CS151 Intro to Data Structures

QuickSort

Announcements

HW06 due next Wednesday 11/29

- Lab08 due next Wednesday too

No lab this week

- Office hours in my lab – Park 200D

HW07 due 12/05

Quick Sort

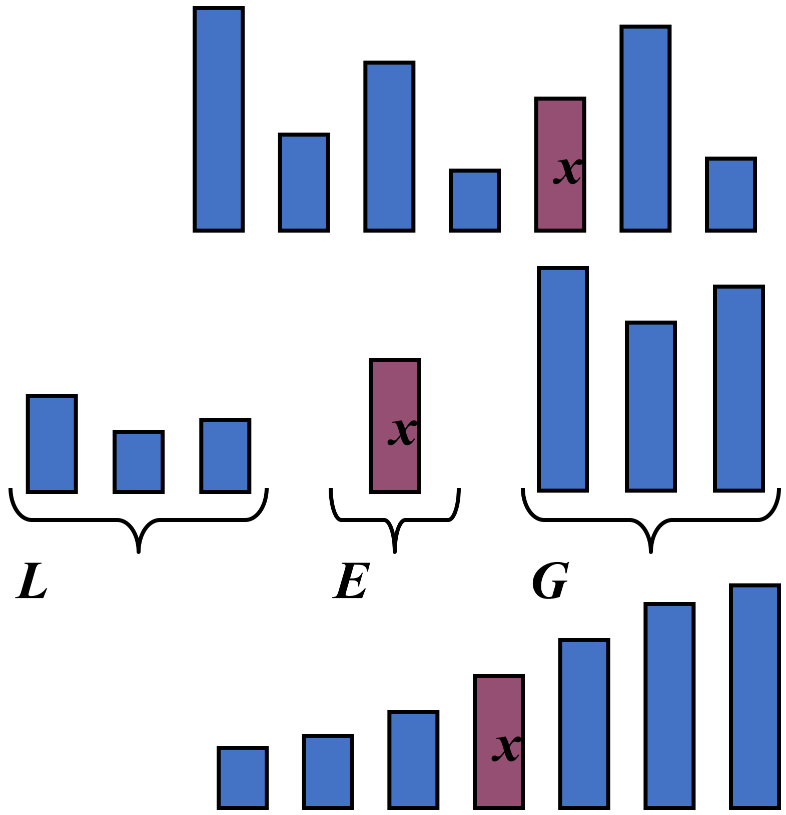
A randomized sorting algorithm based on divide-and-conquer

divide: pick a random element x (pivot) and partition into

- L: $< x$
- E: $= x$
- G: $> x$

conquer: sort L and G

combine: join L, E and G



Pseudo Code

```
quickSort(S):
```

Pseudo Code

```
quickSort(S):
  if (S.size()<2)
        return
```

Pseudo Code – Choose a pivot

```
quickSort(S):
  if (S.size()<2)
      return
  p = S.first()          // first as pivot
```

Pseudo Code - Parition

```
quickSort(S):
  if (S.size()<2)
        return
  p = S.first()            // first as pivot
  L, E, G = partition(S, p)
```

Pseudo Code – QuickSort L and G

```
quickSort(S):
  if (S.size()<2)
        return
  p = S.first()          // first as pivot
  L, E, G = partition(S, p)
  quickSort(L)
  quickSort(G)
```

Pseudo Code – then combine L, E, & G

```
quickSort(S):
  if (S.size()<2)
       return
  p = S.first()          // first as pivot
  L, E, G = partition(S, p)
  quickSort(L)
  quickSort(G)
  S = combine(L,E,G)
```

Pseudo Code – then combine L, E, & G (concat)

```
quickSort(S):
  if (S.size()<2)
        return
  p = S.first()          // first as pivot
  L, E, G = partition(S, p)
  quickSort(L)
  quickSort(G)
  S = L+E+G
```

Partition

```
partition(S, x)
   Input sequence S, pivot x
   Output subsequences L, E, G
   L, E, G = empty sequences






return L, E, G
```

Partition

Remove each y from S and insert into L, E or G

```
partition(S, x)
  Input sequence S, pivot x
  Output subsequences L, E, G
  L, E, G = empty sequences







return L, E, G
```

Partition

Remove each y from S and insert into L, E or G

```
partition(S, x)
  Input sequence S, pivot x
  Output subsequences L, E, G
  L, E, G = empty sequences
  while !S.isEmpty()









return L, E, G
```

Partition

Remove each y from S and insert into L, E or G

```
partition(S, x)
  Input sequence S, pivot x
  Output subsequences L, E, G
  L, E, G = empty sequences
  while !S.isEmpty()
    y = S.removefirst()
    if …
      …
    else if …
      …
    else
      …
  return L, E, G
```

Partition

Remove each y from S and insert into L, E or G

```
partition(S, x)
  Input sequence S, pivot x
  Output subsequences L, E, G
  L, E, G = empty sequences
  while !S.isEmpty()
    y = S.removefirst()
    if y < x
      …
    else if y = x
      …
    else
      …
return L, E, G
```

Partition

Remove each y from S and insert into L, E or G

```
partition(S, x)
  Input sequence S, pivot x
  Output subsequences L, E, G
  L, E, G = empty sequences
  while !S.isEmpty()
    y = S.removefirst()
    if y < x
      L.addLast(y)
    else if y = x
      E.addLast(y)
    else
      G.addLast(y)
  return L, E, G
```

Partition

Remove each y from S and insert into L, E or G

Each insertion and removal is $O(1)$

partition is $O(n)$

```
partition(S, x)
  Input sequence S, pivot x
  Output subsequences L, E, G
  L, E, G = empty sequences
  while !S.isEmpty()
    y = S.removefirst()
    if y < x
      L.addLast(y)
    else if y = x
      E.addLast(y)
    else
      G.addLast(y)
  return L, E, G
```

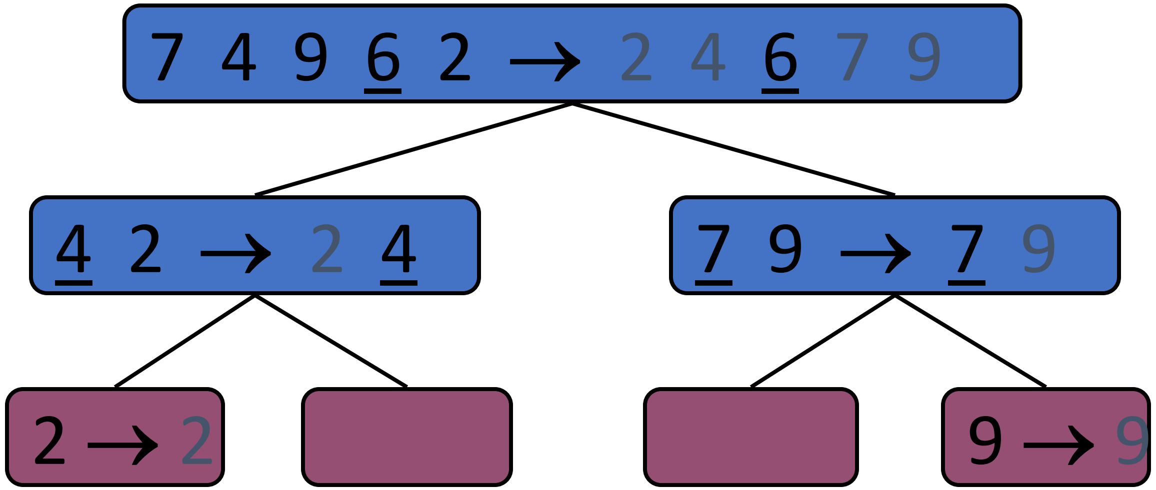
Pseudo Code

```
quickSort(S):
  if (S.size()<2) return
  p = S.first()          // first as pivot
  (L, E, G) = partition(S, p)
  quickSort(L)
  quickSort(G)
  S = L+E+G
```

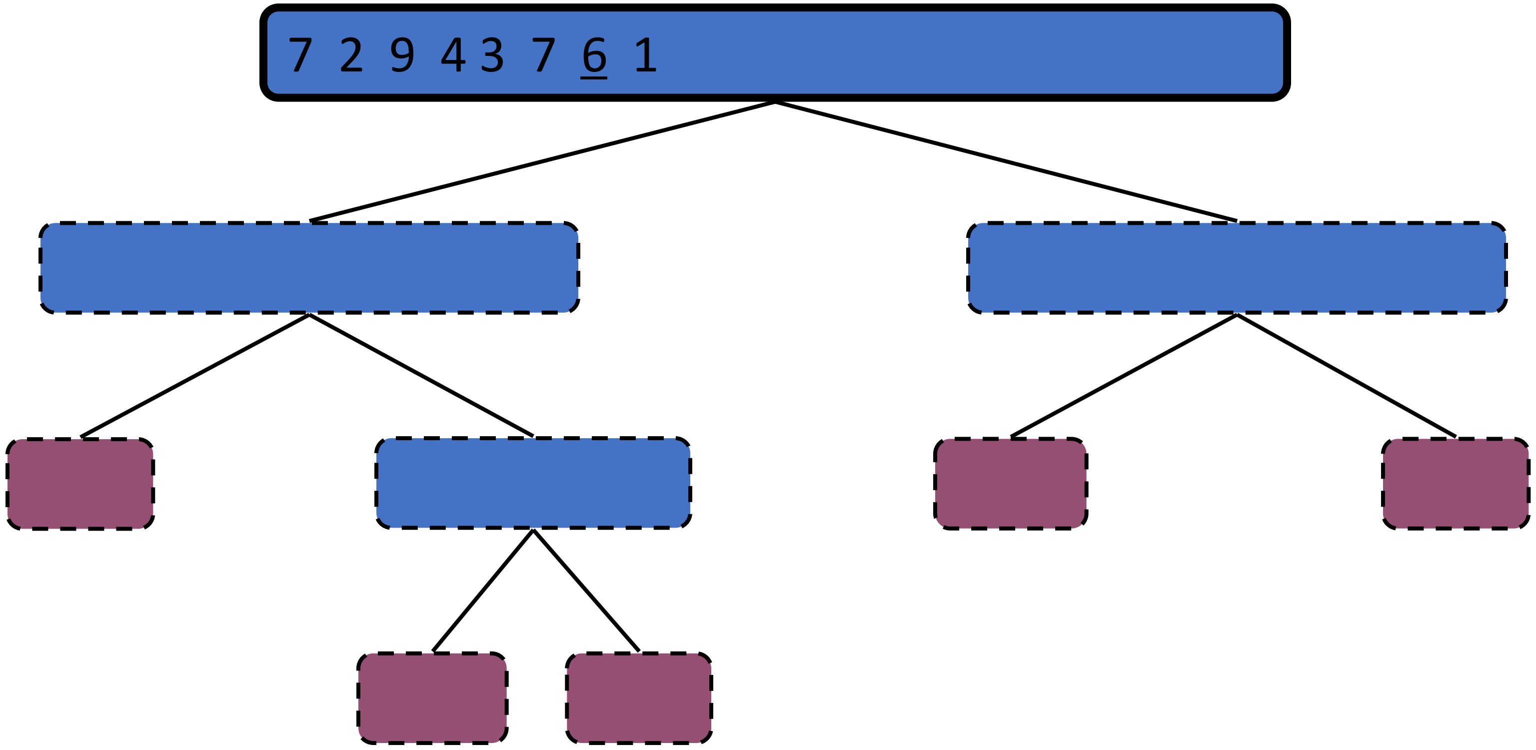
Quick Sort Tree

Execution is depicted by a binary tree

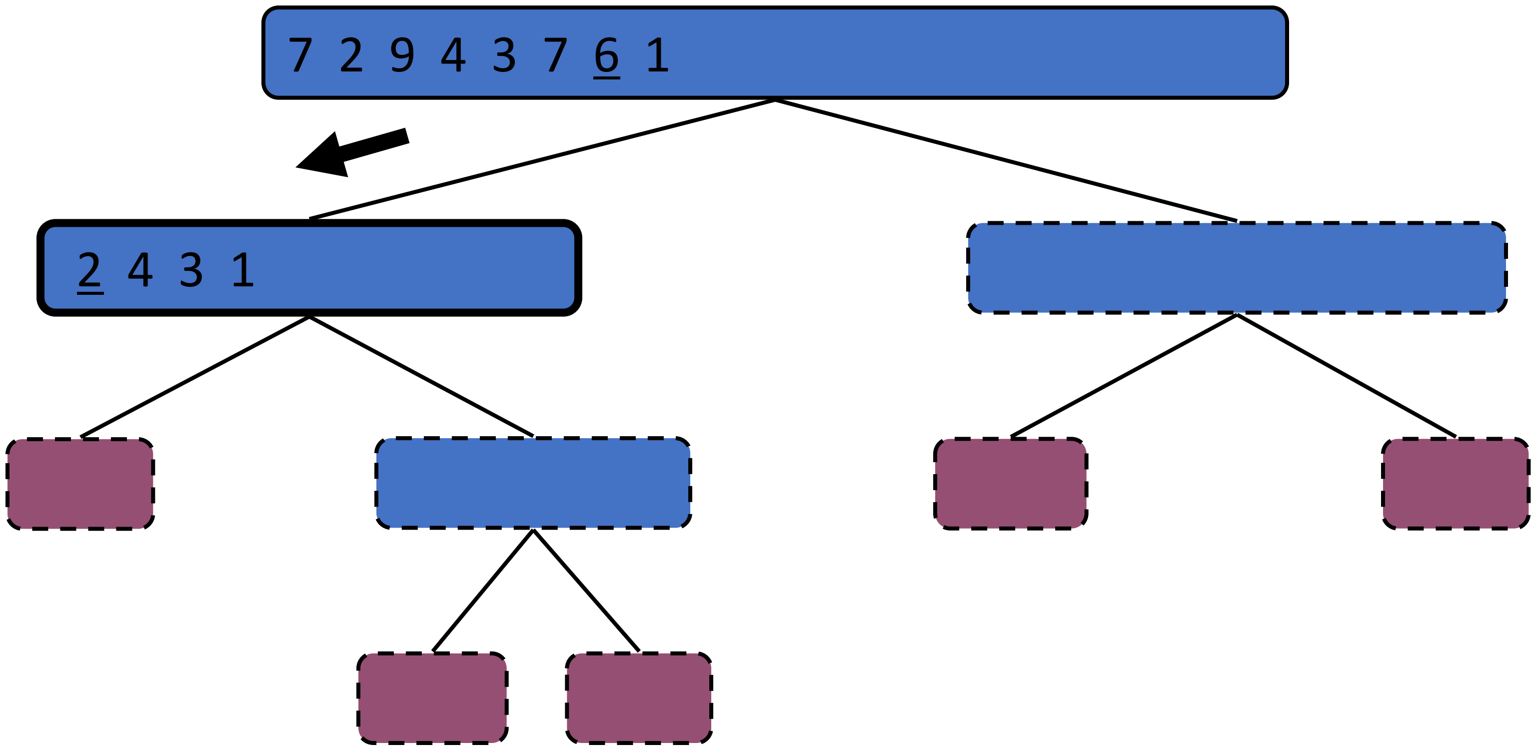
- each node is a recursive call
- leaves are sequences of size 0 or 1



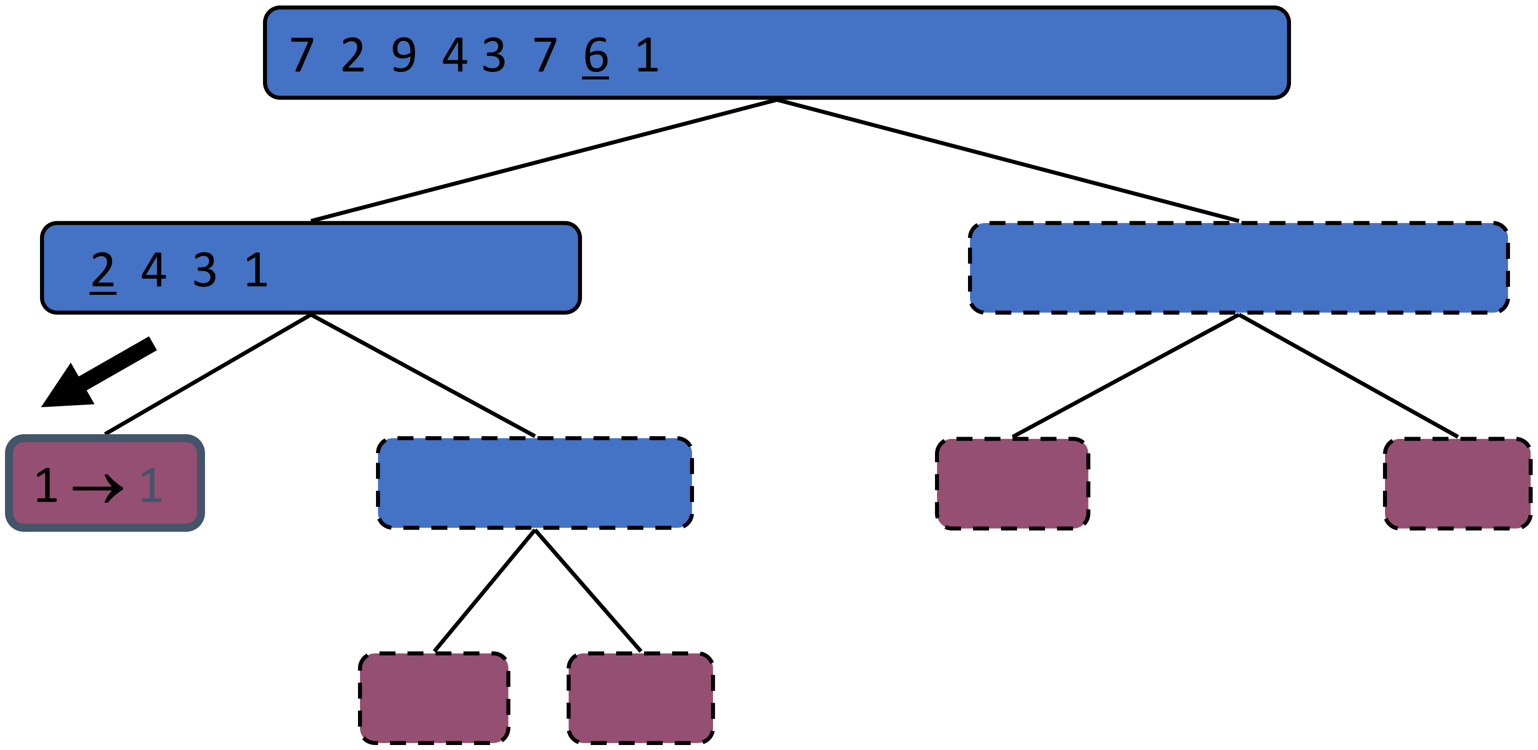
Example



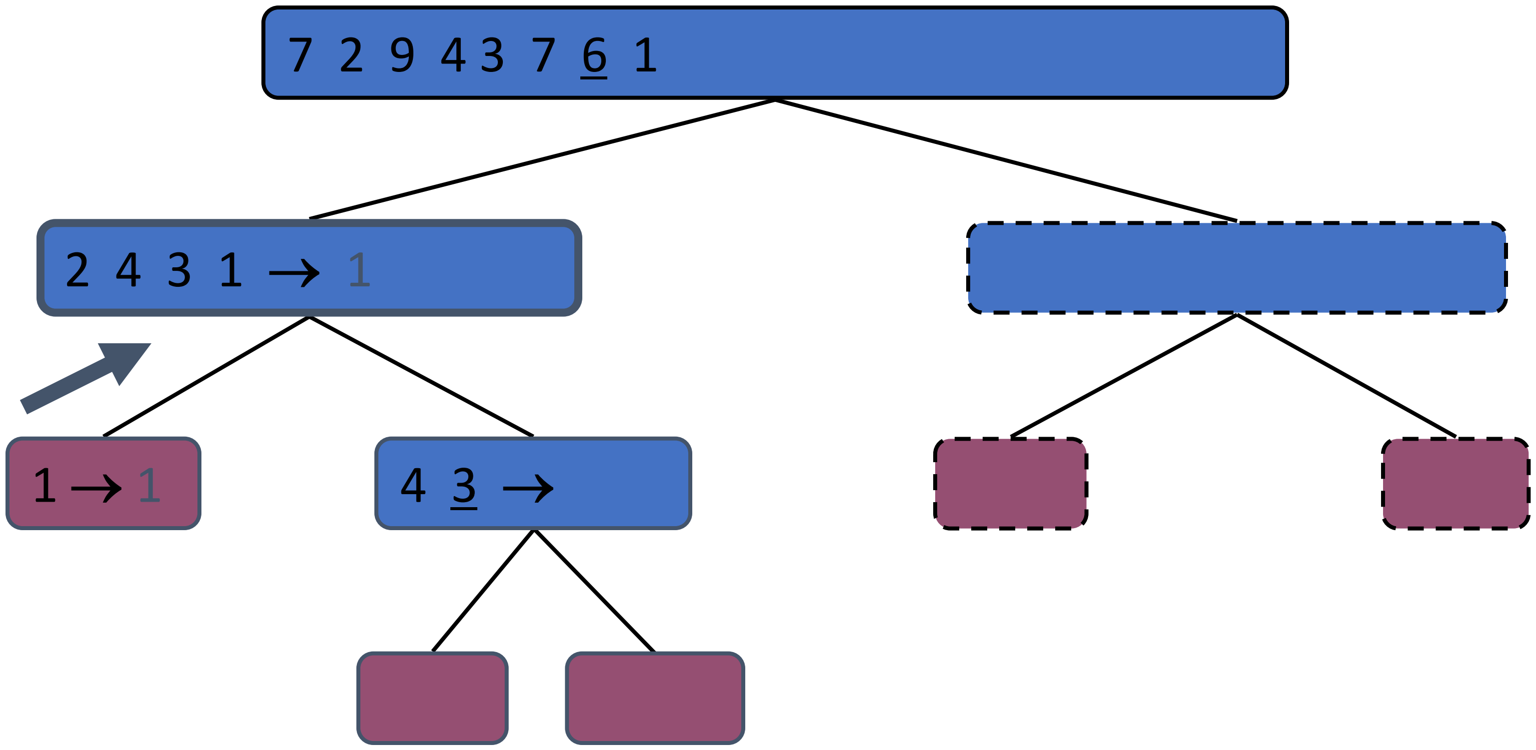
Example



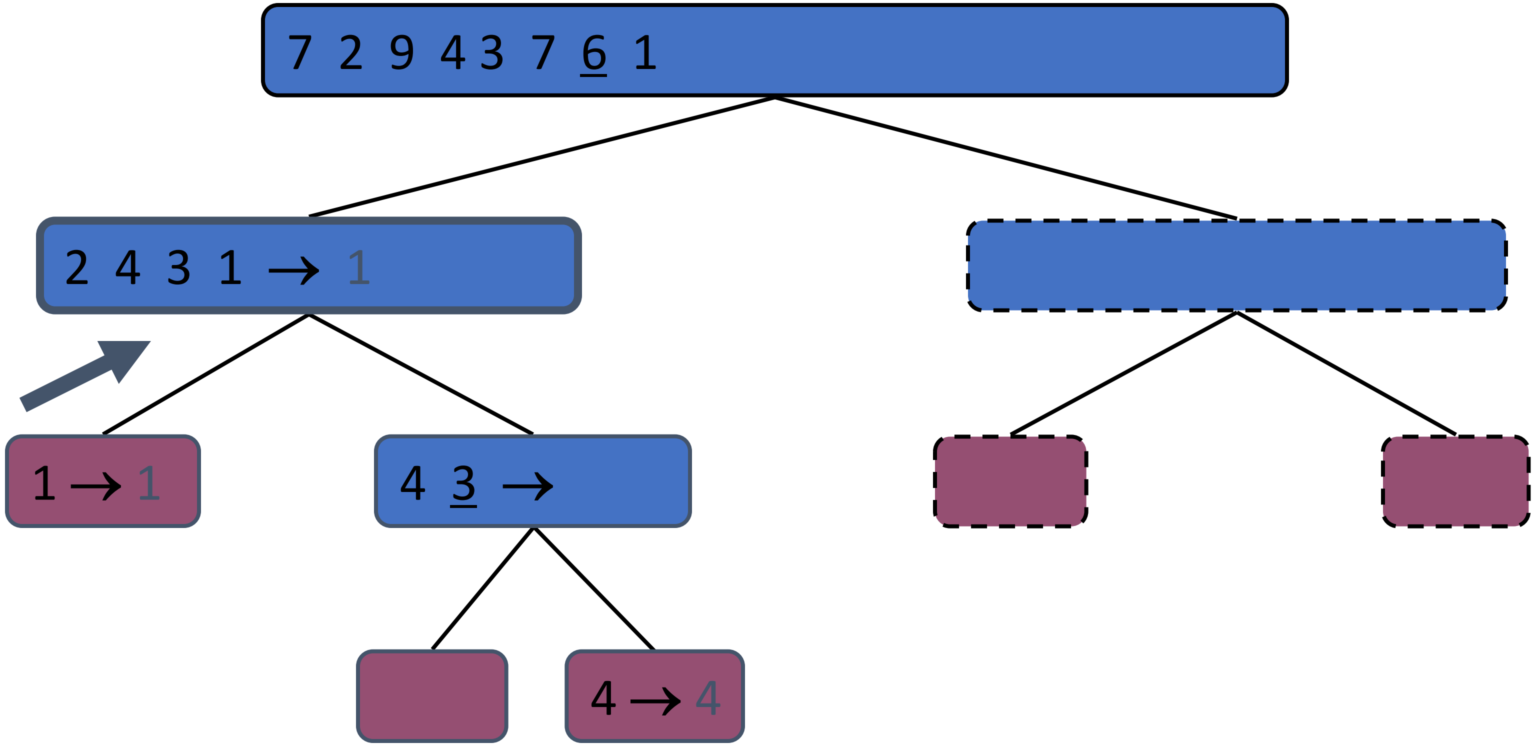
Example



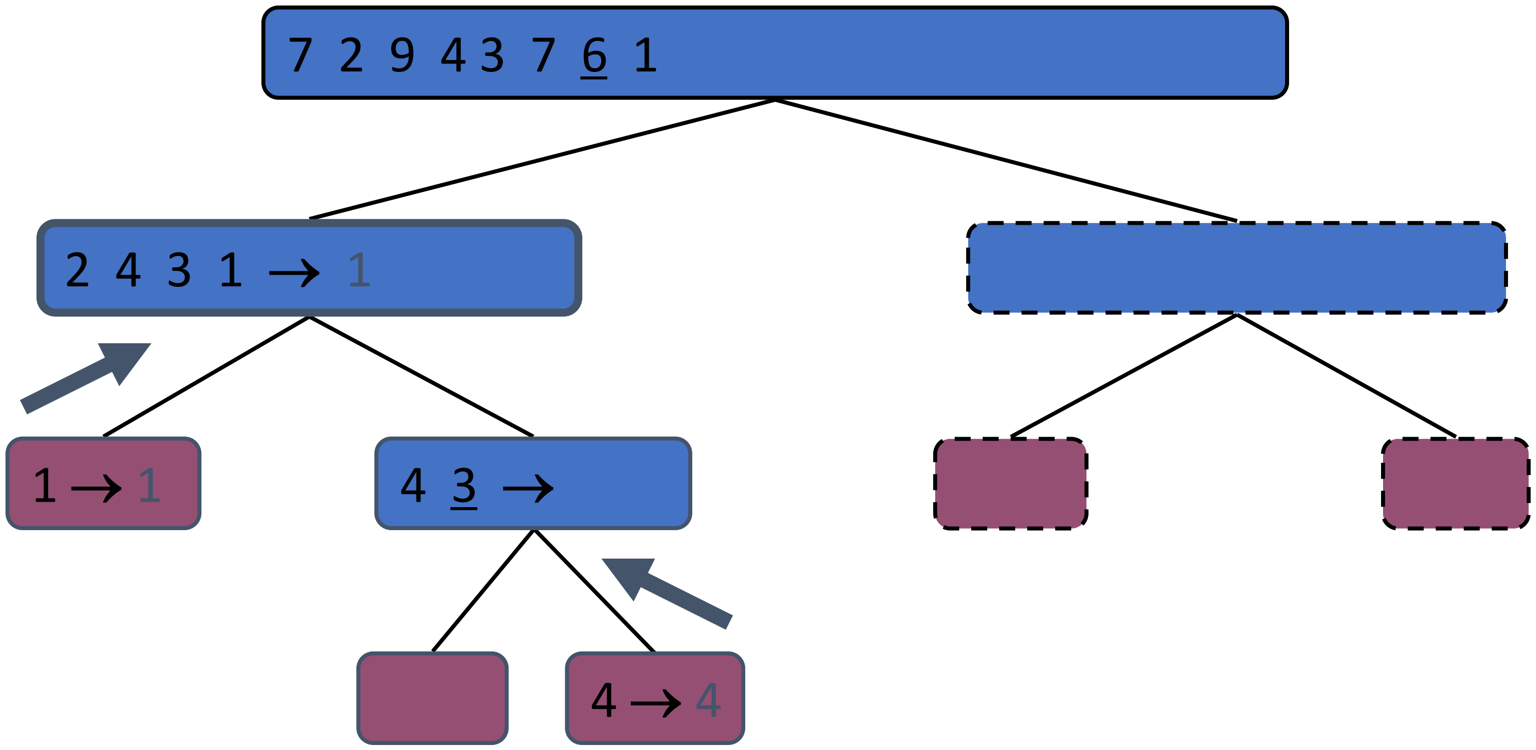
Example



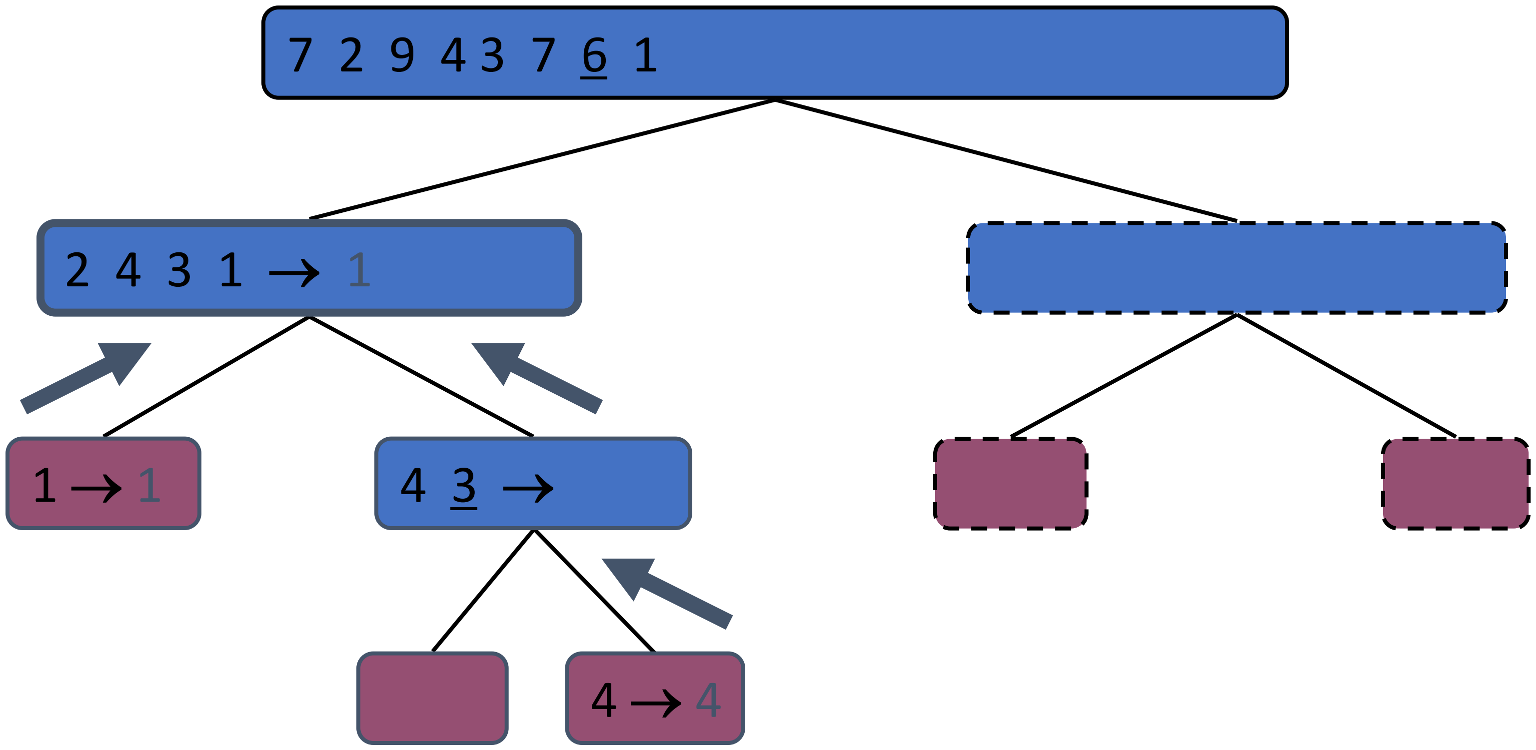
Example



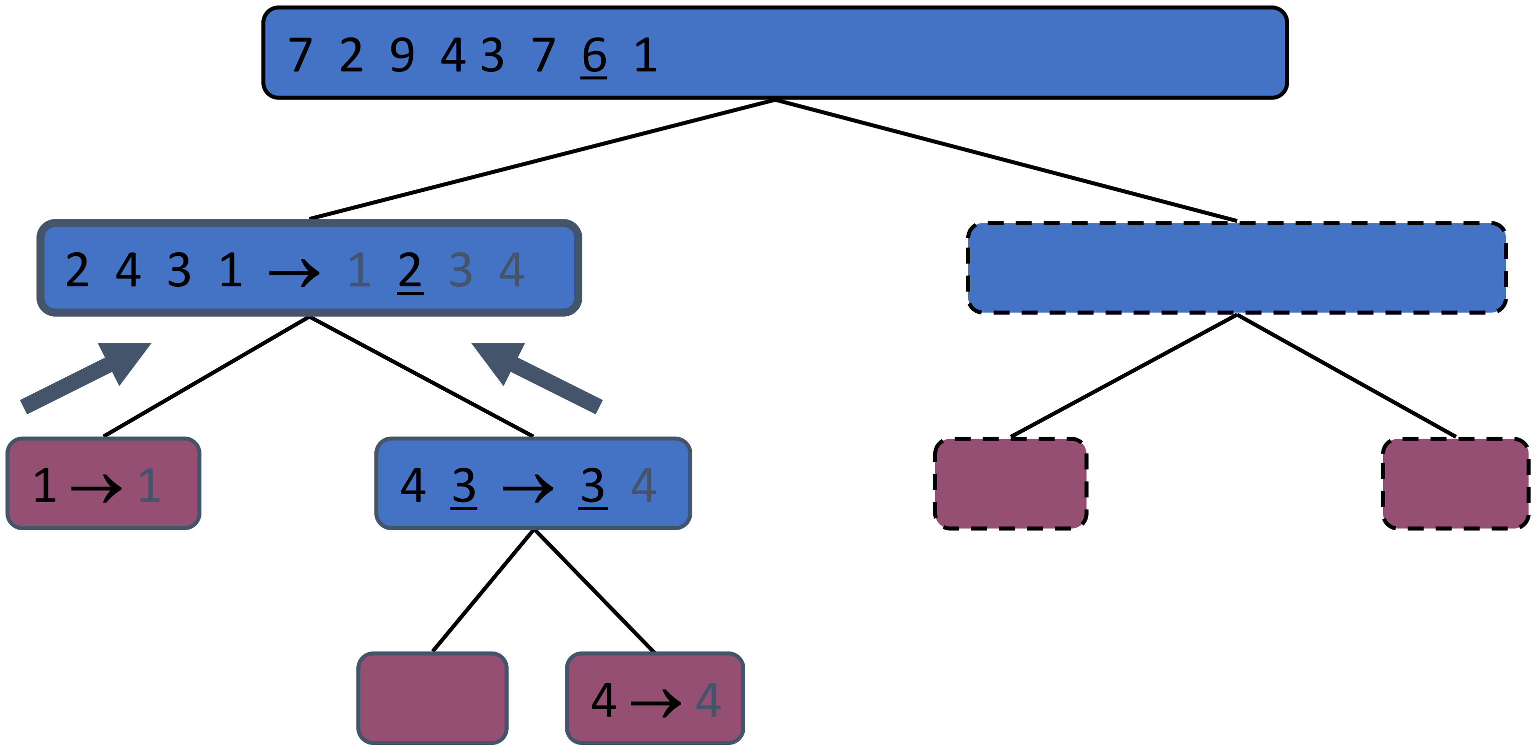
Example



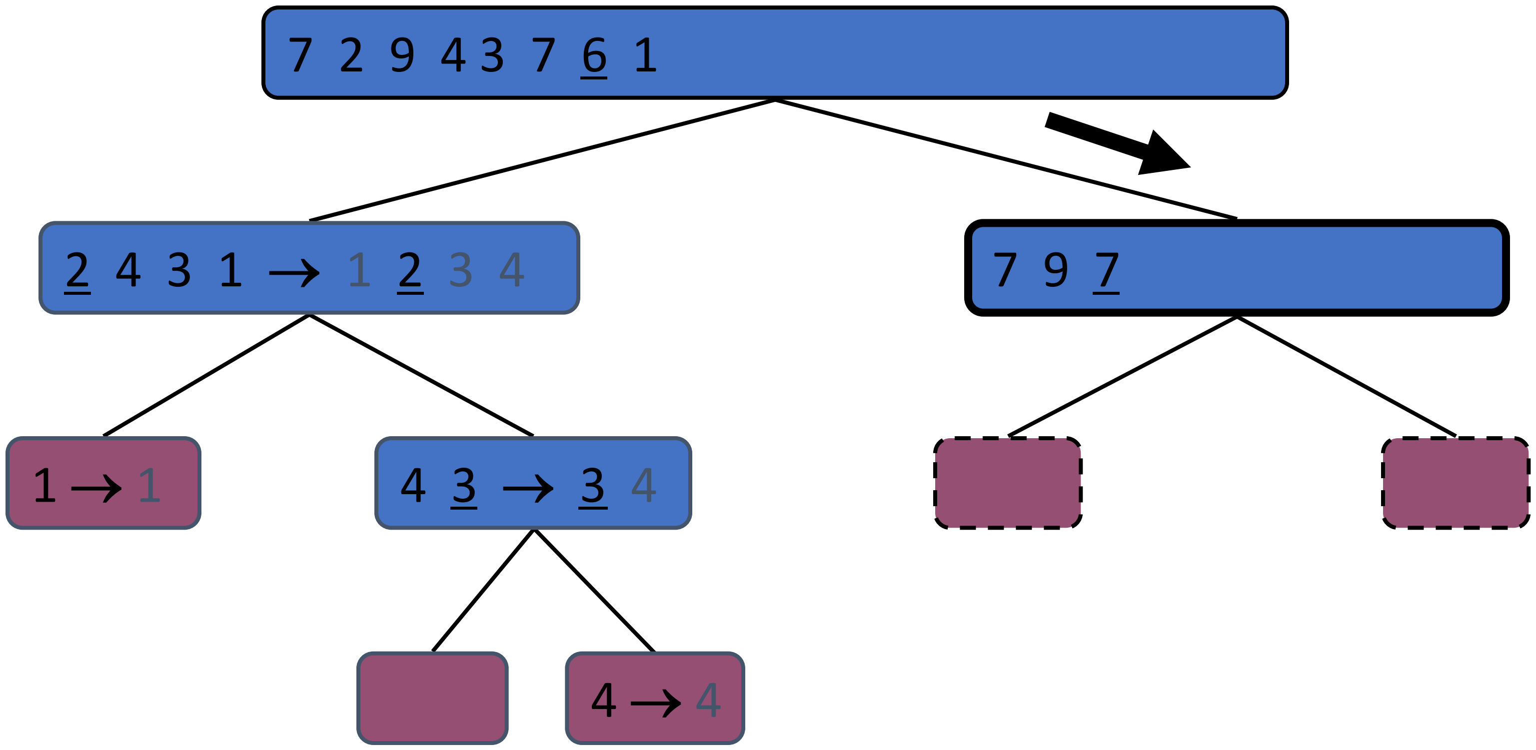
Example



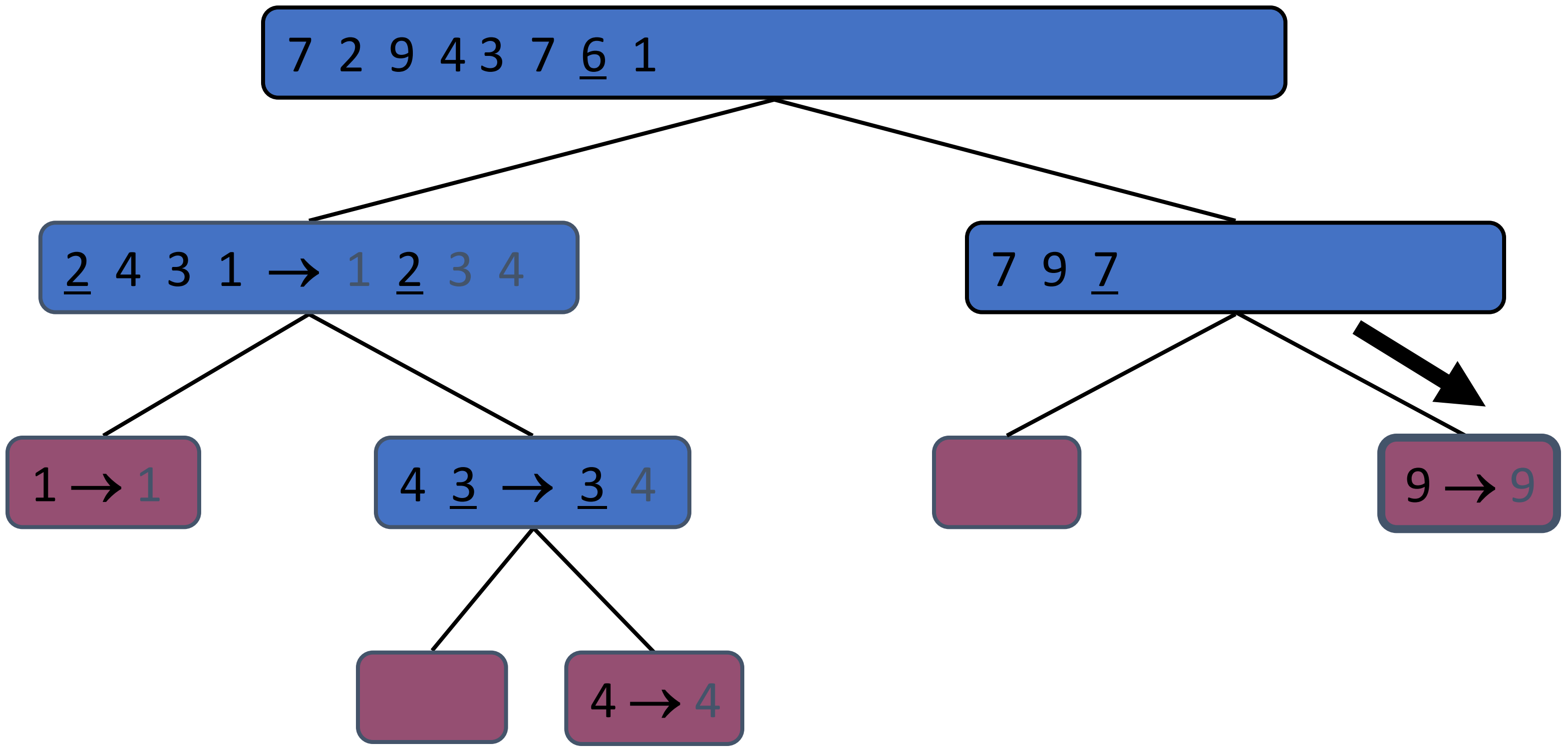
Example



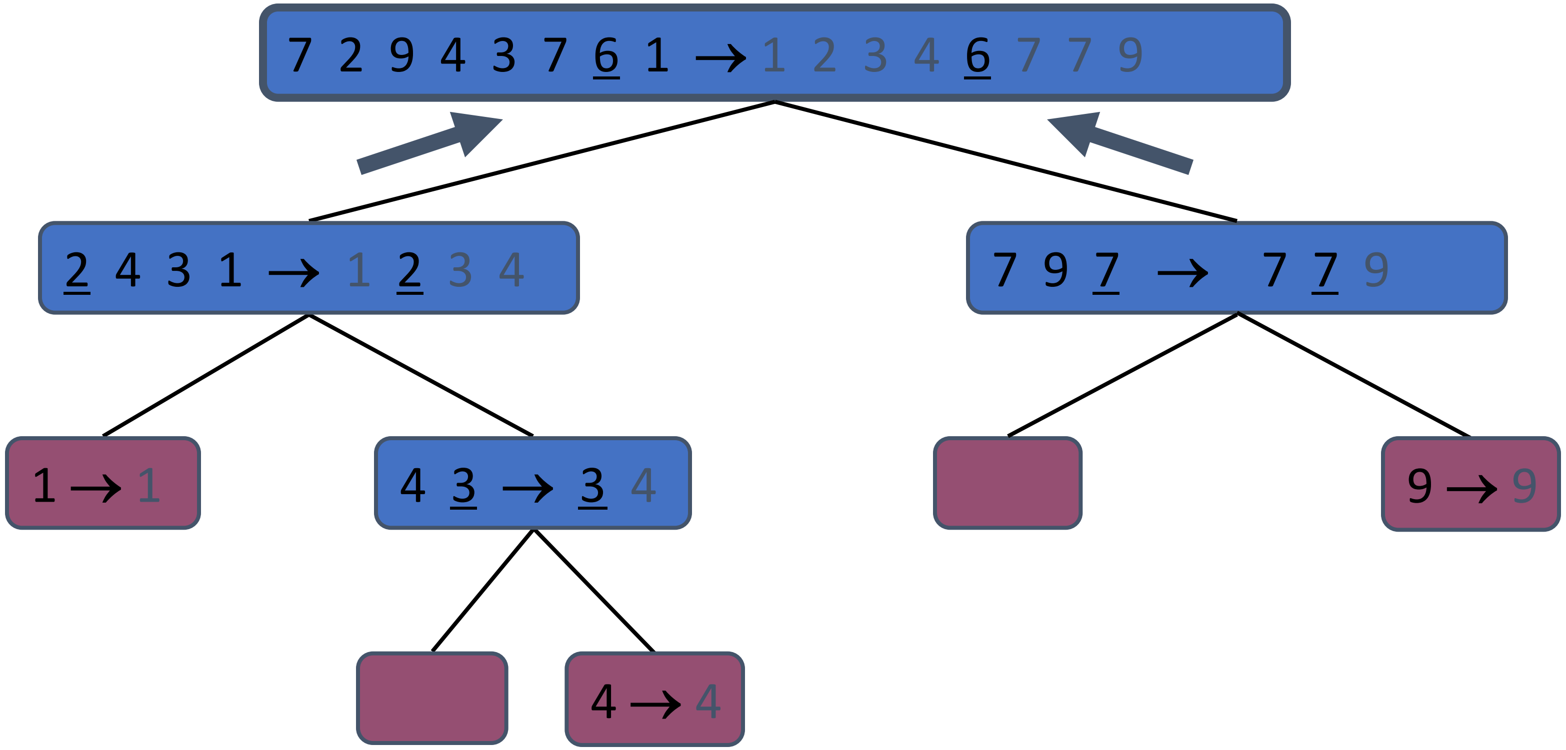
Example



Example



Example

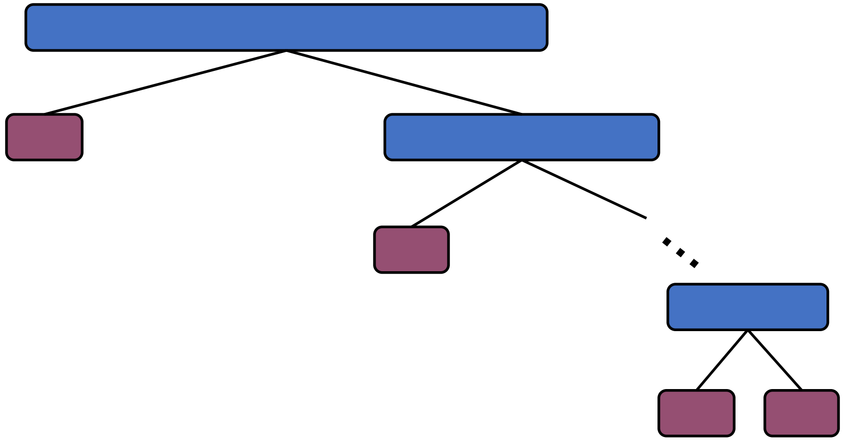


Worst-case Running Time

When the pivot is the min or max

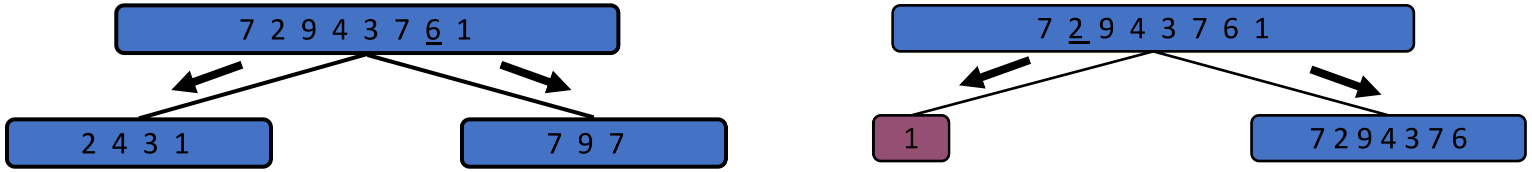
- one of L or G has size $n-1$
- $T(n) = n + (n-1) + \cdots + 2 + 1 = O(n^2)$

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



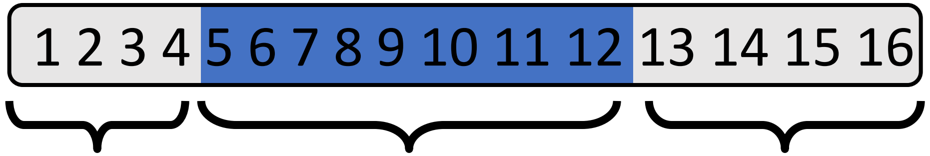
Expected Running Time

Good pivot: sizes of L and G are each less than $\frac{3}{4}n$



Good call **Bad call**

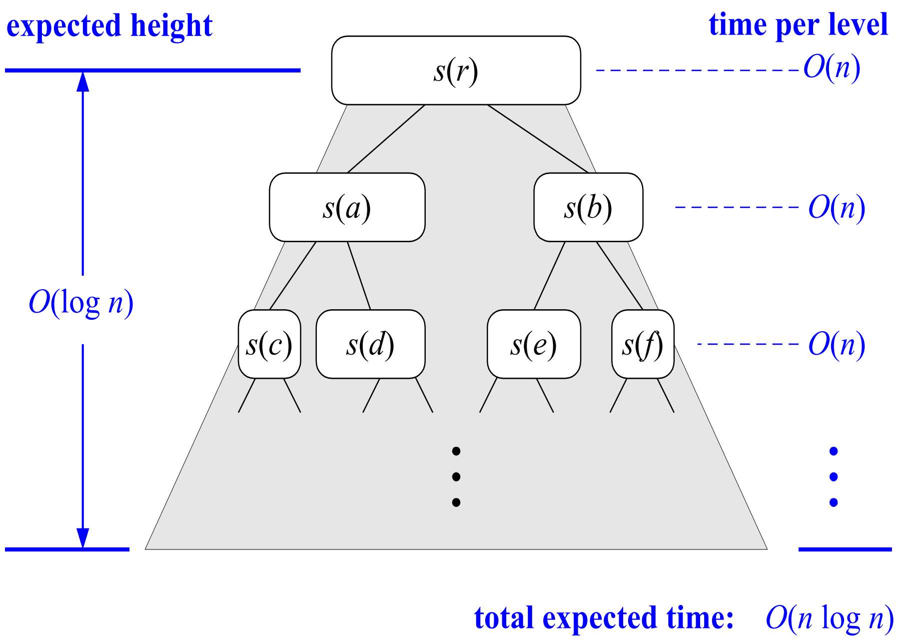
A call is good with probability 50% - ½ of the pivots cause good calls



Bad pivots **Good pivots** **Bad pivots**

Expected Run Time

- node at depth i
  - $\frac{i}{2}$ ancestors are good
  - size of input is at most $(\frac{3}{4})^{\frac{i}{2}}n$
- Expected height of tree is $O(logn)$
- Amount of work at each level is $O(n)$
- Overall expected run time is $O(nlogn)$



In-place Quick Sort

partition rearranges the input list

3 pieces, 2 indices

- $L: [0, l-1], E: [l, r], G: [r+1, n-1]$
- recursive calls on $[0, l-1]$ and $[r+1, n-1]$

```
inPlaceQuickSort(S, s, e)
  if s ≥ e
    return
  i = random int in [s,e]
  x = S(i)
  (l,r)←inPlacePartition(x)
  inPlaceQuickSort(S,s,l-1)
  inPlaceQuickSort(S,r+1,e)
```

In-place Quick Sort

partition rearranges the input list

2 pieces, 1 index

- L: $[0, l-1]$, $E \cup G$: $[l+1, n-1]$
- recursive calls on $[0, l-1]$ and $[l+1, n-1]$

```
inPlaceQuickSort(S, s, e)
  if s ≥ e
   return
  l←inPlacePartition(S, s, e)
  inPlaceQuickSort(S,s,l-1)
  inPlaceQuickSort(S,l+1,e)
```

In-place Partitioning (Hoare’s)

Use two indices to split into L and $E \cup G$

`l` `r`

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9 (pivot = 6)

Repeat until `l` and `r` cross:

- Move `l` to the right to find $\geq x$
- Move `r` to the left to find $< x$
- swap elements at `l` and `r`

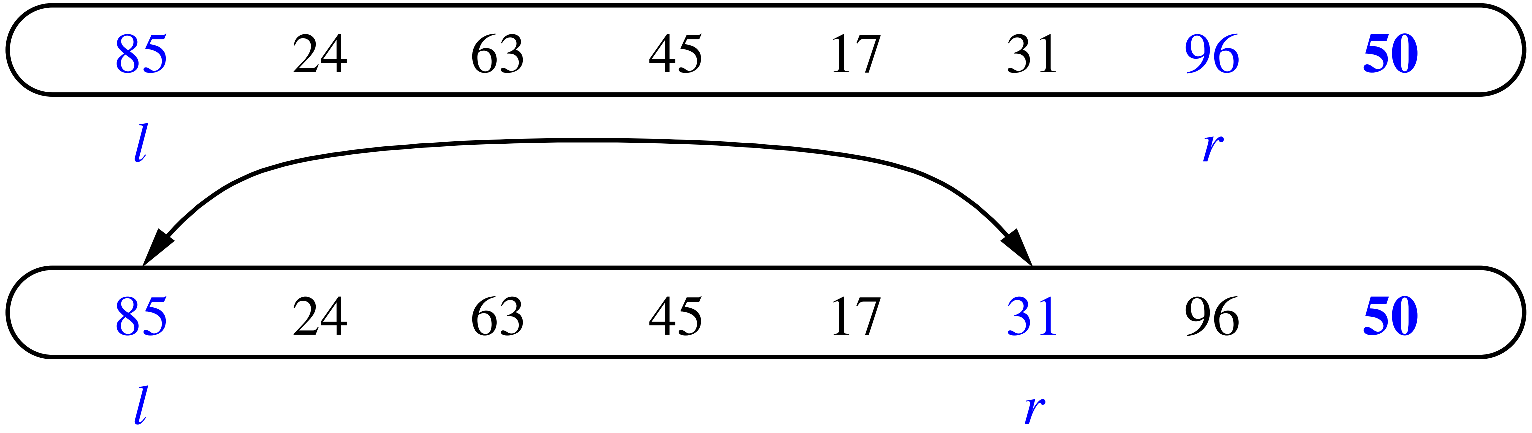
`l` `r`

3 2 5 1 0 7 3 5 9 2 7 9 8 9 7 **6** 9

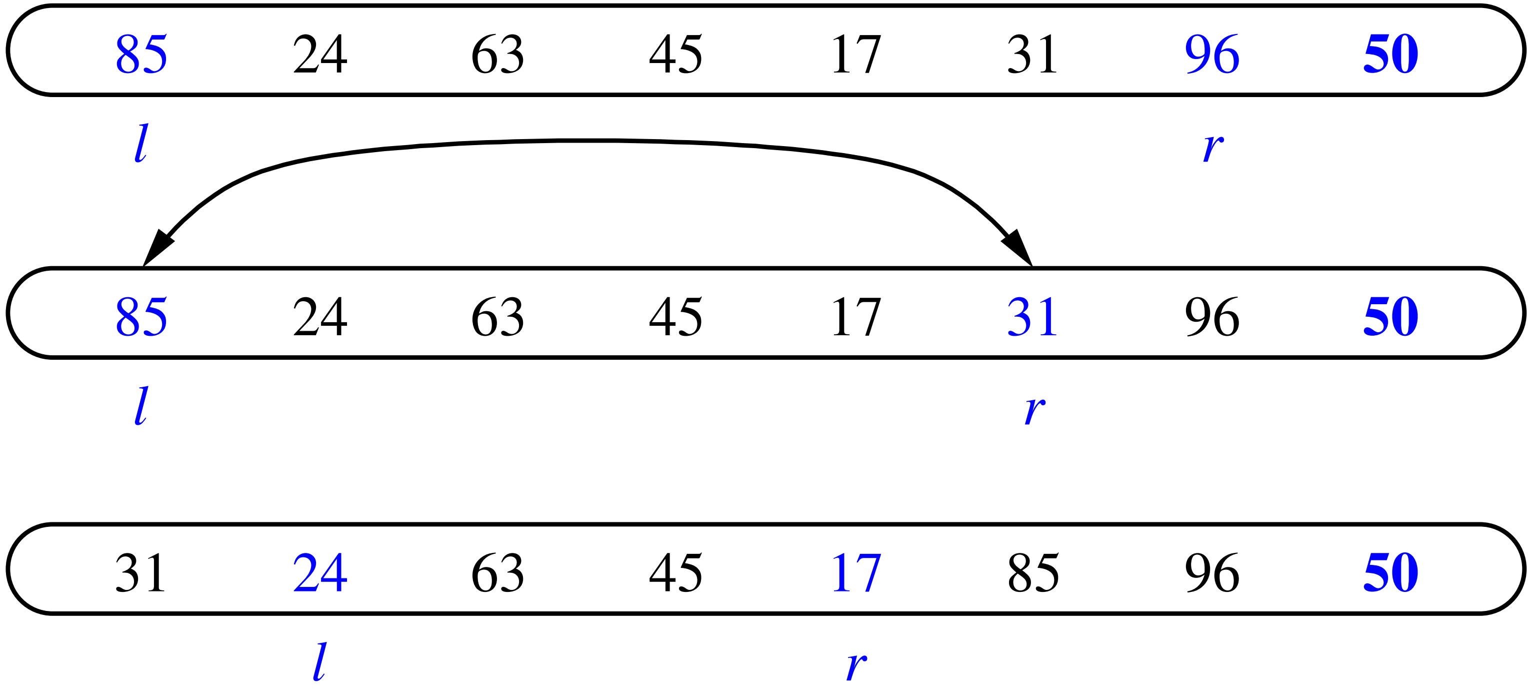
Example

85	24	63	45	17	31	96	**50**
l						r	

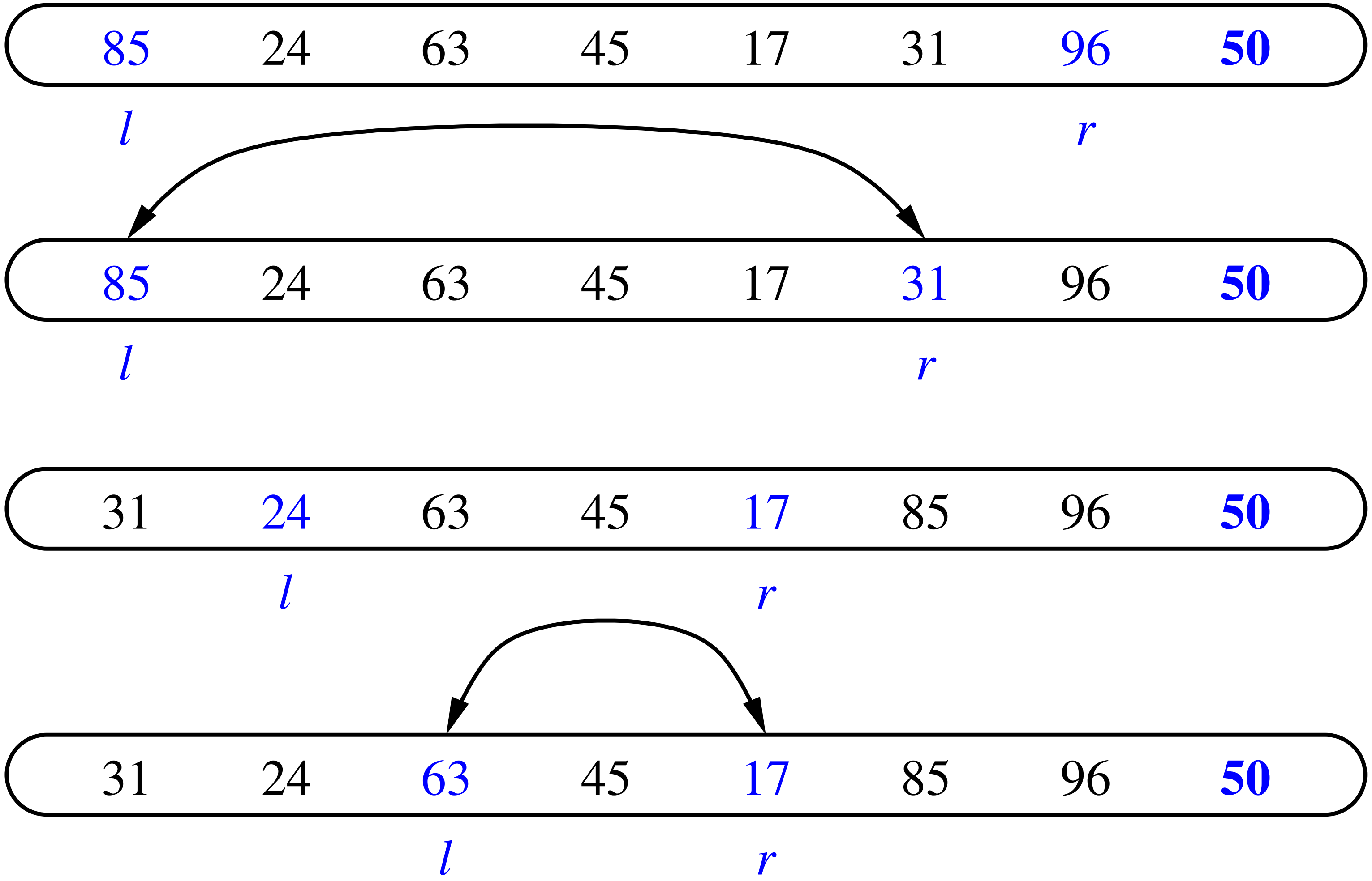
Example



Example


85 24 63 45 17 31 96 50
l r
85 24 63 45 17 31 96 50
l r
31 24 63 45 17 85 96 50
l r

Example


85 24 63 45 17 31 96 50
l r
85 24 63 45 17 31 96 50
l r
31 24 63 45 17 85 96 50
l r
31 24 63 45 17 85 96 50
l r

Example

31	24	17	45	63	85	96	**50**
			l, r				

Example

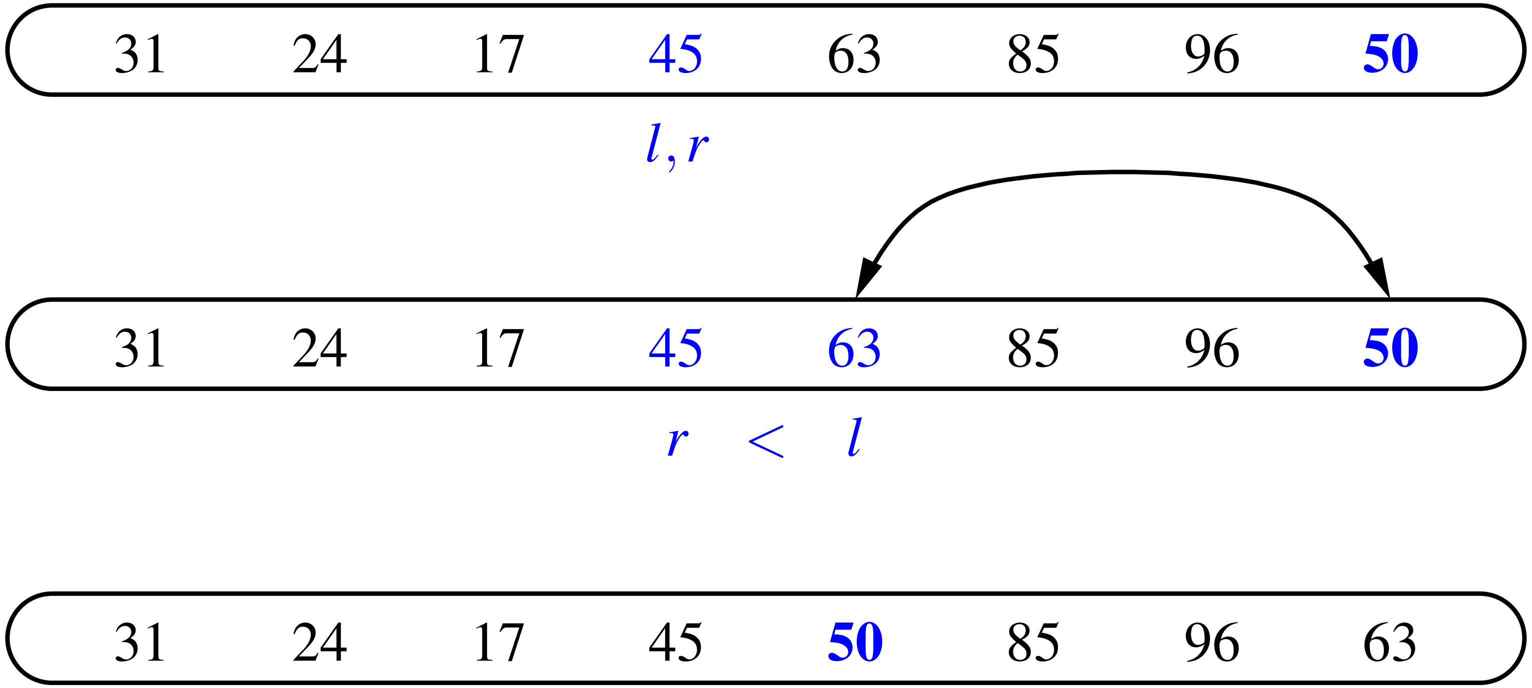
31 24 17 45 63 85 96 **50**

l, r

31 24 17 45 63 85 96 **50**

$r < l$

Example


31 24 17 45 63 85 96 50
l, r
31 24 17 45 63 85 96 50
r < l
31 24 17 45 50 85 96 63

Summary of Sorting Algorithms

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (small inputs <1k)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (small inputs <1k)
quick-sort	$O(nlogn)$ expected	▪ in-place, randomized ▪ fastest (large inputs 1K-1M)
heap-sort	$O(nlogn)$	▪ in-place ▪ fast (large inputs 1K-1M)
merge-sort	$O(nlogn)$	▪ sequential data access ▪ fast (huge inputs >1M)

quicksort vs mergesort

Quicksort is not a good choice for small n – too much bookkeeping

Why is quicksort is often preferred over mergesort?

- mergesort requires extra memory
- worst-case of quicksort is avoidable with randomized pivot choice
- quicksort exhibits good cache locality and works particularly well on arrays

mergesort is preferred for very large n and linked lists