Divide and Conquer

Steps for Divide and Conquer

(1) Divide a problem into smaller subproblems.

We say P is a subproblem of Q if P = Q but P has a smaller instance size.

--- Example ---

To compute the n-th Fibonacci number Fib(n), we can reduce the computation of Fib(n) to that of Fib(n-1) and Fib(n-2).

Fib(n-1) and Fib(n-2) are called the subproblems of Fib(n).

Steps for Divide and Conquer

(2) Conquer the subproblems separately.

--- Example ---

To compute the n-th Fibonacci number Fib(n), we can reduce the computation of Fib(n) to that of Fib(n-1) and Fib(n-2).

The computation of Fib(n-1) and Fib(n-2) can be irrelevant, so one can compute their values separately.

Steps for Divide and Conquer

(3) Combine the results.

--- Example ---

To compute the n-th Fibonacci number Fib(n), we can reduce the computation of Fib(n) to that of Fib(n-1) and Fib(n-2).

The computation of Fib(n-1) and Fib(n-2) can be irrelevant, so one can compute their values separately.

Finally, we sum the return value of Fib(n-1) and Fib(n-2). The sum gives the output of Fib(n).

Quick Sort

Quick Sort

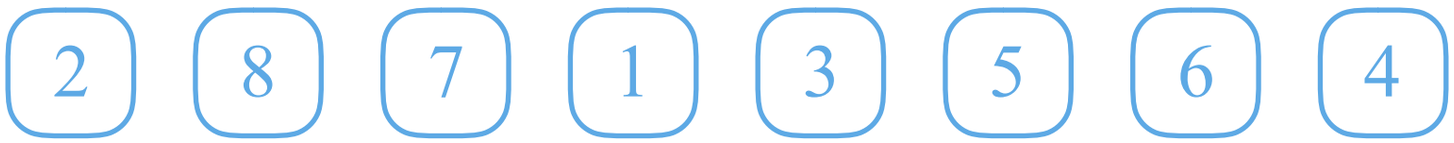
Input: an array A of n integers.

Output: the same array with the n integers ordered nondecrementally.

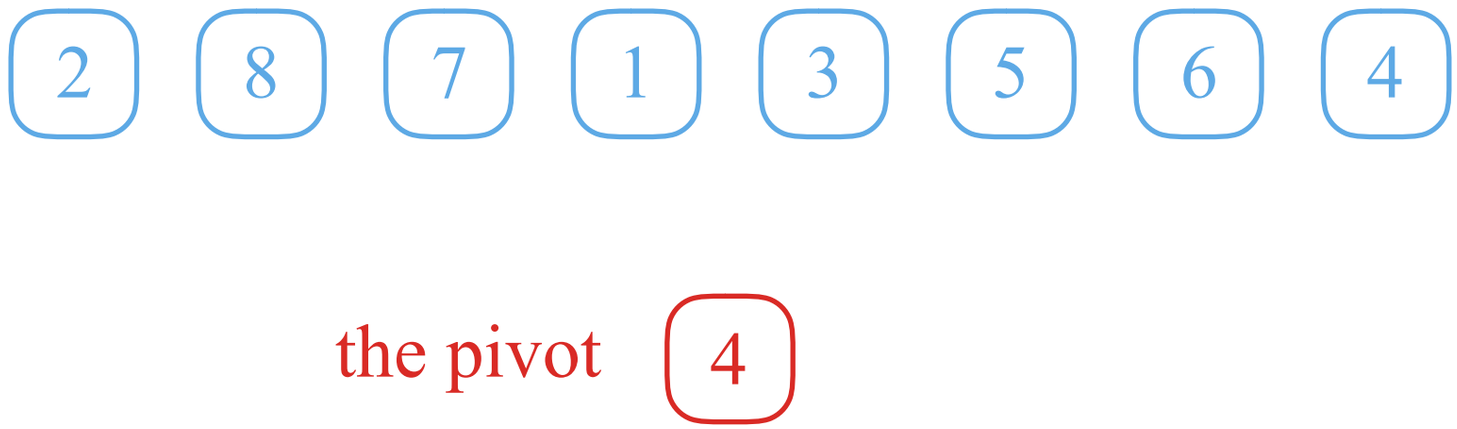
--- Pseudo Code --- // Describe an algorithm in high-level view.

```
QuickSort (A, n){
    pick an arbitrary integer k (called pivot) from A;
    partition A into two subarrays S and L so that
        (1) S contains all elements in A less than or equal to k;
        (2) L contains the rest;

    QuickSort(S, |S|); QuickSort(L, |L|);

    // no combining step is needed if S is placed at the prefix of A and
    // L is placed at the suffix of A
}
```

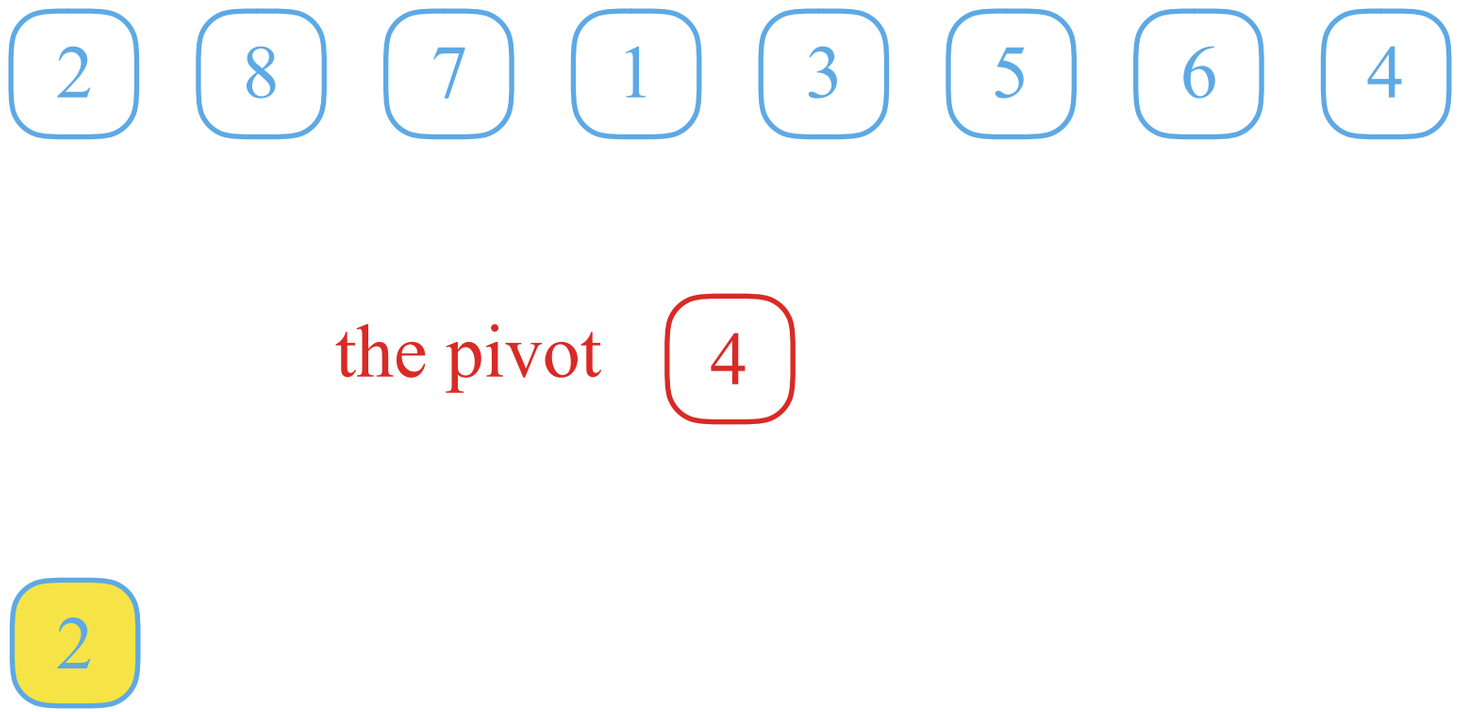
Illustration



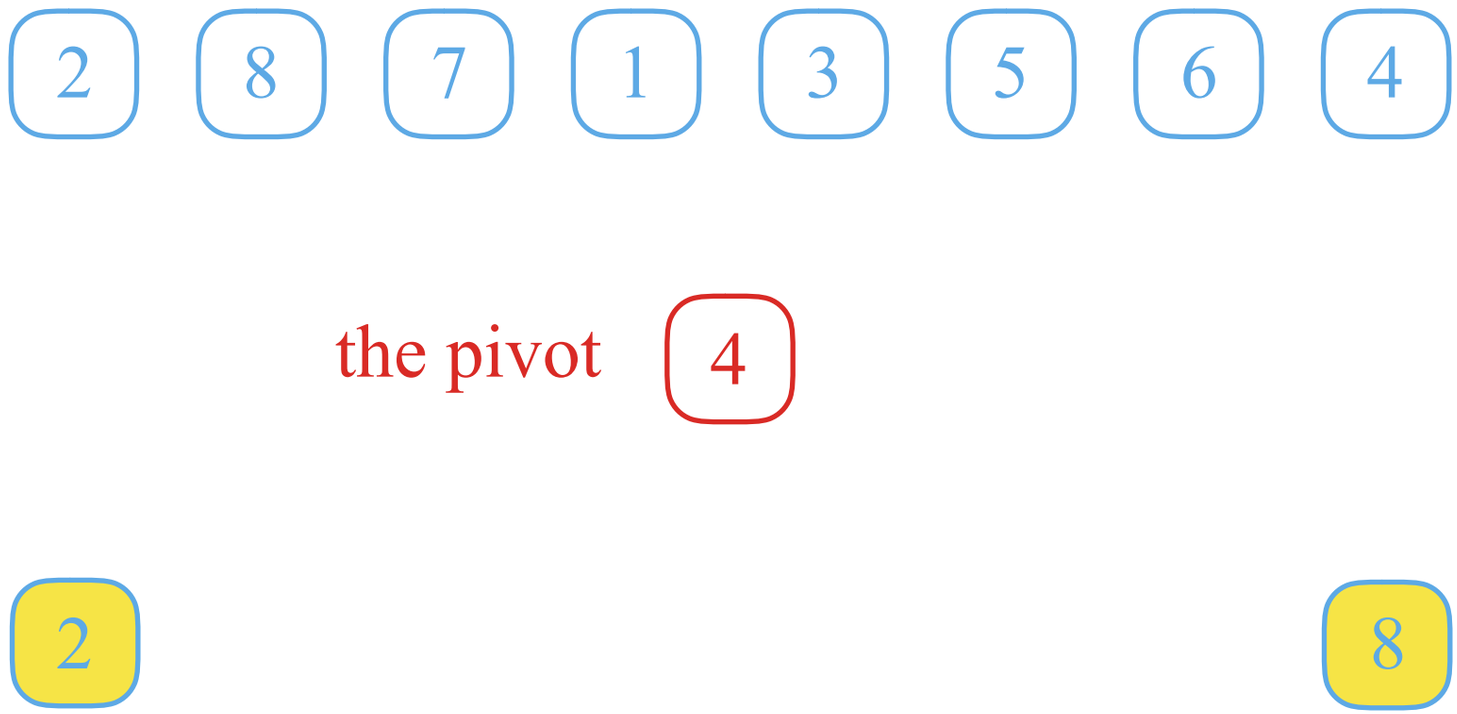
Illustration



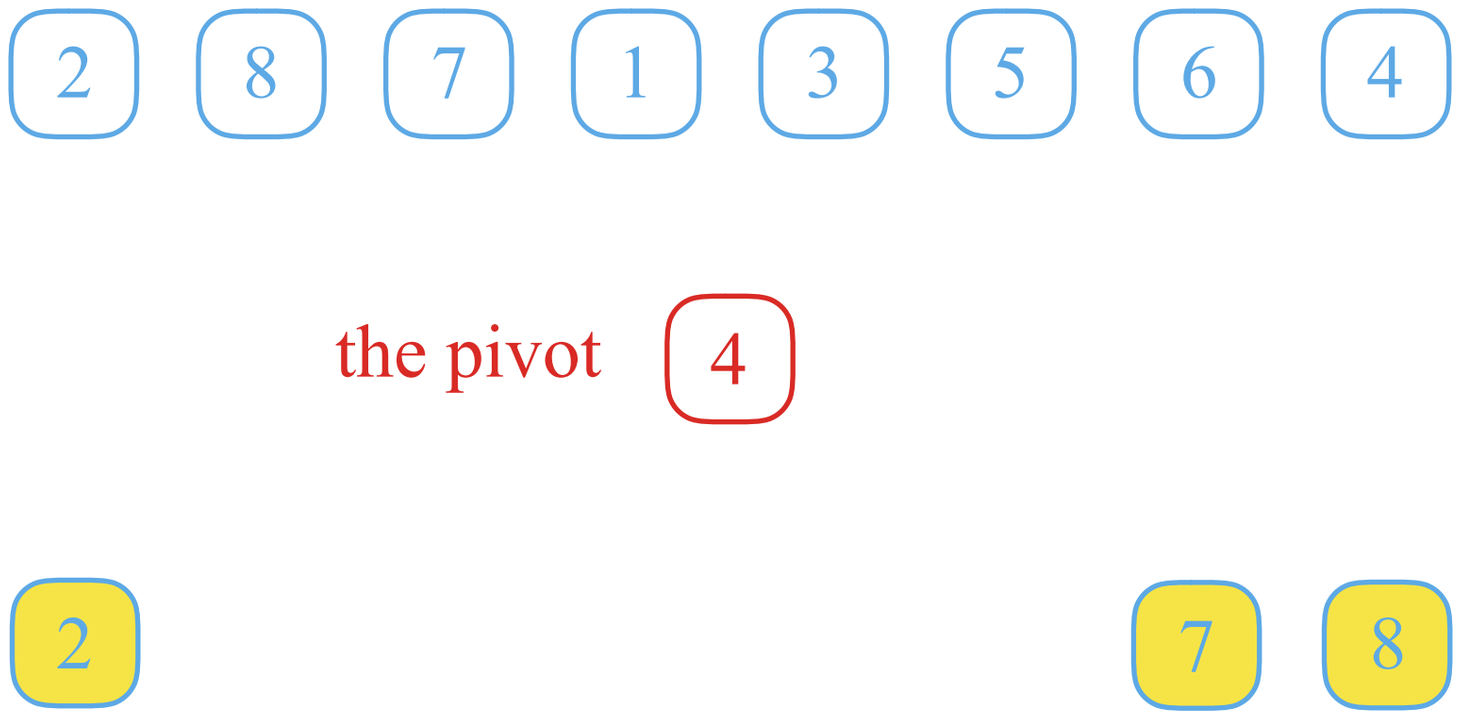
Illustration



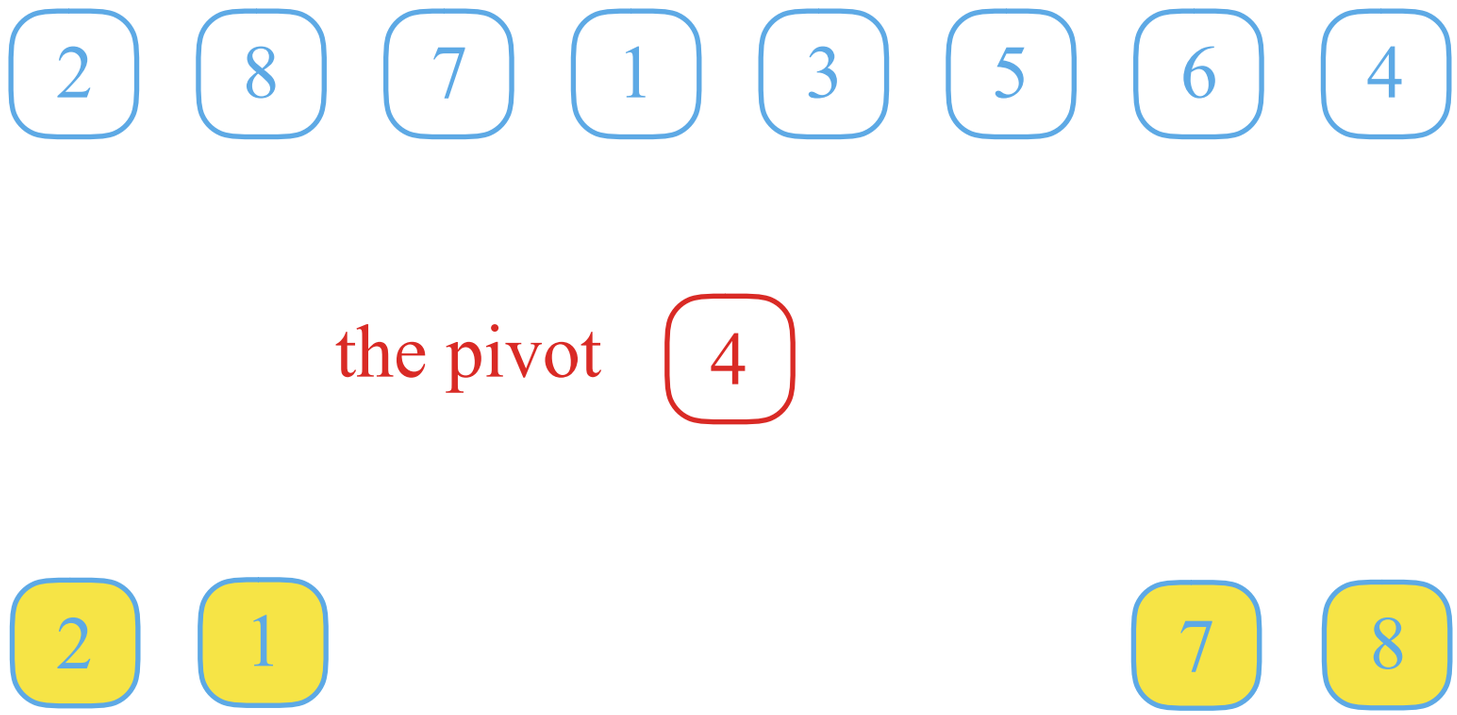
Illustration



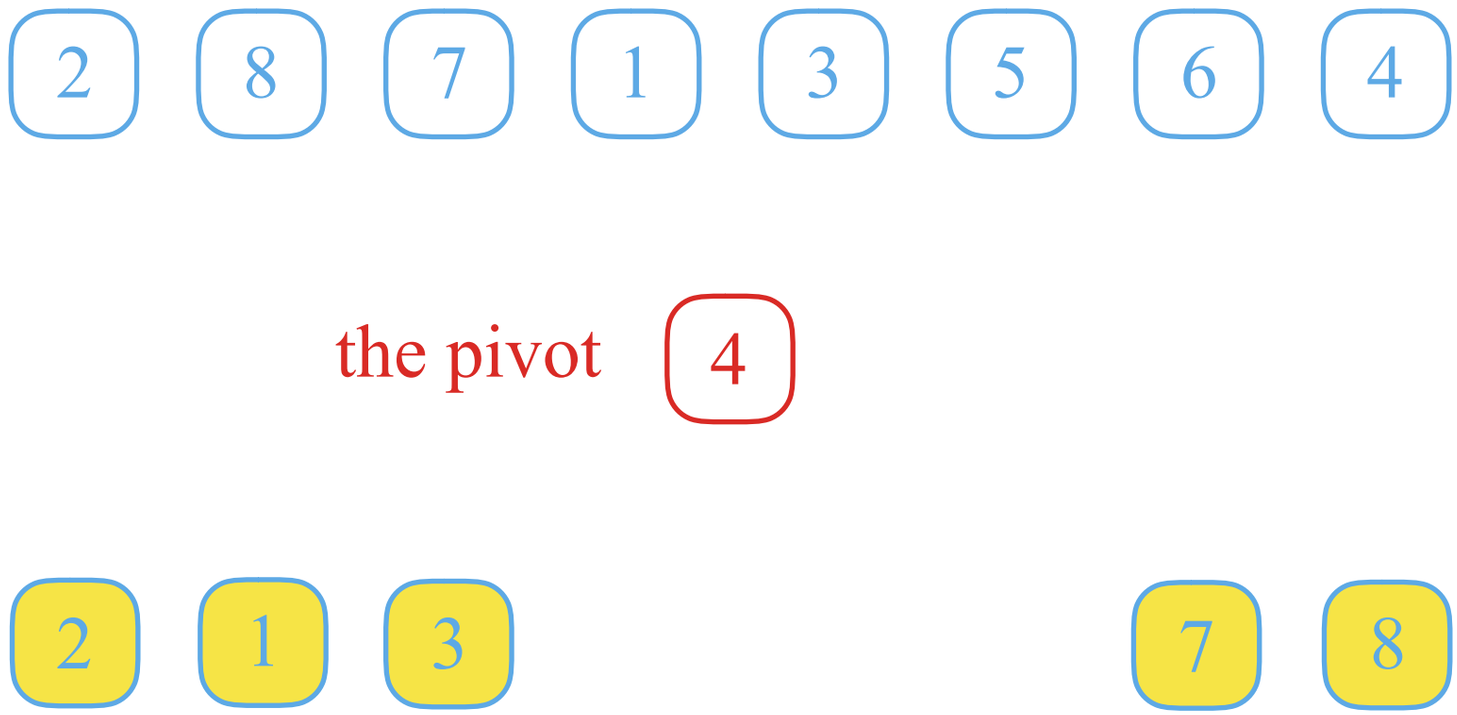
Illustration



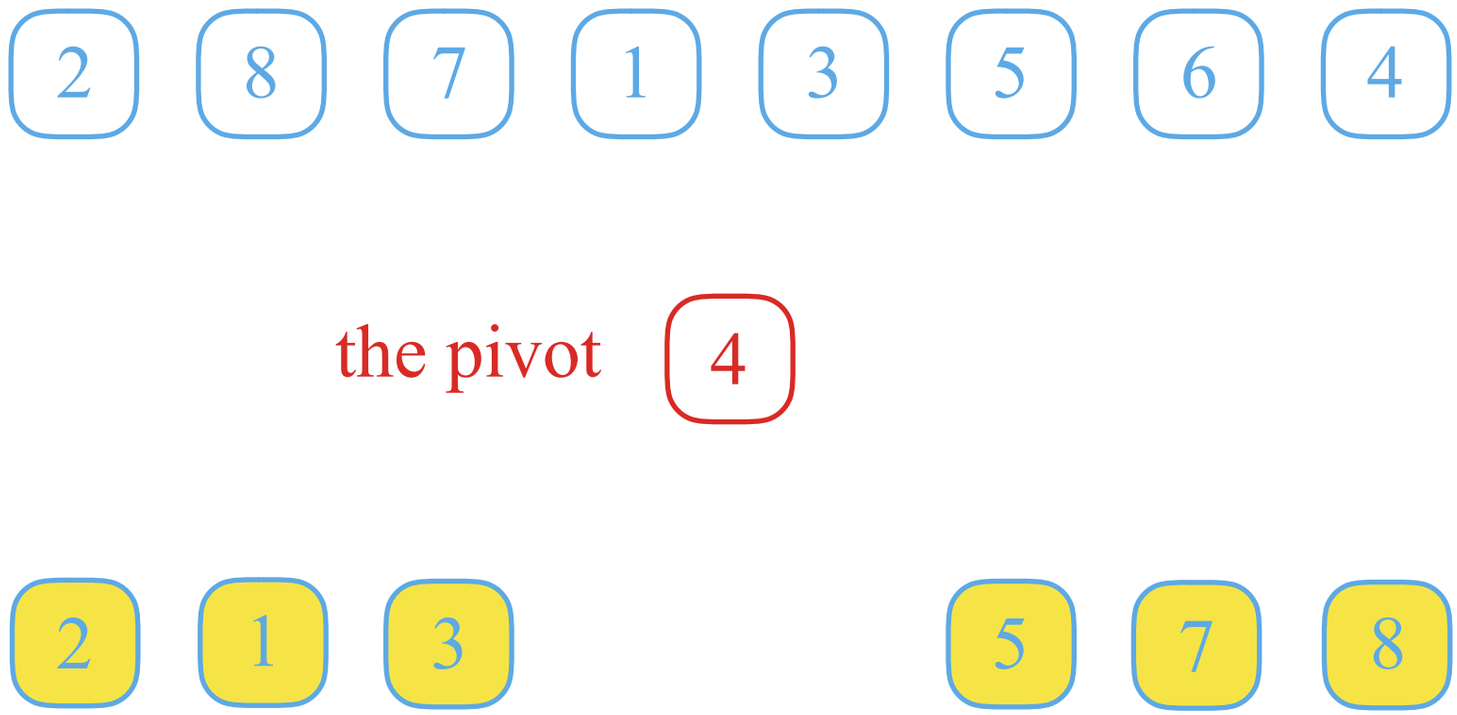
Illustration



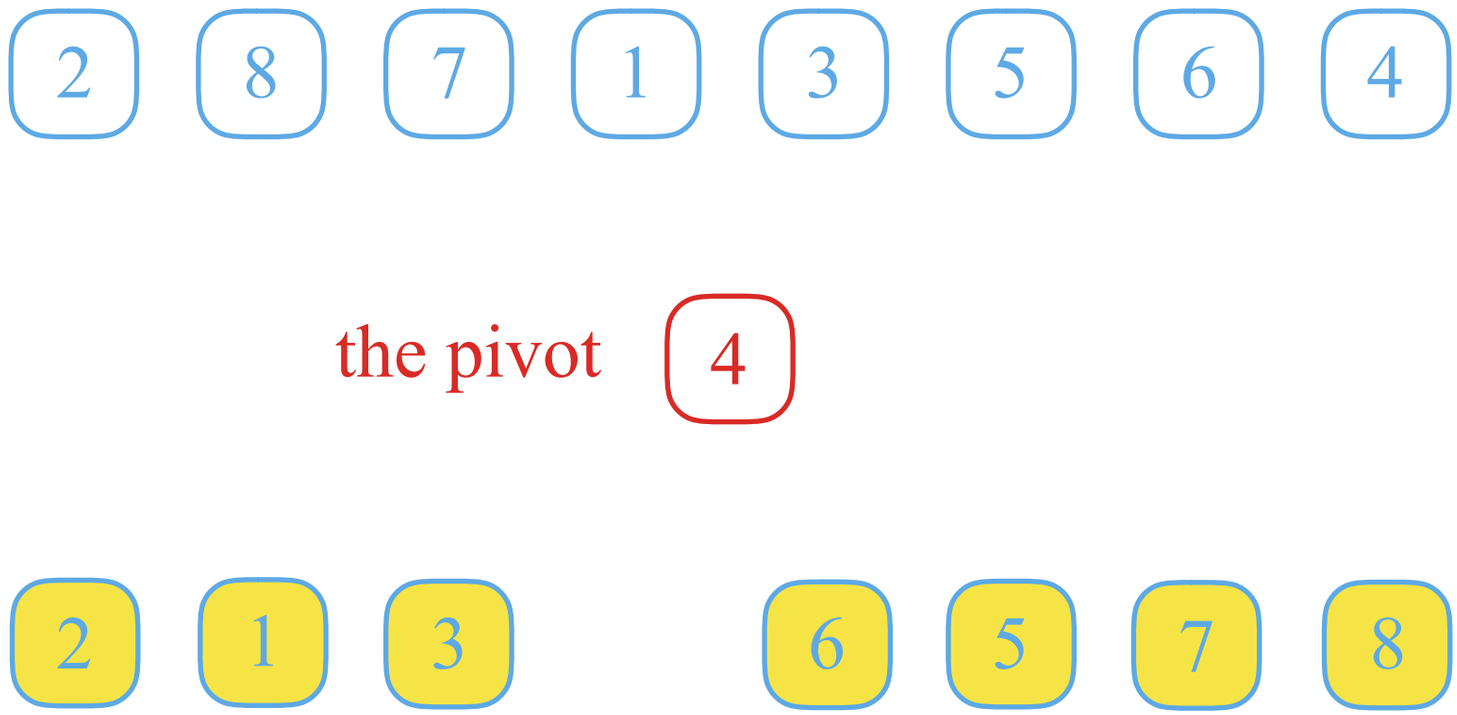
Illustration



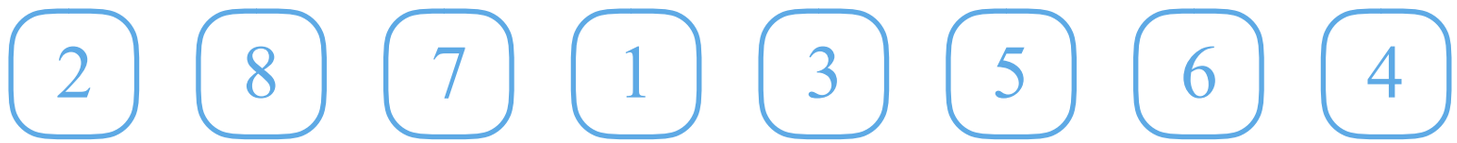
Illustration



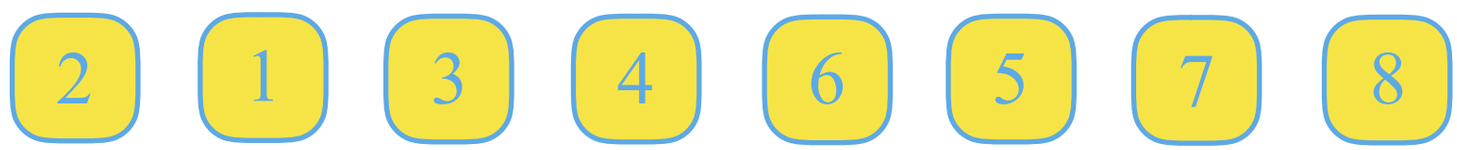
Illustration



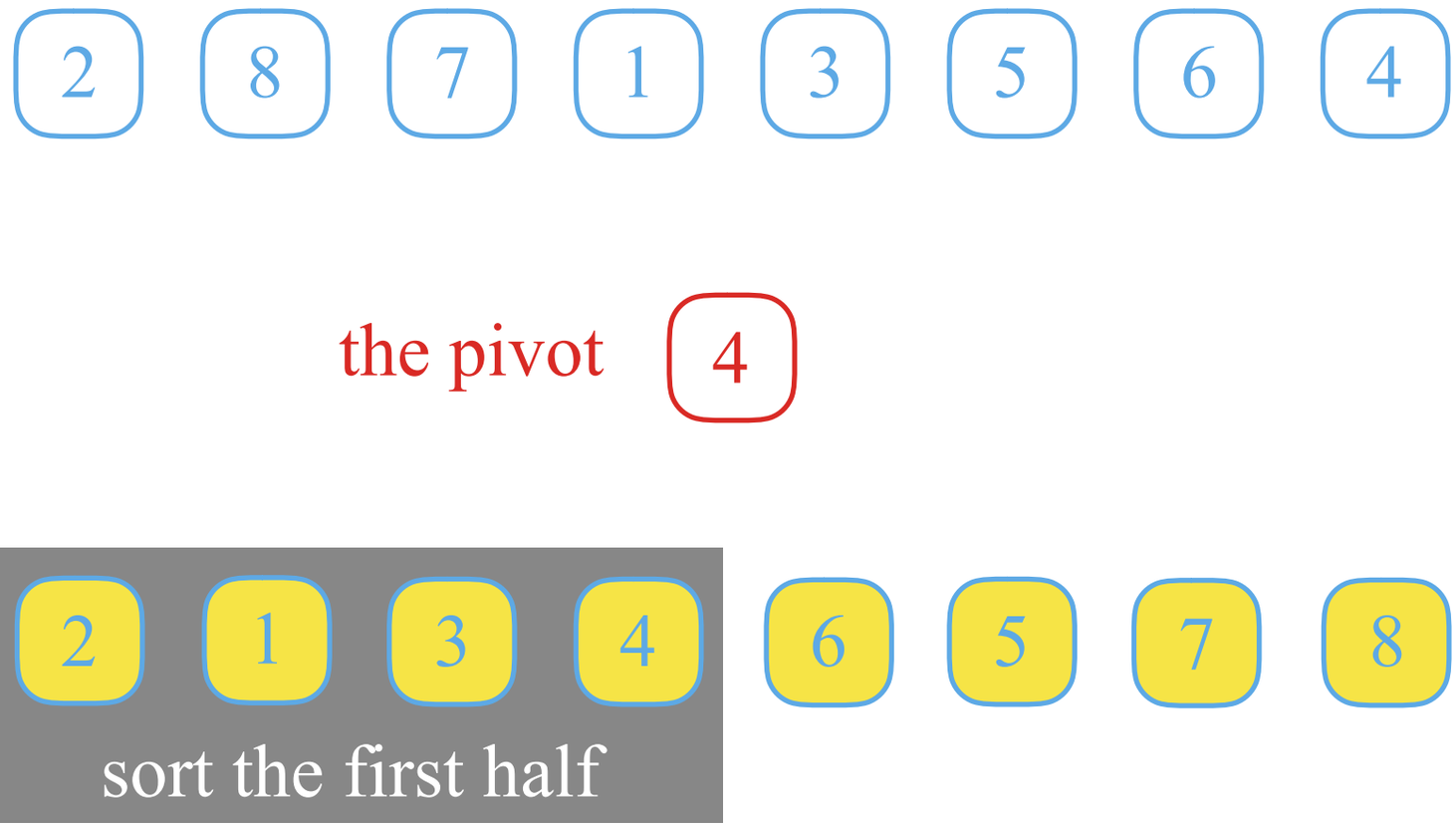
Illustration



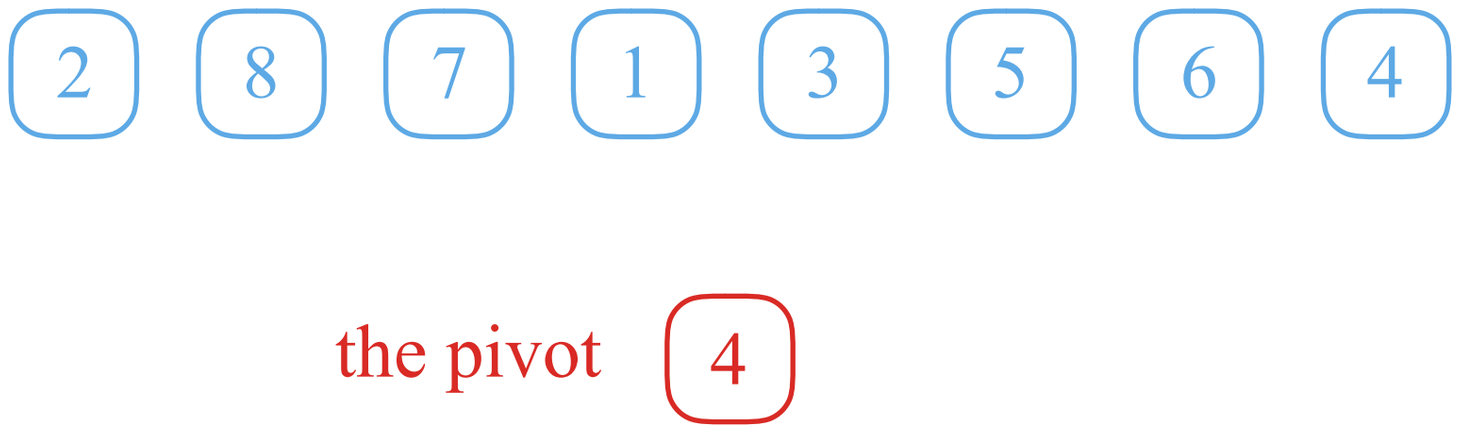
the pivot 4

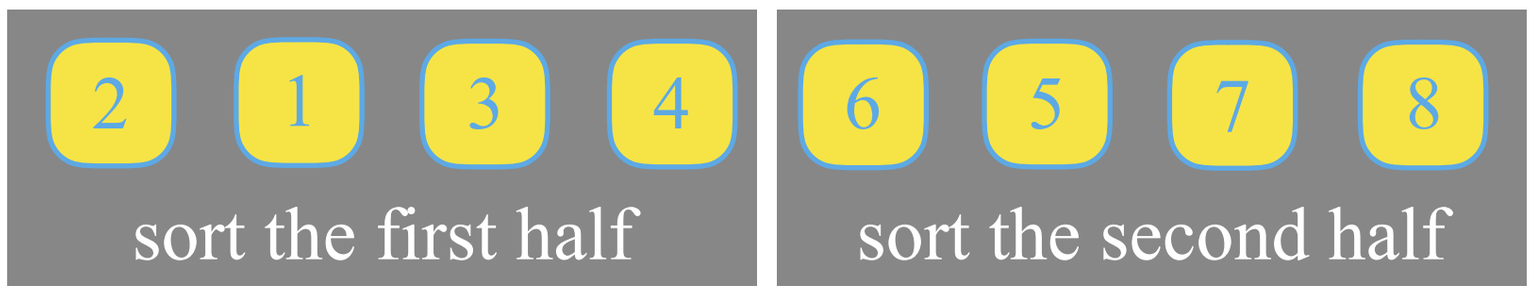


Illustration



Illustration





C++ Code (O(runtime) additional space)

```
void QuickSort(int *A, int n){
    int *buf = new int [n];
    int sfirst = 0, llast = n-1;
    int pivot = A[n-1];

    for(int i=0; i<n; ++i){
        if(A[i] ≤ pivot){
            buf[sfirst++] = A[i];
        }else{
            buf[llast--] = A[i];
        }
    }
    memcpy(A, buf, sizeof(A[0])*n);

    QuickSort(A, sfirst); QuickSort(A+sfirst, n-sfirst);
}
```

C++ Code (O(n) additional space)

```
void QuickSort(int *A, int n, int *buf){
    int *buf = new int [n];
    int sfirst = 0, llast = n-1;
    int pivot = A[n-1];

    for(int i=0; i<n; ++i){
        if(A[i] ≤ pivot){
            buf[sfirst++] = A[i];
        }else{
            buf[llast--] = A[i];
        }
    }
    memcpy(A, buf, sizeof(A[0])*n);

    QuickSort(A, sfirst, buf); QuickSort(A+sfirst, n-sfirst, buf);
}
```

Exercise

Try to implement Quick Sort using O(1) additional space.

It is called *in-place* Quick Sort.

In-Place Quick Sort

```
void QuickSort(int *A, int n){
    int slast = 0;
    int pivot = A[n-1];

    for(int i=0; i<n; ++i){
        if(A[i] ≤ pivot){
            int swap = A[i];
            A[i] = A[slast];
            A[slast++] = swap;
        }
    }

    QuickSort(A, slast); QuickSort(A+slast, n-slast);
}
```

Running Time

If either S or L is always empty (extremely unbalanced)

The running time of Quick Sort can be described by the following recurrence relation:

$$T(n) = \begin{cases} T(n-1) + O(n) & \text{if } n \geq 2 \\ O(1) & \text{if } n = 1 \end{cases}$$

Suppose that the $O(n) < c_1 n$ for every $n \geq 2$ and the $O(1) < c_2$, then we can rewrite the recurrence relation as, where $c = \max\{c_1, c_2\}$:

$$T(n) \leq \begin{cases} T(n-1) + cn & \text{if } n \geq 2 \\ c & \text{if } n = 1 \end{cases}$$

If either S or L is always empty (extremely unbalanced)

The running time of Quick Sort can be described by the following recurrence relation:

$$T(n) = \begin{cases} T(n-1) + O(n) & \text{if } n \geq 2 \\ O(1) & \text{if } n = 1 \end{cases}$$

Suppose that the $O(n) < c_1 n$ for every $n \geq 2$ and the $O(1) < c_2$, then we can rewrite the recurrence relation as, where $c = \max\{c_1, c_2\}$:

$$T(n) \leq \begin{cases} T(n-1) + cn & \text{if } n \geq 2 \\ c & \text{if } n = 1 \end{cases}$$

$T(n) \leq T(n\text{-}1) + cn \leq T(n\text{-}2) + cn + c(n\text{-}1)$
$\leq \cdots \leq c(n + n\text{-}1 + \cdots + 1) \leq cn^2 = O(n^2)$

If ||S|-|L|| is always at most 1 (extremely balanced)

The running time of Quick Sort can be described by the following recurrence relation:

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{if } n \geq 2 \\ O(1) & \text{if } n = 1 \end{cases}$$

By Master Theorem, we get T(n) = O(n log n).

If |S|:|L| is always 9:1 (almost balanced)

The running time of Quick Sort can be described by the following recurrence relation:

$$T(n) = \begin{cases} T(\lfloor n/10 \rfloor) + T(\lceil 9n/10 \rceil) + O(n) & \text{if } n \geq 10 \\ O(1) & \text{if } n < 10 \end{cases}$$

Use the recurrence-tree method, and observe that

(1) each level contributes O(n) running time, and

(2) there are $O(\log_{10/9} n)$ levels.

We guess the running time is O(n log n), and verify the guess by the substibution method.

For arbitrary ratio |S|:|L|

The running time of Quick Sort can be described by the following recurrence relation:

$$T(n) = \begin{cases} T(x_t) + T(n - x_t) + O(n) & \text{for subproblem } t \text{ with } n \geq 2 \\ O(1) & \text{if } n = 1 \end{cases}$$

Use the recurrence-tree method, and observe that

(1) each level contributes O(n) running time, and

(2) there are O(n) levels.

We guess the running time is $O(n^2)$, and verify the guess by the substibution method.

Summary

Quick Sort runs in $O(n^2)$ time for all instances.

Sometimes Quick Sort runs in $O(n \log n)$ time.

Sometimes Quick Sort runs in $\Omega(n^2)$ time. For example, when

$A = \{n, n-1, ..., 1\}$.

Thus we conclude that the worst-case running time of Quick Sort is

$\Theta(n^2)$.

The Power of Randomness

Randomized Quick Sort

Input: an array A of n integers.

Output: the same array with the n integers ordered nondecrementally.

--- Pseudo Code --- // Describe an algorithm in high-level view.

```
QuickSort (A, n){
    pick a random integer k (called random pivot) from A;
    partition A into two subarrays S and L so that
        (1) S contains all elements in A less than or equal to k;
        (2) L contains the rest;

    QuickSort(S, |S|); QuickSort(L, |L|);

    // no combining step is needed if S is placed at the prefix of A and
    // L is placed at the suffix of A
}
```

Usually we get an almost balanced partition

Formally, we say a parition is balanced if

the random pivot has rank in the range [n/4, 3n/4]

breaking ties arbitrarily.

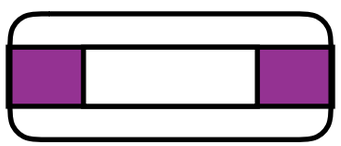
In other words, both S and L has size at least n/4. (Why?)

The probability that a partition is balanced is

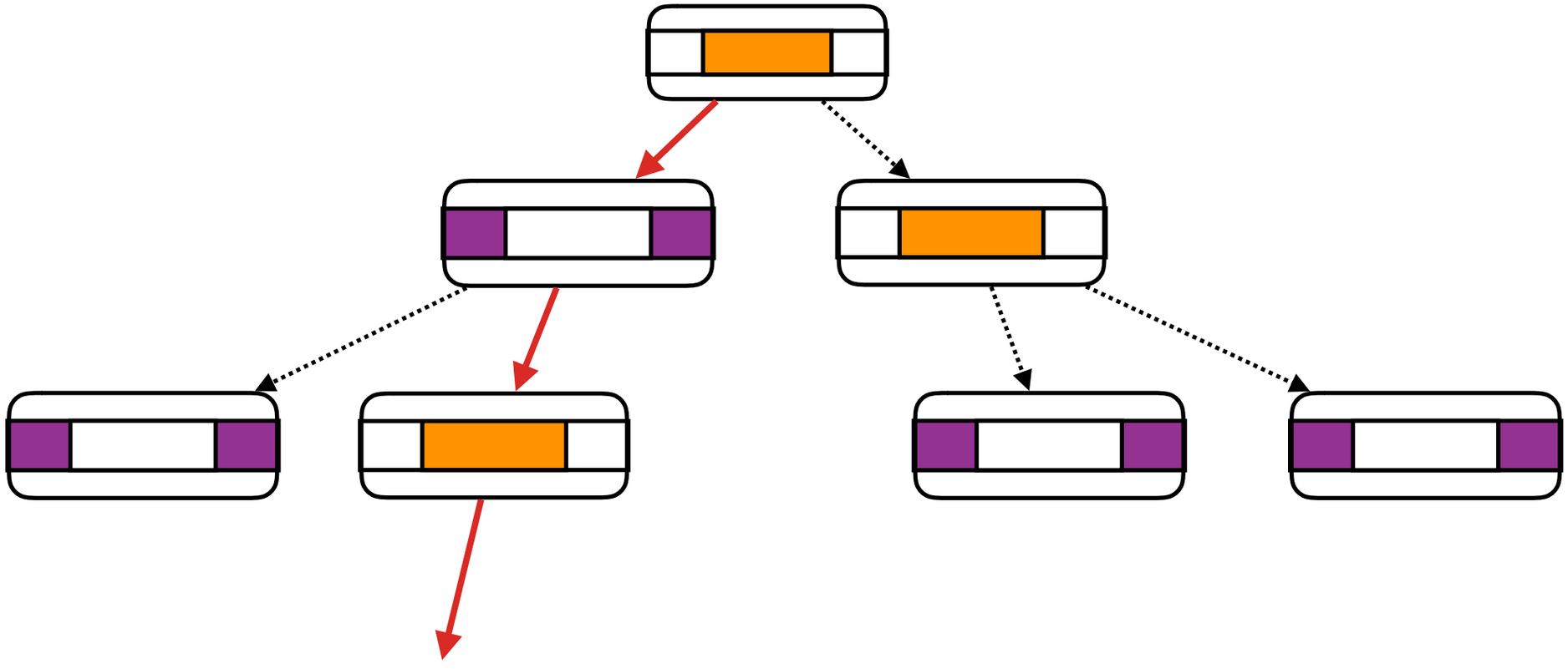
$(3n/4-n/4+1)/n > 1/2$.

Observe a roof-to-leaf path in the recursion tree

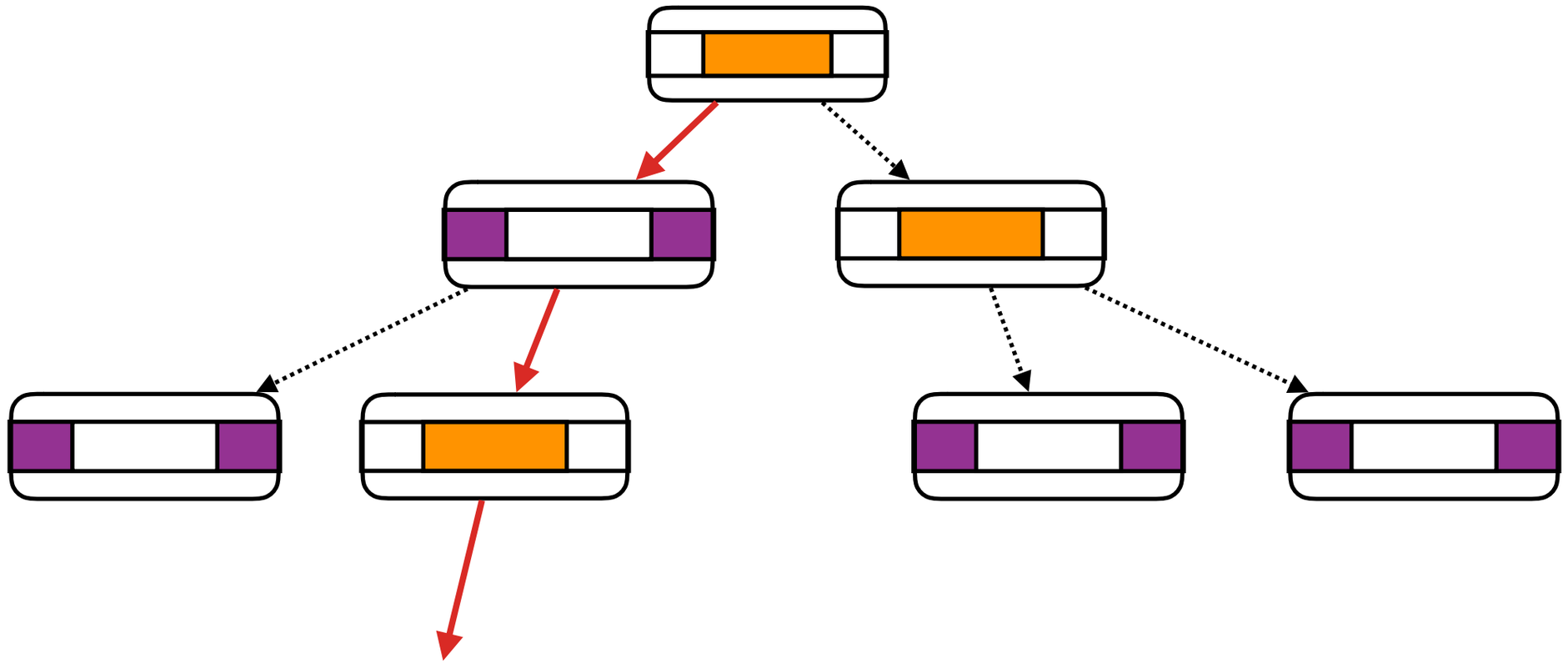
This denotes a computation node whose random pivot has rank in the range [n/4, 3n/4]. Call it **a good node**.



This denotes a computation node whose random pivot has rank outside the range [n/4, 3n/4]. Call it **a bad node**.



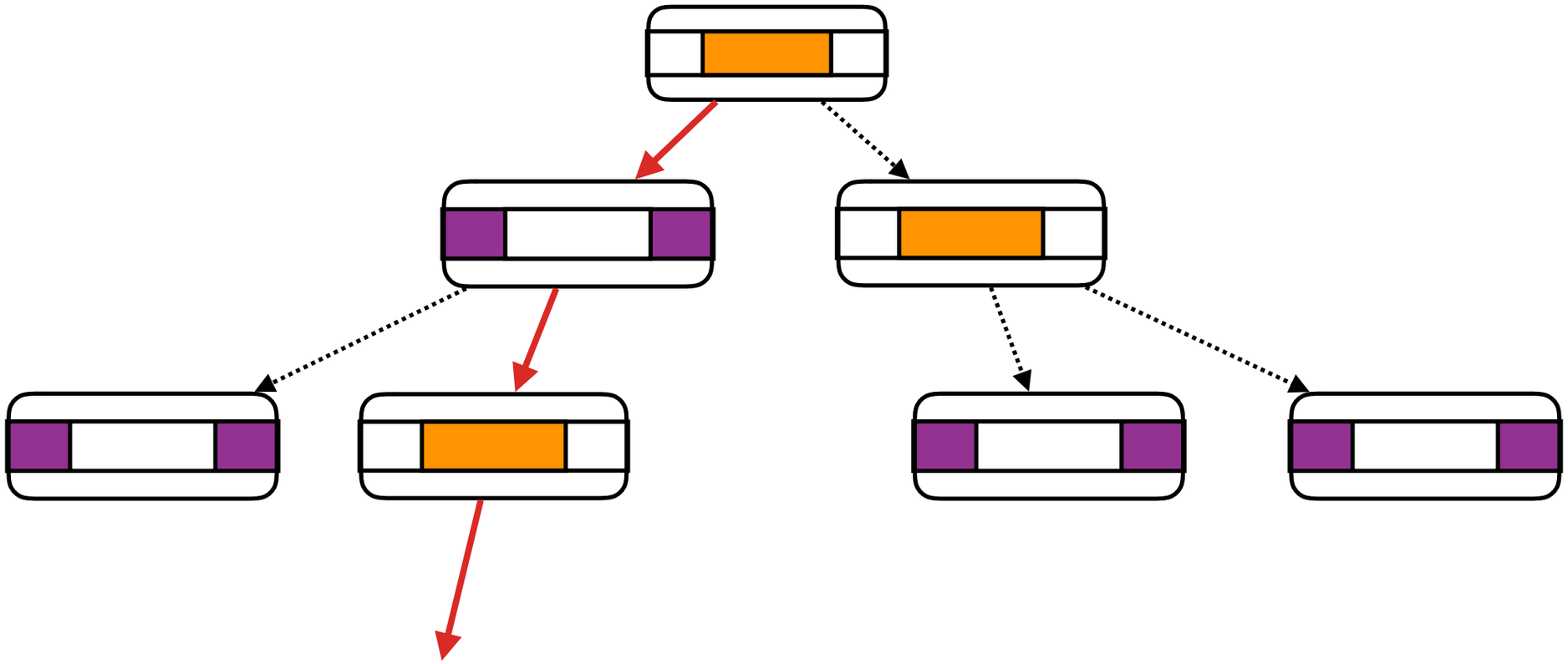
Observe a roof-to-leaf path in the recursion tree



We have $O(\log_{4/3} n)$ good nodes on the root-to-leaf path P. This implies that we have $O(\log_{4/3} n)$ bad nodes on P *with high probability* because good nodes and bad nodes are equally likely to appear.

By *Chernoff Bound*, the above claim fails with probability at most $1/n^{100}$.

Upper-bound the total failure probability



The first root-to-leaf path has length $\omega(\log_{4/3} n)$ with probability $1/n^{100}$.

The second one has length $\omega(\log_{4/3} n)$ with probability $1/n^{100}$.

...

The last one (n-th one) has length $\omega(\log_{4/3} n)$ with probability $1/n^{100}$.

By the *Union bound*, all the root-to-leaf paths has length $O(\log_{4/3} n)$ with probability $n/n^{100} = 1/n^{99}$.

Summary

With probability at least $1 - 1/n^{99}$, the recursion tree of Quick Sort

(1) has height $O(\log_{4/3} n)$, and
(2) each level contributes $O(n)$ running time.

So the running time is $O(n \log n)$ w.h.p.

Note that the probability argument depends on the sampled random numbers, rather than on the input. In other words, one cannot find an input that makes the randomized Quick Sort running slower than $O(n \log n)$ with a fair probability, say 0.0001.

The stl::sort uses Quick Sort as its main procedure.

Selection

Selection Problem

Input: an array A of n integers and an index k in [1, n].

Output: the k-th smallest element in A (tie-breaking by indices). It is also called *k-th order statistics*.

This problem can be solved by sorting in O(n log n) time.

Our goal is to have an algorithm that runs in expected O(n) time.

Quick Select

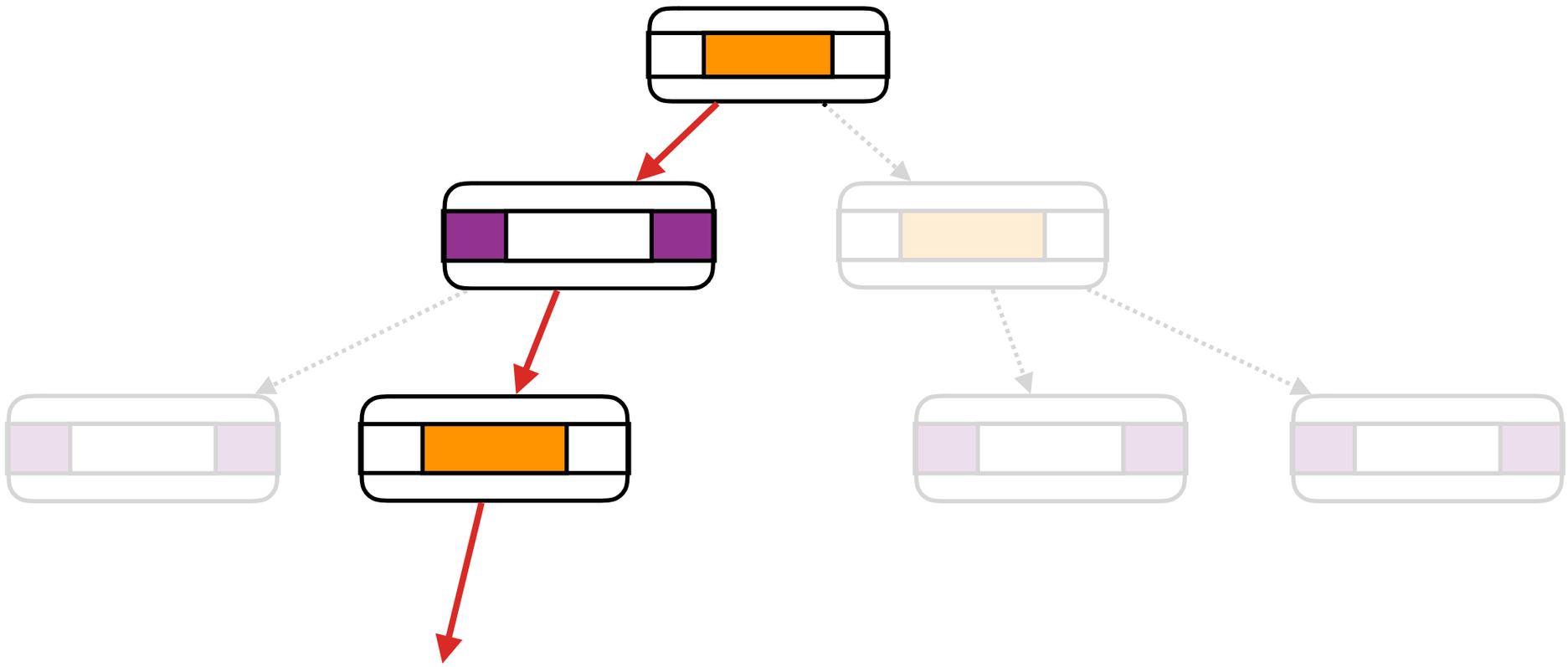
--- Pseudo Code ---

```
QuickSelect (A, k){
    pick a random integer k (called random pivot) from A;
    partition A into two subarrays S and L so that
        (1) S contains all elements in A less than or equal to k;
        (2) L contains the rest;

    if(k ≤ |S|){ // the k-th order statistics is in S
        QuickSelect(S, k);
    }else{ // the k-th order statistics is in L
        QuickSelect(L, k-|S|);
    }
}
```

C++ Code

```
int QuickSelect(int *A, int n, int k){
    if(n == 1){
        if(k == 1) return A[0];
        assert(false);
    }
    int pivot = A[rand()%n];
    int slast = 0;

    for(int i=0; i<n; ++i){
        if(A[i] ≤ pivot){
            int swap = A[i]; A[i] = A[slast]; A[slast++] = swap;
        }
    }
    if(slast ≤ k) return QuickSelect(A, slast, k);
    return QuickSelect(A+slast, n-slast, k-slast);
}
```

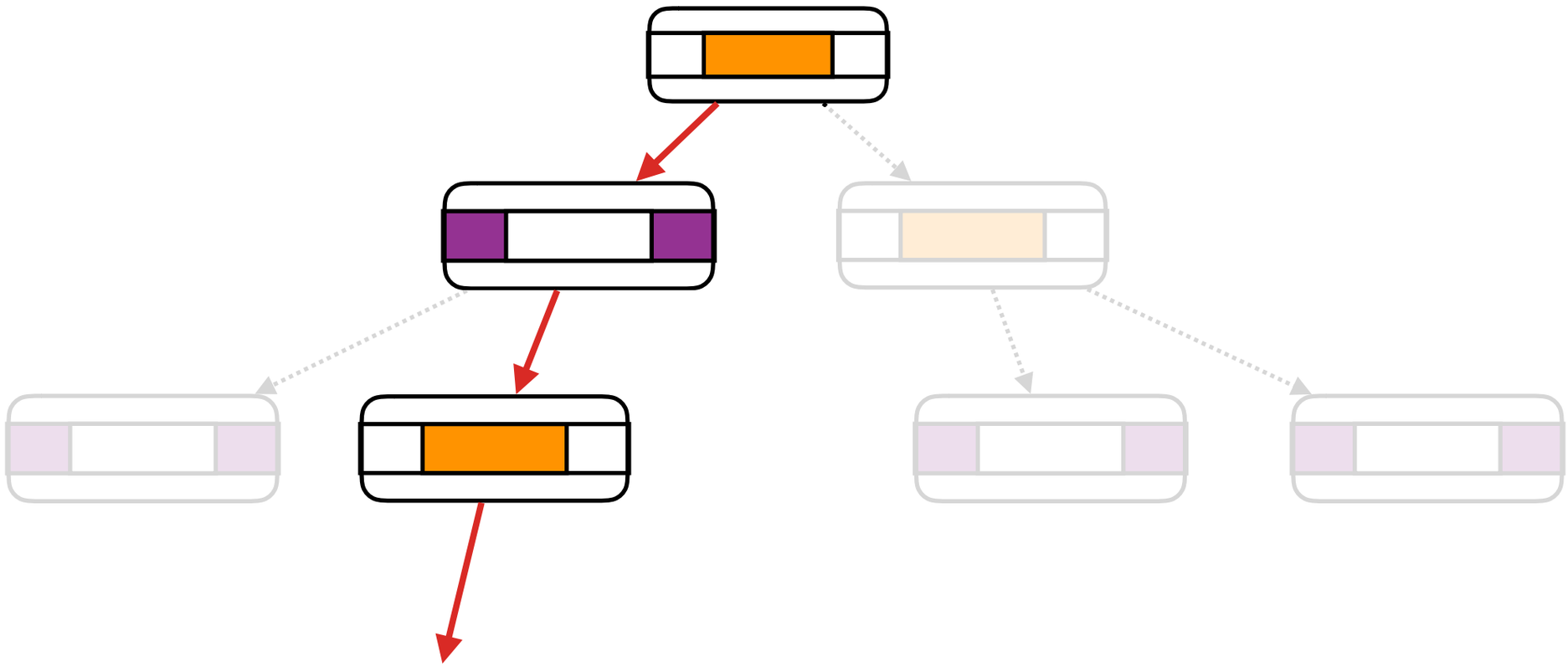
Running Time



A good pivot can reduce the problem size by a factor of at least 3/4. If we always find a good pivot, then the total running time is

$$n + (3/4)\, n + (3/4)^2\, n + \ldots \leq 4n/3 = O(n).$$

Running Time

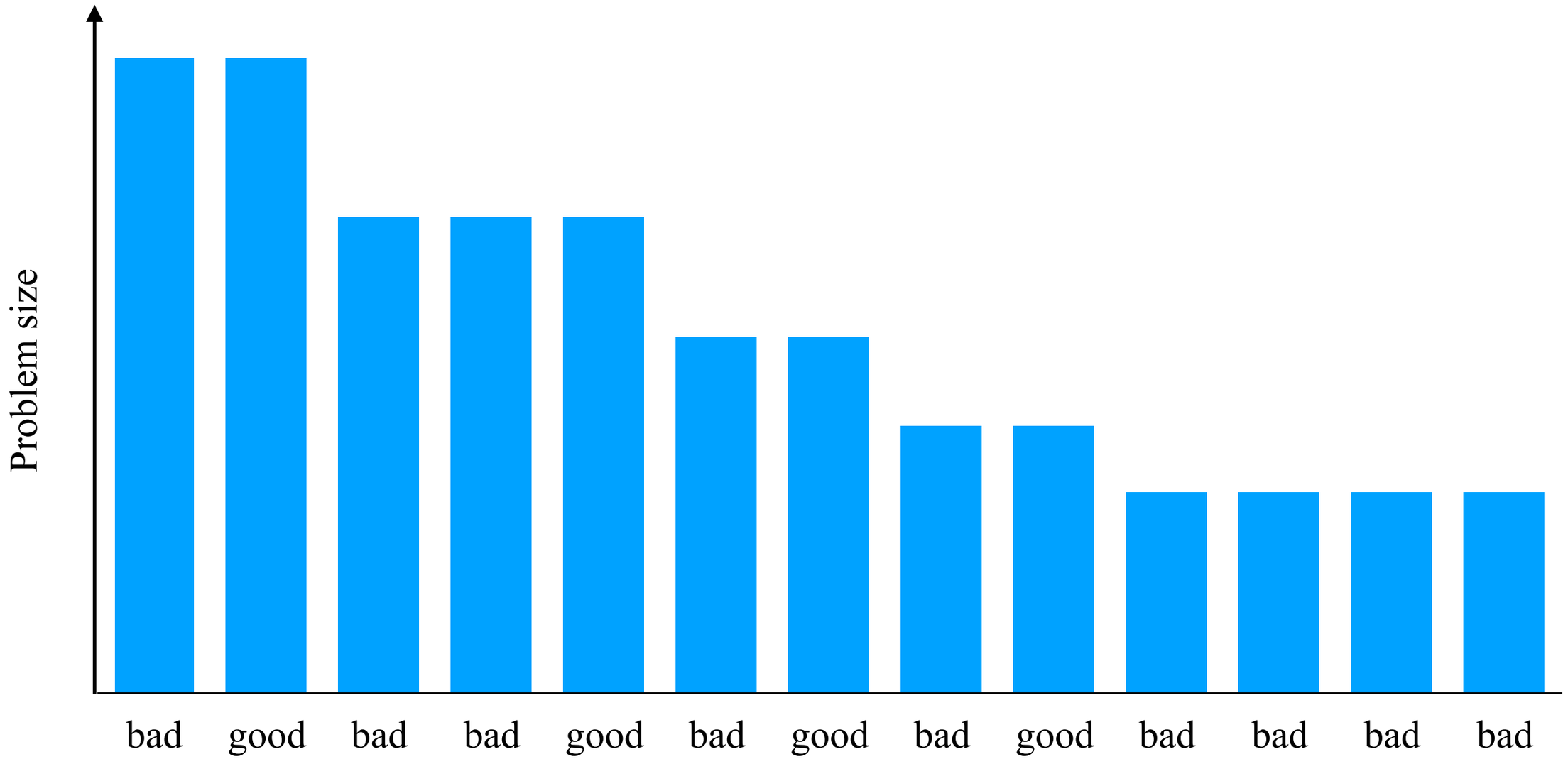


A bad pivot may reduce the problem size by simply 1. If we always find a bad pivot, then the total running time is

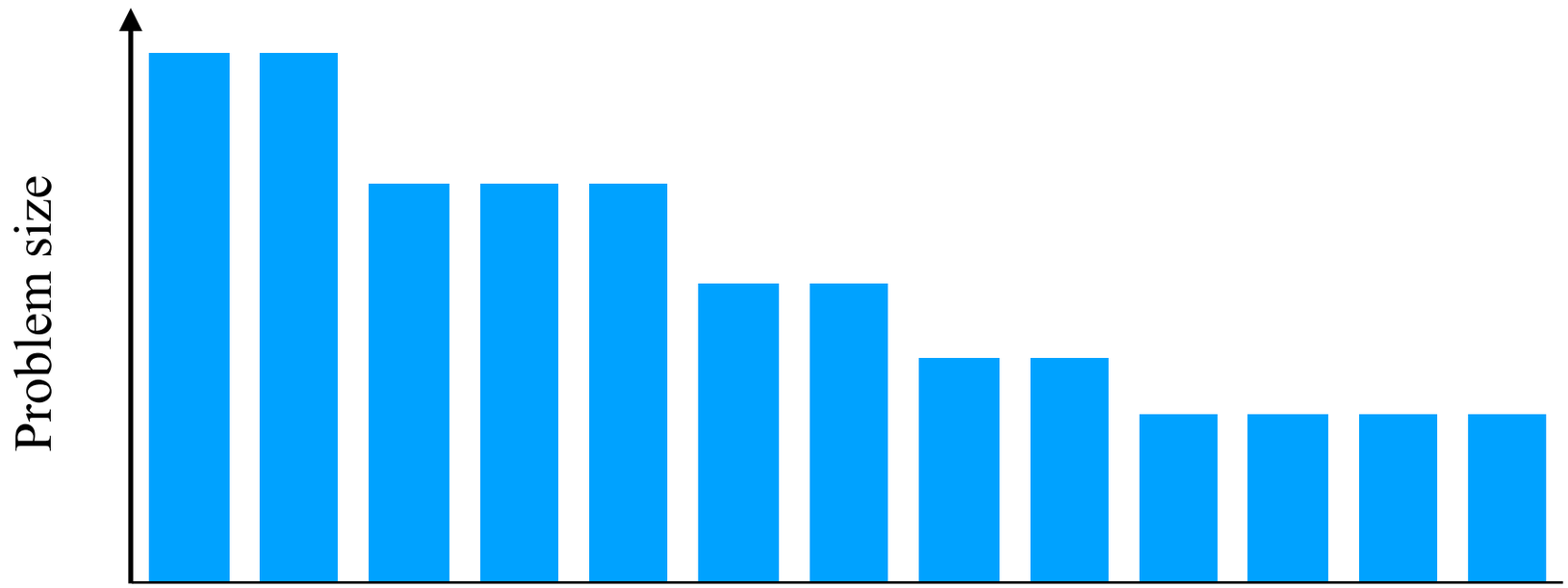
$$n + n\text{-}1 + n\text{-}2 + \ldots + 1 = O(n^2).$$

Running Time

We assume that a good pivot reduce the problem size by a factor of 3/4 and a bad pivot does not reduce the problem size. This assumption works fine if we simply need an upper bound of the running time.



Running Time



The contribution of good nodes to the runtime is at most 4n/3.

The contribution of bad nodes that immediately follow a good node (ℓ_1-bad nodes) to the expected runtime is at most 4n/3 * 1/2.

The contribution of bad nodes that immediately follow a ℓ_1-bad node to the expected runtime is at most 4n/3 * 1/4.

Hence, the expected total runtime is at most 4n(1+1/2+1/4+ ...)/3 = 8n/3.

Deviates from the expectation

We know the runtime of Quick Select is at most 8n/3.

This is an average among all random choices. No matter which random choice is made, the runtime cannot be negative.

Hence, by *Markov inequality* we know with probability 1/2 Quick Select runs in time less than 16n/3.