Fi =

$[(1+sqrt(5))^i–(1-sqrt(5))^i]/(2^i sqrt(5))$

Outline

Sorting!

- -What's a sorting algorithm?
- -Insertion sort
- -Merge sort
- -Divide & conquer (Master's thm)
- -Quicksort

Sorting problem

Input: sequence of numbers =
$\{a_1, a_2, \ldots a_n\}$

Output: different order =
$\{a_1', a_2', \ldots a_n'\}$, where
$a_1' \leq a_2' \leq \ldots \leq a_n'$

Insertion sort

General idea:

-Examine one element at a time

-Insert into correct place in an already sorted sequence

-Repeat...

Insertion sort

Where to put a 10 of spades?
A 6 of hearts?

Insertion sort

Where to put a 10 of spades?

A 6 of hearts? Between 5 and 7

Insertion sort

```
Input: A[1,2, ... n]
for j = 2 to n
    i=j-1
    key = A[j] // why do we need this?
    while i > 0 AND A[i] > key
        A[i+1] = A[i]
        i = i – 1
    A[i+1] = key
```

Insertion sort

Sort: {4, 5, 3, 8, 1, 6, 2}

Insertion sort

Sort: {4, 5, 3, 8, 1, 6, 2}

{4} - done

{4, 5} – done

{4, 5, 3}, {4,3,5}, {3,4,5} – done

{3, 4, 5, 8} – done

{3, 4, 5, 8, 1}, {3, 4, 5, 1, 8},

{3, 4, 1, 5, 8}, {3, 1, 4, 5, 8},

{1, 3, 4, 5, 8} - done

Insertion sort

Sort: {4, 5, 3, 8, 1, 6, 2}

{1, 3, 4, 5, 8} – done

{1, 3, 4, 5, 8, 6}, {1, 3, 4, 5, 6, 8}

-done

{1, 3, 4, 5, 6, 8, 2},{1, 3, 4, 5, 6, 2, 8}

{1, 3, 4, 5, 2, 6, 8},{1, 3, 4, 2, 5, 6, 8}

{1, 3, 2, 4, 5, 6, 8},{1, 2, 3, 4, 5, 6, 8}

-done and done

Insertion sort

Worst case runtime?

Average case?

Insertion sort

Worst case runtime?
Outer loop runs n times and inner loop runs j-1 times
$1+2+3+ \ldots + n-1 = ?$

Average case?

Insertion sort

Worst case runtime?

Outer loop runs n times and inner loop runs j-1 times

$1+2+3+ \ldots + n-1 = n(n-1)/2 = O(n^2)$

Average case?

inner loop $(j-1)/2$ times $= O(n^2)$

Insertion sort

Correctness:

Base: Initial list is 1 element, sorted

Step: Inner loop places everything bigger than key after it and everything smaller before. Before & after will be sorted as it started sorted

Termination: Terminates after n A[n] placed, so whole list sorted

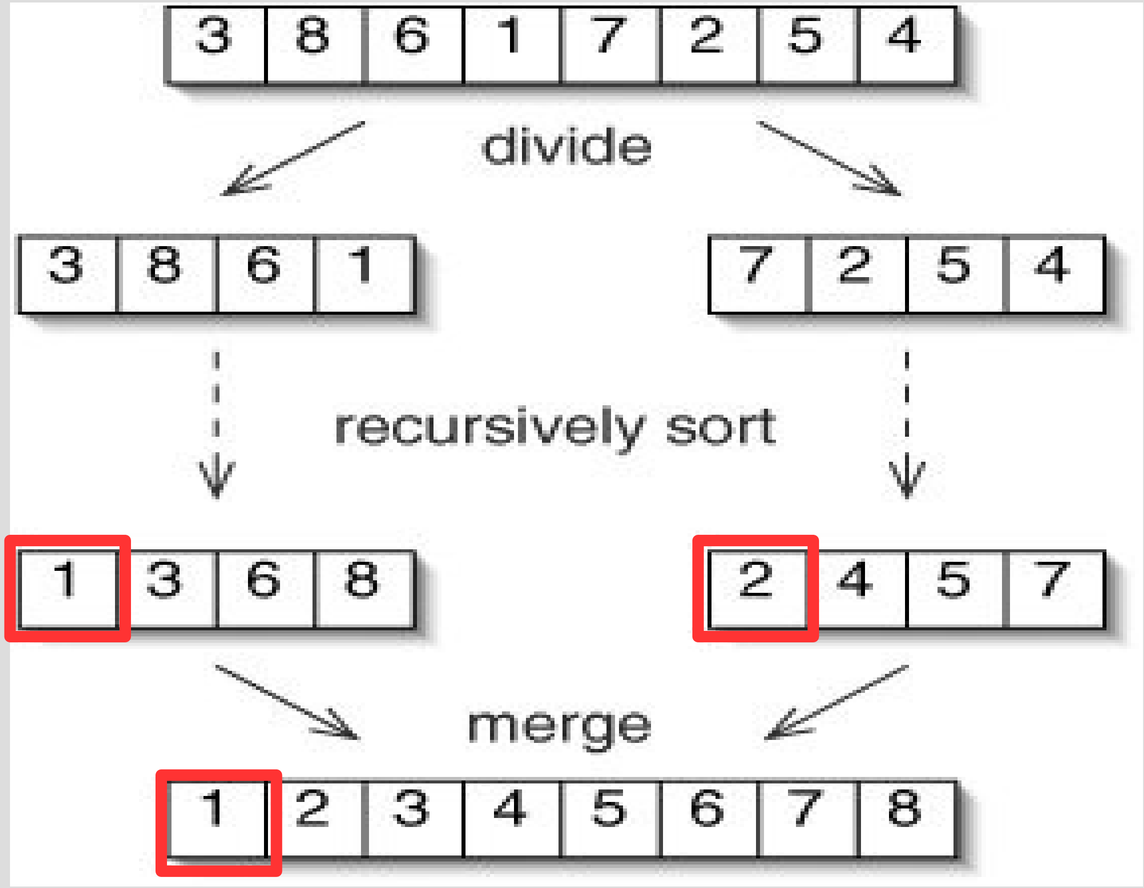
Merge sort

1. Split pile in half

2. Sort each half (possibly recursively with merge sort)

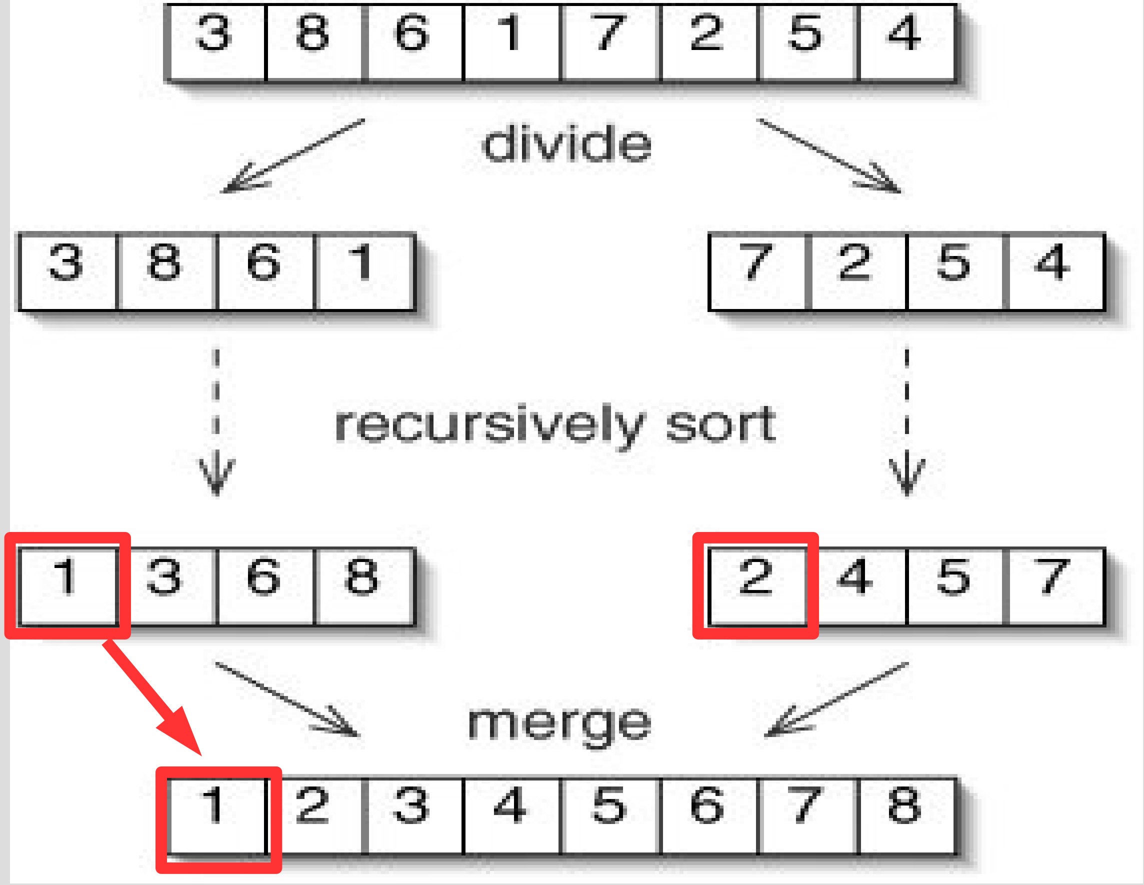
3. Recombine lists

Merge sort

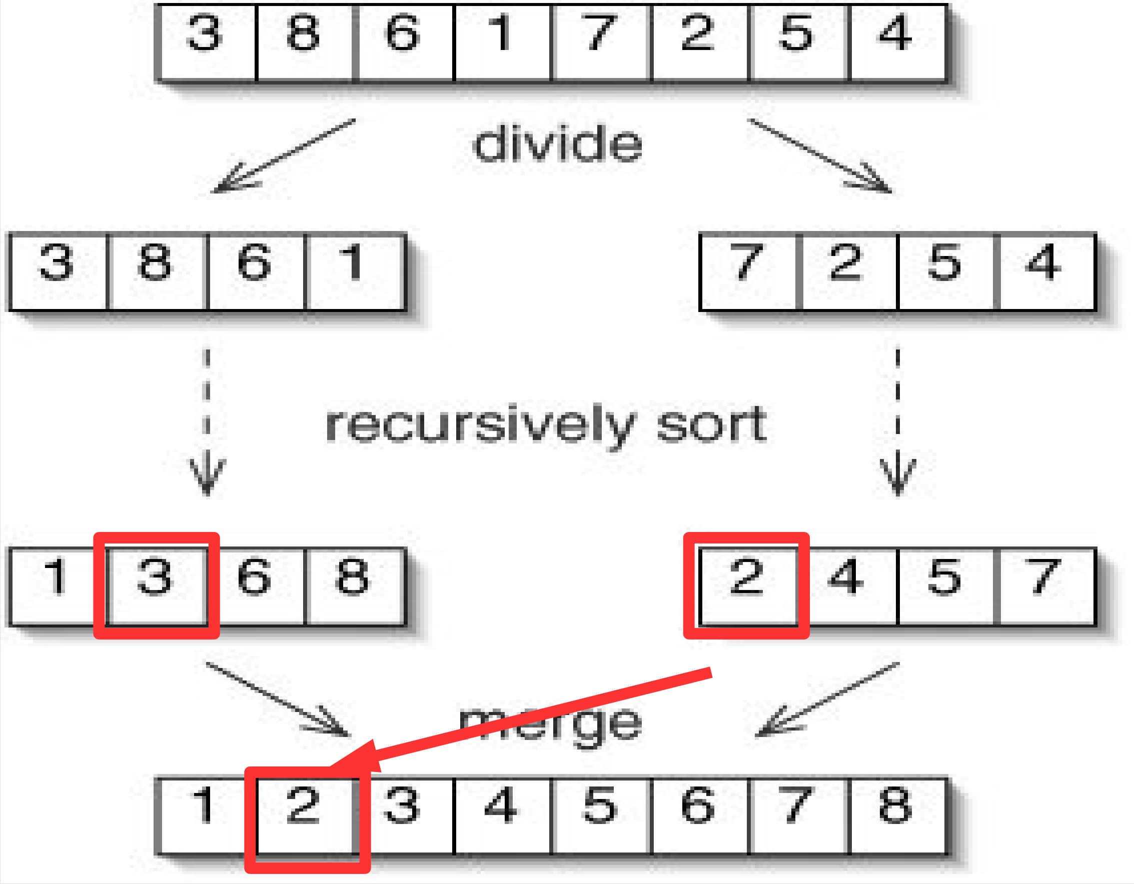


3 8 6 1 7 2 5 4
divide
3 8 6 1
7 2 5 4
recursively sort
1 3 6 8
2 4 5 7
merge
1 2 3 4 5 6 7 8

Merge sort

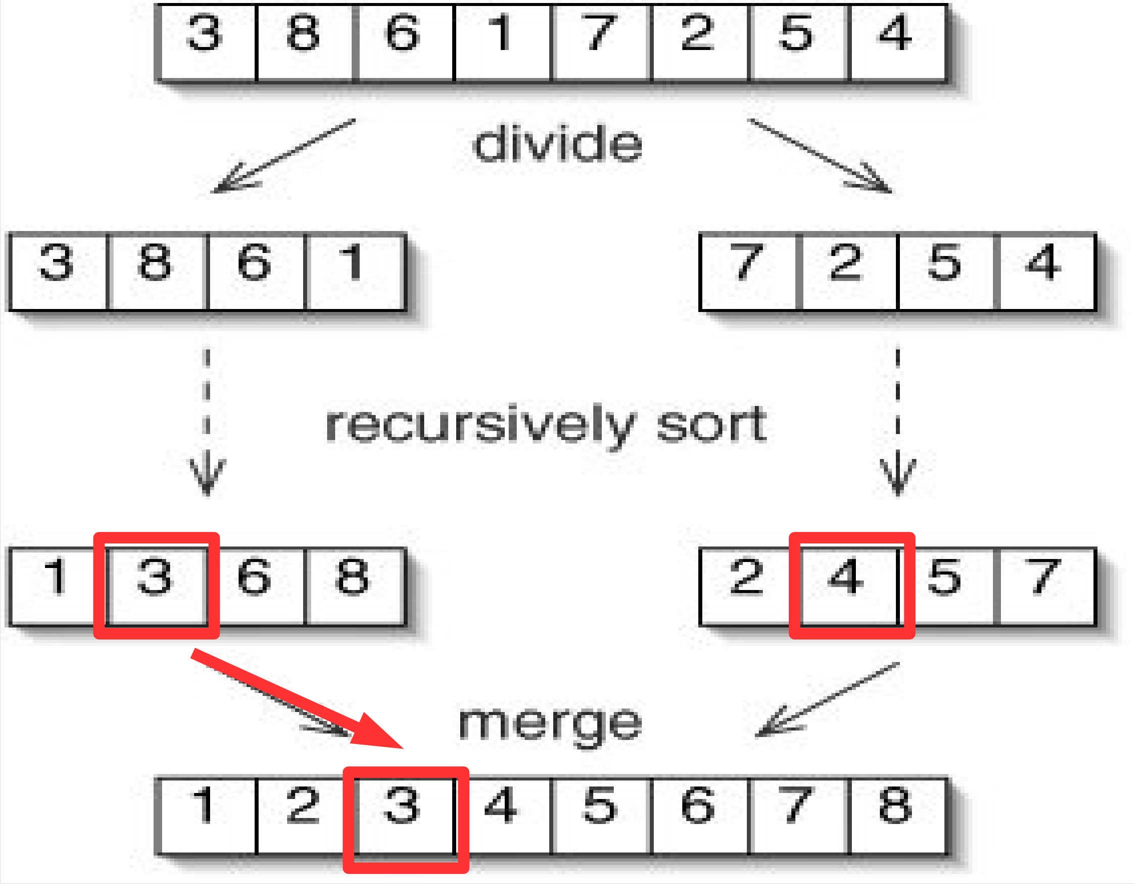

3 8 6 1 7 2 5 4
divide
3 8 6 1
7 2 5 4
recursively sort
1 3 6 8
2 4 5 7
merge
1 2 3 4 5 6 7 8

Merge sort



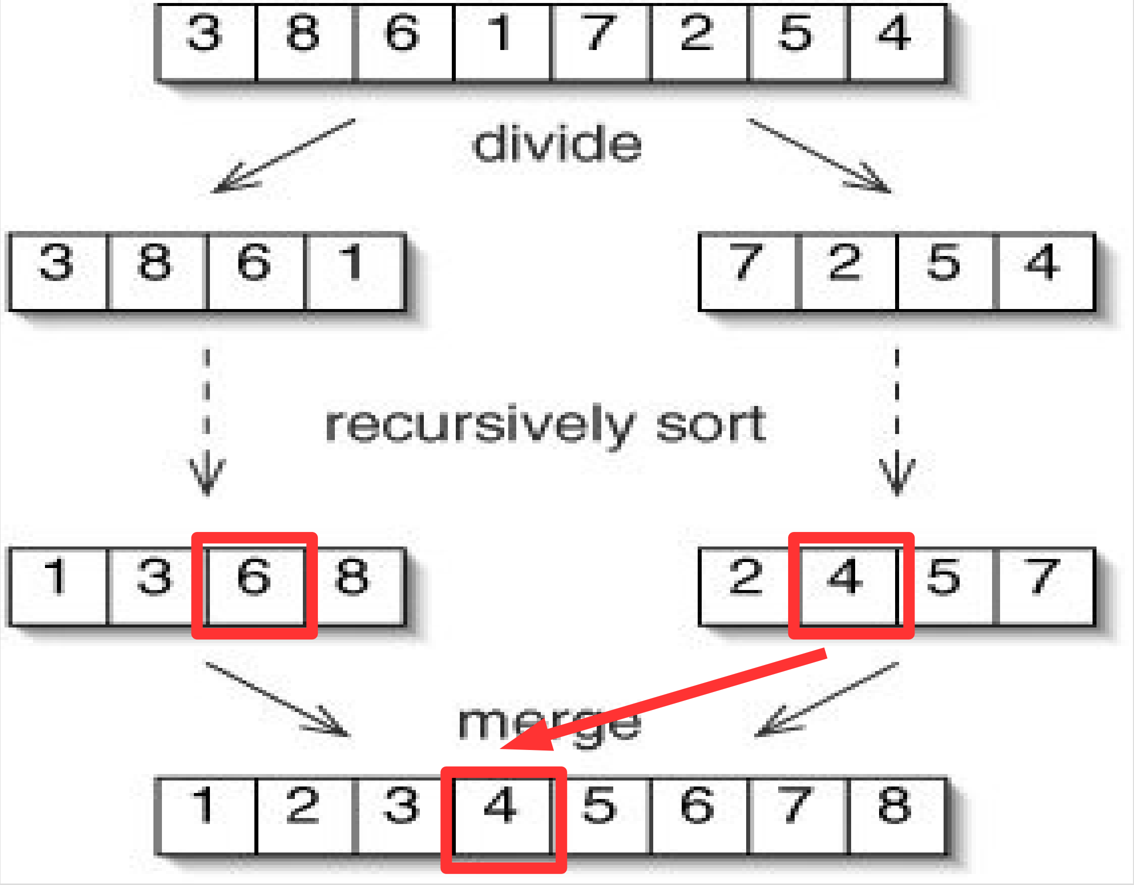
3 8 6 1 7 2 5 4
divide
3 8 6 1
7 2 5 4
recursively sort
1 3 6 8
2 4 5 7
merge
1 2 3 4 5 6 7 8

Merge sort

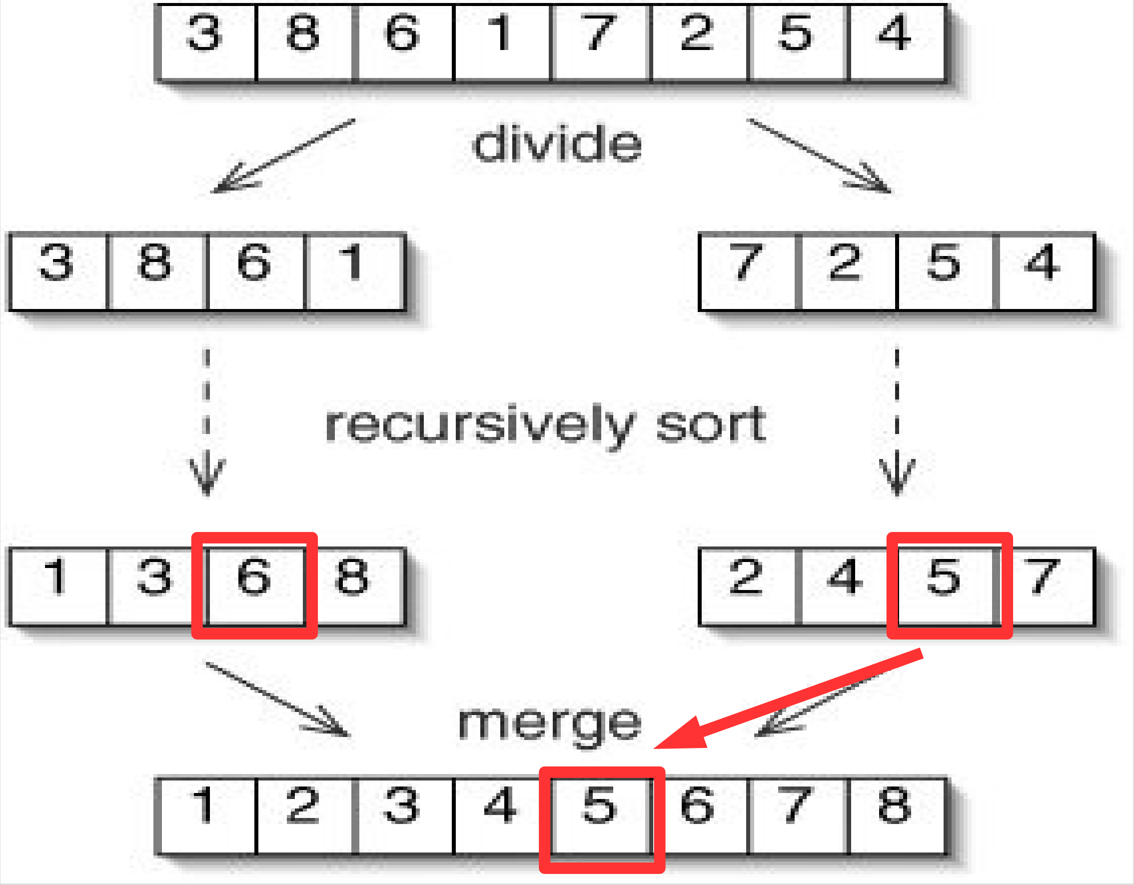


3 8 6 1 7 2 5 4
divide
3 8 6 1
7 2 5 4
recursively sort
1 3 6 8
2 4 5 7
merge
1 2 3 4 5 6 7 8

Merge sort

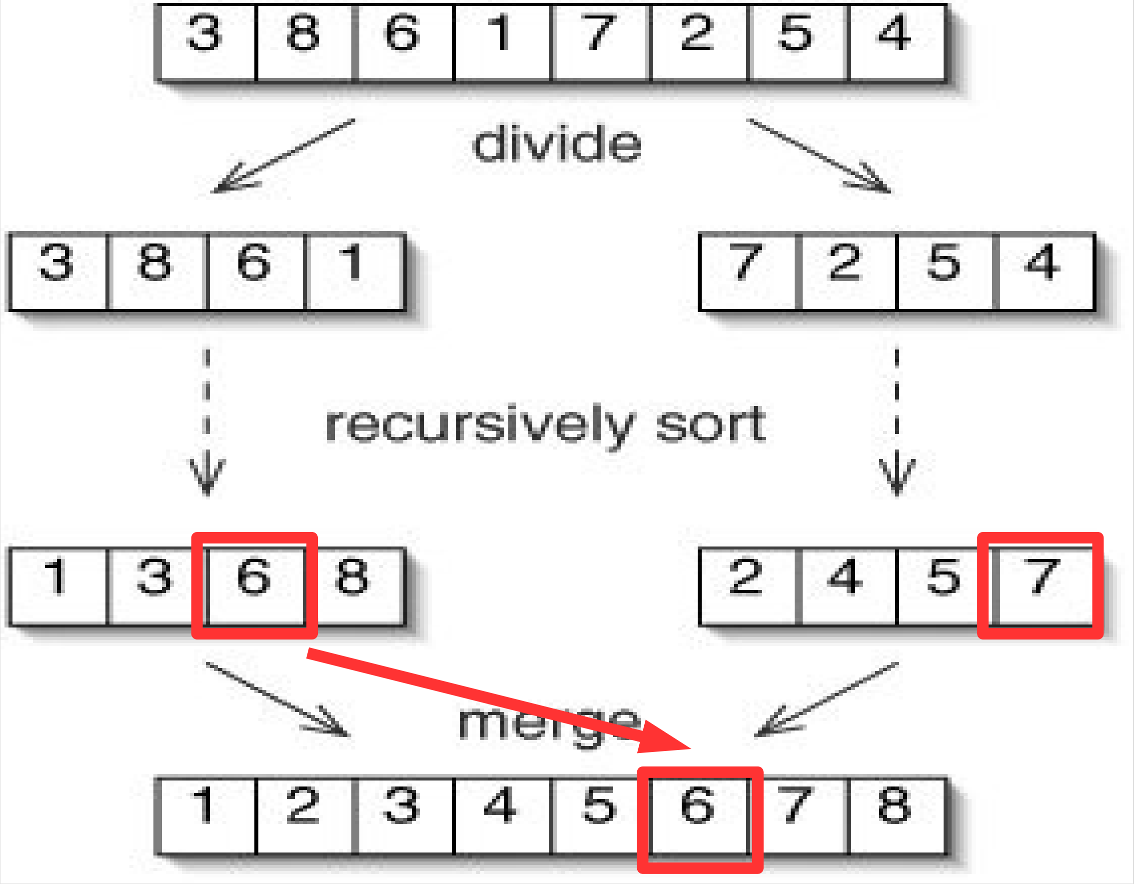

3 8 6 1 7 2 5 4
divide
3 8 6 1
7 2 5 4
recursively sort
1 3 6 8
2 4 5 7
merge
1 2 3 4 5 6 7 8

Merge sort



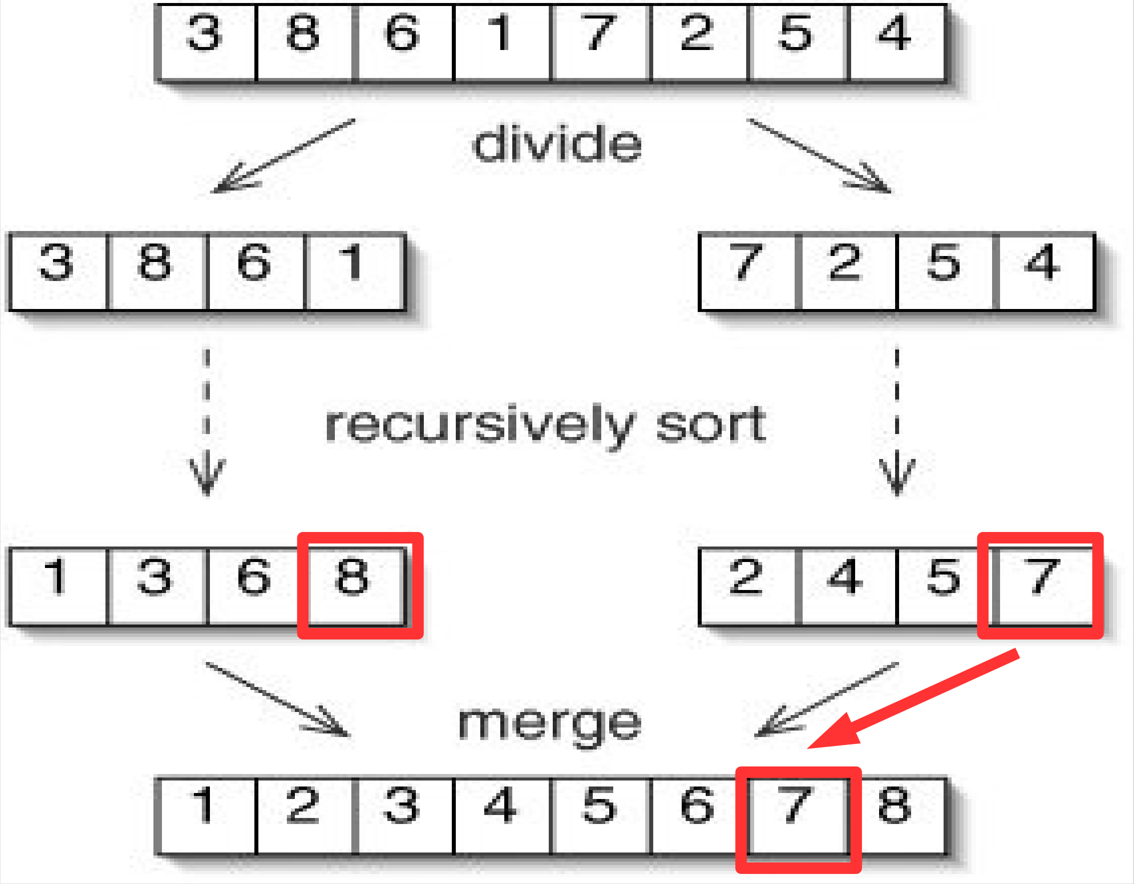
3 8 6 1 7 2 5 4
divide
3 8 6 1
7 2 5 4
recursively sort
1 3 6 8
2 4 5 7
merge
1 2 3 4 5 6 7 8

Merge sort

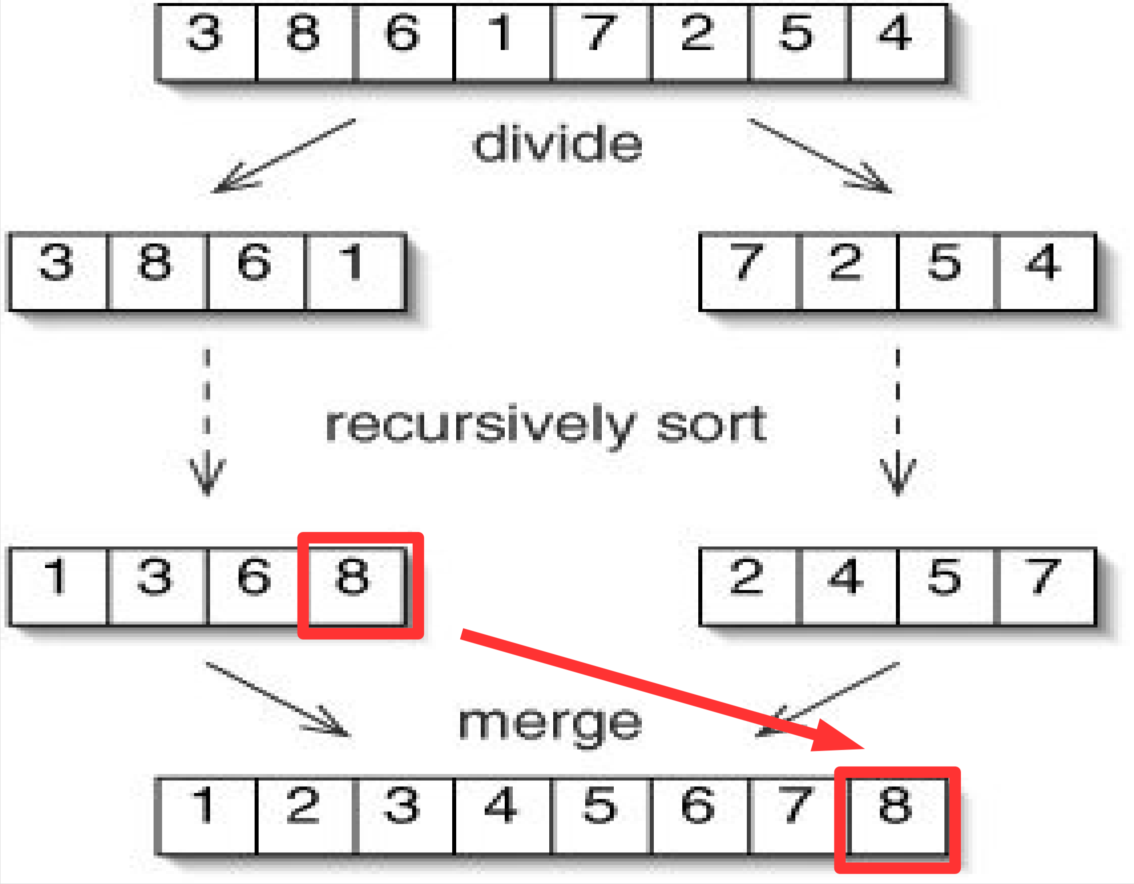


3 8 6 1 7 2 5 4
divide
3 8 6 1
7 2 5 4
recursively sort
1 3 6 8
2 4 5 7
merge
1 2 3 4 5 6 7 8

Merge sort


3 8 6 1 7 2 5 4
divide
3 8 6 1
7 2 5 4
recursively sort
1 3 6 8
2 4 5 7
merge
1 2 3 4 5 6 7 8

Merge sort


3 8 6 1 7 2 5 4
divide
3 8 6 1
7 2 5 4
recursively sort
1 3 6 8
2 4 5 7
merge
1 2 3 4 5 6 7 8

Merge sort

```
Merge(L[1, ..., n_l], R[1, ..., n_r]
i=1, j=1, k=1
while i < n_l OR j < n_r
   if L[i] < R[j]
      A[k] = L[i], i=i+1
   else
      A[k] = R[j], j=j+1
   k = k+1
```

Merge sort

Sort: {4, 5, 3, 8, 1, 6, 2}

Merge sort

Sort: {4, 5, 3, 8, 1, 6, 2} - Split

{4, 5, 3}{8, 1, 6, 2} - Split

{4, 5}{3}{8,1}{6,2} – Split

{4}{5}{3}{8}{1}{6}{2} – Merge

{4, 5}{3} {1, 8} {2, 6} – Merge

{3, 4, 5} {1, 2, 6, 8} – Merge

{1, 2, 3, 4, 5, 6, 8}

Merge sort

Corectness:

Base: A[] empty (sorted), at L&R[1]

Step: In the while loop, the smallest element in L[] or R[] will be added as the largest element in A[]

Termination: while loop end after all elements in L[] and R[] have been added

Merge sort

Run time:

$T(n) =$

Merge sort

Run time: (recurrence relation)

$T(n) = \{O(1)$ if $n=1$, otherwise...

Divide + $2T(n/2)$ + Merge}

$T(n) = \{O(1)$ if $n=1$, otherwise...

$O(1) + 2T(n/2) + O(n)\}$

$T(n) = O(n \lg n)$

Divide & conquer

Master's theorem: (proof 4.6)

For $a \geq 1$, $b \geq 1$, $T(n) = a\,T(n/b) + f(n)$

If f(n) is... (3 cases)

$O(n^c)$ for $c < \log_b a$, $T(n)$ is $\Theta(n^{\log_b a})$

$\Theta(n^{\log_b a})$, then $T(n)$ is $\Theta(n^{\log_b a} \lg n)$

$\Omega(n^c)$ for $c > \log_b a$, $T(n)$ is $\Theta(f(n))$

Master's theorem: TL;DR

If you have something of the form:

$T(n) = a\, T(n/b) + f(n)$

acts like $n^{\log_b a}$

Case 1: $f(n)$ grows faster, then
overall growth just $f(n)$

Case 2: $n^{\log_b a}$ grows faster, then
overall growth just $n^{\log_b a}$

Case 3: Both grow same, tack on lg n:
$n^{\log_b a} \lg(n)$

Divide & conquer

Which works better for multi-cores: insertion sort or merge sort?
Why?

Divide & conquer

Which works better for multi-cores: insertion sort or merge sort?
Why?

Merge sort! After the problem is split, each core and individually sort a sub-list and only merging needs to be done synchronized

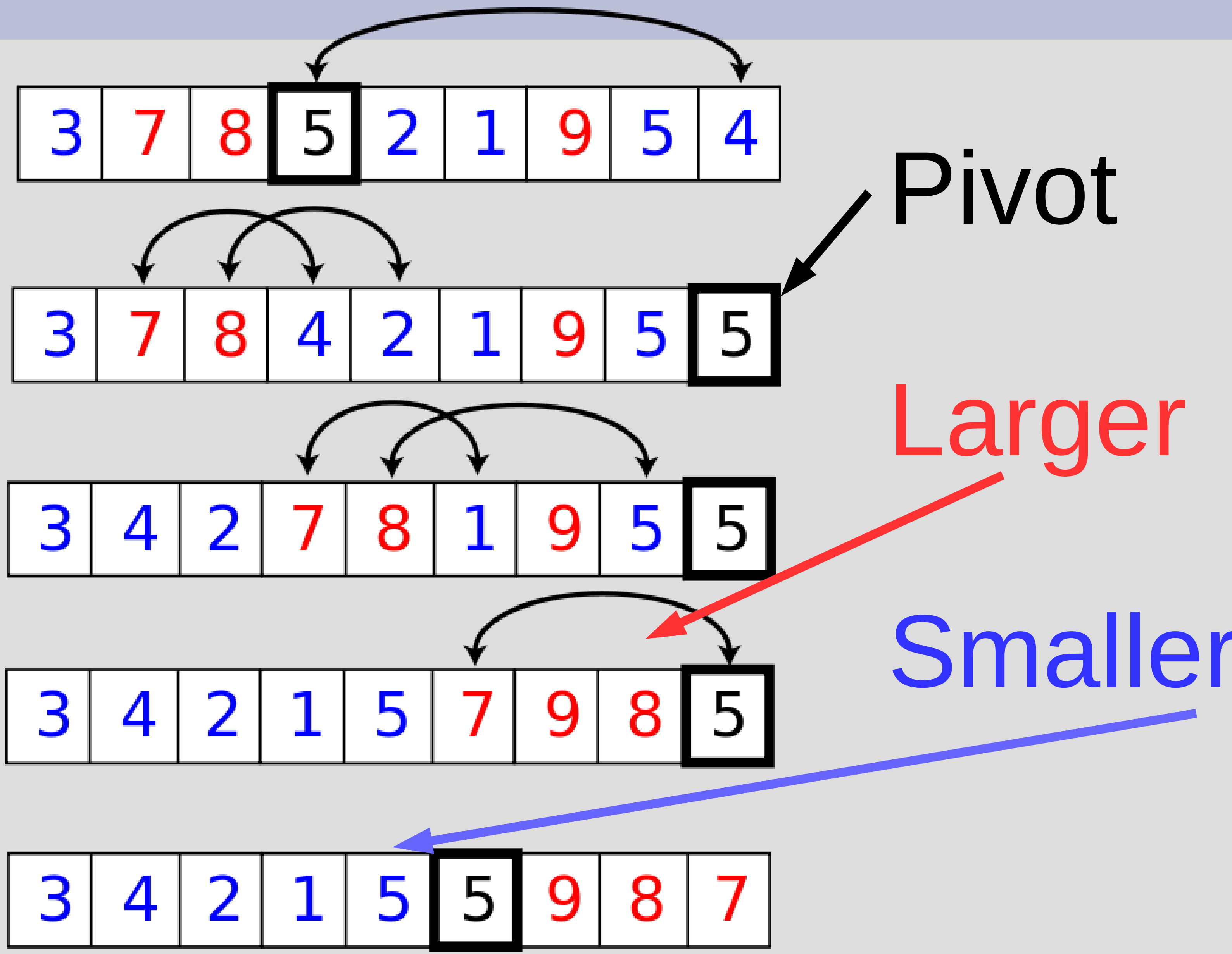
Quicksort

1. Pick a pivot (any element!)

2. Sort the list into 3 parts:
   - Elements smaller than pivot
   - Pivot by itself
   - Elements larger than pivot

3. Recursively sort smaller & larger

Quicksort



3 7 8 5 2 1 9 5 4
Pivot
3 7 8 4 2 1 9 5 5
Larger
3 4 2 7 8 1 9 5 5
Smaller
3 4 2 1 5 7 9 8 5
3 4 2 1 5 5 9 8 7

Quicksort

```
Partition(A, pivot)
x = A[pivot]
i = A.start – 1
for j = A.start to A.end -1
   if A[j] ≤ x
      i = i + 1
      swap A[i] and A[j]
swap A[i+1] with A[r]
```

Quicksort

Sort: {4, 5, 3, 8, 1, 6, 2}

Quicksort

Sort: {4, 5, 3, 8, 1, 6, 2} – Pivot = 2

{4, 5, 3, 8, 1, 6, 2} – sort 4

{4, 5, 3, 8, 1, 6, 2} – sort 5

{4, 5, 3, 8, 1, 6, 2} – sort 3

{4, 5, 3, 8, 1, 6, 2} – sort 8

{4, 5, 3, 8, 1, 6, 2} – sort 1, swap 4

{1, 5, 3, 8, 4, 6, 2} – sort 6

{1, 5, 3, 8, 4, 6, 2},{1, 2, 5, 3, 8, 4, 6}

Quicksort

For quicksort, you can pick any pivot you want

The algorithm is just easier to write if you pick the last element (or first)

Quicksort

Sort: {4, 5, 3, 8, 1, 6, 2} - Pivot = 3

{4, 5, 2, 8, 1, 6, 3} – swap 2 and 3

{4, 5, 2, 8, 1, 6, 3}

{4, 5, 2, 8, 1, 6, 3}

{2, 5, 4, 8, 1, 6, 3} – swap 2 & 4

{2, 5, 4, 8, 1, 6, 3} (first red ^)

{2, 1, 4, 8, 5, 6, 3} – swap 1 and 5

{2, 1, 4, 8, 5, 6, 3}{2, 1, 3, 8, 5, 6, 4}

Quicksort

Correctness:

Base: Initially no elements are in the "smaller" or "larger" category

Step (loop): If A[j] < pivot it will be added to "smaller" and "smaller" will claim next spot, otherwise it it stays put and claims a "larger" spot

Termination: Loop on all elements...

Quicksort

Runtime:

Worst case?

Average?

Quicksort

Runtime:

Worst case?

Always pick lowest/highest element, so $O(n^2)$

Average?

Quicksort

Runtime:
Worst case?
Always pick lowest/highest element, so $O(n^2)$

Average?
Sort about half, so same as merge sort on average

Quicksort

Runtime:
Worst case?
Always pick lowest/highest element, so $O(n^2)$

Average?
Sort about half, so same as merge sort on average

Quicksort

Can bound number of checks against pivot:

Let $X_{i,j}$ = event A[i] checked to A[j]

$\text{sum}_{i,j} X_{i,j}$ = total number of checks

$E[\text{sum}_{i,j} X_{i,j}] = \text{sum}_{i,j} E[X_{i,j}]$

$= \text{sum}_{i,j}$ Pr(A[i] check A[j])

$= \text{sum}_{i,j}$ Pr(A[i] or A[j] a pivot)

Quicksort

$= \text{sum}_{i,j}$ Pr(A[i] or A[j] a pivot)

$= \text{sum}_{i,j}$ (2 / j-i+1) // j-i+1 possibilties

$< \text{sum}_i$ O(lg n)

= O(n lg n)

Quicksort

Which is better for multi core, quicksort or merge sort?

If the average run times are the same, why might you choose quicksort?

Quicksort

Which is better for multi core, quicksort or merge sort?
Neither, quicksort front ends the processing, merge back ends

If the average run times are the same, why might you choose quicksort?

Quicksort

Which is better for multi core, quicksort or merge sort?
Neither, quicksort front ends the processing, merge back ends

If the average run times are the same, why might you choose quicksort?
Uses less space.

Sorting!

So far we have been looking at comparative sorts (where we have to compare the numbers)

The minimum running time for this type of algorithm is $\Theta(n \lg n)$