Quicksort

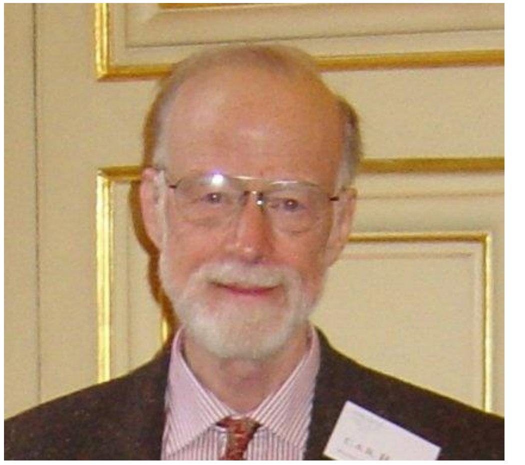
"There's nothing in your head the sorting hat can't see. So try me on and I will tell you where you ought to be."

-The Sorting Hat, *Harry Potter and the Sorcerer's Stone*

Quicksort

- Quicksort is the most popular fast sorting algorithm:
  – It has an average running time case of Θ(n*log n)
- The other fast sorting algorithms, Mergesort and Heapsort, also run in Θ(n*log n) time, but
  – have fairly large constants hidden in their algorithms
  – tend to move data around more than desirable

Quicksort

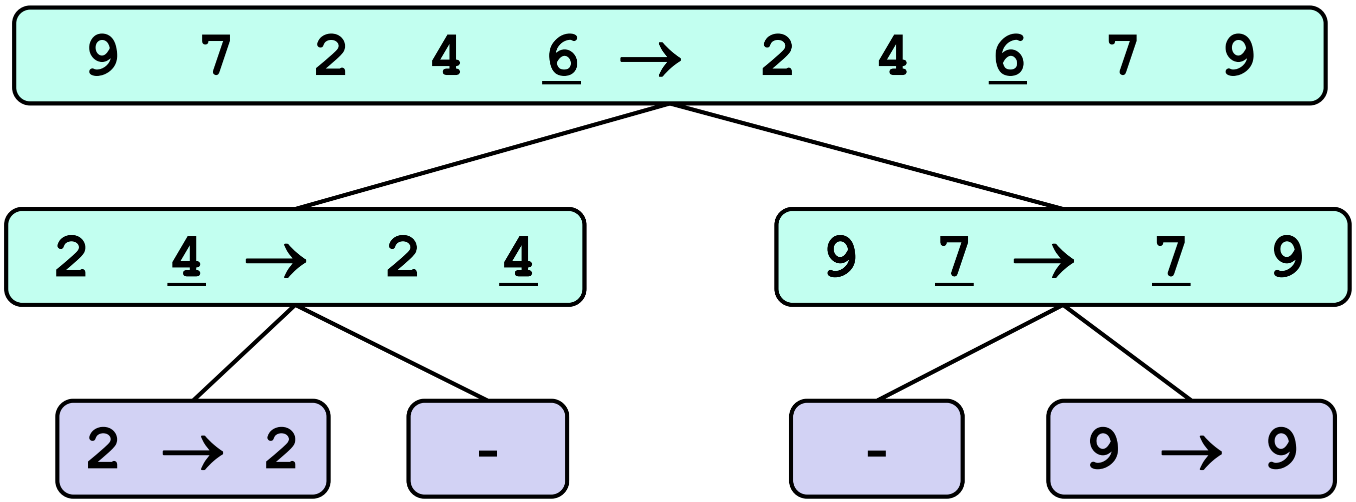


- Invented by C.A.R. (Tony) Hoare
- A Divide-and-Conquer approach that uses recursion:
  - If the list has 0 or 1 elements, it's sorted
  - Otherwise, pick any element ***p*** in the list. This is called the ***pivot value***
  - Partition the list, minus the pivot, into two sub-lists:
    - one of values less than the pivot and another of those greater than the pivot
    - equal values go to either
  - Return the Quicksort of the first list followed by the Quicksort of the second list.

Quicksort Tree

Use binary tree to represent the execution of Quicksort

- Each node represents a recursive call of Quicksort
- Stores
  - Unsorted sequence before the execution and its pivot
  - Sorted sequence at the end of the execution
- The leaves are calls on sub-sequences of size 0 or 1



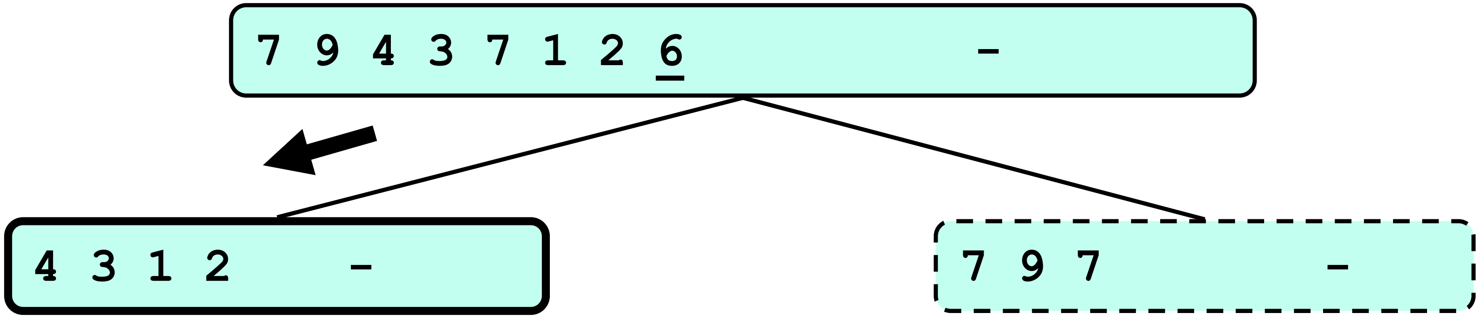
Example: Quicksort Execution

- Pivot selection – last value in list: **6**

```
7 9 4 3 7 1 2 6     -
```

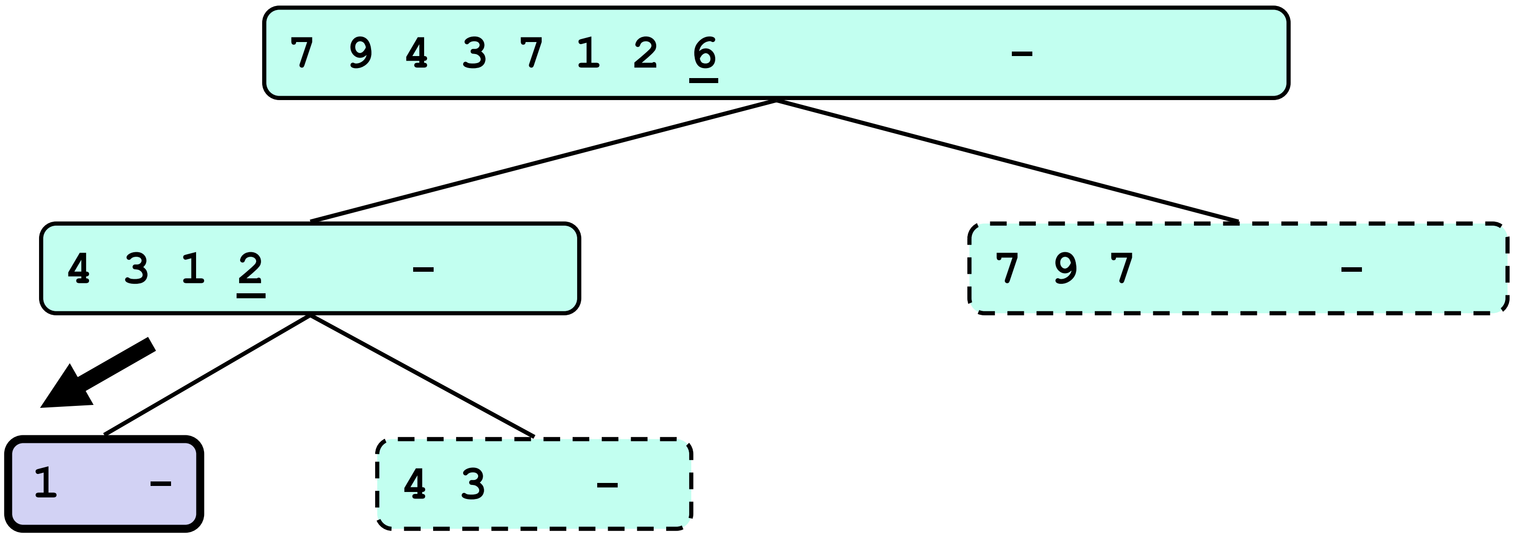
Example: Quicksort Execution

- Partition around **6**
- Recursive call on sub-list with smaller values



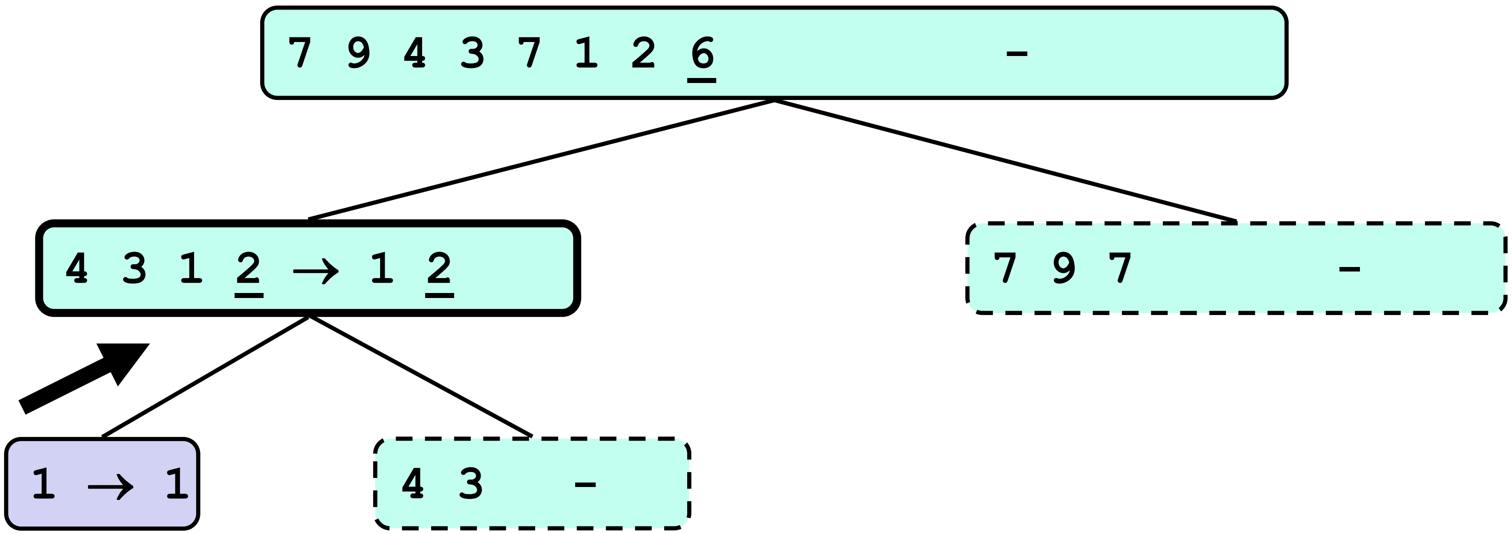
Example: Quicksort Execution

- Partition around **2**
- Recursive call on sub-list of smaller values



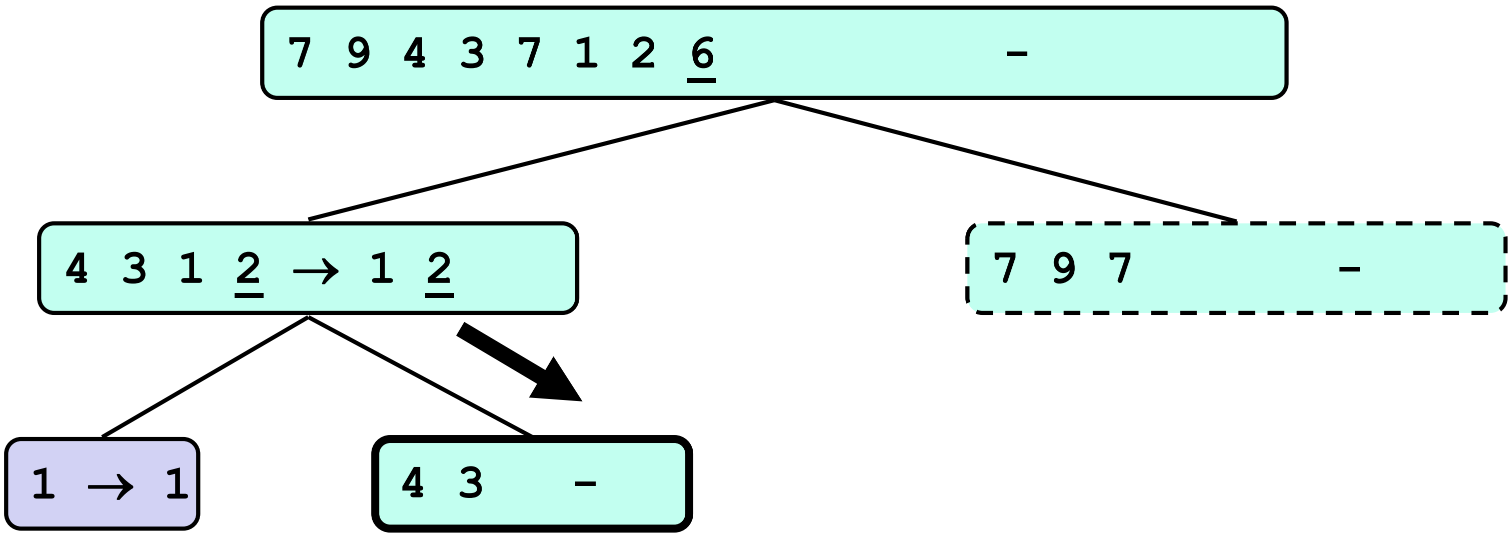
Example: Quicksort Execution

- Base case
- Return from sub-list of smaller values



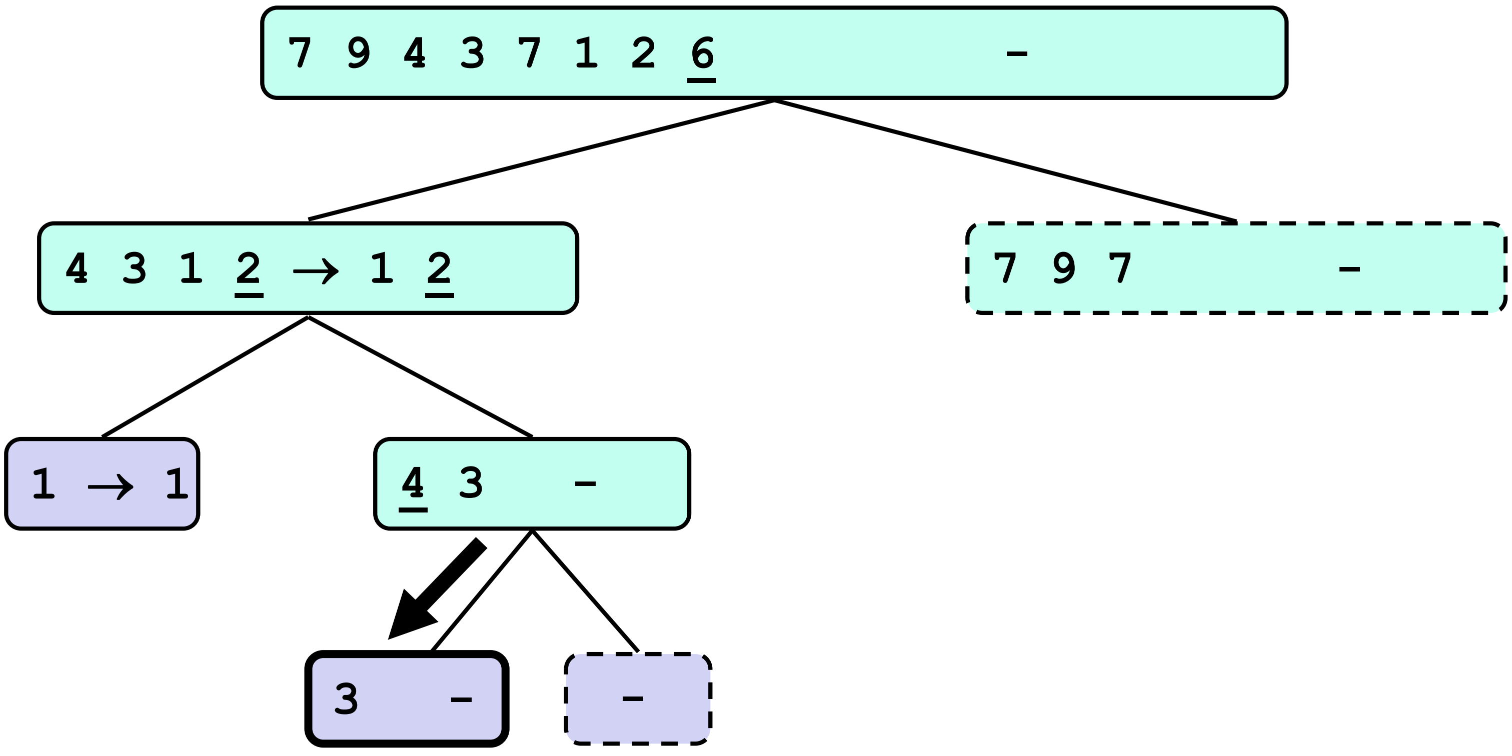
Example: Quicksort Execution

- Recursive call on sub-list of larger values



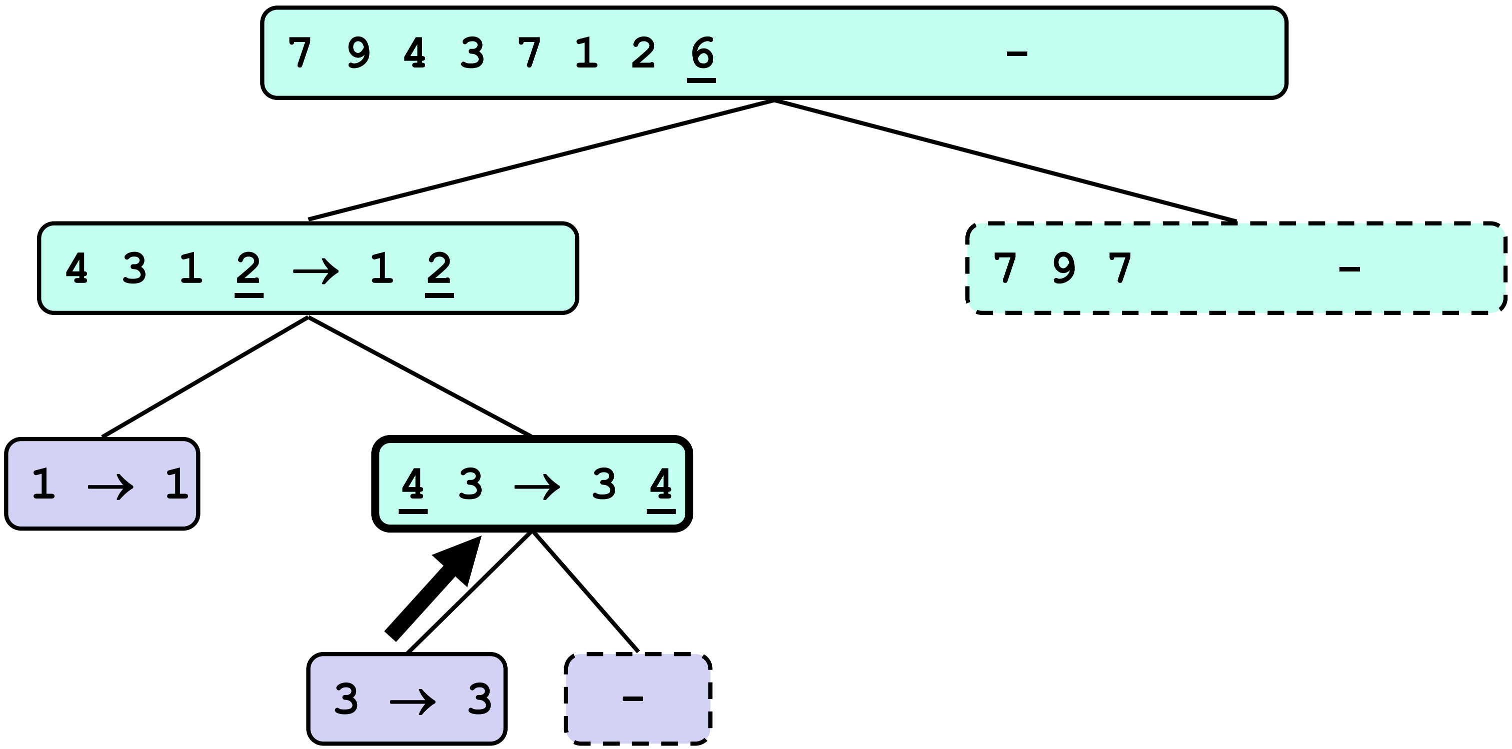
Example: Quicksort Execution

- Partition around **4**
- Recursive call on sub-list of smaller values



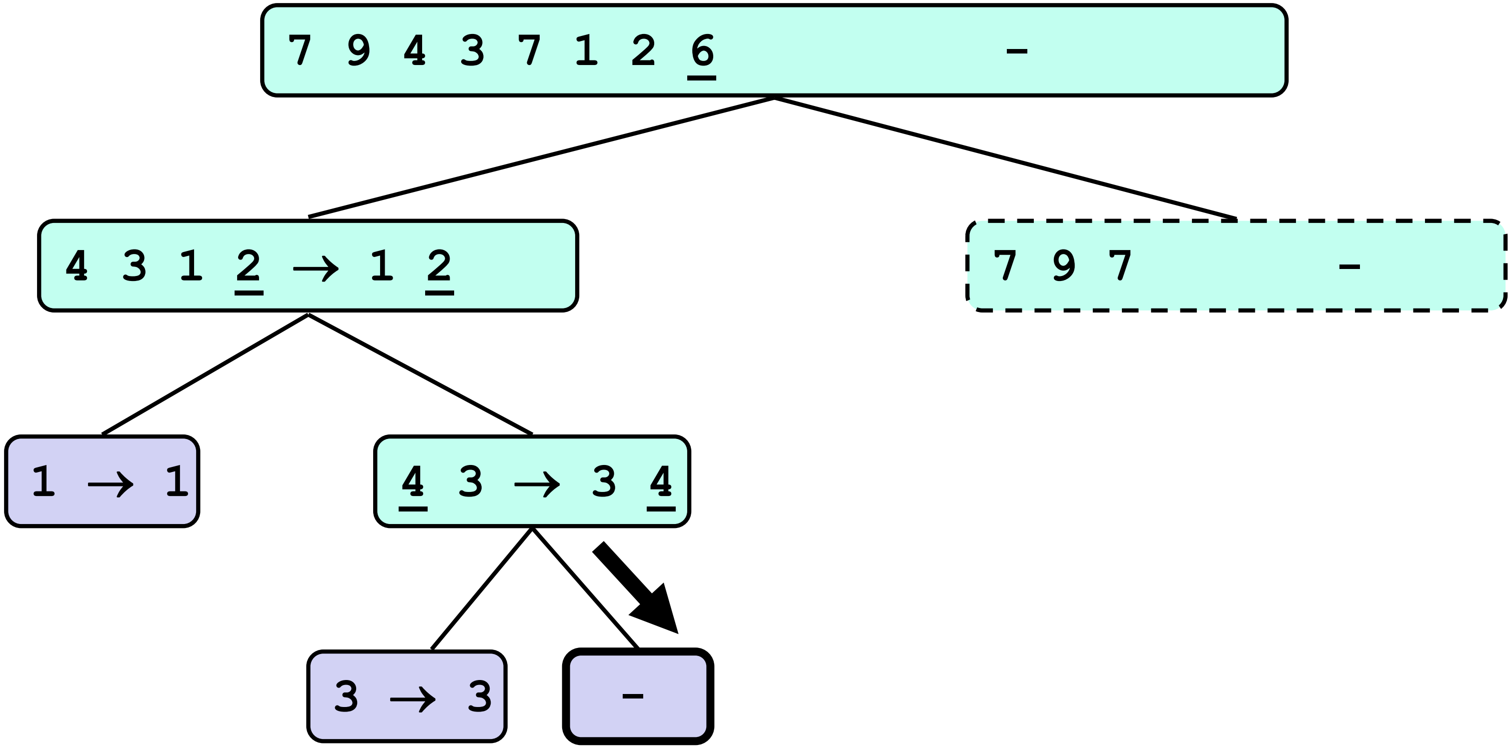
Example: Quicksort Execution

- Base case
- Return from sub-list of smaller values



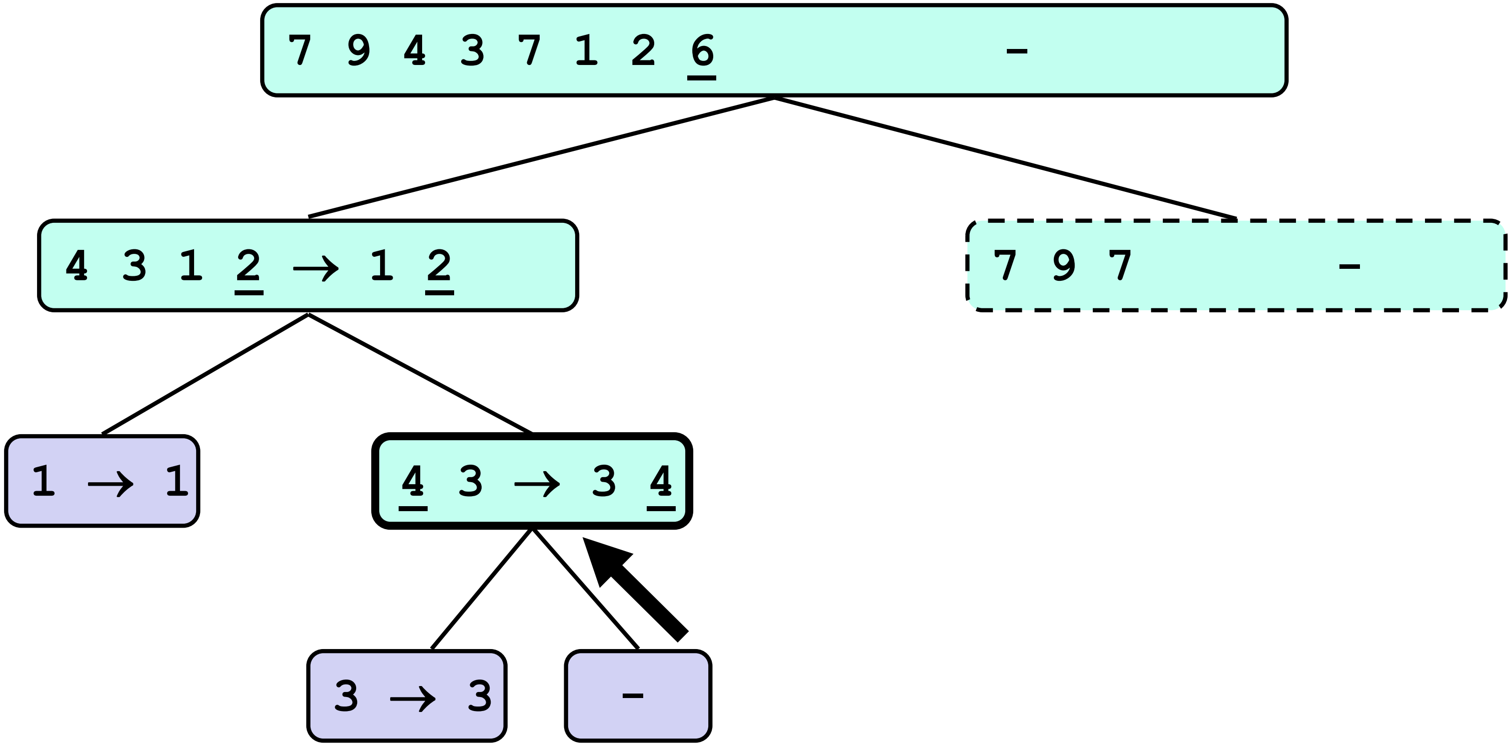
Example: Quicksort Execution

- Recursive call on sub-list of larger values



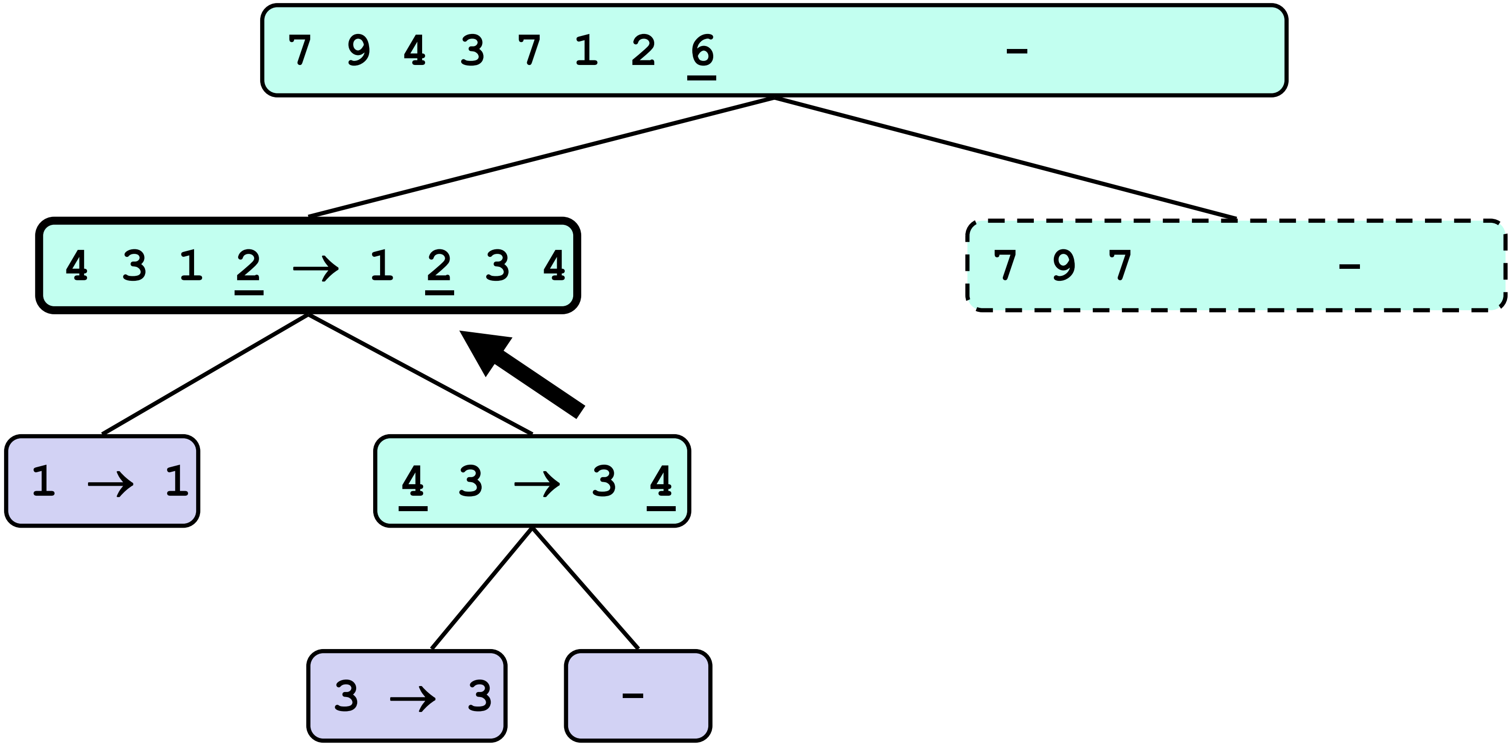
Example: Quicksort Execution

- Base case
- Return from sub-list of larger values



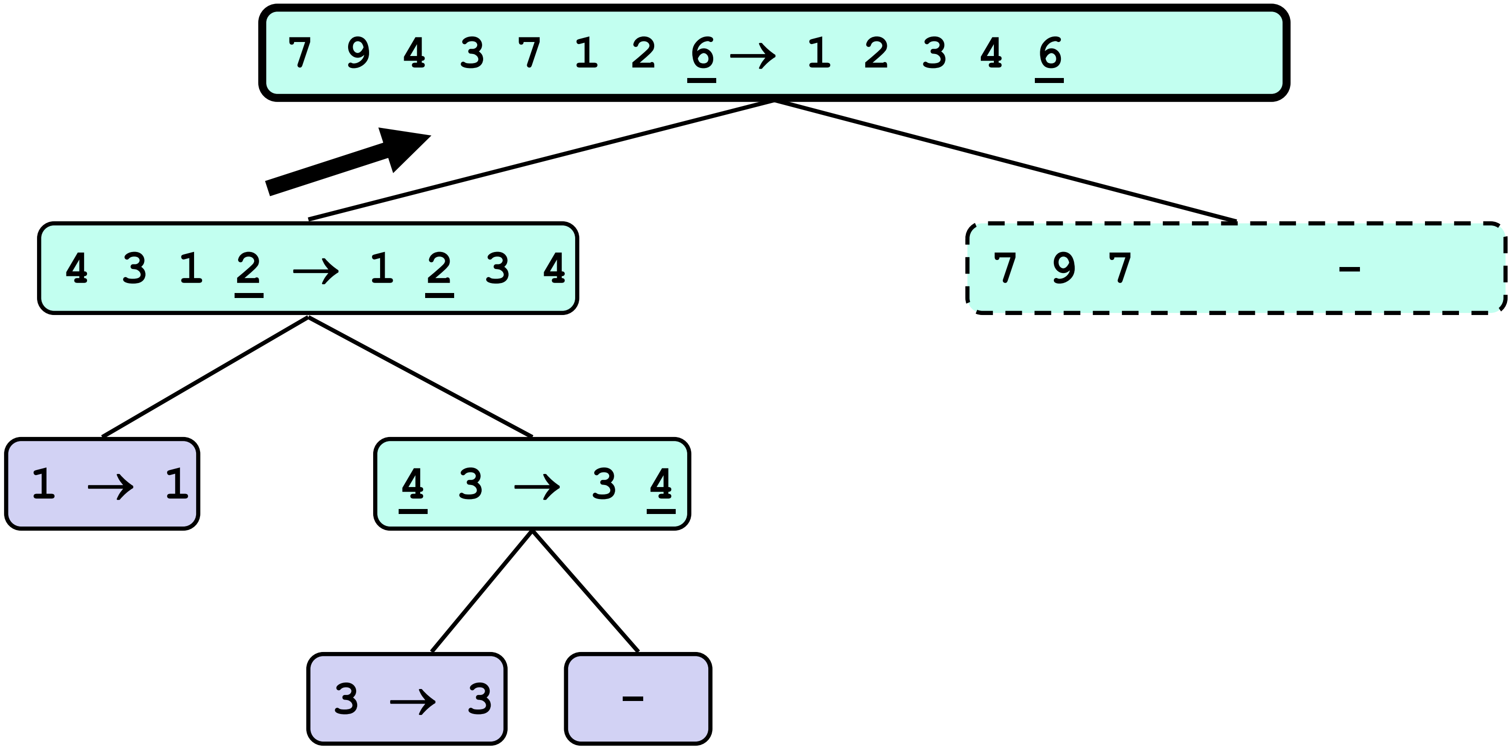
Example: Quicksort Execution

- Return from sub-list of larger values



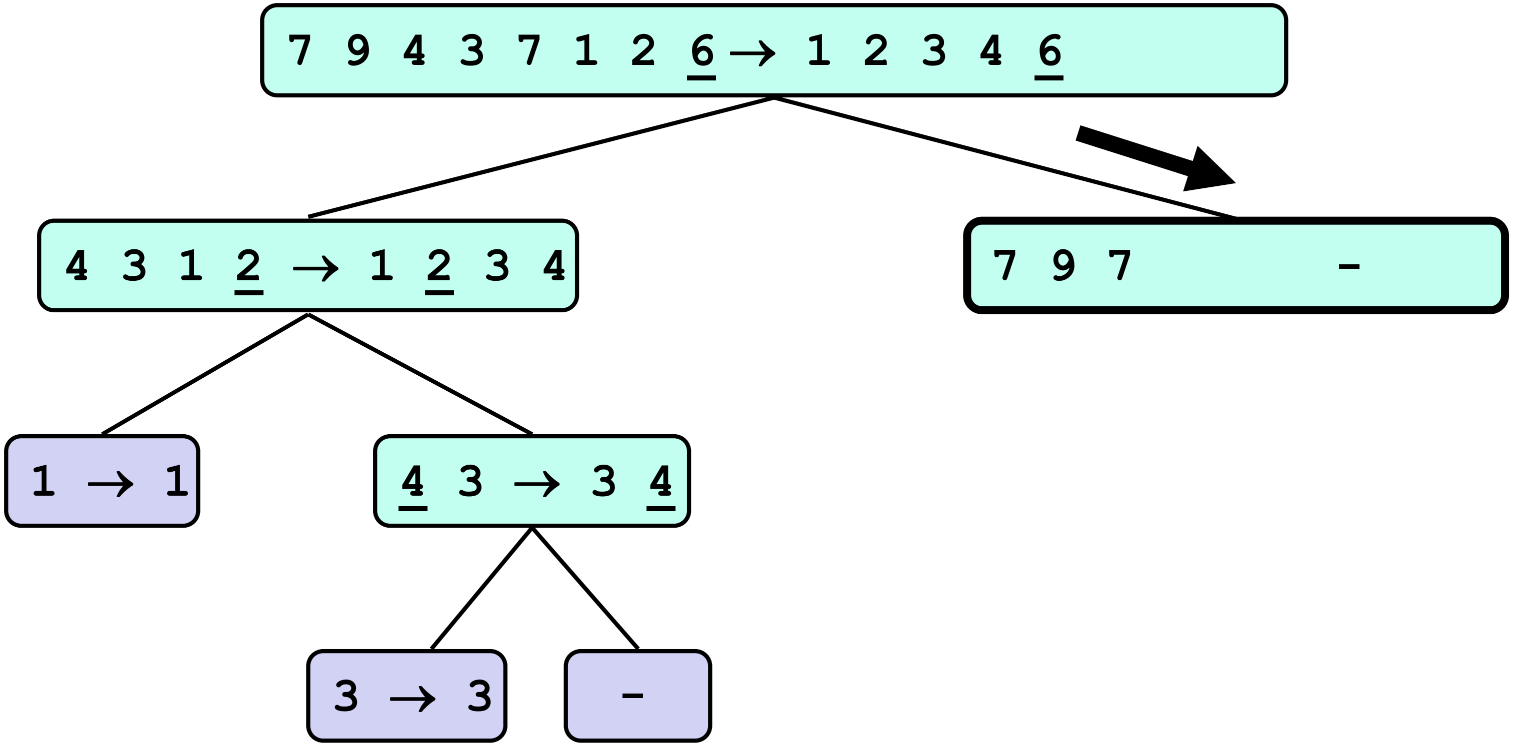
Example: Quicksort Execution

- Return from sub-list of smaller values



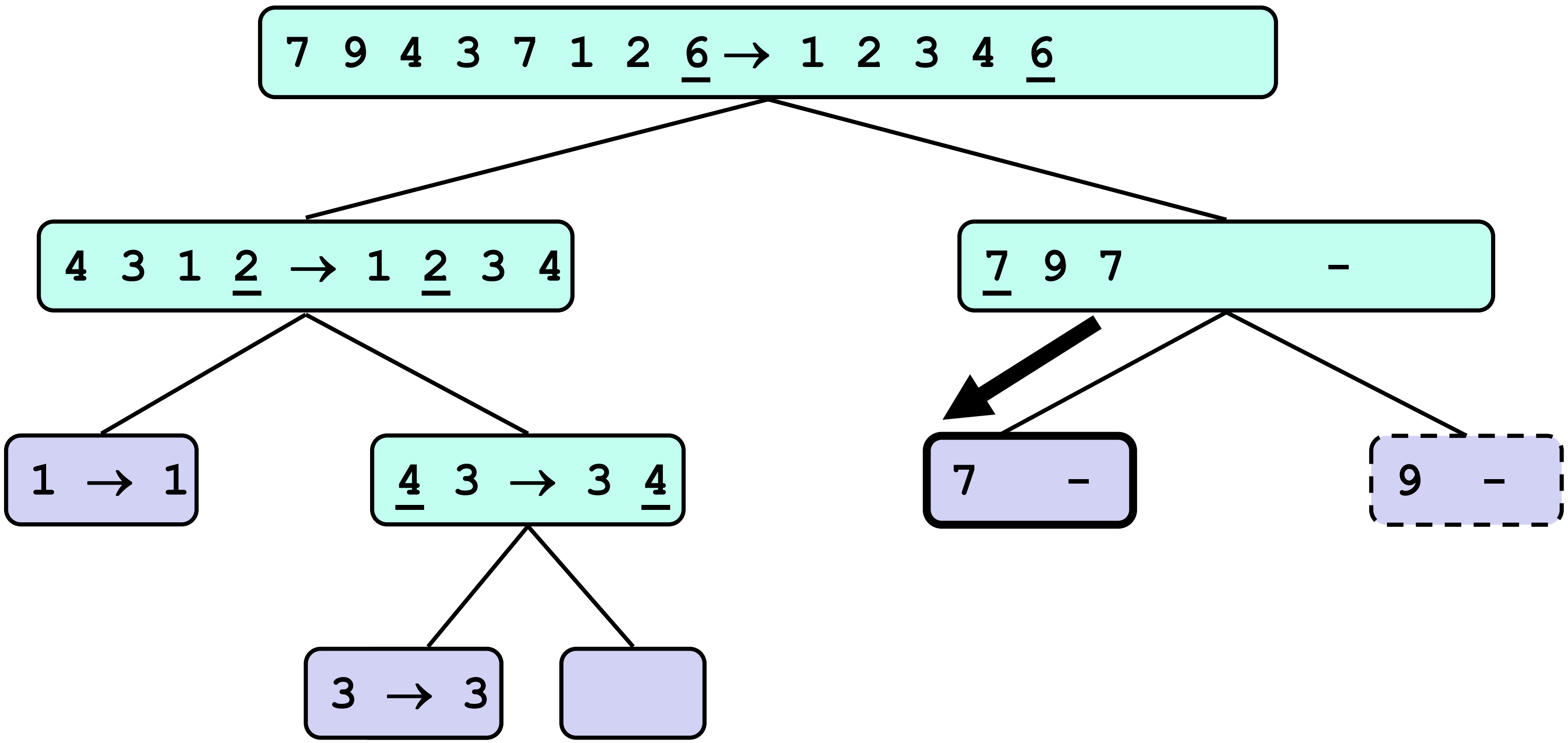
Example: Quicksort Execution

- Recursive call on sub-list of larger values



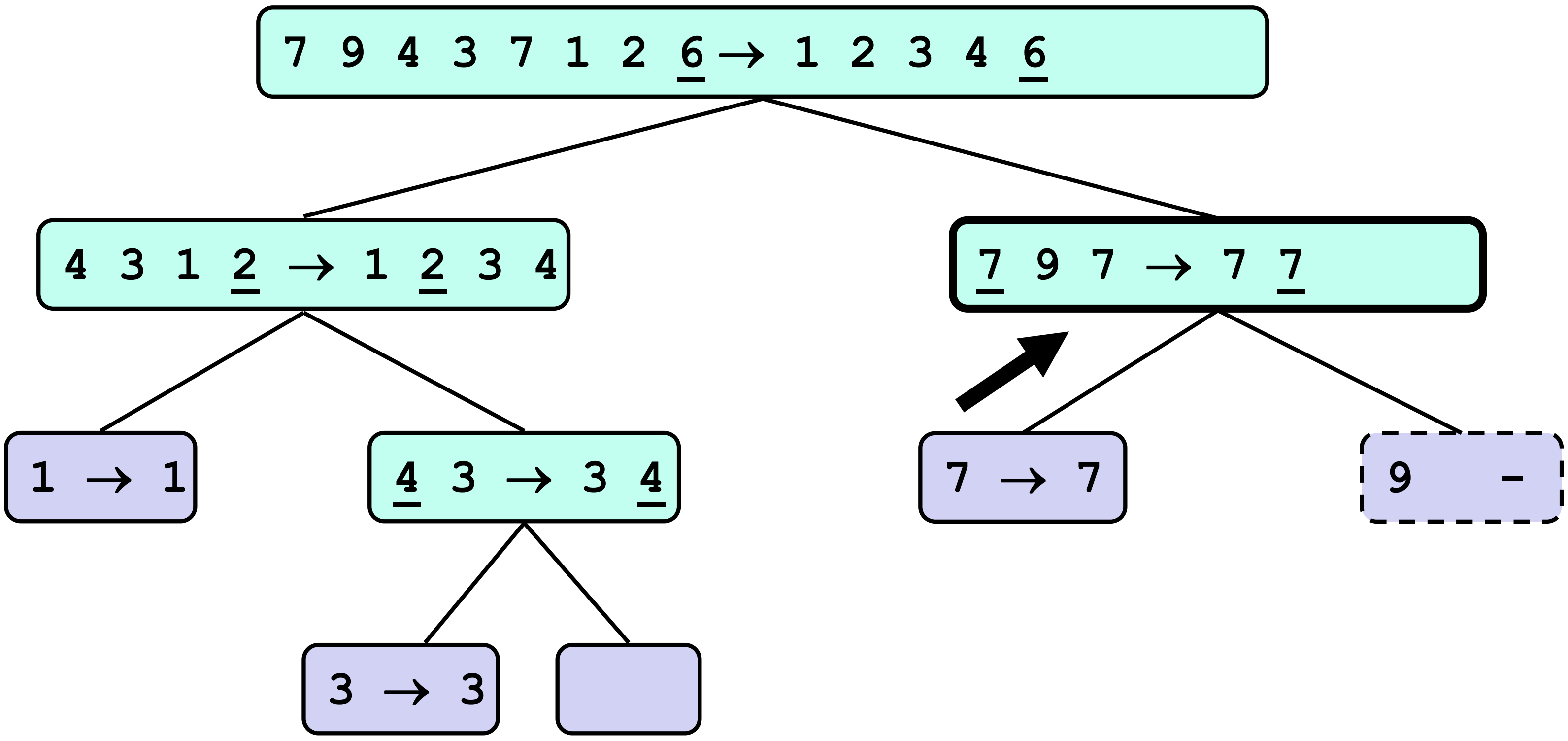
Example: Quicksort Execution

- Partition around **7**
- Recursive call on sub-list of smaller values



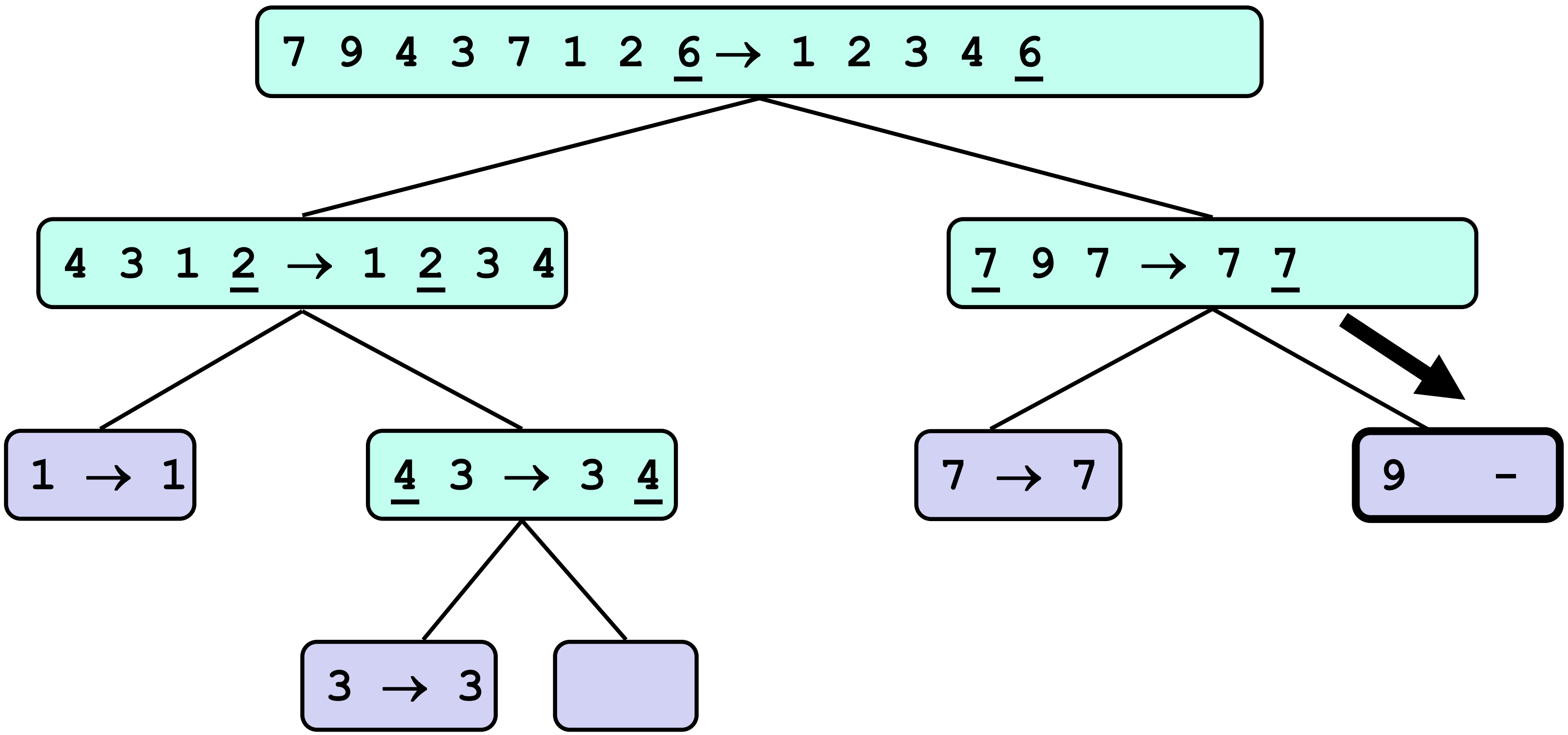
Example: Quicksort Execution

- Base case
- Return from sub-list of smaller values



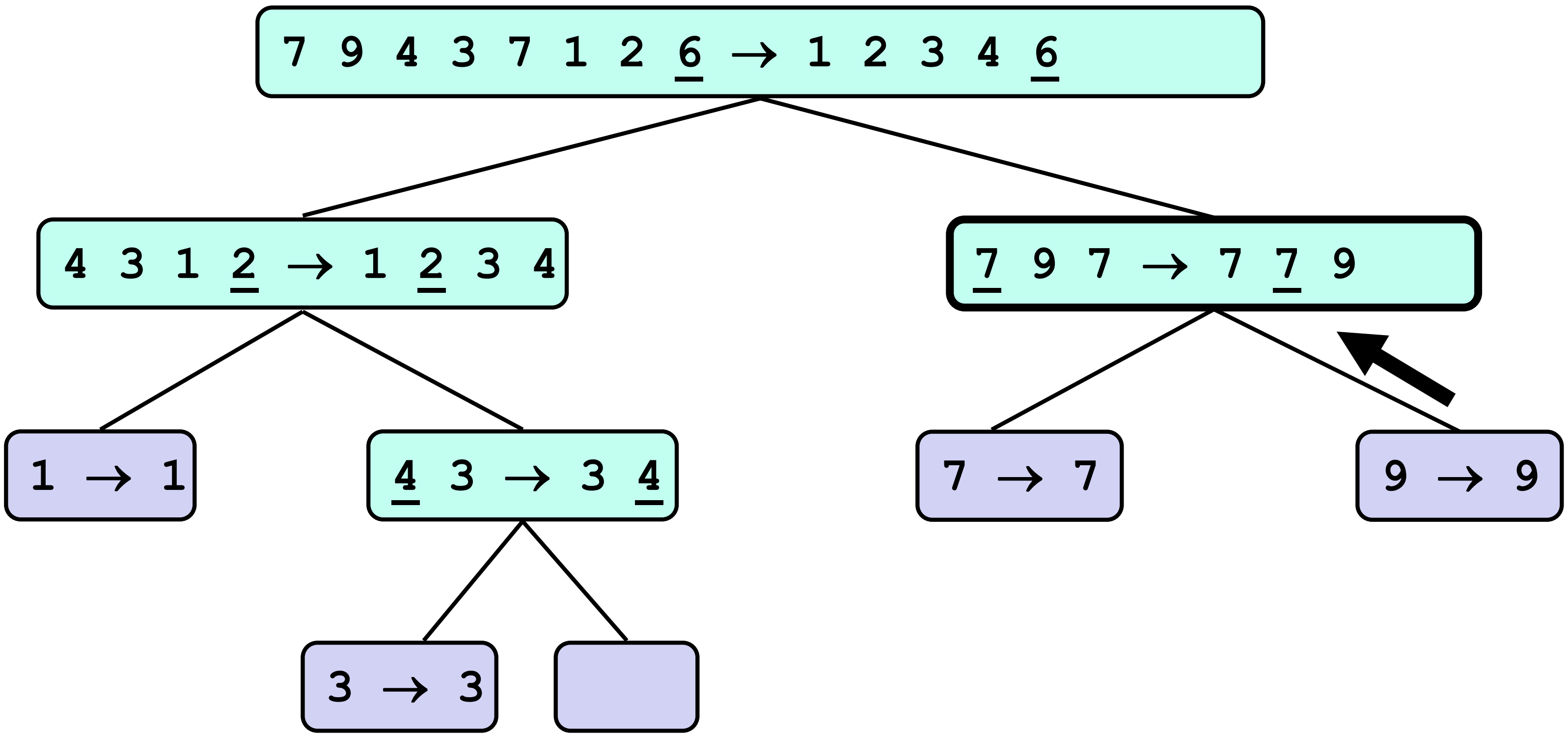
Example: Quicksort Execution

- Recursive call on sub-list of larger values



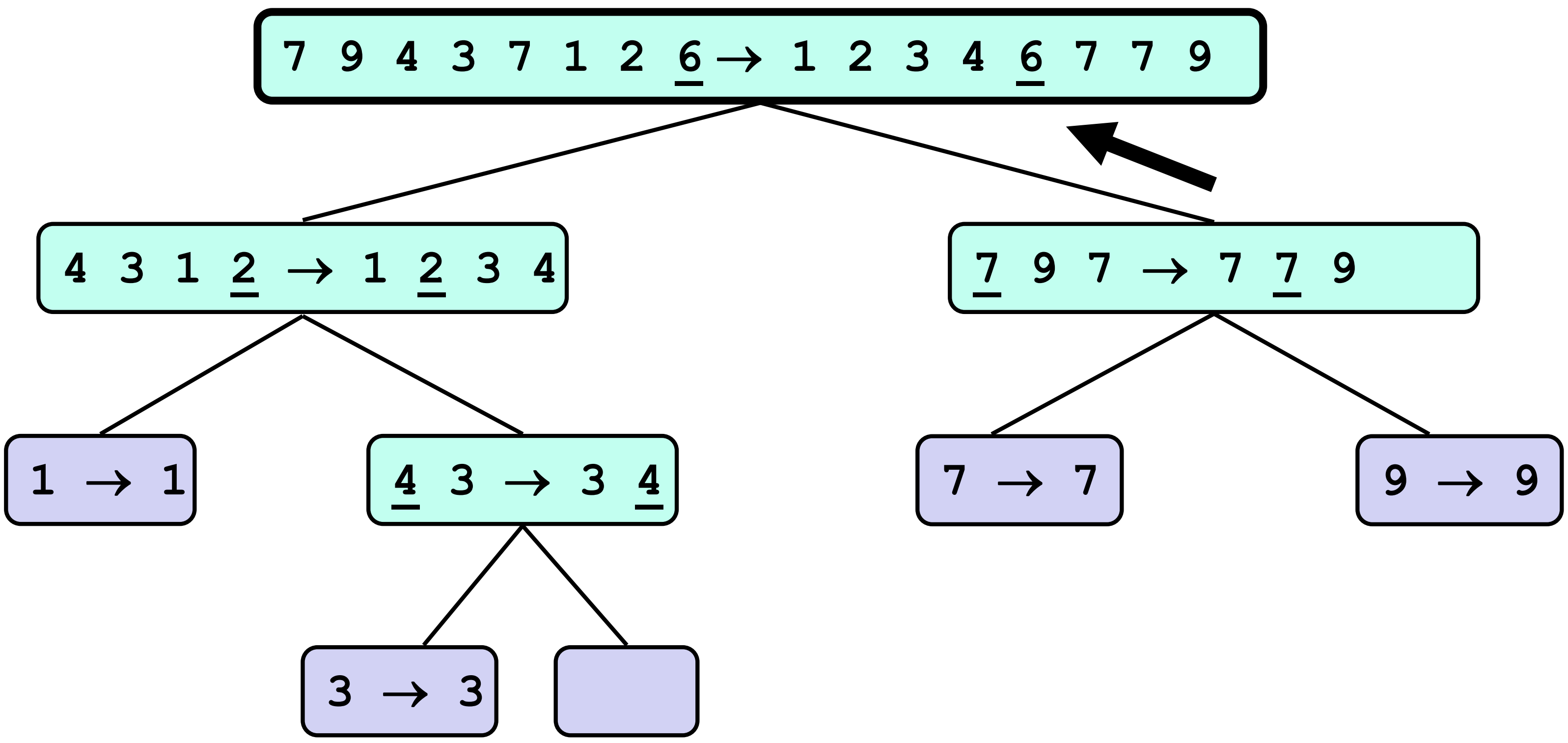
Example: Quicksort Execution

- Base case
- Return from sub-list of larger values



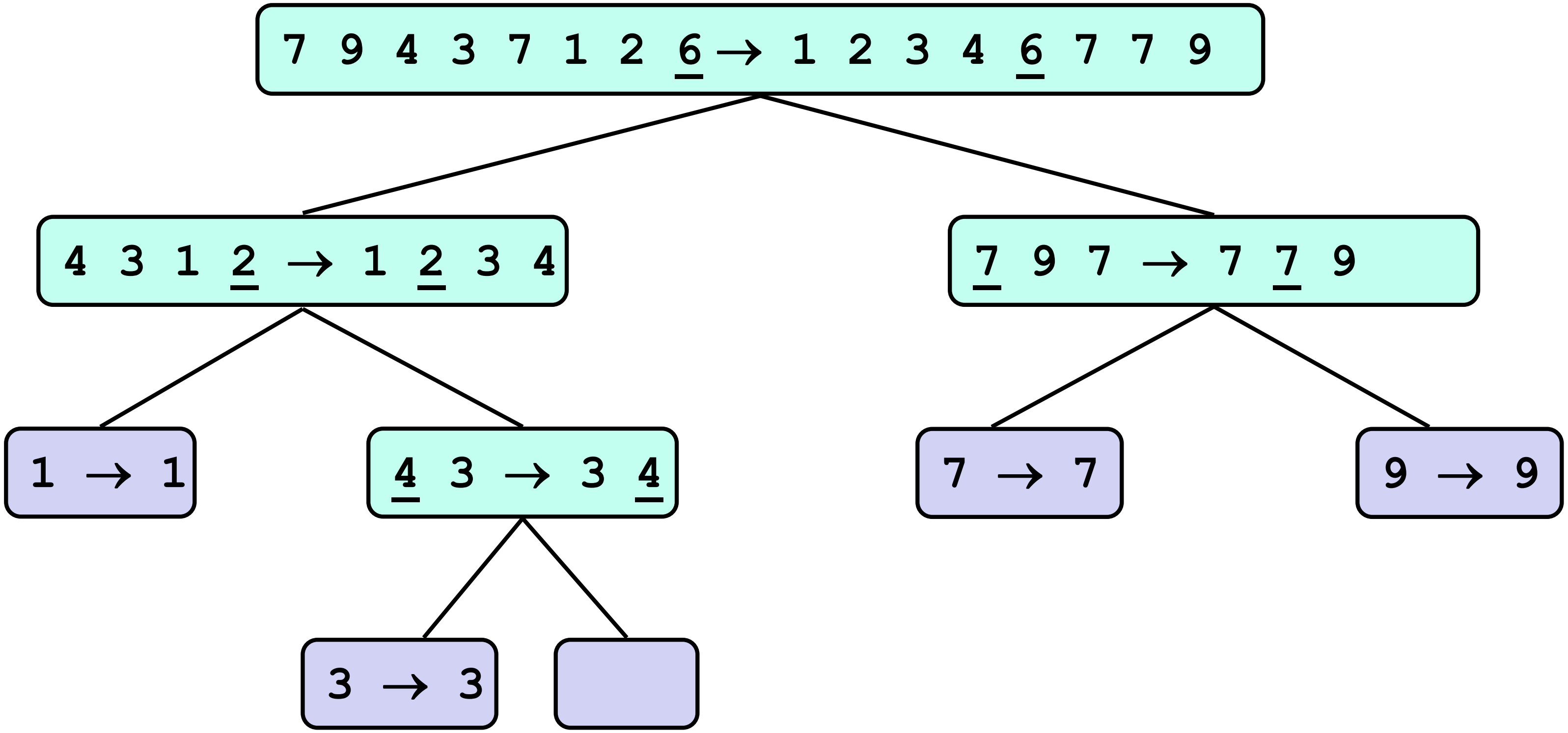
Example: Quicksort Execution

- Return from sub-list of larger values



Example: Quicksort Execution

- Done



Quicksort Algorithm

```
// A: array of values
// p: beginning index of sub-list being sorted
// r: ending index of sub-list being sorted
// Initial call: Quicksort(A, 1, A.length)
```

$\text{QUICKSORT}(A, p, r)$

if $p < r$

  then $q \leftarrow \text{PARTITION}(A, p, r)$

    $\text{QUICKSORT}(A, p, q-1)$

    $\text{QUICKSORT}(A, q+1, r)$

Quicksort Partition Algorithm

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
  do if $A[j] \leq x$
    then $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

Example: Quicksort Partition

`Partition(A, 1, A.length)`

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
  do if $A[j] \leq x$
    then $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

```
x = A[7]
i = p - 1 (0)
j = 1
A[1] ≤ 6
i = 0 + 1
swap A[1], A[1]
```

1	2	3	4	5	6	7
2	8	7	1	3	5	6

1	2	3	4	5	6	7
2	8	7	1	3	5	6

Example: Quicksort Partition

`Partition(A, 1, A.length)`

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
  do if $A[j] \leq x$
    then $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

`j = 2`
`A[2] > 6`
`no change`

1	2	3	4	5	6	7
2	8	7	1	3	5	6

Example: Quicksort Partition

```
Partition(A, 1, A.length)
```

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
    do if $A[j] \leq x$
        then $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

```
j = 3
A[3] > 6
no change
```

1	2	3	4	5	6	7
2	8	7	1	3	5	6

Example: Quicksort Partition

`Partition(A, 1, A.length)`

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
  do if $A[j] \leq x$
    then $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

```
j = 4
A[4] ≤ 6
i = 1 + 1
swap A[2], A[4]
```

1	2	3	4	5	6	7
2	8	7	1	3	5	6

1	2	3	4	5	6	7
2	1	7	8	3	5	6

Example: Quicksort Partition

```
Partition(A, 1, A.length)
```

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
  do if $A[j] \leq x$
    then $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

```
j = 5
A[5] ≤ 6
i = 2 + 1
swap A[3], A[5]
```

1	2	3	4	5	6	7
2	1	7	8	3	5	6

1	2	3	4	5	6	7
2	1	3	8	7	5	6

Example: Quicksort Partition

`Partition(A, 1, A.length)`

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
  do if $A[j] \leq x$
    then $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i+1] \leftrightarrow A[r]$
return $i + 1$

```
j = 6
A[6] ≤ 6
i = 3 + 1
swap A[4], A[6]
```

1	2	3	4	5	6	7
2	1	3	8	7	5	6

1	2	3	4	5	6	7
2	1	3	**5**	7	**8**	6

Example: Quicksort Partition

```
Partition(A, 1, A.length)
```

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
  do if $A[j] \leq x$
    then $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

```
j = 7
for loop done

swap A[5], A[7]
return 5
```

1	2	3	4	5	6	7
2	1	3	5	7	8	6

1	2	3	4	5	6	7
2	1	3	5	**6**	8	**7**

Example: Quicksort Partition

`Partition(A, 1, A.length)`

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
  do if $A[j] \leq x$
    then $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$

Partition complete

1	2	3	4	5	6	7
2	1	3	5	6	8	7

Runtime Analysis: Best Case

- What is best case running time?
  - Assume keys are random, uniformly distributed.
  - Recursion:
    1. Partition splits list in two sub-lists of size (`n/2`)
    2. Quicksort each sub-list
  - Depth of recursion tree? `O(log n)`
  - Number of accesses in partition? `O(n)`
- Best case running time: `O(n*log n)`

Runtime Analysis: Worst Case

- What is worst case running time?
  - List already sorted
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(n)$
  - Number of accesses per partition? $O(n)$
- Worst case running time: $O(n^2)$

Average Case for Quicksort

- If the list is already sorted, Quicksort is terrible: $O(n^2)$
  - It is possible to construct other bad cases
- However, Quicksort is *usually* $O(n*\log n)$
  - The constants are so good, Quicksort is generally the fastest known algorithm
  - Most real-world sorting is done by Quicksort

Tweaking Quicksort

- Almost anything you can try to "improve" Quicksort will actually slow it down
- One *good* tweak is to switch to a different sorting method when the subarrays get small (say, 10 or 12)
  - Quicksort has too much overhead for small array sizes
- For large arrays, it *might* be a good idea to check beforehand if the array is already sorted
  - But there is a better tweak than this

Picking a Better Pivot

- Before, we picked the *first* element of the sub-list to use as a pivot
  - If the array is already sorted, this results in $O(n^2)$ behavior
  - It's no better if we pick the *last* element
- We could do an *optimal* quicksort if we always picked a pivot value that exactly cuts the array in half
  - Such a value is called a *median*: half of the values in the list are larger, half are smaller
  - The easiest way to find the median is to *sort* the list and pick the value in the middle (!)

Median of Three

- Obviously, it doesn't make sense to sort the list in order to find the median
- Instead, compare just *three* elements of sub-list: first, last, and middle
  – Take the *median* (middle value) of these three as pivot
- If rearrange (sort) these three numbers so that the smallest is in the first position, the largest in the last position, and the other in the middle
  – Simplifies and speeds up the partition loop

Summary of Sorting Algorithms

Algorithm	Time	Notes
Selection Sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
Insertion Sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
Quicksort	$O(n*\log n)$ (expected)	▪ in-place, randomized ▪ fastest (good for large inputs)
Heapsort	$O(n*\log n)$	▪ in-place ▪ fast (good for large inputs)
Mergesort	$O(n*\log n)$	▪ sequential data access ▪ fast (good for huge inputs)

Quicksort Algorithm

```
public void quicksort(Comparable list[], int lo, int hi)
{
        if(lo < hi)
        {
                int p = partition(list, lo, hi);
                quicksort(list, lo, p - 1);
                quicksort(list, p + 1, hi);
        }
}

public static void swap(Object a[], int index1, int index2)
{
        Object tmp = a[index1];
        a[index1] = a[index2];
        a[index2] = tmp;
}
```

Quicksort Partition Algorithm

```
public int partition(Comparable list[], int lo, int hi)
{
        // pivot at start position
        int pivot = list[lo];
        // keep track of last value <= pivot
        int i = lo;
        // move smaller values down in list
        for(int j = lo + 1; j <= hi)
        {
                if(list[j] <= pivot)
                {
                        i++;
                        swap(list[i], list[j]);
                }
        }
        // put pivot in correct position in partitioned list
        swap(list[i], list[lo]);
        // index of pivot value
        return i;
}
```

Comparison of Various Sorts

Num Items	Selection	Insertion	Quicksort
1000	16	5	0
2000	59	49	6
4000	271	175	5
8000	1056	686	0
16000	4203	2754	11
32000	16852	11039	45
64000	expected?	expected?	68
128000	expected?	expected?	158
256000	expected?	expected?	335
512000	expected?	expected?	722
1024000	expected?	expected?	1550

times in milliseconds