COMP 446 / 546 ALGORITHM DESIGN AND ANALYSIS

LECTURE 3 DIVIDE AND CONQUER

ALPTEKİN KÜPÇÜ

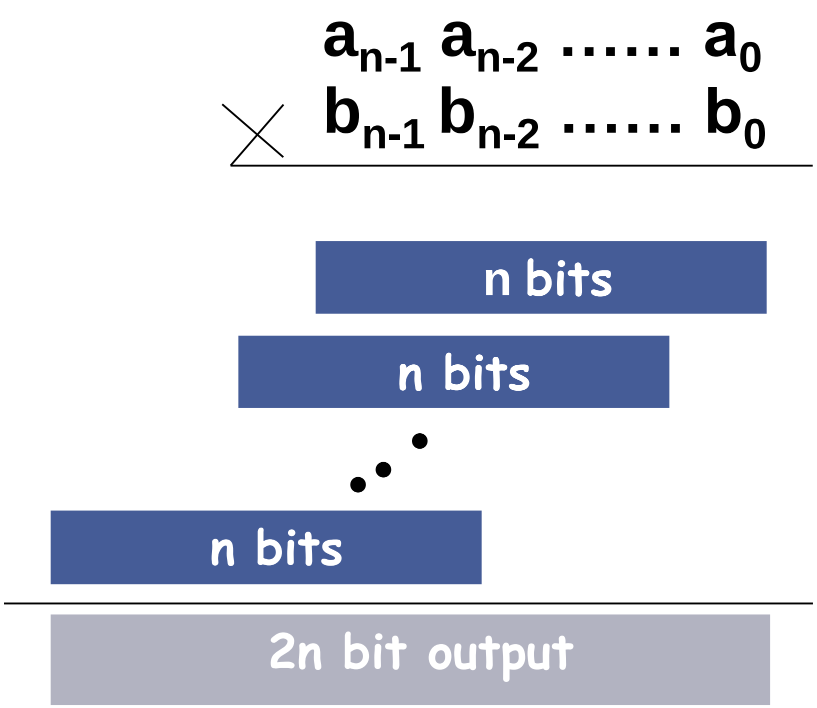
Based on slides of Shafi Goldwasser, David Luebke, George Kollios, Roger Crawfis, and Cevdet Aykanat

RECAP

- What have we seen?
  - Worst-case vs. average-case analysis
  - Stable Marriage Problem
    - Fun problem applicable to many scenarios
    - Algorithm design must take into account the goals
    - Simple correctness proofs
  - Insertion Sort
    - A daily algorithm (sorting a deck of cards)
  - Merge Sort
    - Analyzing recurrence relations
    - Divide-and-Conquer paradigm
- Let’s remember:
  - Binary Search

MULTIPLYING LARGE INTEGERS

- Given two n-bit integers a and b, compute c=ab
- Naive (grade-school) algorithm
- Cost?
- Total work $\Theta(n^2)$



MULTIPLYING LARGE INTEGERS: DIVIDE AND CONQUER

Divide: Write $a = A_1 2^{n/2} + A_0$
$b = B_1 2^{n/2} + B_0$

for A_0, A_1, B_0, B_1 n/2 bit integers

Assume $n=2^k$ w.l.o.g.

Conquer: $ab = A_1B_1\ 2^n + A_1B_0\ 2^{n/2} + B_1A_0\ 2^{n/2} + A_0B_0$
$= A_1B_1\ 2^n + (A_1B_0 + B_1A_0)\ 2^{n/2} + A_0B_0$

Reduces to 4 multiplications of n/2-bit integers (done recursively), plus 3 additions and 2 shifts.
Additions and shifts cost $\Theta(n)$.

MULTIPLYING LARGE INTEGERS: DIVIDE AND CONQUER

$$T(n) = 4\,T(n/2)+cn$$

 # subproblems

Subproblem size
recursive calls

Work dividing and combining
additions and shifts

- Case 1 of Master Theorem
- $T(n)=\Theta(n^2)$

No better than the grade school algorithm???

MULTIPLYING LARGE INTEGERS: KARATSUBA'S ALGORITHM

Compute $x = A_1B_1$, $y = A_0B_0$, $z = (A_0+A_1)(B_0+B_1)$

Let $ab = x\,2^n + (z-y-x)\,2^{n/2} + y$

Multiply (n,a,b)

- a and b are n-bit integers
- assume n is a power of 2 for simplicity

1. If n<2 then use grade-school algorithm else
2. $A_1 \longleftarrow a \text{ div } 2^{n/2}$; $B_1 \longleftarrow b \text{ div } 2^{n/2}$
3. $A_1 \longleftarrow a \text{ mod } 2^{n/2}$; $B_1 \longleftarrow b \text{ mod } 2^{n/2}$
4. x MULTIPLY $(n/2, A_1, B_1)$
5. y MULTIPLY $(n/2, A_0, B_0)$
6. z MULTIPLY $(n/2, A_0+A_1, B_0+B_1)$
7. Output $x\,2^n + (z-x-y)\,2^{n/2} + y$

MULTIPLYING LARGE INTEGERS: KARATSUBA'S ALGORITHM

$$T(n) = 3\,T(n/2) + cn$$

 # subproblems

Subproblem size
recursive calls

Work dividing and combining
additions and shifts

- Case 1 of Master Theorem
- $T(n) = \Theta(n^{\log_2 3} = n^{1.58496})$

Much better. But can we do even better??

WHY STOP HERE?

- **We can obtain a sequence of asymptotically faster integer multiplication algorithm by splitting the inputs into more pieces.**
- **If we split A and B into k equal parts than the corresponding multiplication algorithm is obtained from an interpolation-based polynomial multiplication algorithm of two degree k-1 polynomials.**
- **Since the product is of degree 2(k-1), we need to evaluate 2k-1 points. Thus there are 2k-1 multiplications each of size n/k and time for splitting and adding is still O(n).**
- **T(n) = (2k-1)T($\frac{n}{k}$)+cn = Θ($n^{\log_k 2k-1}$) $\approx n^{\varepsilon}$ for any $\varepsilon > 1$**
- **Fastest: T(n) = Θ (n log n log log n)**
  - Based on the Fast Fourier Transform

EXPONENTIATION

- Problem: compute a^n, where n is in N.
- Naive algorithm: Compute $a^1, a^2, a^3, \ldots, a^n$ Complexity: $\Theta(n)$ multiplications
- Divide-and-Conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even} \\ a^{n-1/2} \cdot a^{n-1/2} \cdot a & \text{if } n \text{ is odd} \end{cases}$$

$$T(n) = T(n/2) + c \quad \Rightarrow \quad T(n) = \Theta(\log n) \text{ multiplications}$$

MATRIX MULTIPLICATION

$$\left.\begin{array}{l}\text{Input: } A = [a_{ij}]\ ;\ B = [b_{ij}] \\[1em] \text{Output: } C = [c_{ij}] = AB\end{array}\right\}\quad i\ ,\ j = 1,\ 2,\ \ldots\ldots\ n$$

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & \cdots & c_{1n} \\ \vdots & c_{22} & \ddots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ \vdots & a_{22} & \ddots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & \cdots & b_{1n} \\ \vdots & b_{22} & \ddots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & \cdots & b_{nn} \end{pmatrix}$$

$$c_{ij} = \sum_{1 \le k \le n} a_{ik} \,.\, b_{kj}$$

STANDART ALGORITHM

for $i \leftarrow 2$ to length[A]
    do for $j \leftarrow 1$ to n
        do $c_{ij} \leftarrow 0$
            for $k \leftarrow 1$ to n
                do $c_{ij} \leftarrow c_{ij} + a_{ik} \cdot b_{kj}$

Running time = $\Theta(n^3)$
unit multiplications and additions

Non-square matrices?

DIVIDE AND CONQUER ALGORITHM

- **Idea:** n x n matrix = 2 x 2 matrix of (n/2) x (n/2) submatrices

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

C = A · B

$$\left.\begin{aligned} c_{11} &= a_{11}\, b_{11} + a_{12}\, b_{21} \\ c_{12} &= a_{11}\, b_{12} + a_{12}\, b_{22} \\ c_{21} &= a_{21}\, b_{11} + a_{22}\, b_{21} \\ c_{22} &= a_{21}\, b_{21} + a_{22}\, b_{22} \end{aligned}\right\}$$

8 mults of (n/2) x (n/2) submatrices
4 adds of (n/2) x (n/2) submatrices

ANALYSIS OF D&C ALGORITHM

$$T(n) = 8\,T(n/2) + cn^2$$

 # subproblems

Subproblem size

recursive calls on (n/2) x (n/2) matrices

Work dividing submatrices

- Case 1 of Master Theorem
- $T(n) = \Theta(n^{\log_2 8} = n^3)$

No better than the standard algorithm??

STRASSEN'S IDEA

- **Same Idea:** n x n matrix = 2 x 2 matrix of (n/2) x (n/2) submatrices
- But multiply 2 x 2 matrices with only 7 recursive mults

- $P_1 = a_{11} \times (b_{12} - b_{22})$
- $P_2 = (a_{11} + a_{12}) \times b_{22}$
- $P_3 = (a_{21} + a_{22}) \times b_{11}$
- $P_4 = a_{22} \times (b_{21} - b_{11})$
- $P_5 = (a_{11} + a_{22}) \times (b_{11} + b_{22})$
- $P_6 = (a_{11} - a_{22}) \times (b_{21} + b_{22})$
- $P_7 = (a_{11} - a_{21}) \times (b_{11} + b_{12})$

- $C_{11} = P_5 + P_4 - P_2 + P_6$
- $C_{12} = P_1 + P_2$
- $C_{21} = P_4 + P_3$
- $C_{22} = P_5 + P_1 - P_3 - P_7$

7 mults
18 adds/subs

ANALYSIS OF STRASSEN'S ALGORITHM

$$T(n) = 7\,T(n/2) + cn^2$$

 # subproblems

Subproblem size

recursive calls on (n/2) x (n/2) matrices

Work dividing submatrices

- Case 1 of Master Theorem
- T(n)= $\Theta(n^{\log_2 7} = n^{2.81})$

Beats the standard algorithm for n > 30.
Best known today: $\Theta(n^{2.376})$ [Coppersmith-Winograd] only of theoretical interest.

CONCLUSIONS

- **Divide and conquer is just one of several powerful techniques for algoritm design.**
  - Can lead to more efficient algorithms.

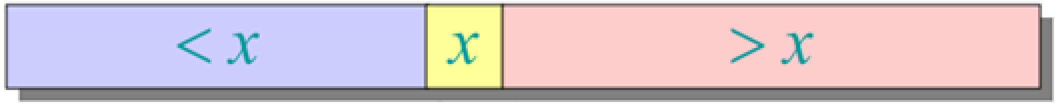
- **Divide and conquer algorithms can be analyzed using recurrence relations.**
  - So practice this math.

QUICKSORT

- **Proposed by C.A.R. Hoare in 1962.**
- **Divide and conquer algorithm but the work is in divide rather than in combine**
- **Different versions:**
  - Basic: Good in average case (for a random input)
  - Randomized: good for all inputs in expectation (Randomized Las Vegas algorithm)
- **Sorts in place (like insertation sort, but unlike merge sort).**
- **Very practical (even though asymptotically not optimal).**

IDEA: DIVIDE AND CONQUER

- **Quicksort an n-element array A:**
- **Divide:**
  1. Pick a pivot element x in A
  2. Partition the array into three sub-arrays
     L(elements < x), E(elements = x), G(elements > x)



Conquer: Recursively sort sub-arrays L and G

Combine: Do nothing.

HOW TO CHOOSE PIVOT X

Basic Quick Sort:

Pivot is the first element: X = A[1]

Time: worst case $O(n^2)$ time

$O(n \log n)$ time for average input

Randomized Quick Sort:

X is chosen at random from the array A

(recursively each time a random choice)

Time: Expected $O(n \log n)$ for all inputs

Randomness gives us more control over runtime.

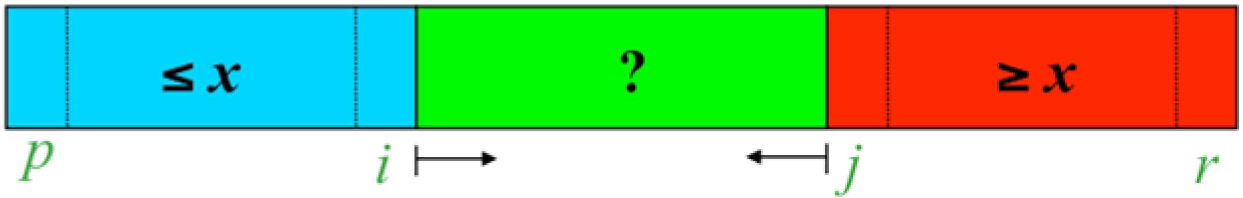
PSEDOCODE FOR BASIC QUICKSORT

```
QUICKSORT (A,p,r)
        if p < r then
                q ← PARTITION(A,p,r)
                QUICKSORT(A,p,q-1)
                QUICKSORT(A,q+1,r)
```

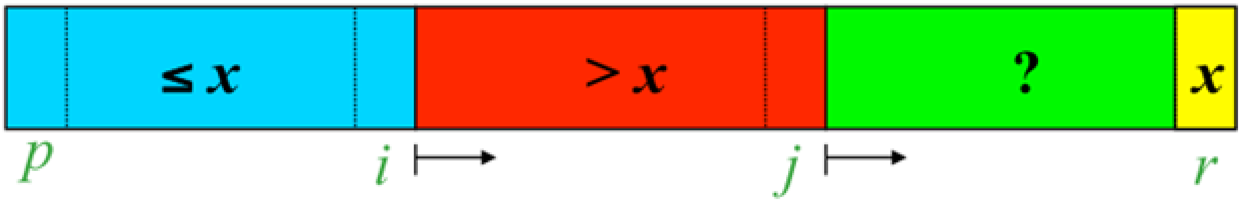
- p and r denote beginning and ending indices
- Initial call: QUICKSORT(A,1,n)

TWO PARTITIONING ALGORTIHMS

1. **Hoare's algorithm: Partitions around the first element of sub-array (pivot x = A[p]). Partitions grow in opposite directions.**



2. **Lomuto's algorithm: Partitions around the last element of sub-array (pivot x = A[r]). Partitions grow in the same direction.**

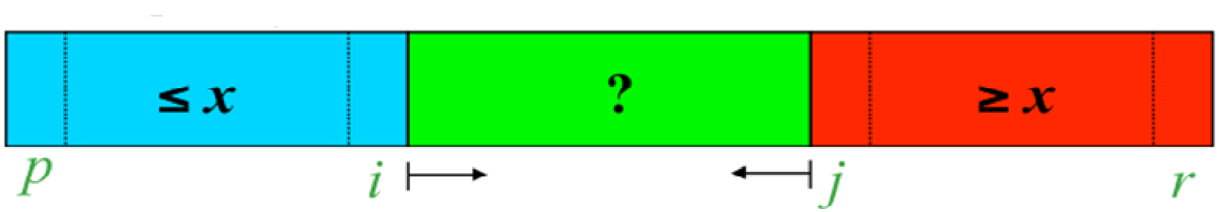


HOARE PARTITIONING

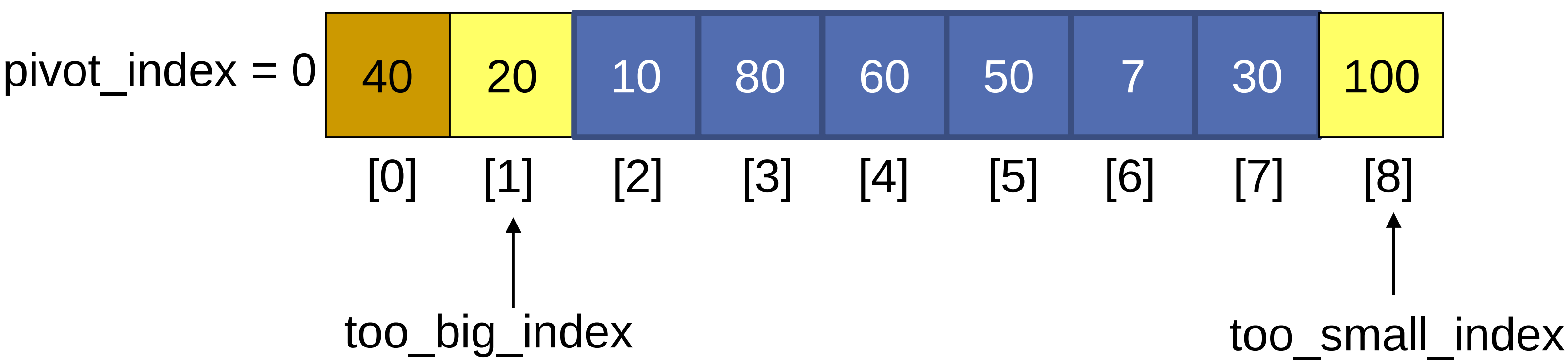
- **Pick first element as pivot**

40	20	10	80	60	50	7	30	100

- **Partition array into sub-arrays:**
  - Less than the pivot
  - Greater than the pivot
  - At the end, put pivot into the middle

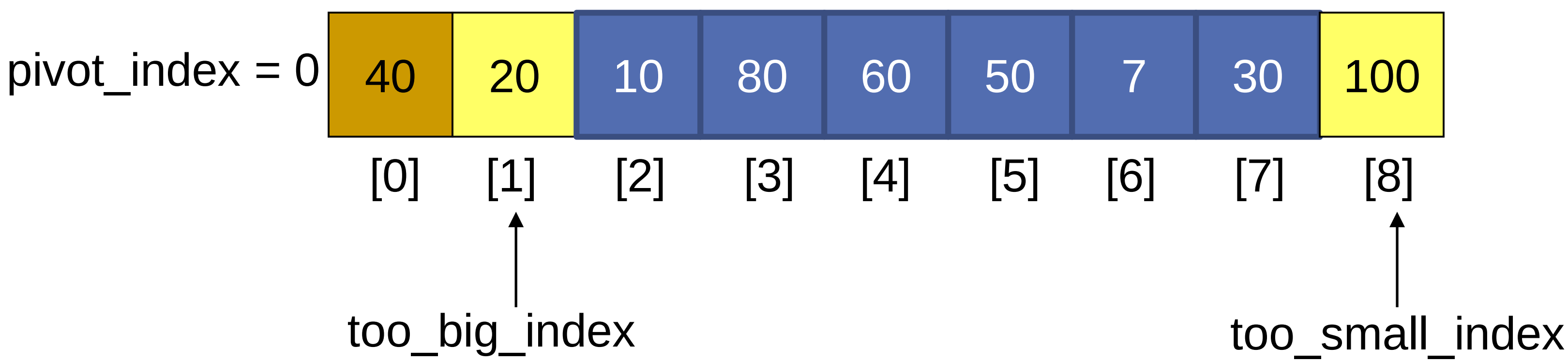


PARTITIONING EXAMPLE



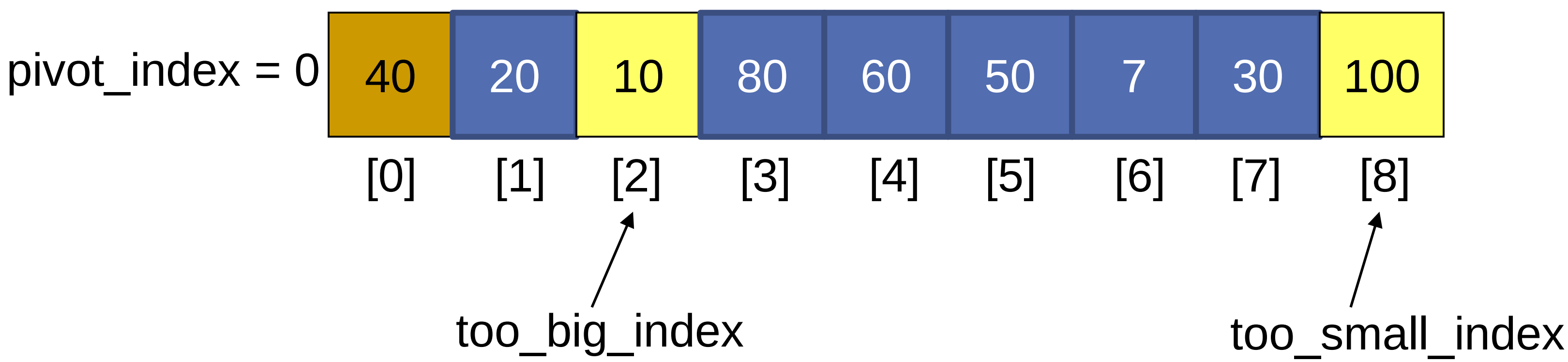
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index



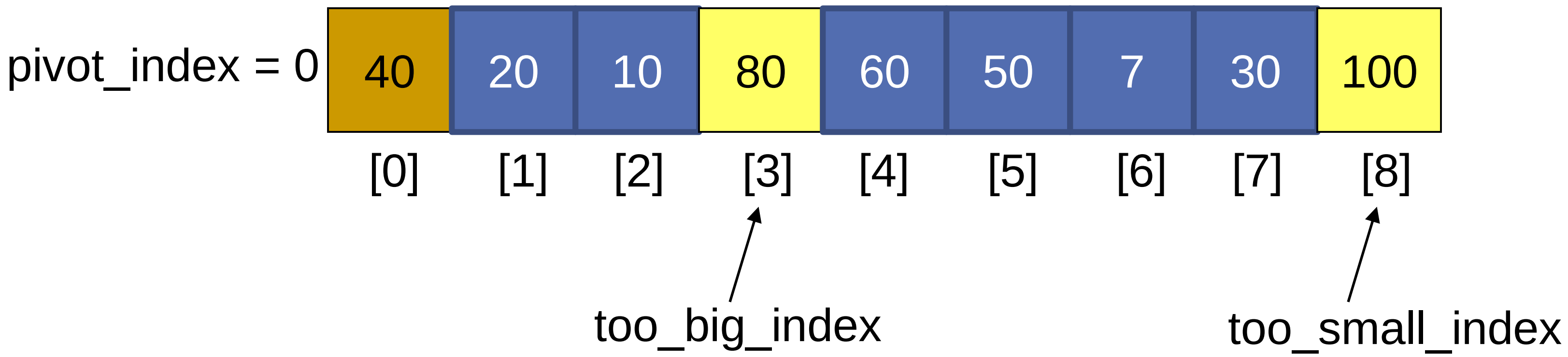
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index



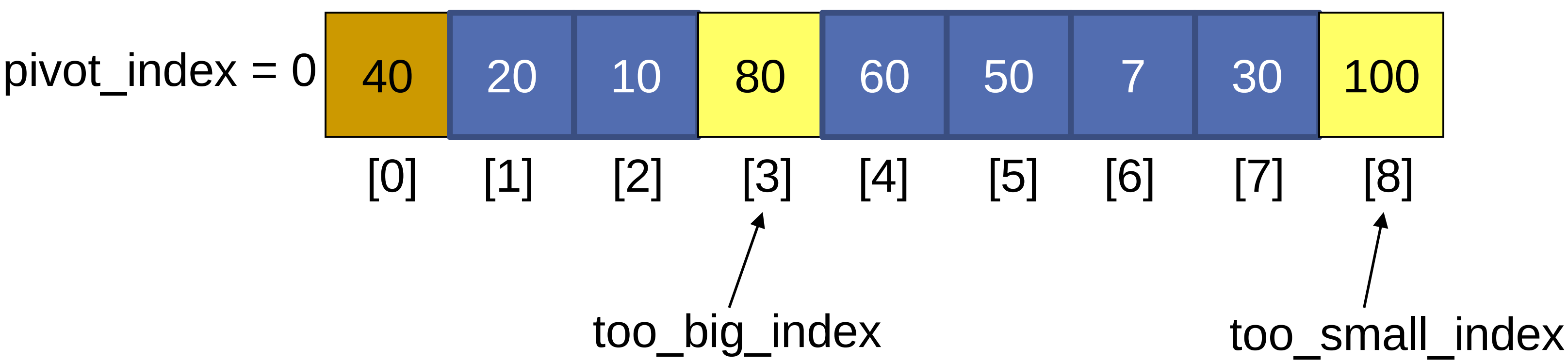
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index



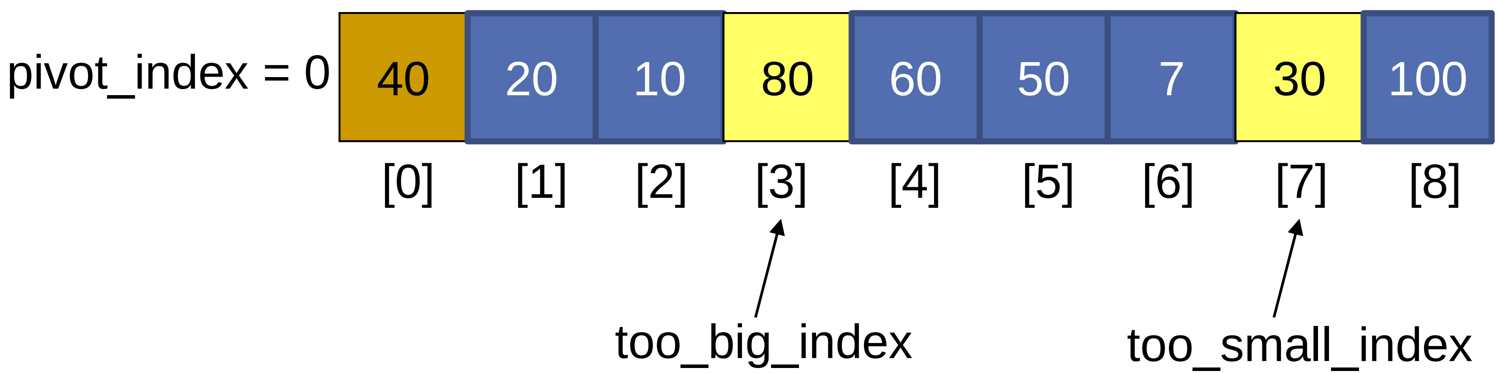
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



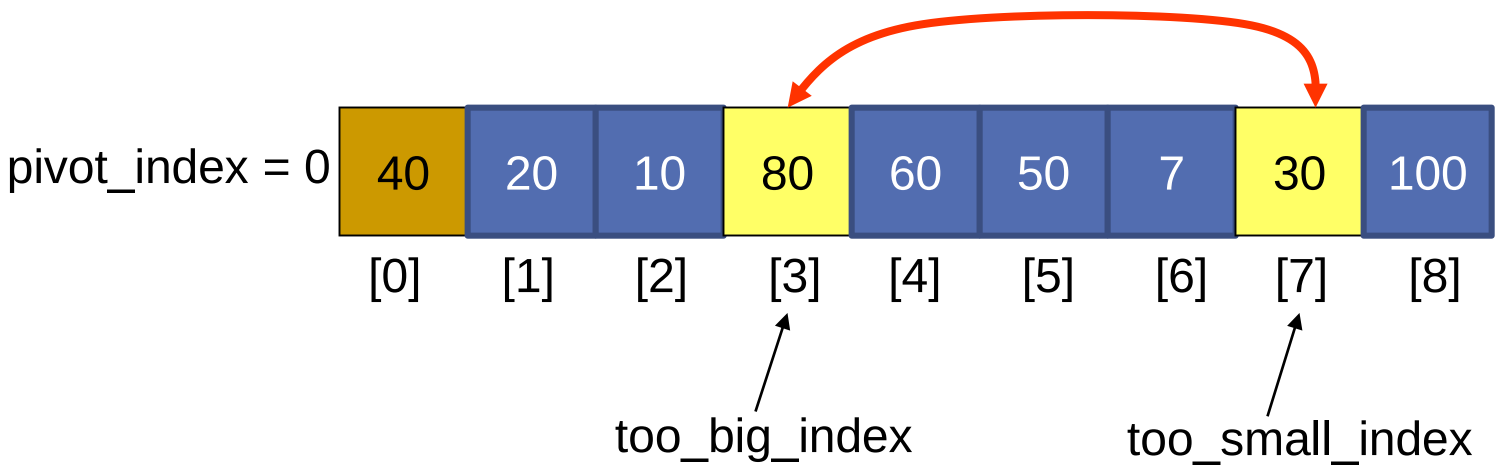
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index



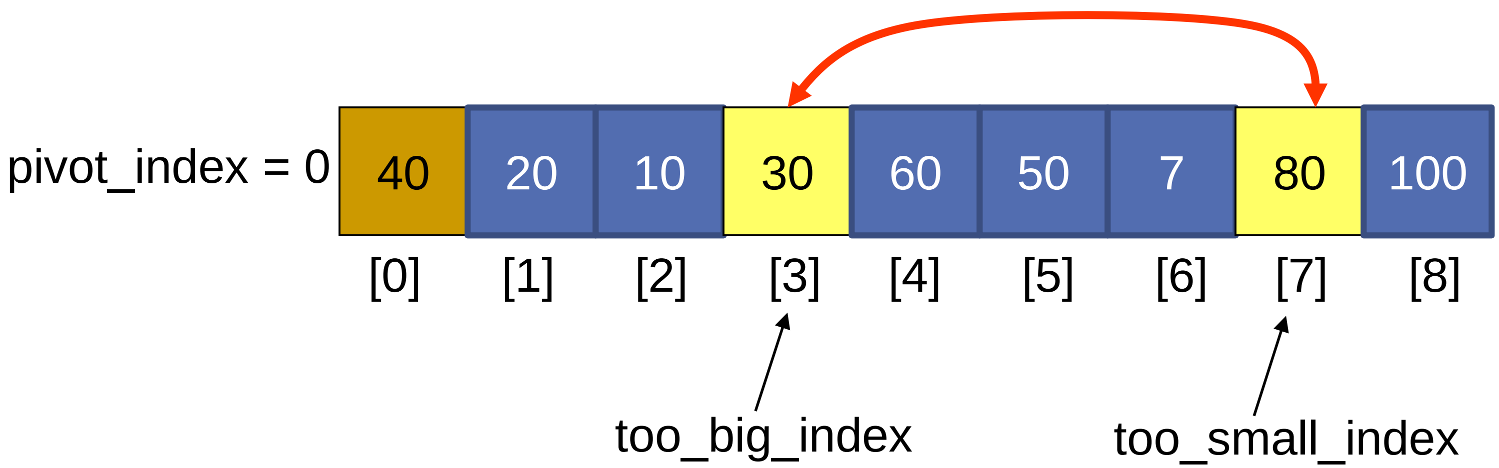
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



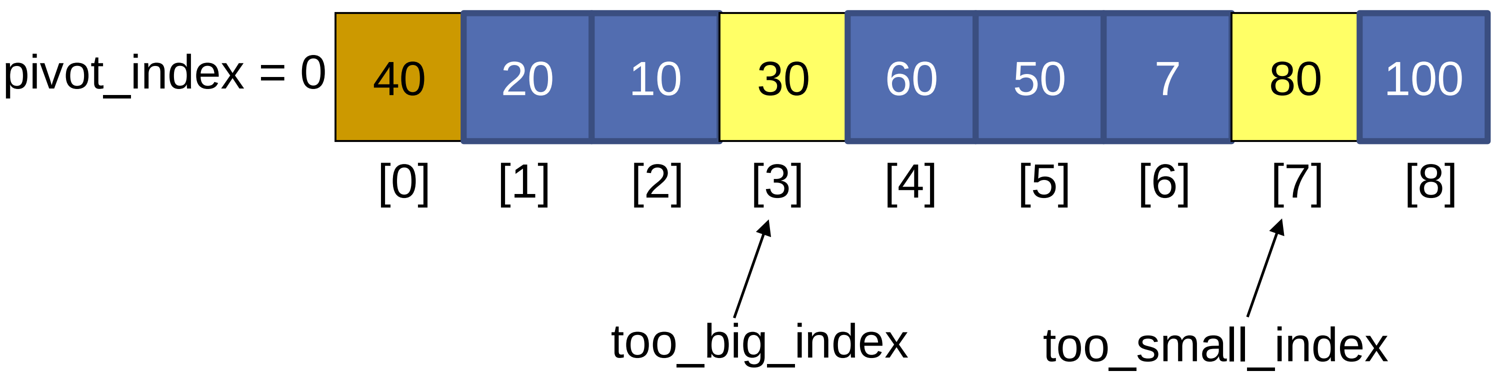
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]



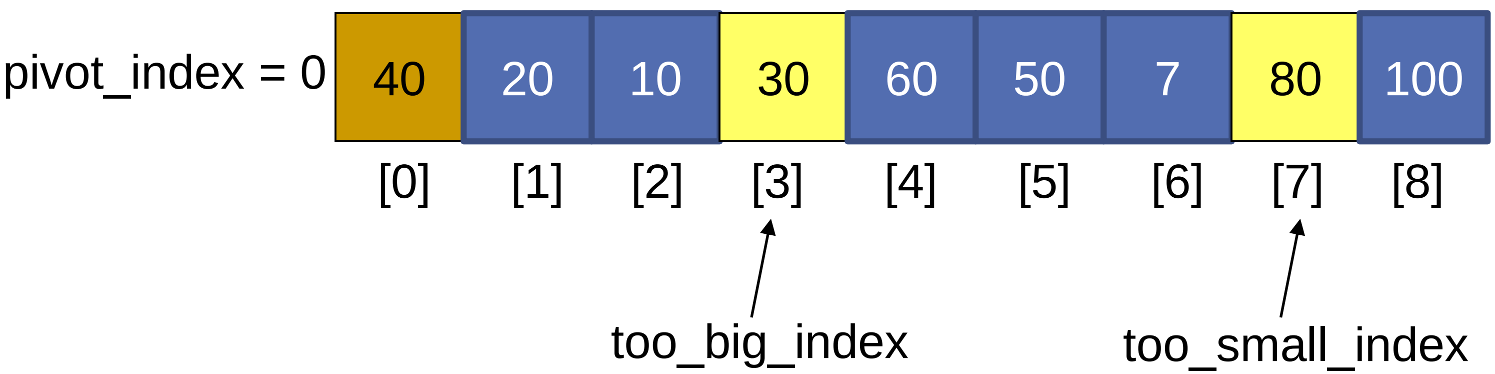
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



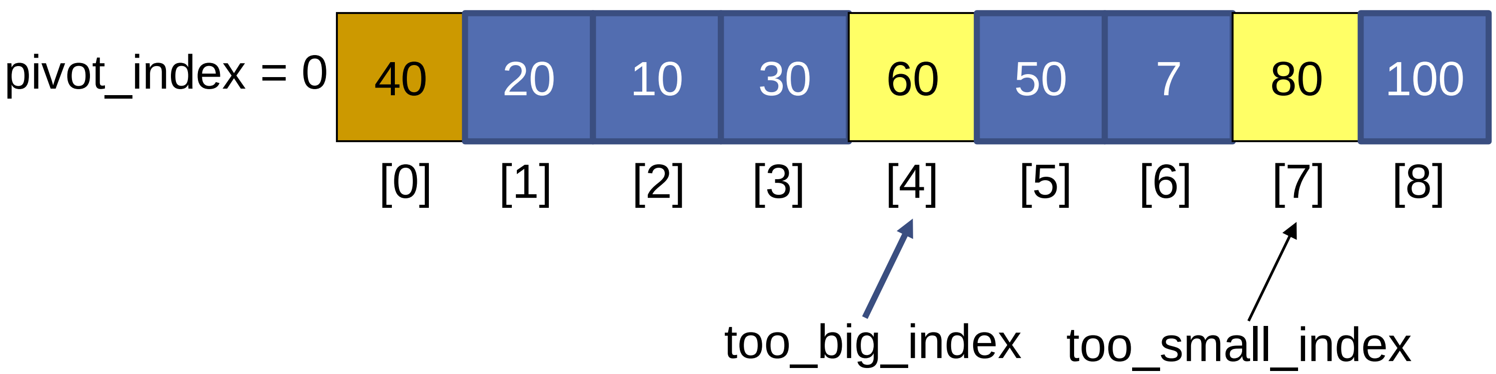
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



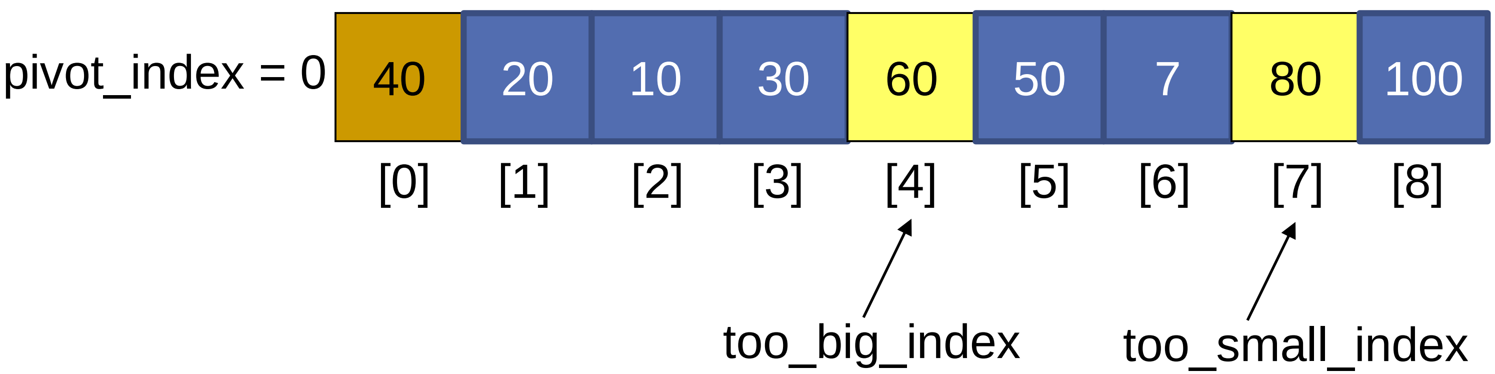
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



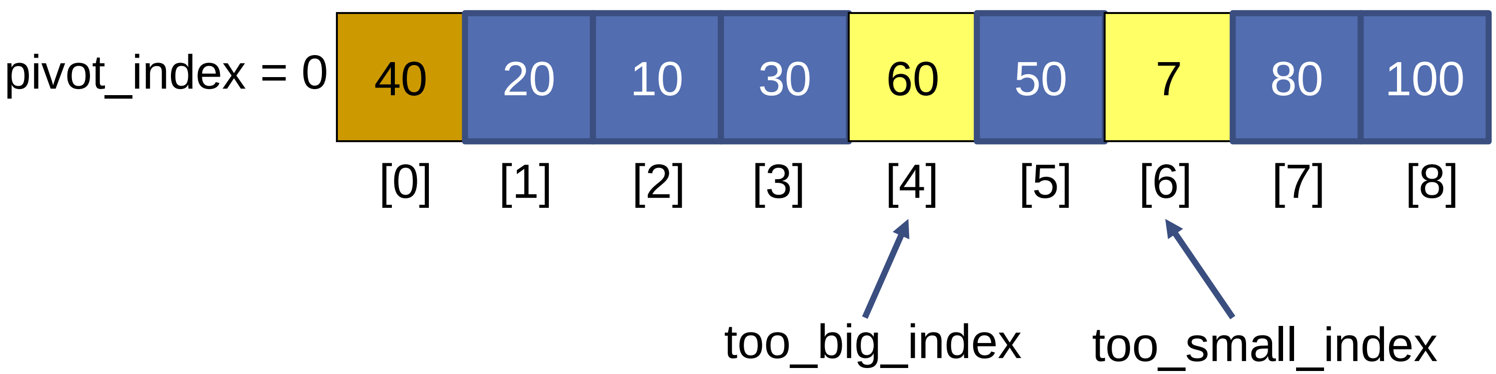
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



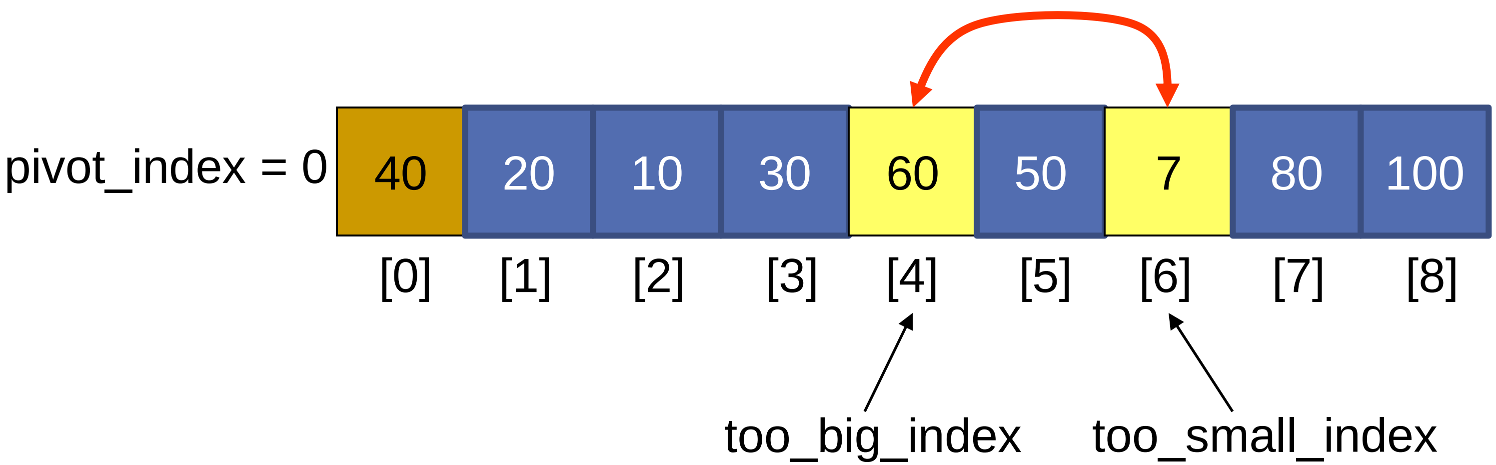
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



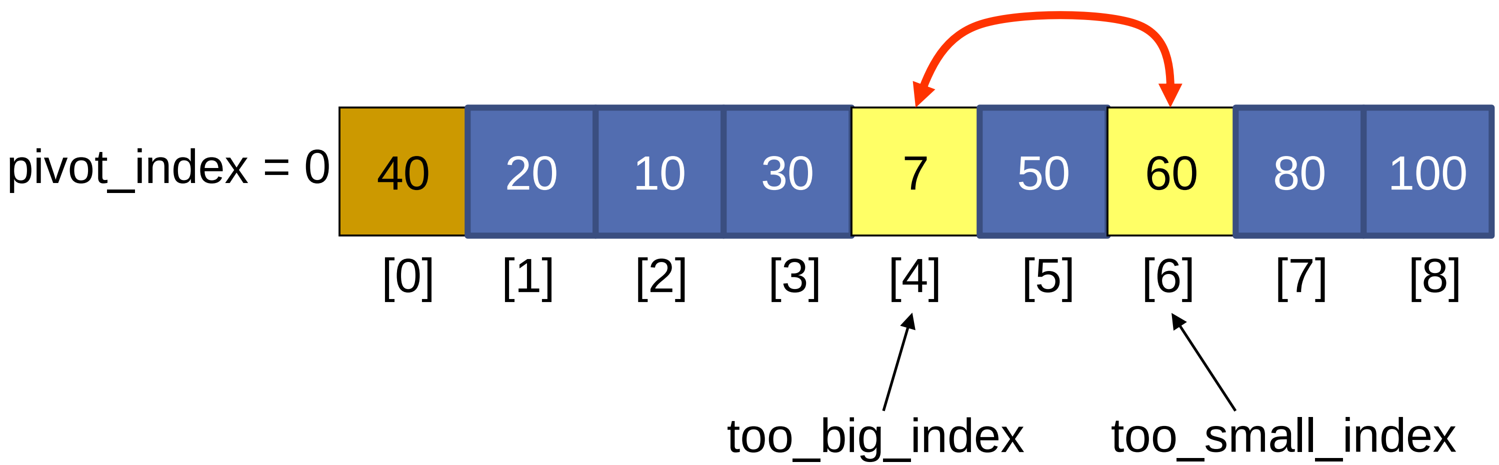
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



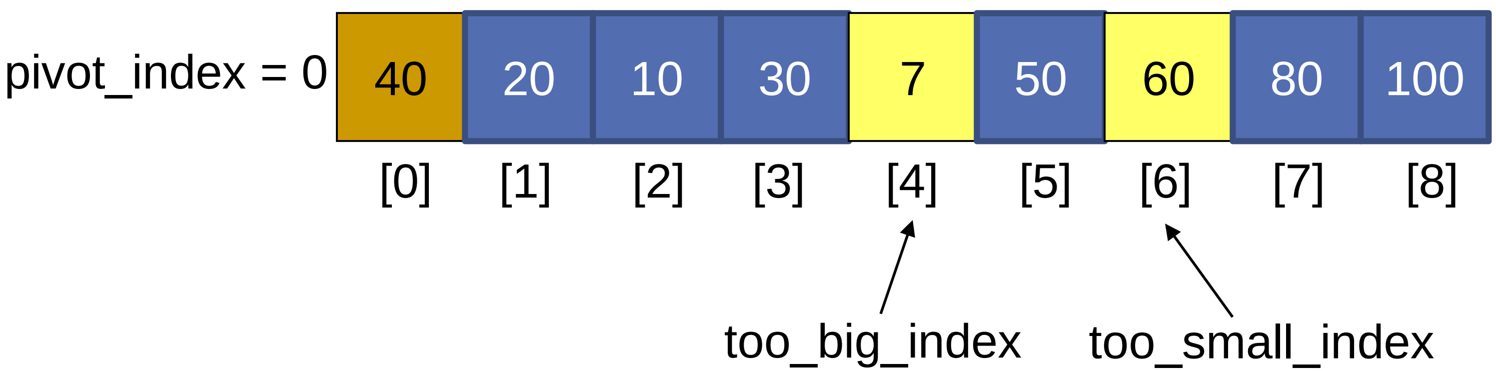
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



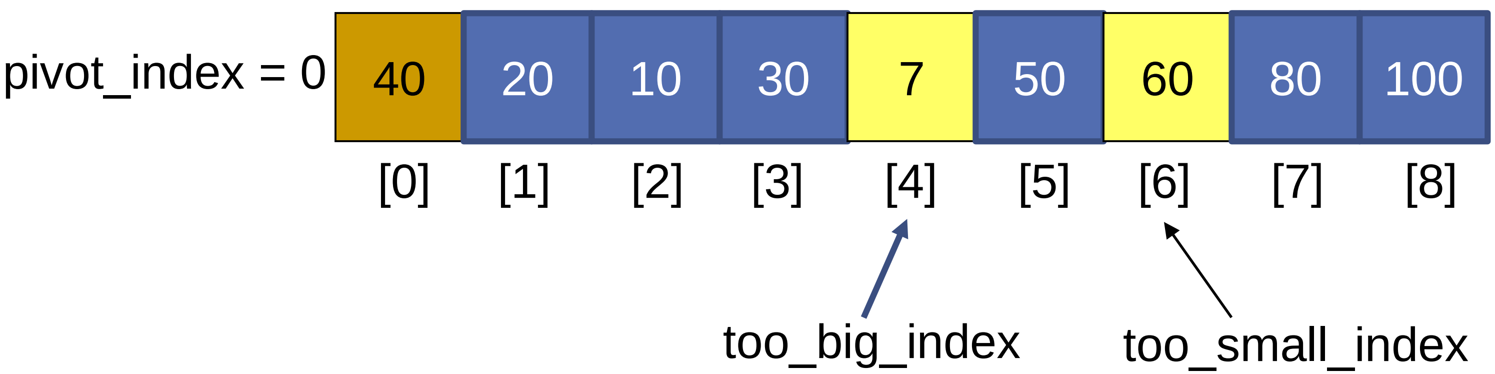
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



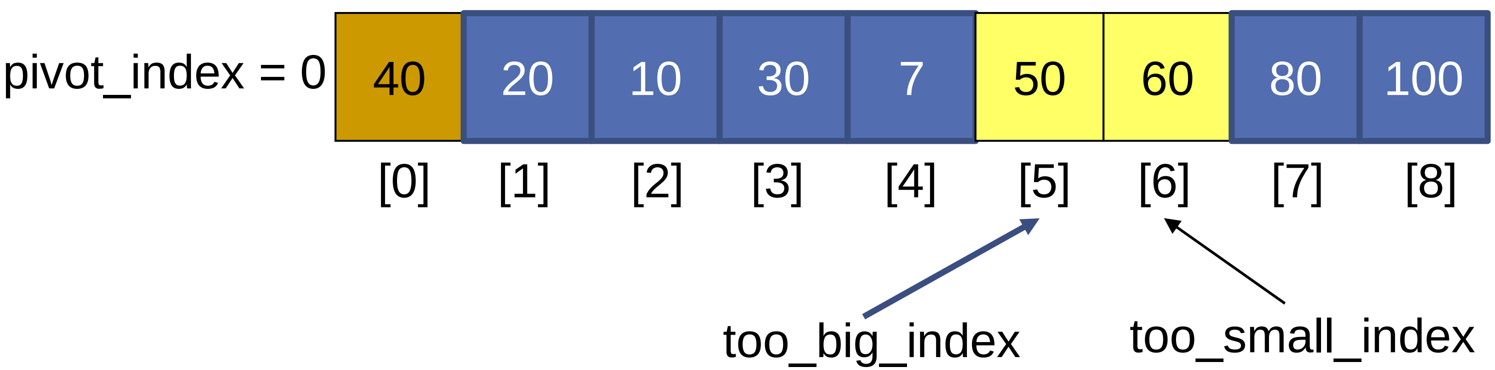
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



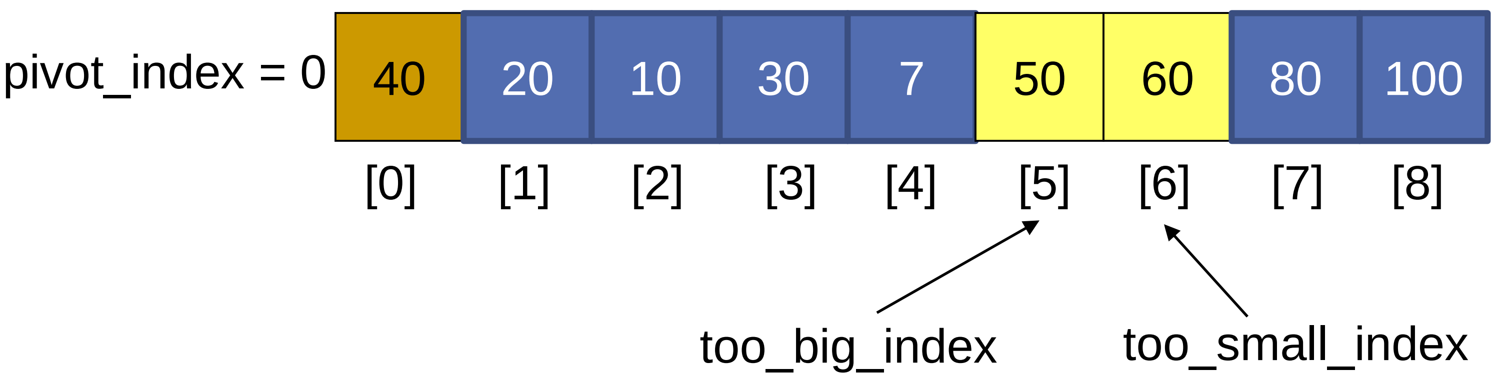
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



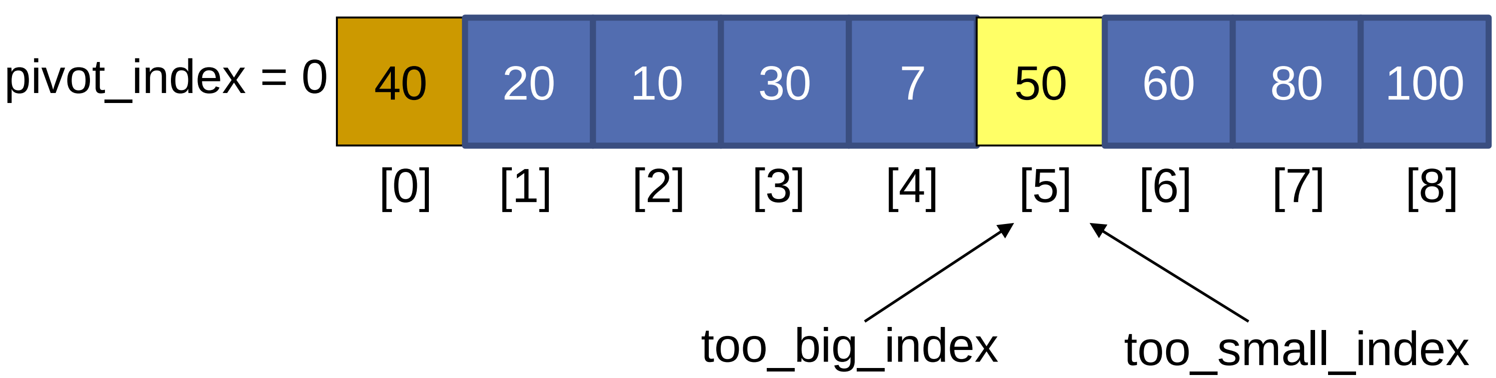
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



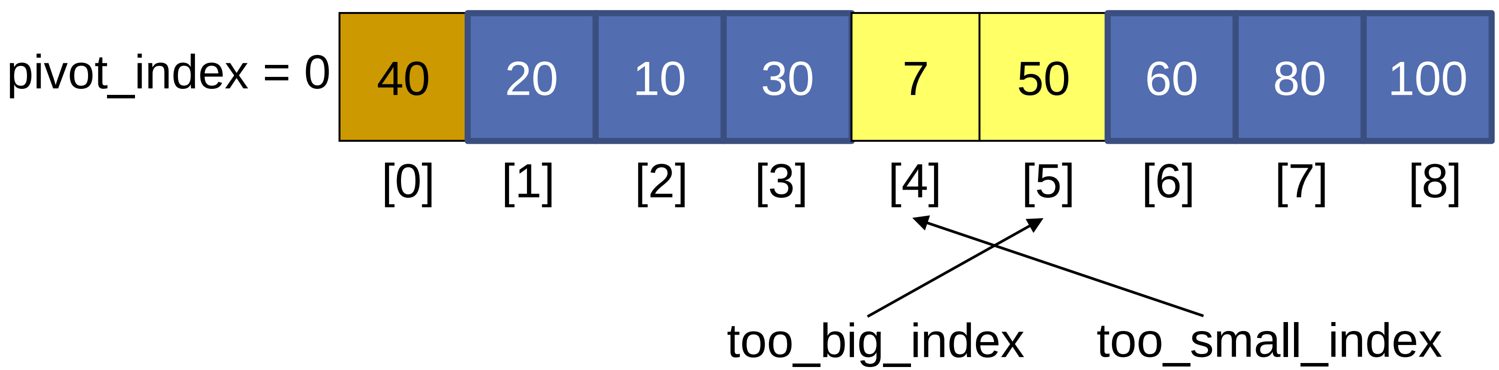
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



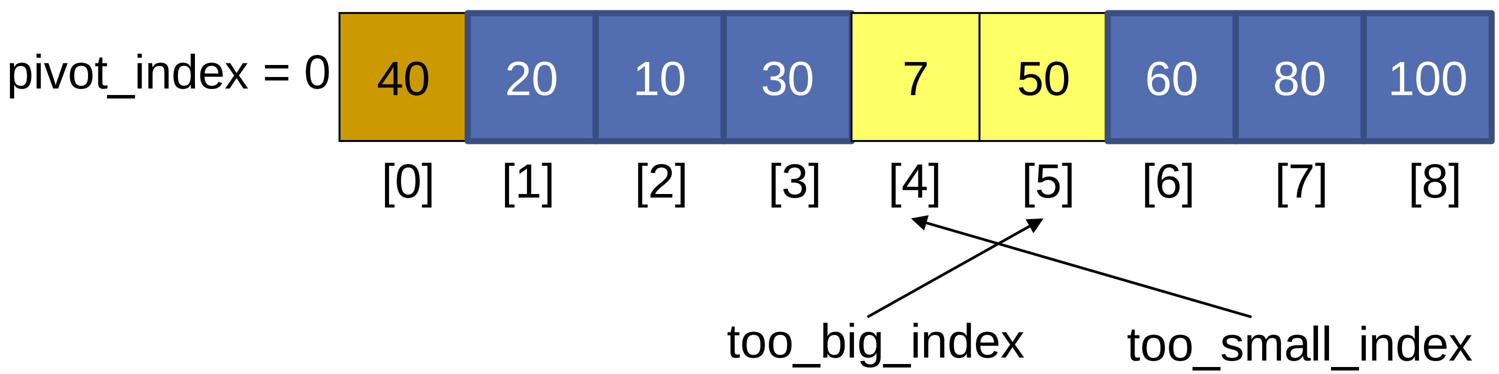
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



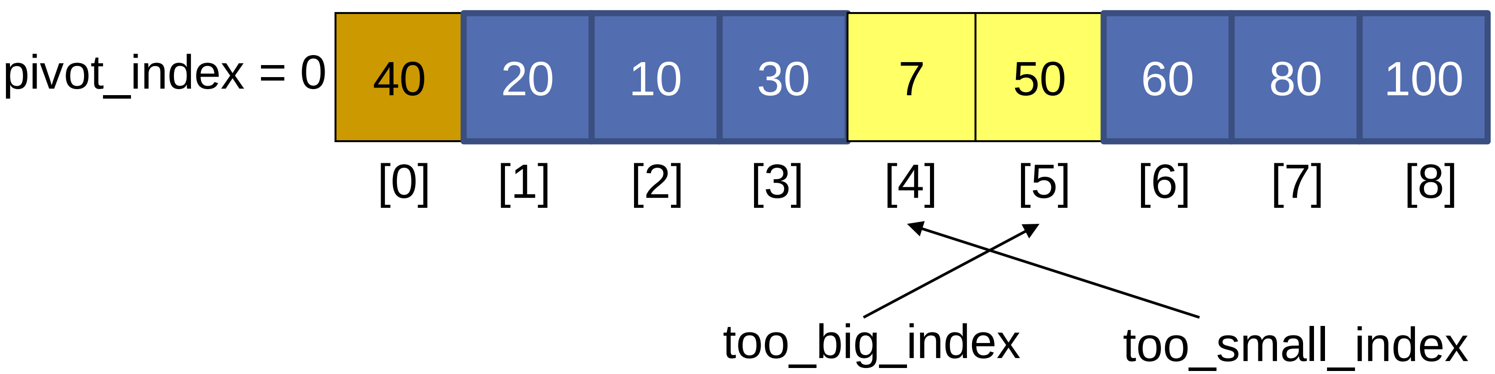
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



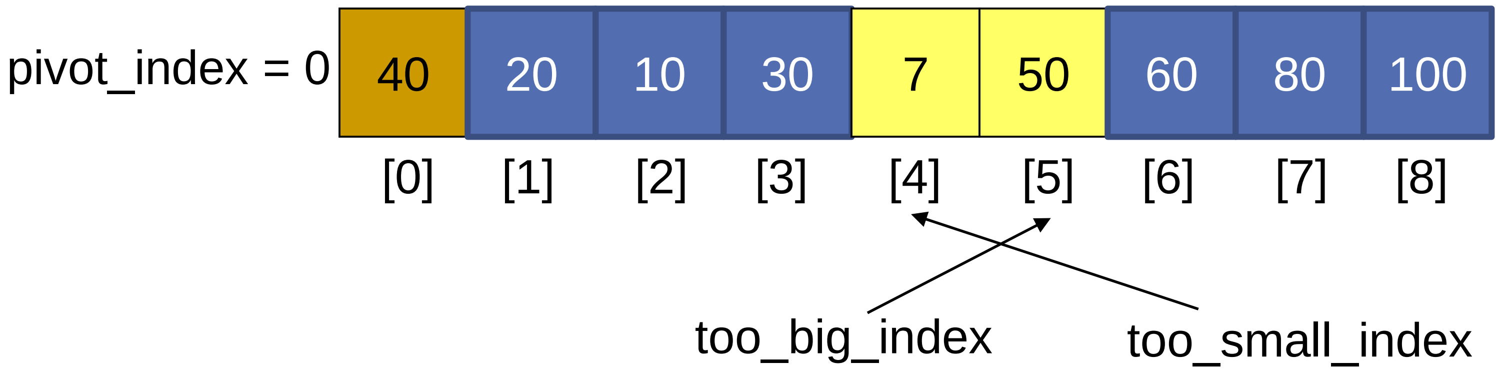
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.



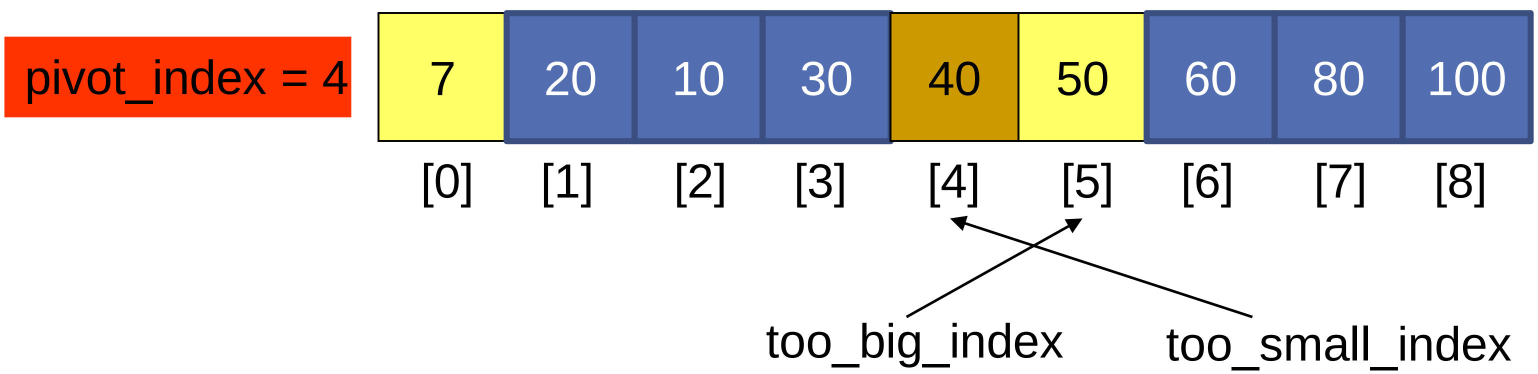
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]

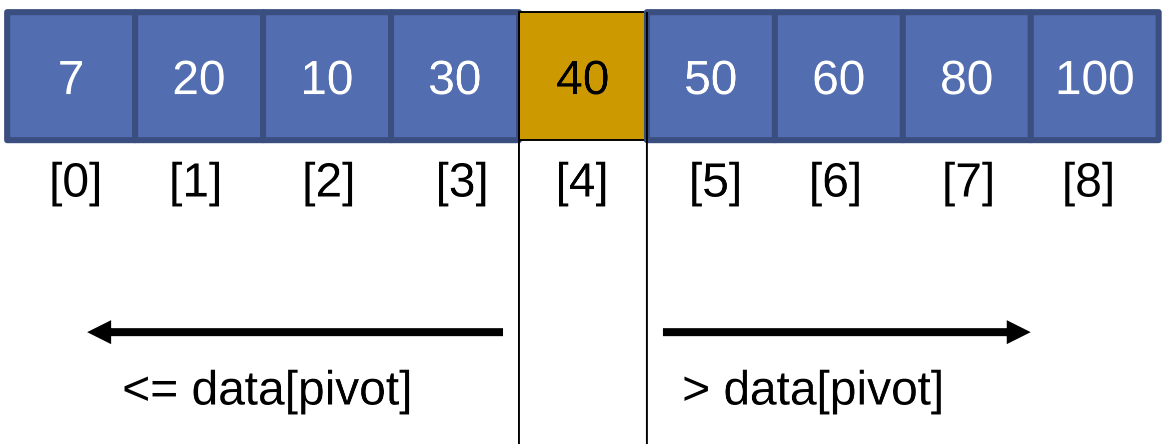


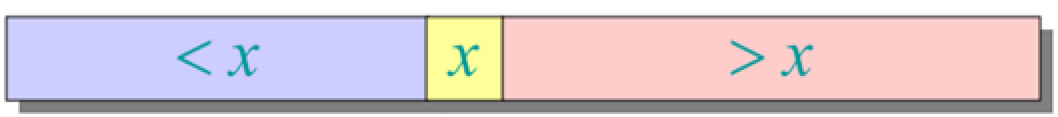
PARTITIONING EXAMPLE

1. While data[too_big_index] <= data[pivot]
   ++too_big_index
2. While data[too_small_index] > data[pivot]
   --too_small_index
3. If too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. While too_small_index > too_big_index, go to 1.
5. Swap data[too_small_index] and data[pivot_index]



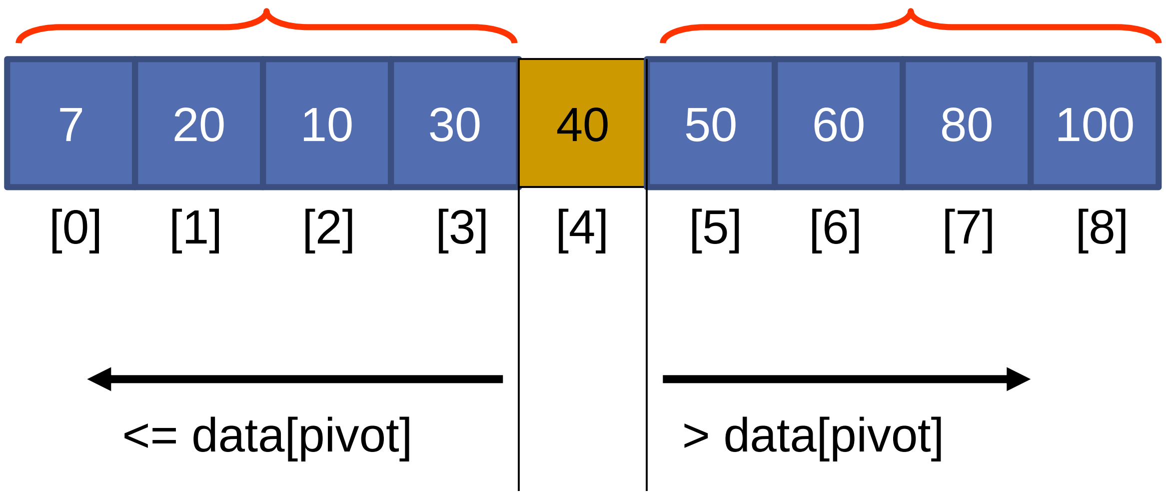
PARTITION RESULT





RUNTIME: O(n)

RECURSION: QUICKSORT SUB-ARRAYS



ANOTHER EXAMPLE OF PARTITIONING

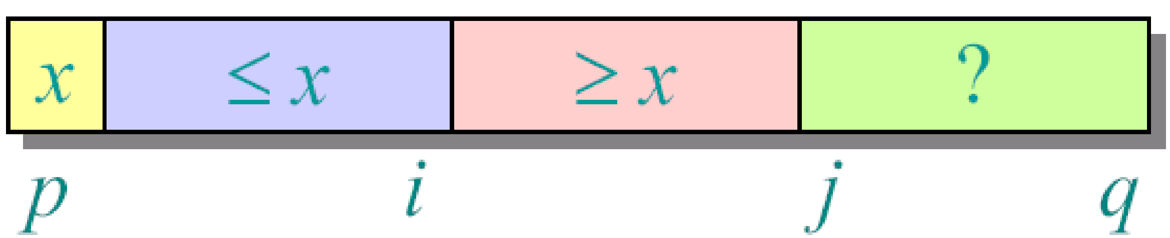
- choose pivot: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- search: 4 3 6 9 2 4 3 1 2 1 8 9 3 5 6
- swap: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- search: 4 3 3 9 2 4 3 1 2 1 8 9 6 5 6
- swap: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- search: 4 3 3 1 2 4 3 1 2 9 8 9 6 5 6
- swap: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- search: 4 3 3 1 2 2 3 1 4 9 8 9 6 5 6
- swap with pivot: 1 3 3 1 2 2 3 4 4 9 8 9 6 5 6

ANOTHER PARTITIONING SUBROUTINE

PARTITION(A, p, q) ▷ $A[p \,..\, q]$
  $x \leftarrow A[p]$ ▷ pivot $= A[p]$
  $i \leftarrow p$
  for $j \leftarrow p + 1$ **to** q
    do if $A[j] \leq x$
      then $i \leftarrow i + 1$
        exchange $A[i] \leftrightarrow A[j]$
  exchange $A[p] \leftrightarrow A[i]$
  return i

Running time $= O(n)$ for n elements.

Invariant:



EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
i	j						

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
i		j					

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
i		→	j				

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11

→ i j

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11

6	5	13	10	8	3	2	11
	i			j			

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
	i			→	j		

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
		i			j		

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
		i				j	

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
			i			j	

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
			i			→	j

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11

i → j

EXAMPLE OF PARTITIONING

6	10	13	5	8	3	2	11
6	5	13	10	8	3	2	11
6	5	3	10	8	13	2	11
6	5	3	2	8	13	10	11
2	5	3	6	8	13	10	11
			i				

LOMUTO PARTITIONING

- Select the last element A[*r*] in the subarray *A*[*p*..*r*] as the *pivot*.
- The array is partitioned into four (possibly empty) regions.
  1. *A*[*p*..*i*] — All entries in this region are **< *pivot***.
  2. *A*[*i*+1..*j* – 1] — All entries in this region are **> *pivot***.
  3. *A*[*r*] = *pivot*.
  4. *A*[*j*..*r* – 1] — Not known how they compare to *pivot*.
- These hold before each iteration of the *for* loop, and constitute a ***loop invariant***.

CORRECTNESS OF PARTITION

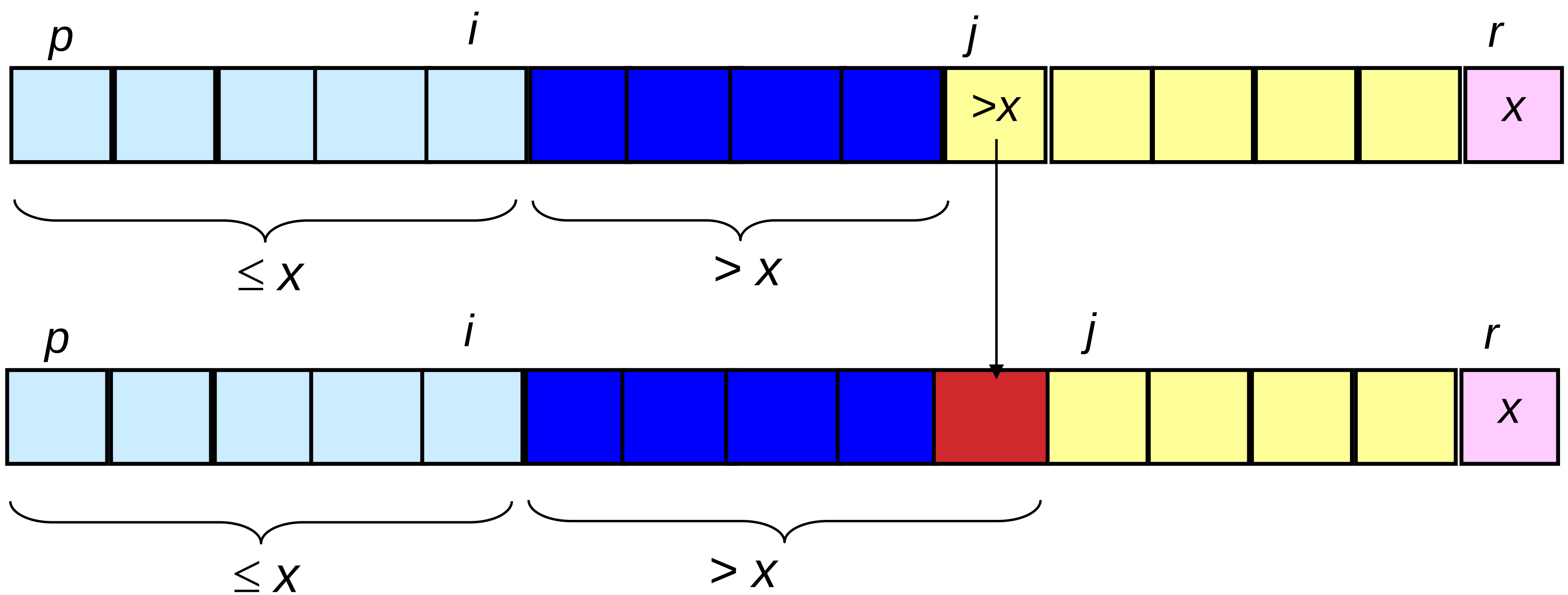
- **Use loop invariant.**
- **Initialization:**
  - Before first iteration
    - *A*[*p*..*i*] and *A*[*i*+1..*j* – 1] are empty
    - *r* is the index of the *pivot*
- **Maintenance:**
  - Use inductive hypothesis

1. *A*[*p*..*i*] **< *pivot***
2. *A*[*i*+1..*j* – 1] **> *pivot***
3. *A*[*r*] = *pivot*

```
Partition(A, p, r)
    x := A[r], i := p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j];
    A[i + 1] ↔ A[r];
    return i + 1;
```

CORRECTNESS OF PARTITION

Case 1: *A*[*j*] > *x* Increment *j* only.

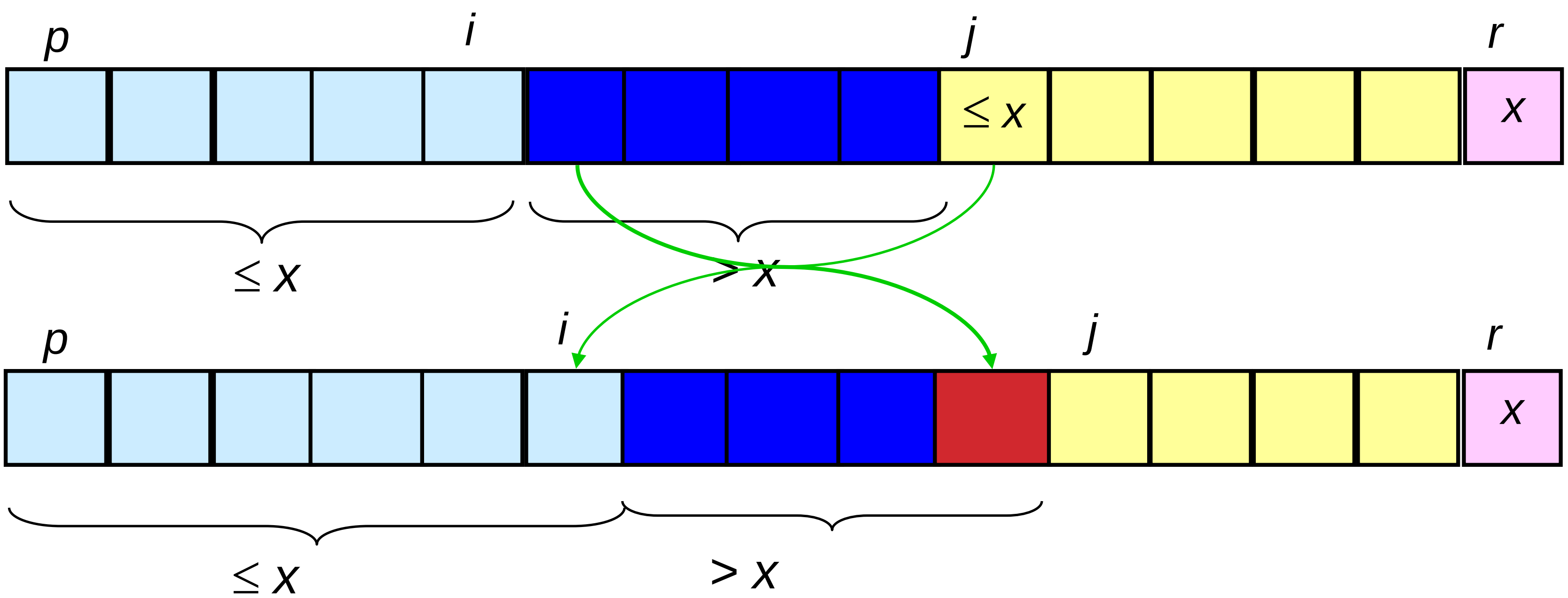


1. *A*[*p*..*i*] **< *pivot***
2. *A*[*i*+1..*j* – 1] **> *pivot***
3. *A*[*r*] = *pivot*

CORRECTNESS OF PARTITION

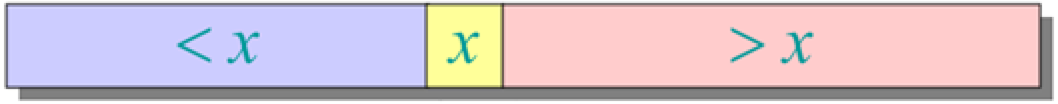
- **Case 2:** ***A[j] ≤ x***
  - Increment *i*
  - Swap *A*[*i*] and *A*[*j*]
  - Increment *j*

1. *A*[*p*..*i*] **< *pivot***
2. *A*[*i*+1..*j* – 1] **> *pivot***
3. *A*[*r*] = *pivot*



CORRECTNESS OF PARTITION

- **Termination:**
  - When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases:
    - $A[p..i] \leq$ ***pivot***
    - $A[i+1..j-1] >$ ***pivot***
    - $A[r] =$ ***pivot***
- **At last, swap $A[i+1]$ and $A[r]$.**
  - Before swap $A[i+1] >$ pivot
  - After swap *pivot* moves from the end of the array to between the two subarrays.
  - Thus, procedure *partition* correctly performs the divide step.



ANALYZING QUICKSORT

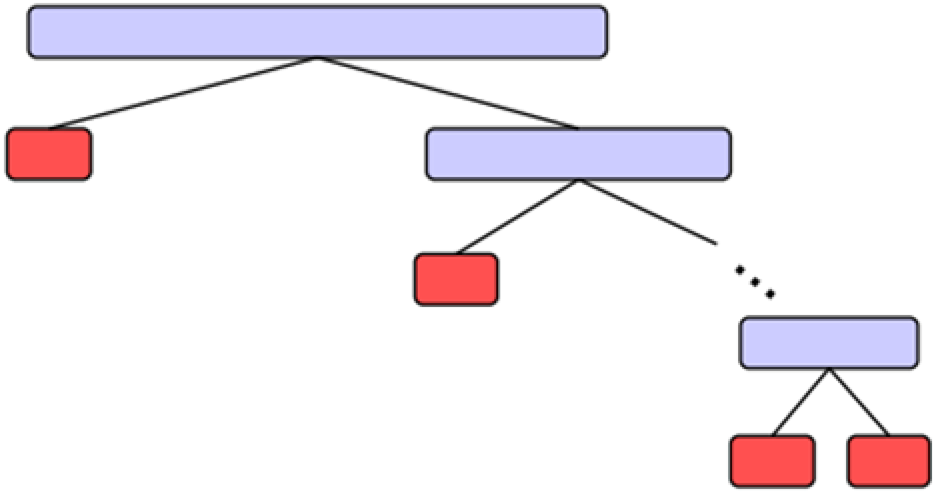
- ***What will be the worst case for the algorithm?***
- ***What will be the best case for the algorithm?***
- ***Which one is more likely?***
- ***Will any particular input cause the worst case?***

ANALYZING QUICKSORT

- ***What will be the worst case for the algorithm?***
  - Partition is always unbalanced
- ***What will be the best case for the algorithm?***
  - Partition is perfectly balanced
- ***Which one is more likely?***
  - Balanced, except...
- ***Will any particular input cause the worst case?***
  - Yes: Already-sorted or reverse-sorted input

WORST CASE

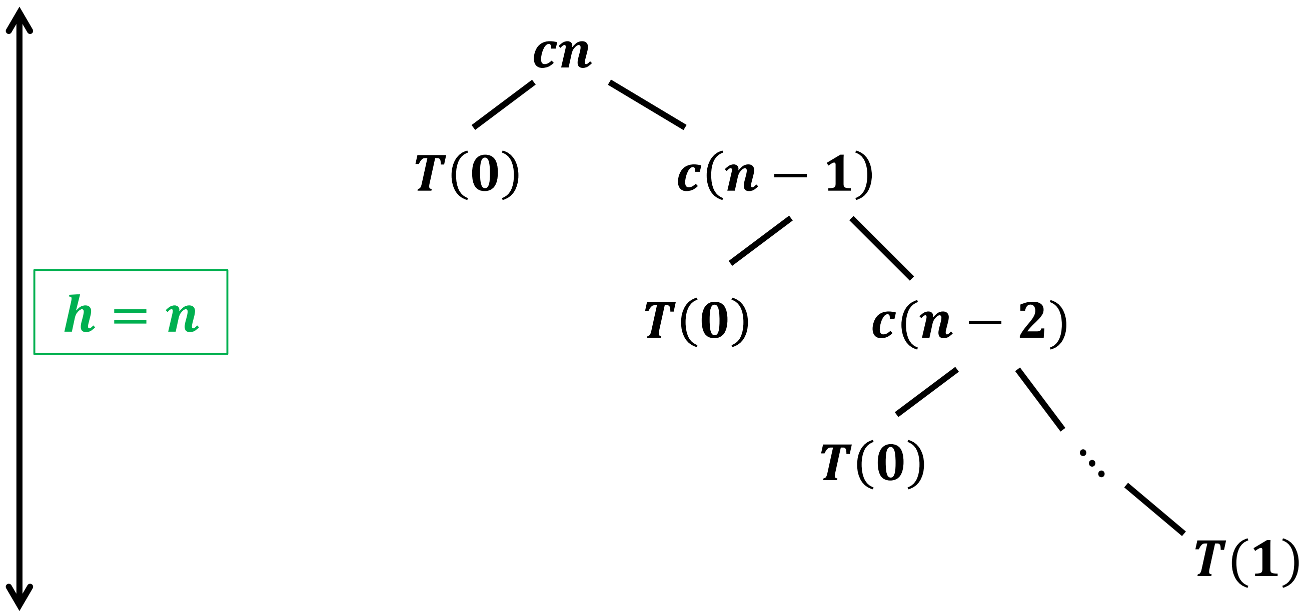
Depth	Partition Time
0	n
1	n-1
…	…
n-1	1



- Total: n + (n + 1) + …… + 2 + 1
- Thus, the worst-case running time of quicksort is $O(n^2)$

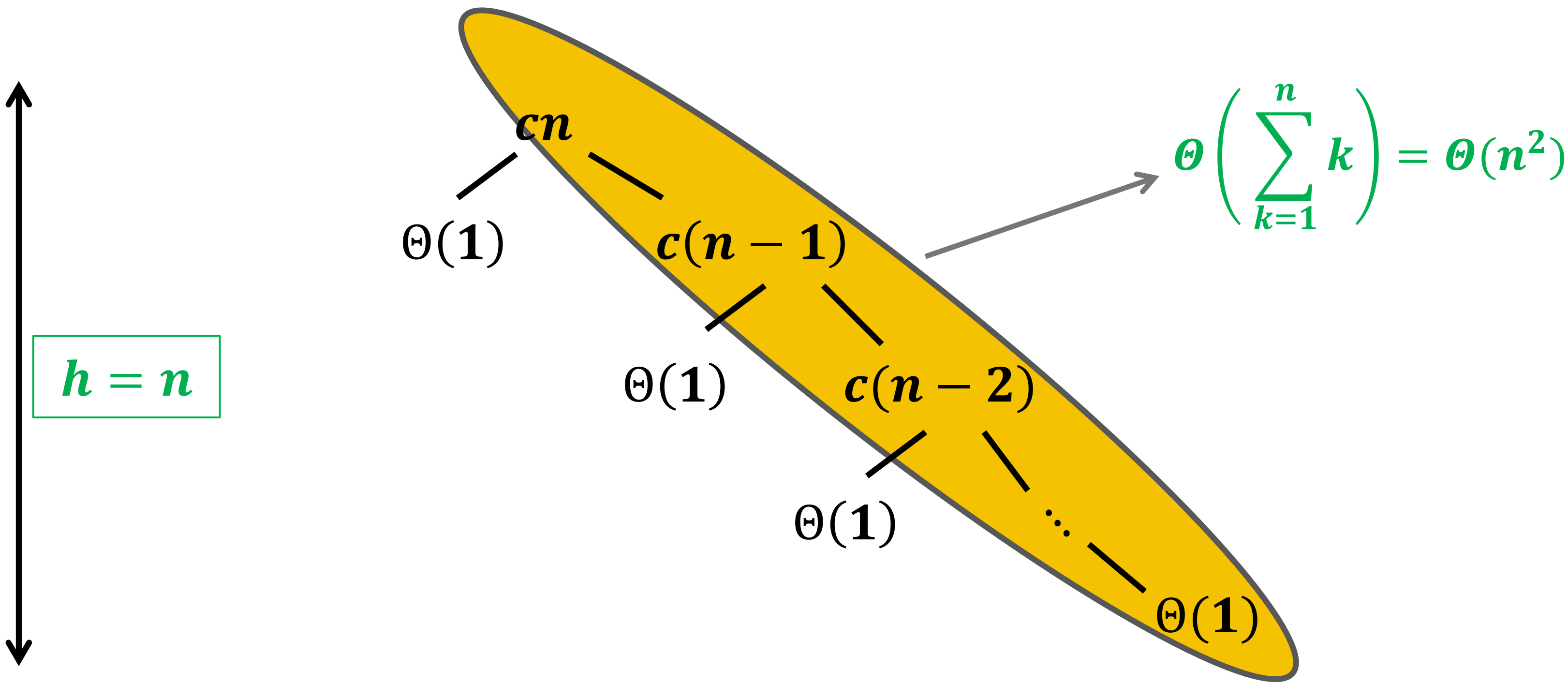
WORST-CASE RECURSION TREE

$$T(n) = T(0) + T(n-1) + cn$$



WORST-CASE RECURSION TREE

$$T(n) = T(1) + T(n-1) + cn$$



$$T(n) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

BEST-CASE

- Partition into two halves
  - $T(n) = 2\,T(n/2) + cn$
  - $T(n) = \Theta(n \log n)$

BEST-CASE

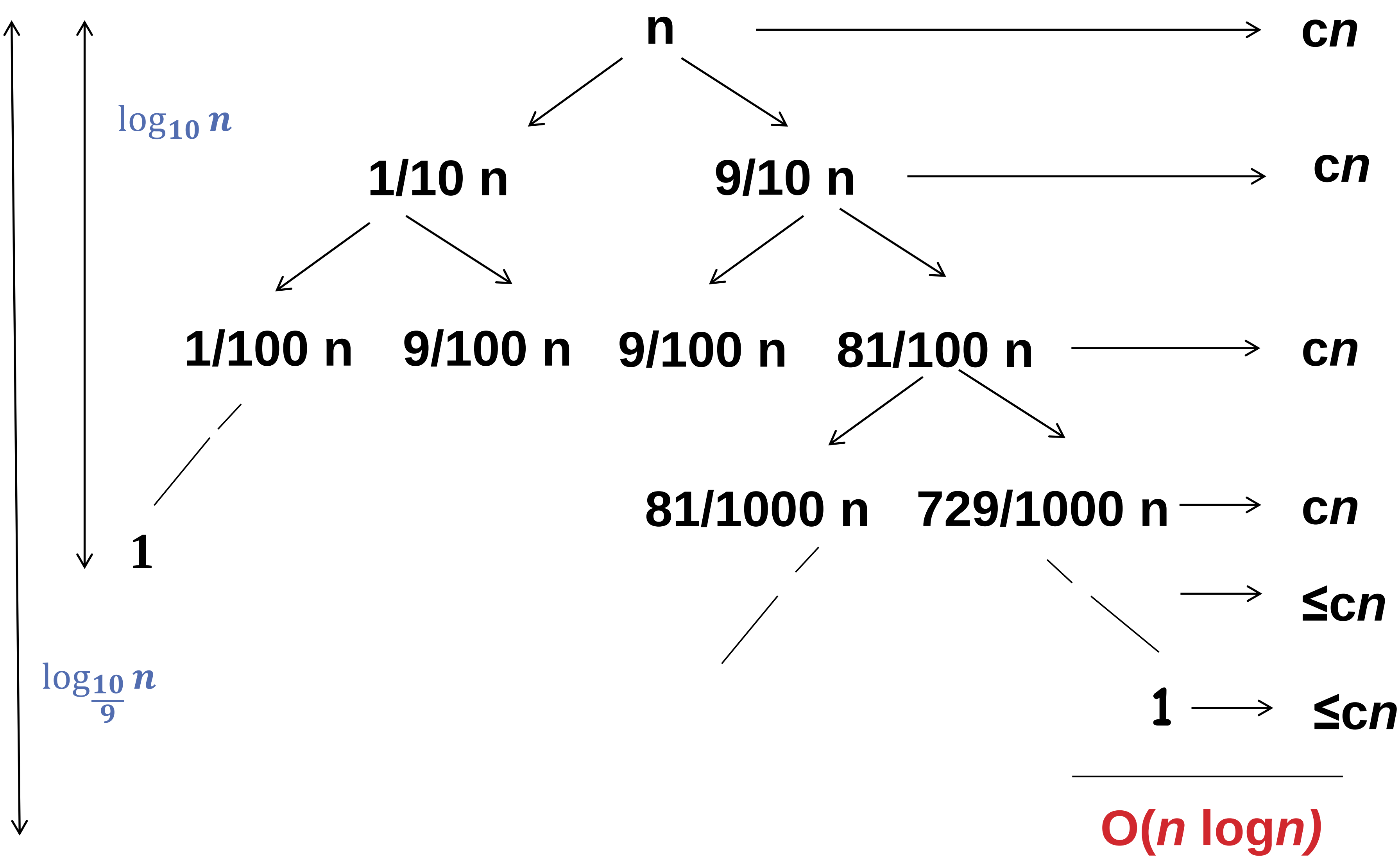
- Partition into two halves
  - $T(n) = 2\,T(n/2) + cn$
  - $T(n) = \Theta(n \log n)$

- What if the split is always $\frac{1}{10} : \frac{9}{10}$?
  - $T(n) = ?$

BEST-CASE

- Partition into two halves
  - T(n) = 2 T(n/2) + cn
  - T(n) = Θ(n log n)

- What if the split is always $\frac{1}{10} : \frac{9}{10}$?
  - $T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + cn$


n
cn
1/10 n
9/10 n
cn
1/100 n
9/100 n
9/100 n
81/100 n
cn
81/1000 n
729/1000 n
cn
≤cn
1
1
≤cn
$\log_{10} n$
$\log_{\frac{10}{9}} n$
O(n logn)

INTUTION FOR AVERAGE CASE

- **Assumption:** All permutations equally likely (realistic??)
- **Unlikely:** Splits always the same way at every level
- **Expectation:** Some splits will be reasonably balanced.

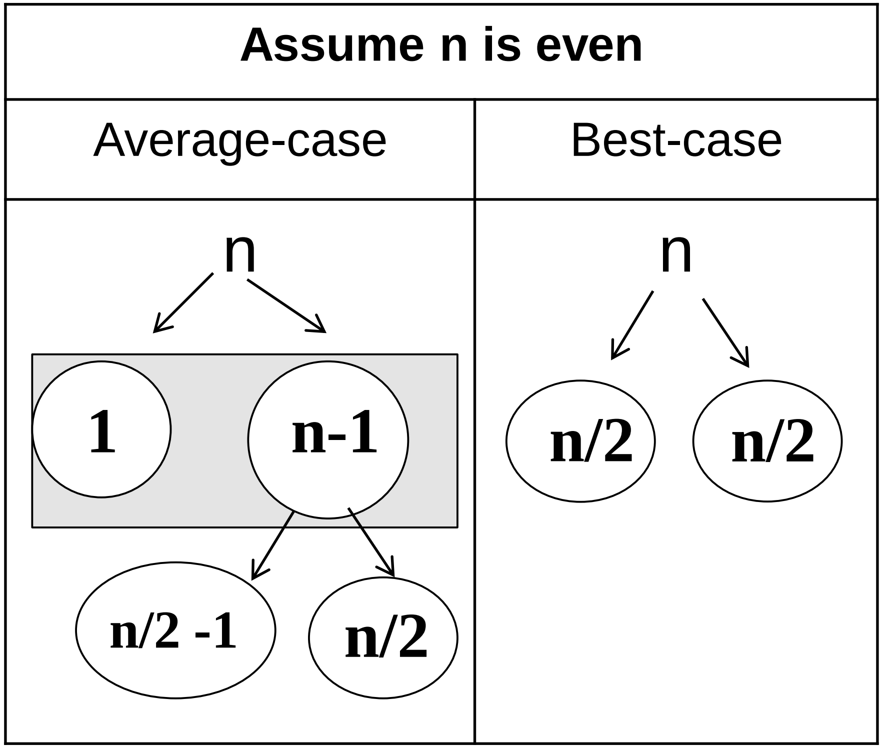
  Some splits will be fairly unbalanced.
- **Average case:** A mix of good and bad splits.

  Good and bad splits distributed randomly through the tree.

  Good and bad splits occur in the alternate levels of the tree.
- **Good split:** Best-case split (n/2 and n/2)
- **Bad split:** Worst-case split (1 and n-1)

INTUTITION FOR AVERAGE CASE

- Two successive levels of average-case produce a half-and-half split.
- Same result as a single level of the best case.
- Thus, after spending $\Theta(n) + \Theta(n-1) = \Theta(n)$ partitioning cost, we reach the best case.
- Total tree height doubles (2 log n).
- Runtime still $\Theta(n \log n)$



CONCLUSIONS

- **Average-case analysis**
  - Hard in general
  - Need to make assumptions on the input distribution or workload
    - May not represent the real scenario
  - Somewhere between best- and worst- cases
    - May be as bad as the worst case (e.g., insertion sort)
    - Or as good as the best case (e.g., quicksort)

- **Next: What if we can enforce input distribution?**