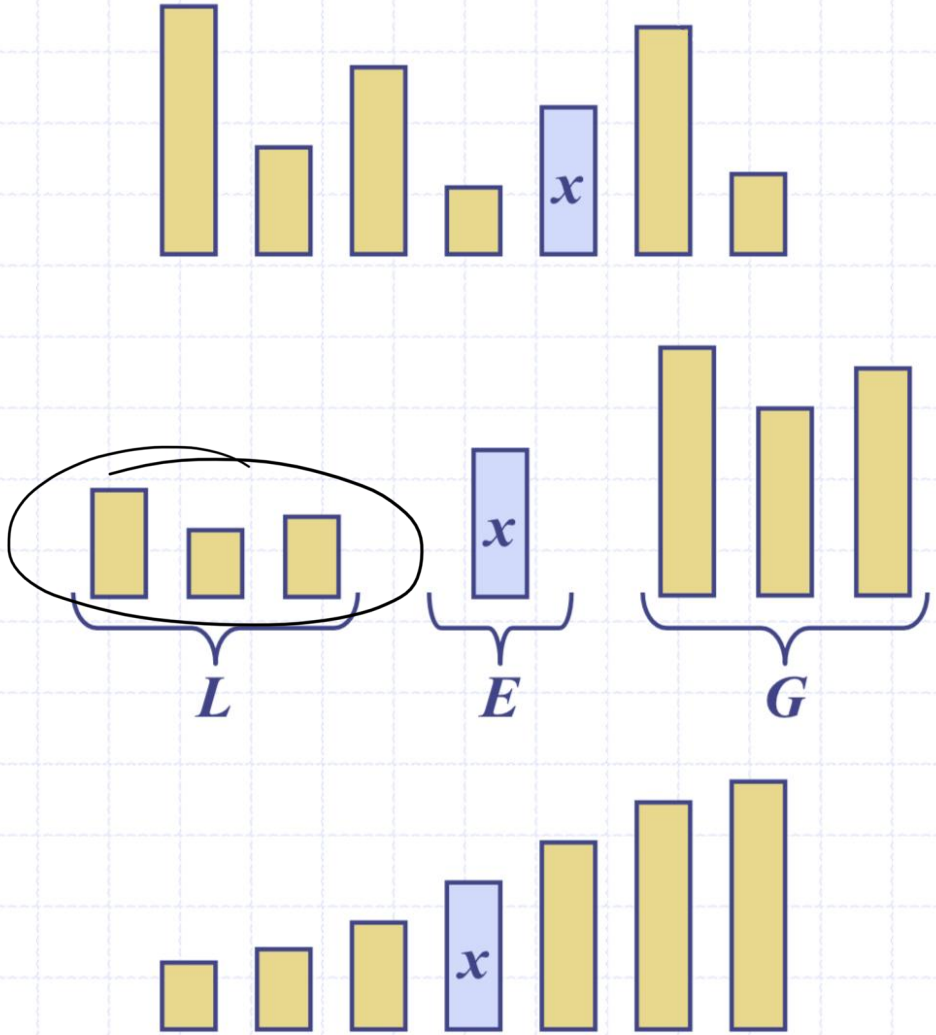
QuickSort

- QuickSort Overview
- 3-Way QuickSort
- QuickSort Analysis
- QuickSort vs. MergeSort

Quick-sort is a randomized sorting algorithm based on the divide-and-conquer paradigm:

- Divide: pick a random element x (called pivot) and partition S into
  - L elements less than x
  - E elements equal x
  - G elements greater than x
- Recur: sort L and G
- Conquer: join L, E and G

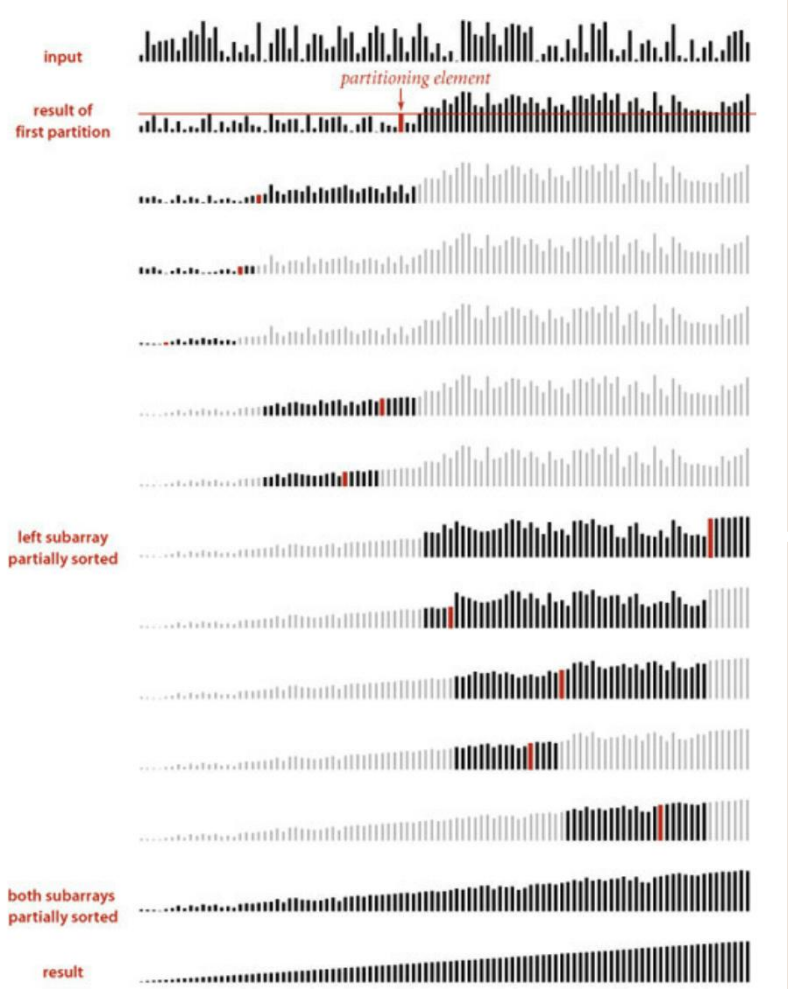


Picture from Tamassia and Goodrich

3-Way Quicksort

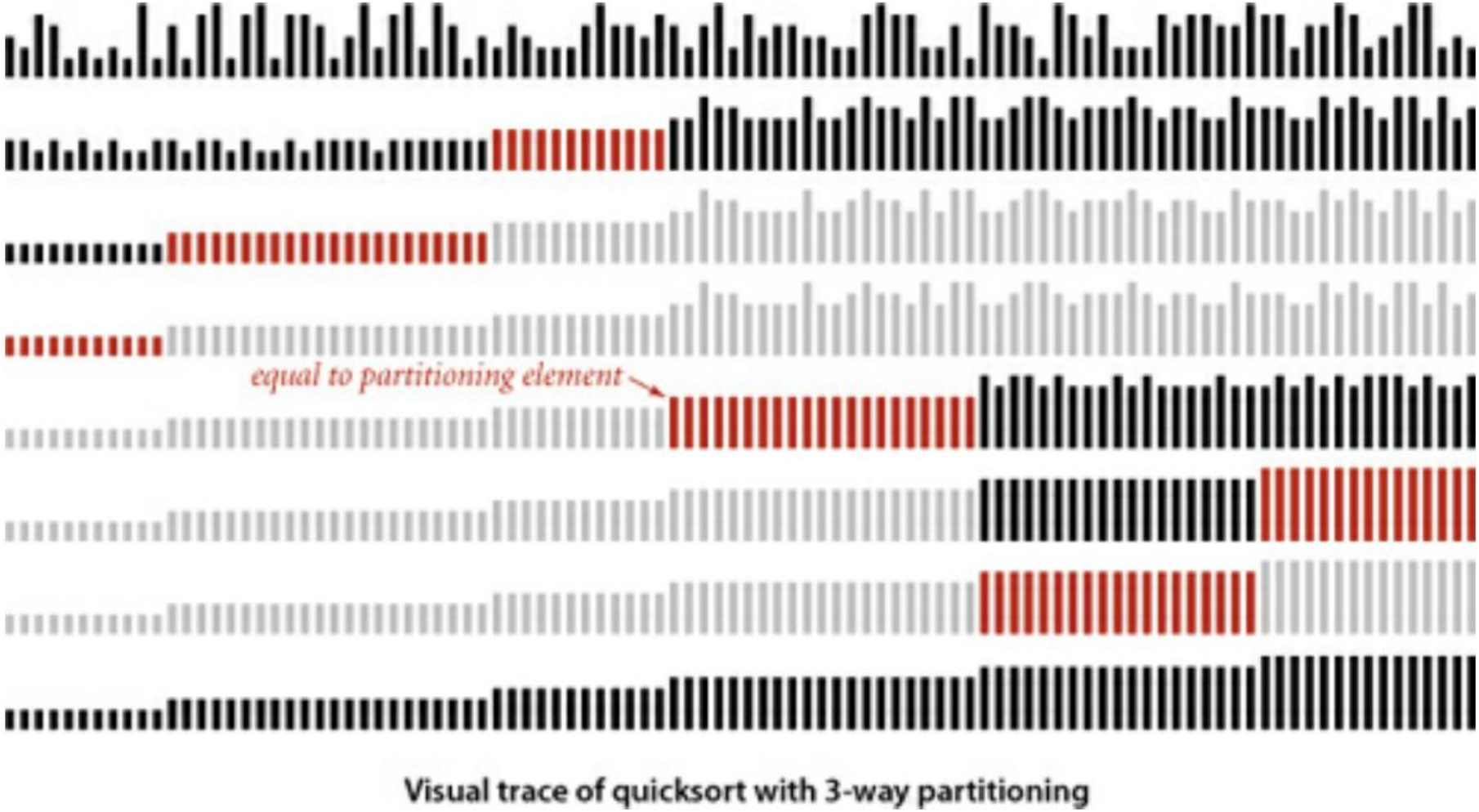
- Instead of partitioning into 2 sets each time, partition into 3 sets:
  - Less than set
  - Equal than set
  - Greater than set
- Helps slightly with many repeated keys

Picture from Tamassia and Goodrich



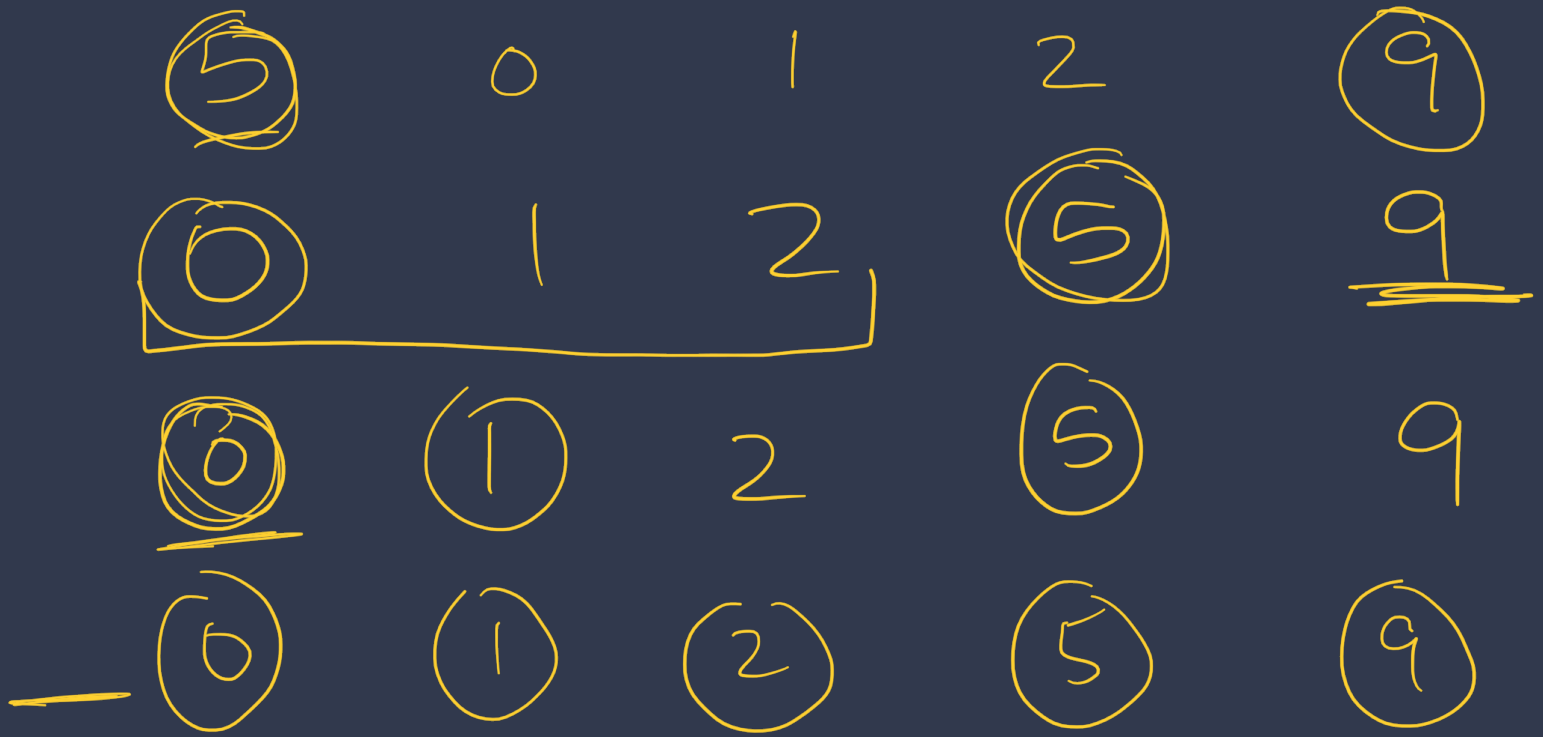
Quicksort with median-of-3 partitioning and cuto f or small subarrays

Picture from Sedgewick and Wayne



Visual trace of quicksort with 3-way partitioning

Picture from Sedgewick and Wayne


5 0 1 2 9
0 1 2 5 9
0 1 2 5 9
0 1 2 5 9

```
Algorithm quickSort
Input: an array A of size N
Output: A, sorted
pivot(A, 0, N-1)

procedure pivot (Array A, int i, int j)
      if(i >= j)
            return
      if(i < 0 || i >= A.length || j >= A.length)
            return
      int x = A[i]
      int p = i;
      int f = i + 1;
      for(int k = f; k <= j; k++)
            if(A[k] < x)
                  swap A[f] and A[k]
                  swap A[p] and A[f]
                  p++
                  f++
            else if(A[k] == x)
                  swap A[f] and A[k]
                  f++
      end for
      pivot(A, i, p-1);
      pivot(A, f, j)
end pivot
```

$$T(N) = 2T(N/2) + N$$

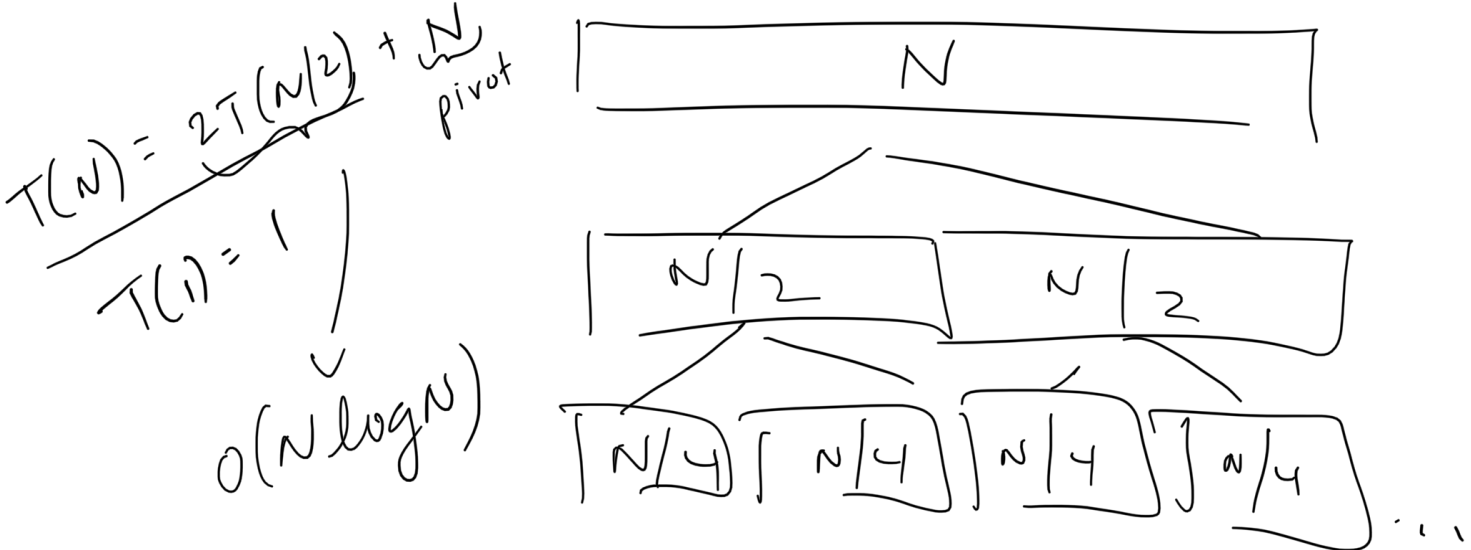
$$T(N) = T(1) + T(N-1) + N$$

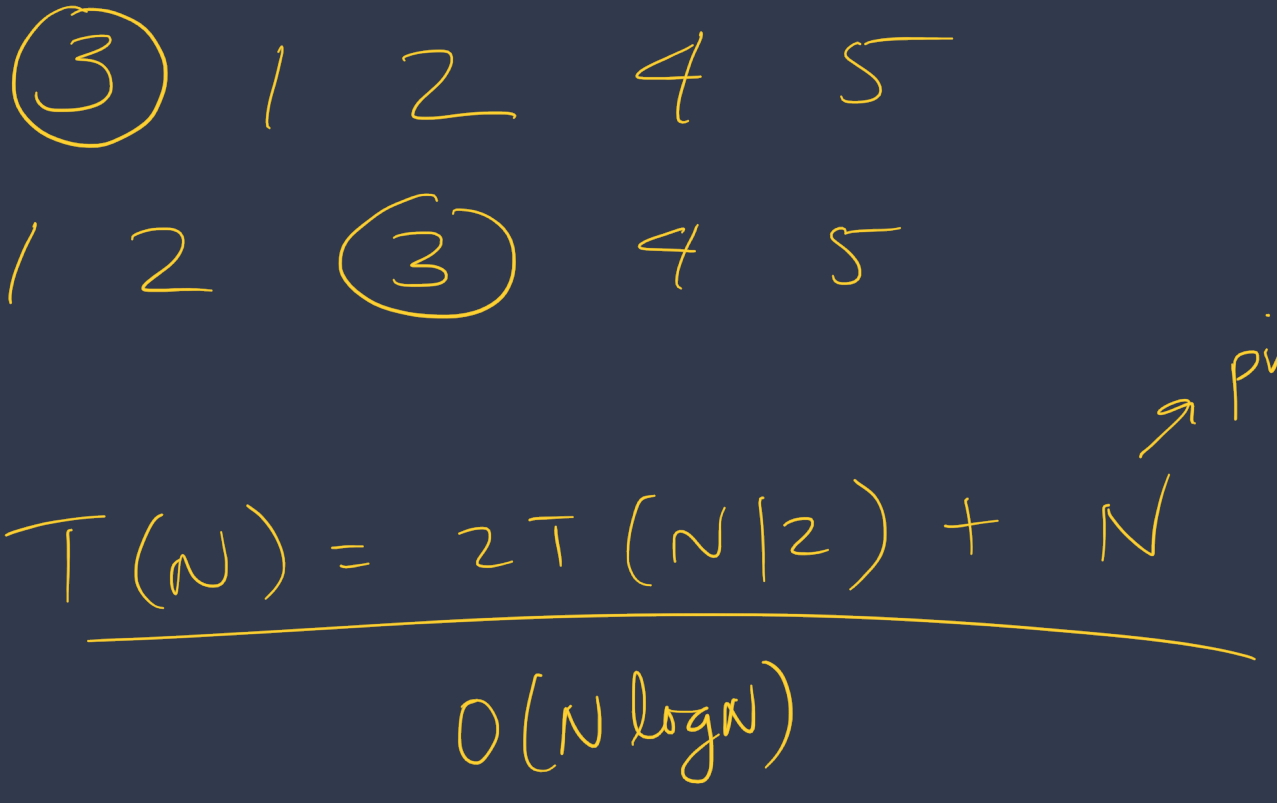
```
Algorithm quickSort
Input: an array A of size N
Output: A, sorted
pivot(A, 0, N-1)

procedure pivot (Array A, int i, int j)
      if(i >= j)
            return
      if(i < 0 || i >= A.length || j >= A.length)
            return
      int x = A[i]
      int p = i;
      int f = i + 1;
      for(int k = f; k <= j; k++)
            if(A[k] < x)
                  swap A[f] and A[k]
                  swap A[p] and A[f]
                  p++
                  f++
            else if(A[k] == x)
                  swap A[f] and A[k]
                  f++
      end for
      pivot(A, i, p-1)
      pivot(A, f, j)
end pivot
```

Best-Case Running Time

If the pivot is always the median value, the list gets divided evenly into two halves (or close to halves), which gives us the same situation as MergeSort.




3 1 2 4 5
1 2 3 4 5
T(N) = 2T(N/2) + N
pivot
O(N log N)

Worst-case Running Time

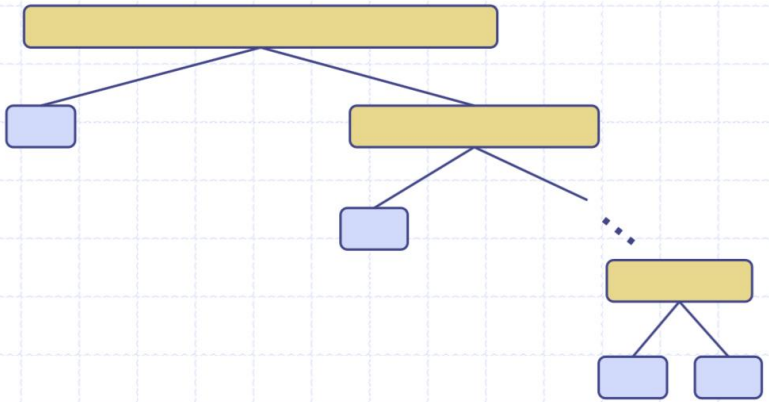
- When does the worst-case running time occur?
  - When the pivot is the unique minimum or maximum element
- In such cases, one of L and G has size $n - 1$ and the other size 0
- The running time is proportional to the sum

$$n + (n-1) + \ldots + 2 + 1$$

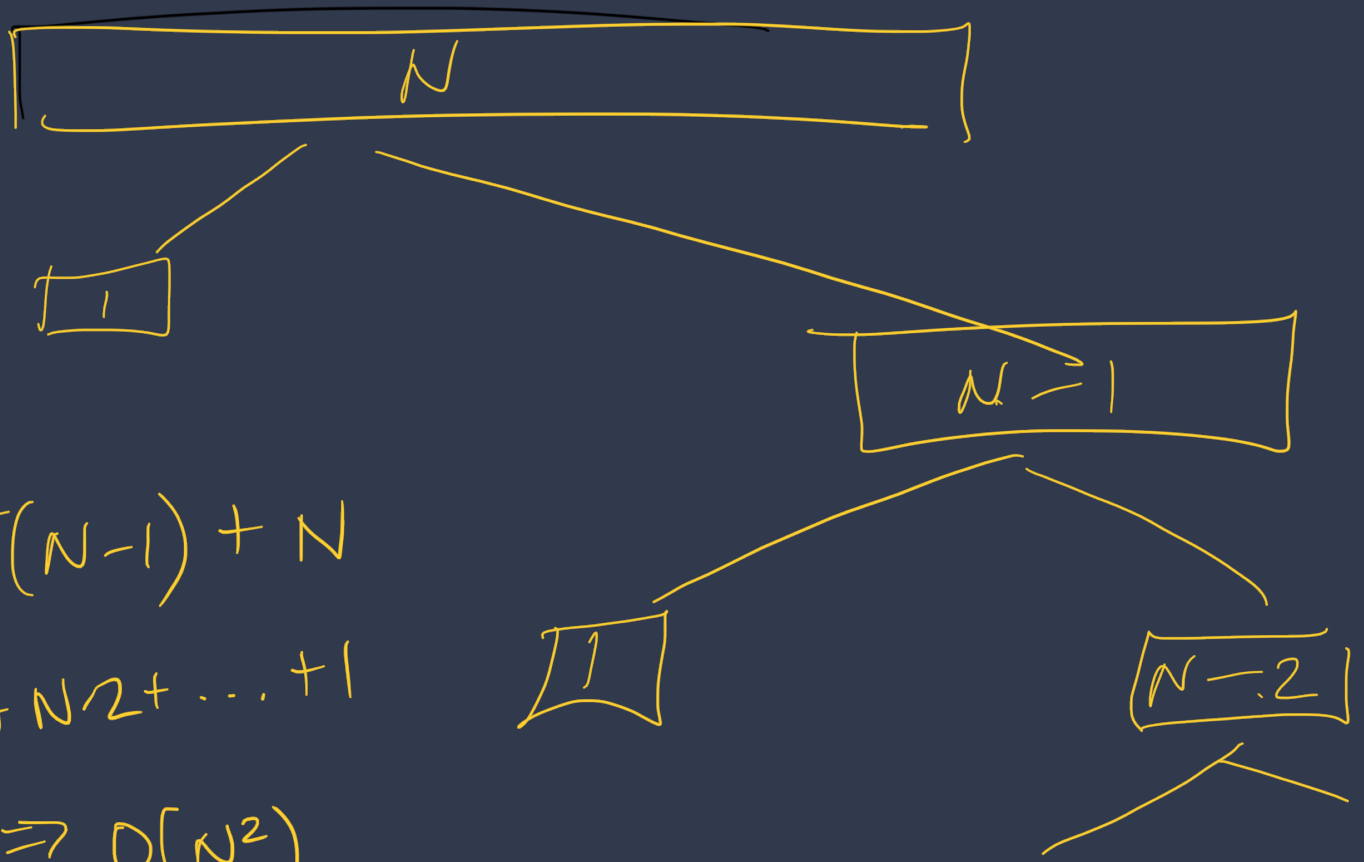
- Thus, the worst-case running time of quick-sort is
  - $O(n^2)$

$$T(N) = T(N-1) + N$$

depth	time
0	n
1	$n-1$
…	…
$n-1$	1



Picture from Tamassia and Goodrich



$T(1) = 1$

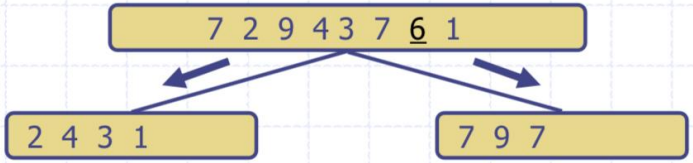
$T(N) = T(N-1) + N$

$= N + N-1 + N-2 + \dots + 1$

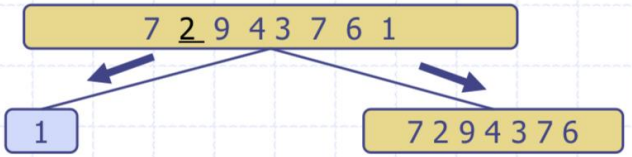
$\Rightarrow O(N^2)$

Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size s
  - **Good call:** the sizes of L and G are each less than $3s/4$
  - **Bad call:** one of L and G has size greater than $3s/4$

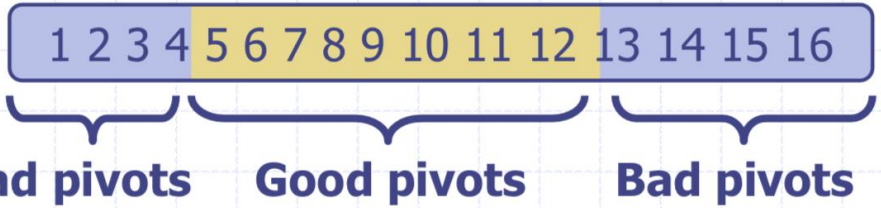


Good call



Bad call

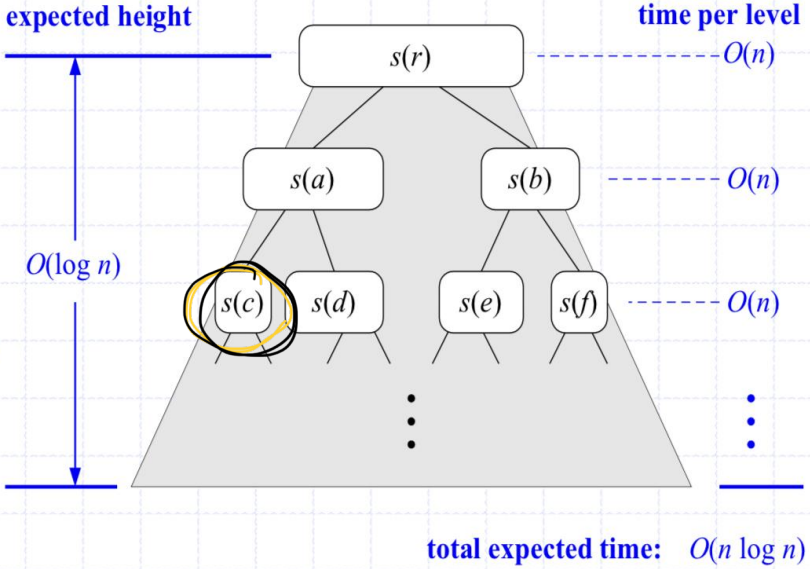
- A call is **good** with what probability?
  - 1/2 of the possible pivots cause good calls:



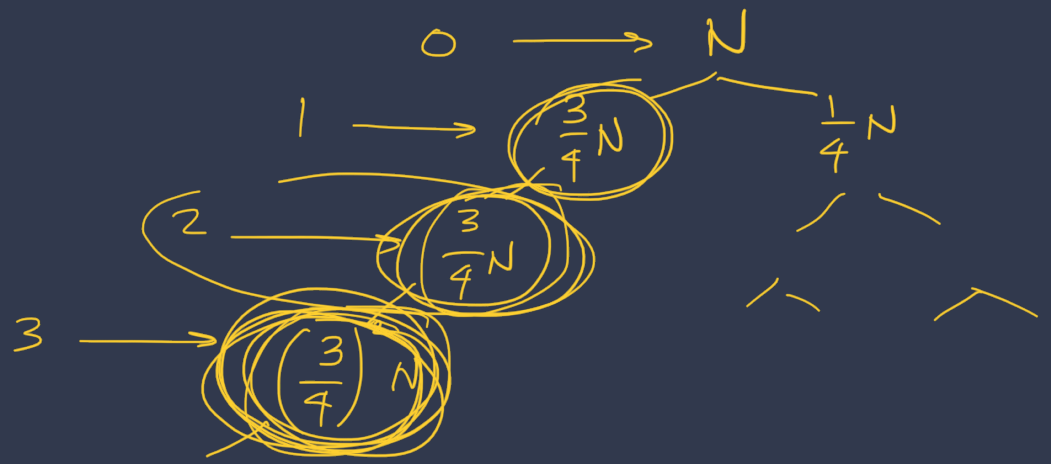
Picture from Tamassia and Goodrich

Expected Running Time, Part 2

- (Probabilistic Fact: The expected number of coin tosses required in order to get k heads is $2k$)
- For a node of depth i, we expect
  - How many of the ancestor calls to be good?
    - $i/2$ ancestors
  - The size of the input sequence for the current call is at most
    - $(3/4)^{i/2}n$
- Therefore, we have
  - For a node of depth $2\log_{4/3}n$, the expected input size is one
  - Thus, the expected height of the quick-sort tree is $O(\log n)$
- The amount or work done at the nodes of the same depth is $O(n)$
- Hence, the expected running time of quick-sort is $O(n \log n)$



Picture from Tamassia and Goodrich



average :

$\left(\frac{3}{4}\right)^0 N$ $O(N \log N)$

$\left(\frac{3}{4}\right)^1 N$

$\left(\frac{3}{4}\right)^{i/2} N$

$\frac{\left(\frac{3}{4}\right)^{i/2} N}{\left(\frac{3}{4}\right)^{i/2}} = \frac{1}{\left(\frac{3}{4}\right)^{i/2}} = \left(\frac{4}{3}\right)^{i/2}$

$\log_{4/3} N = \log_{4/3} \left(\frac{4}{3}\right)^{i/2}$

$2 \log_{4/3} N = i$