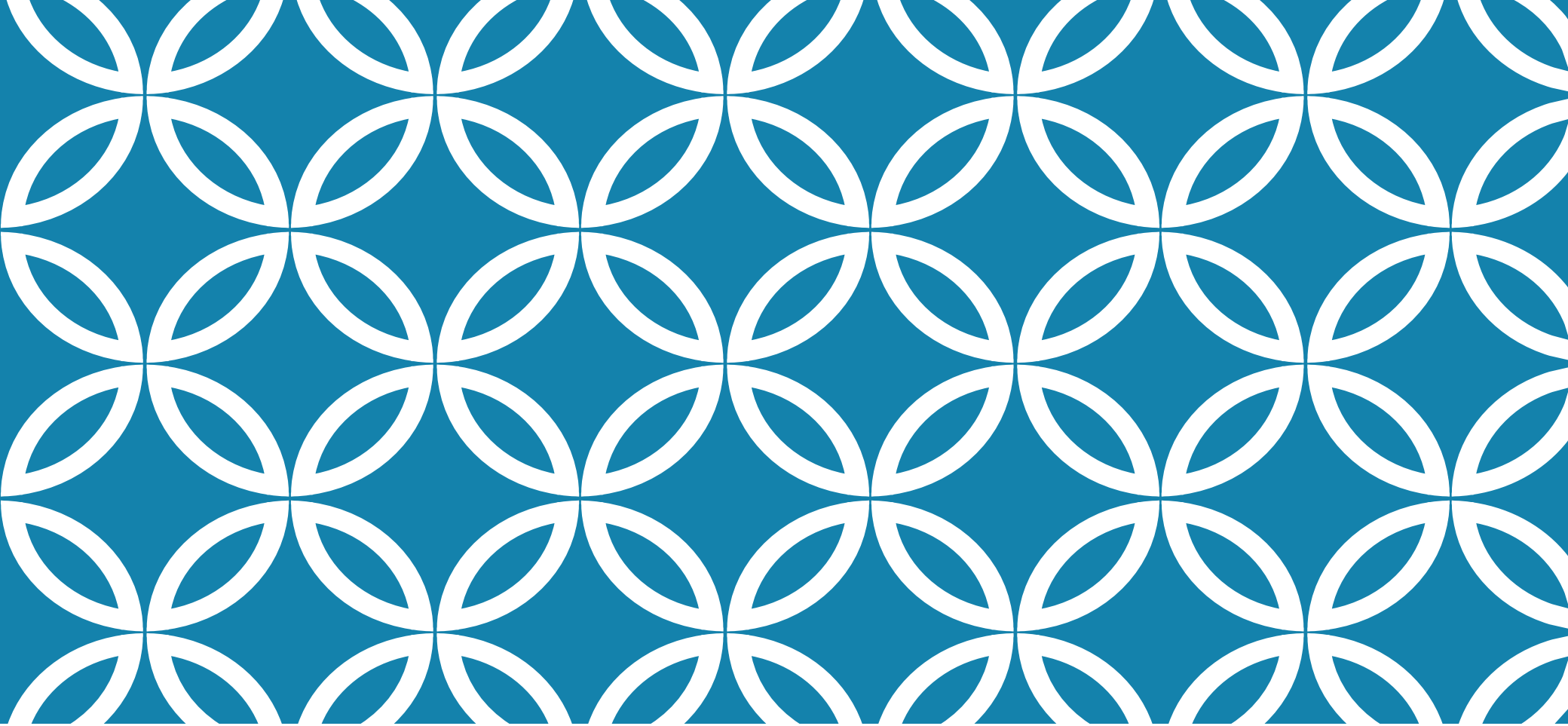
095946- ADVANCED ALGORITHMS AND PARALLEL PROGRAMMING

Fabrizio Ferrandi

a.a. 2021-2022



QUICKSORT

- DIVIDE AND CONQUER
- PARTITIONING
- WORST-CASE ANALYSIS
- INTUITION
- RANDOMIZED QUICKSORT
- ANALYSIS

Material adapted from Erik D. Demaine and Charles E. Leiserson slides

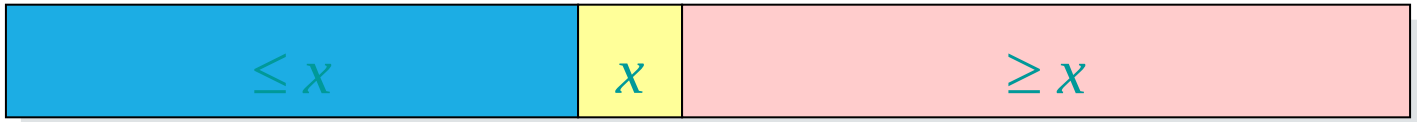
QUICKSORT

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).

DIVIDE AND CONQUER

Quicksort an n-element array:

1. *Divide:* Partition the array into two subarrays around a *pivot* x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.



2. *Conquer:* Recursively sort the two subarrays.
3. *Combine:* Trivial.

Key: *Linear-time partitioning subroutine.*

PARTITIONING SUBROUTINE

```
PARTITION(A, p, q)  ▷ A[p . . q]
    x ← A[p]        ▷ pivot = A[p]
    i ← p
    for j ← p + 1 to q
        do if A[j] ≤ x
                then  i ← i + 1
                      exchange A[i] ↔ A[j]
    exchange A[p] ↔ A[i]
    return i
```

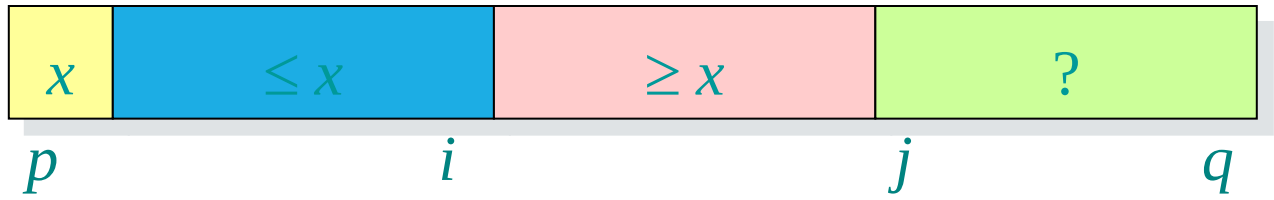
we move the values according to this pivot -> in that case is gonna to be the first element

in the worse case we are spending O(n)

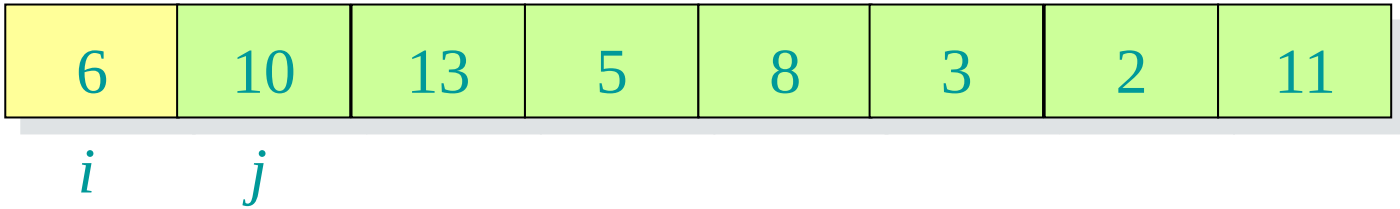
i'm increasing i

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

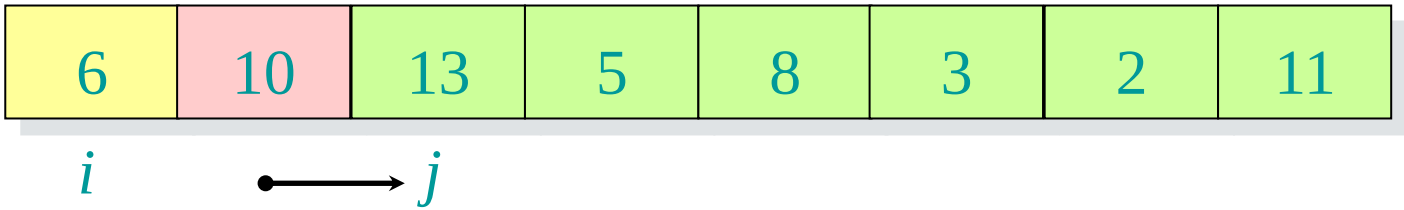
Invariant:



EXAMPLE OF PARTITIONING

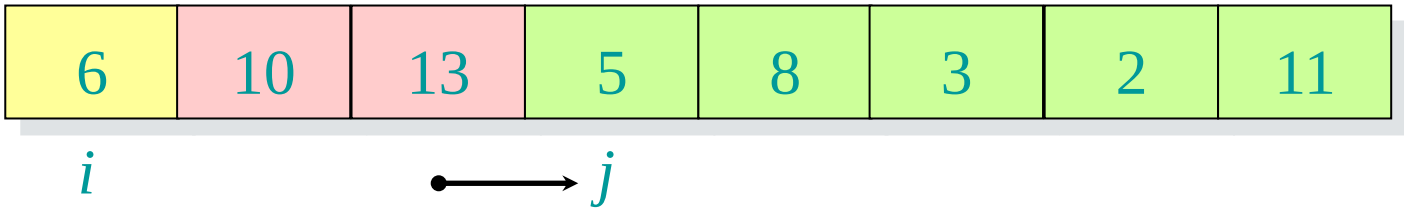


EXAMPLE OF PARTITIONING

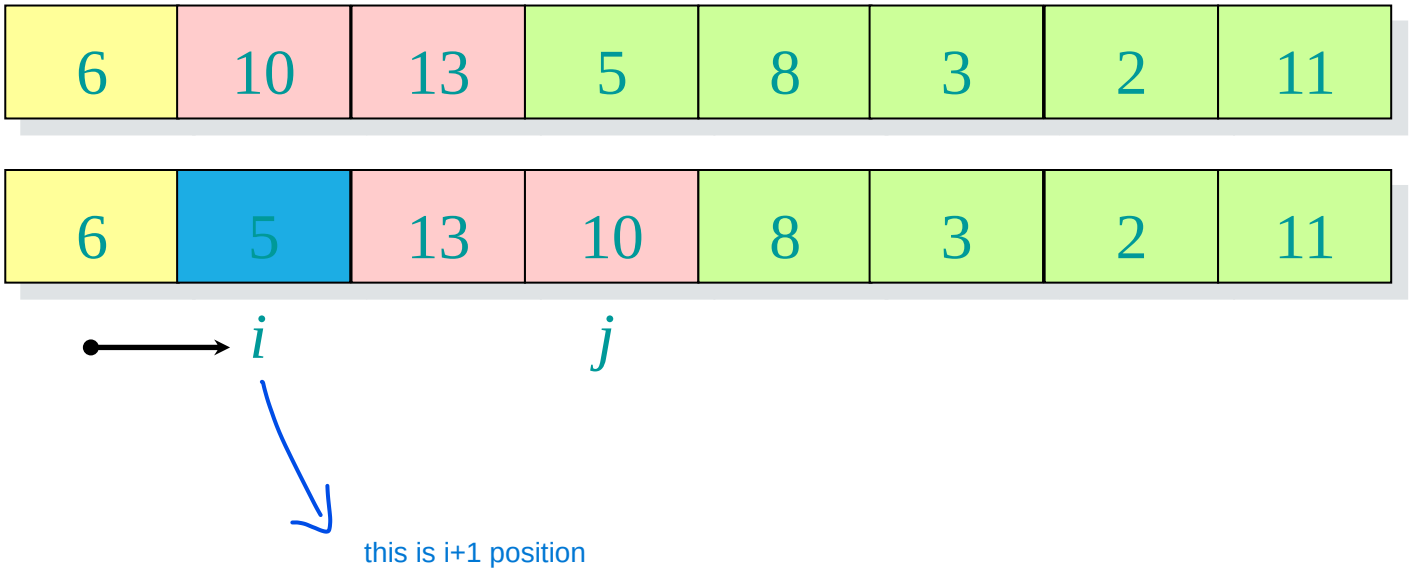


10 is greater so we move on

EXAMPLE OF PARTITIONING



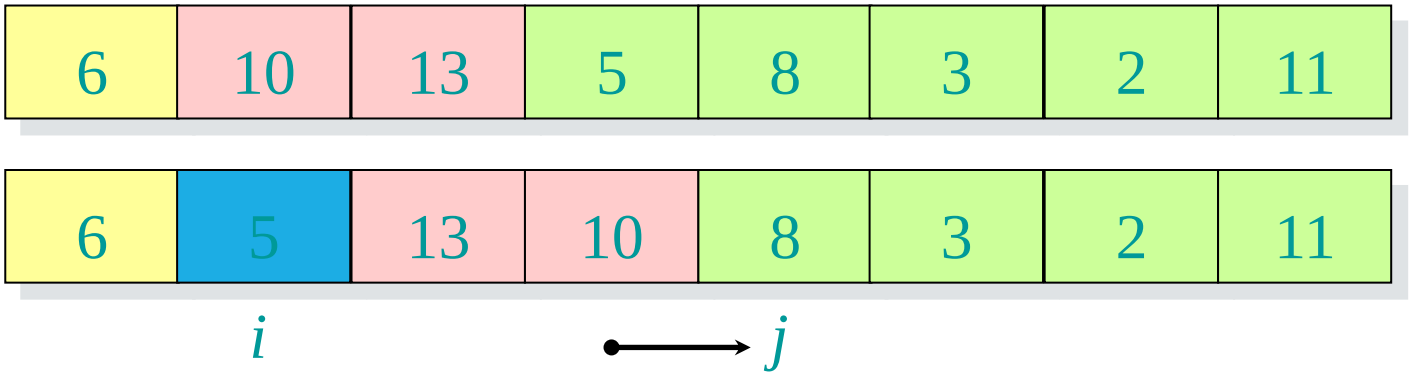
EXAMPLE OF PARTITIONING



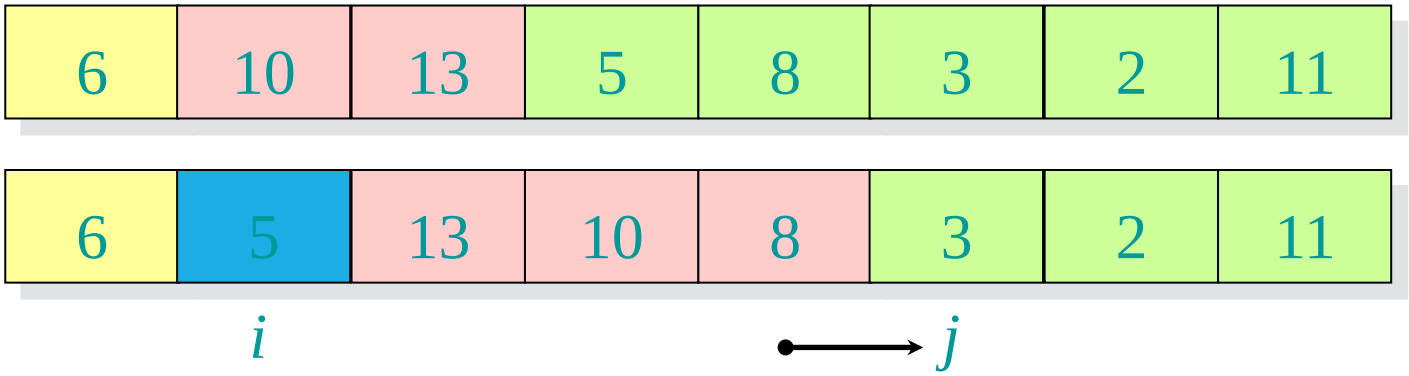
this is i+1 position

5<6

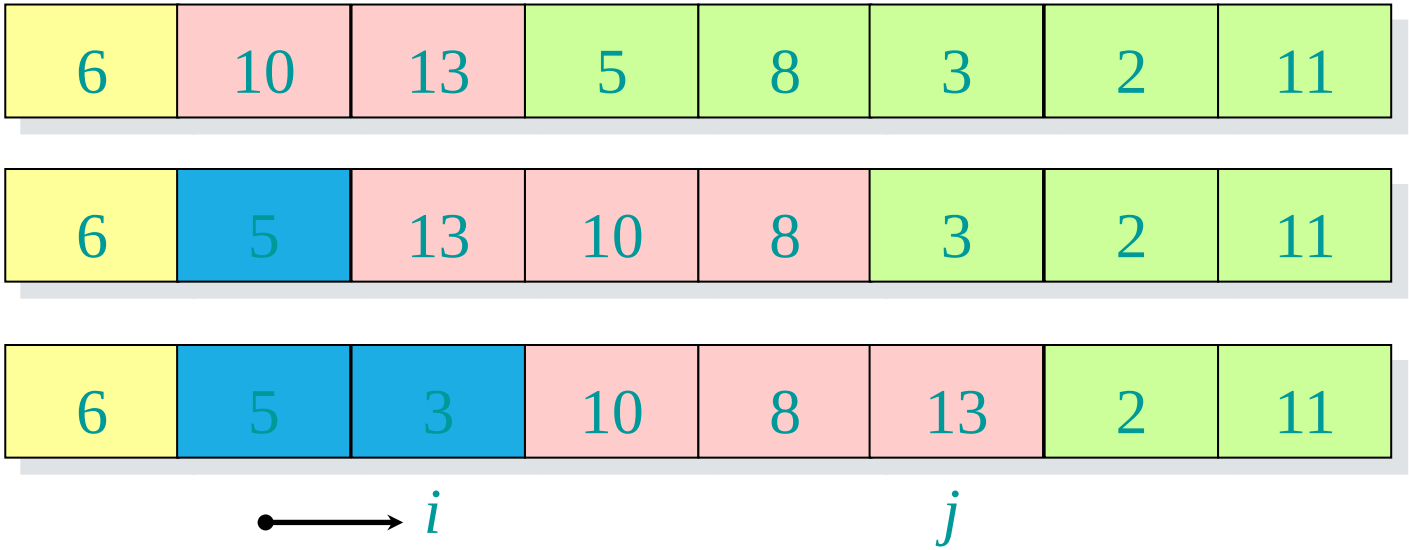
EXAMPLE OF PARTITIONING



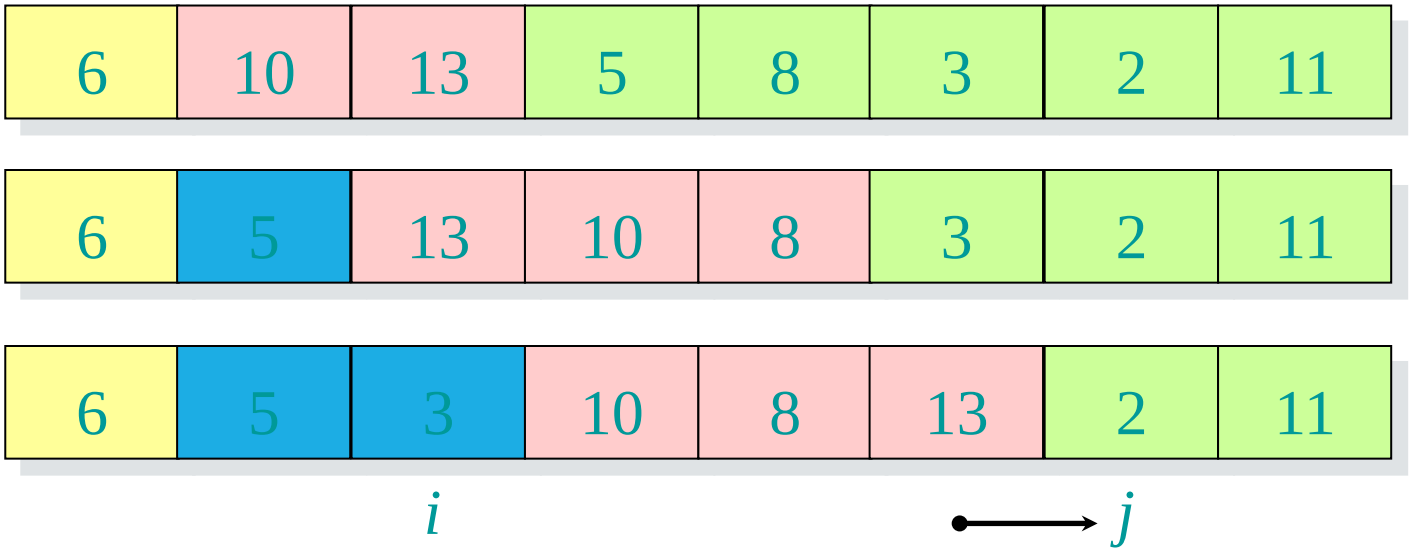
EXAMPLE OF PARTITIONING



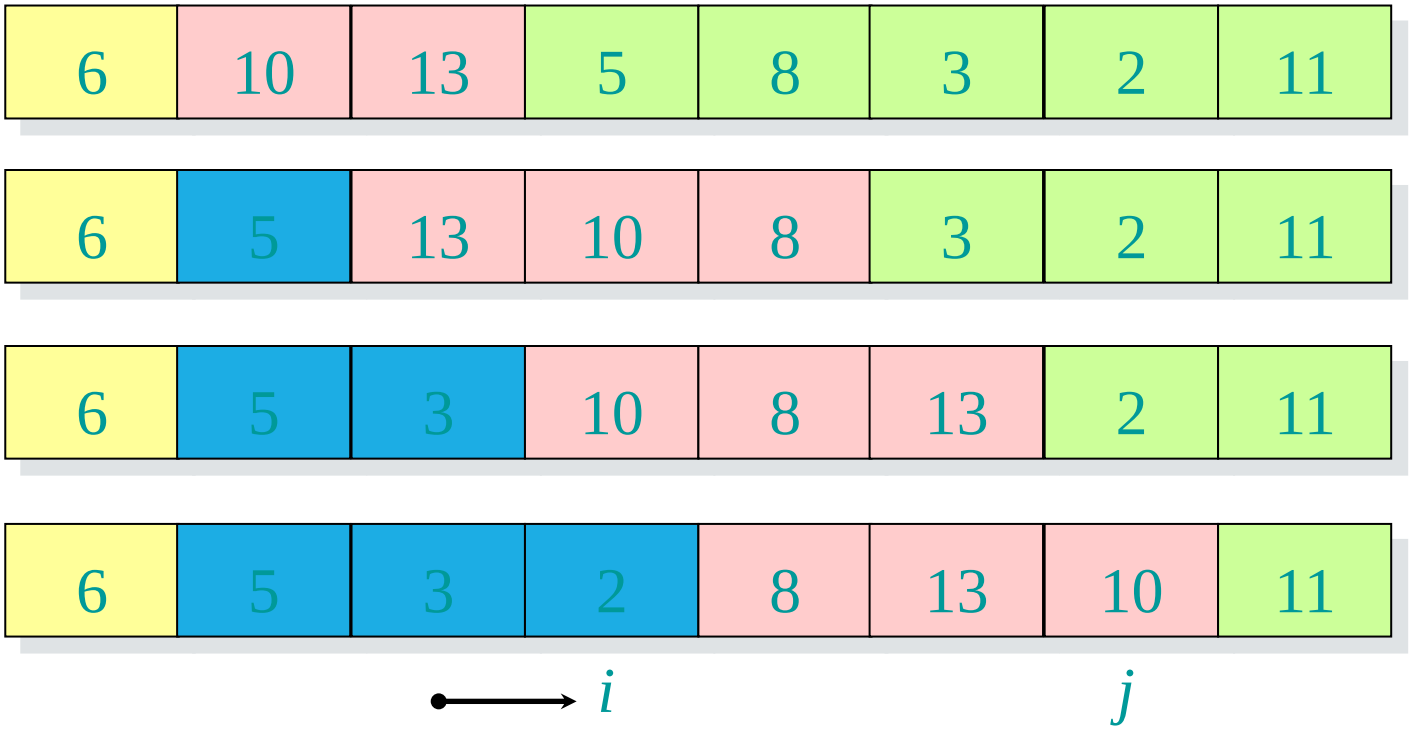
EXAMPLE OF PARTITIONING



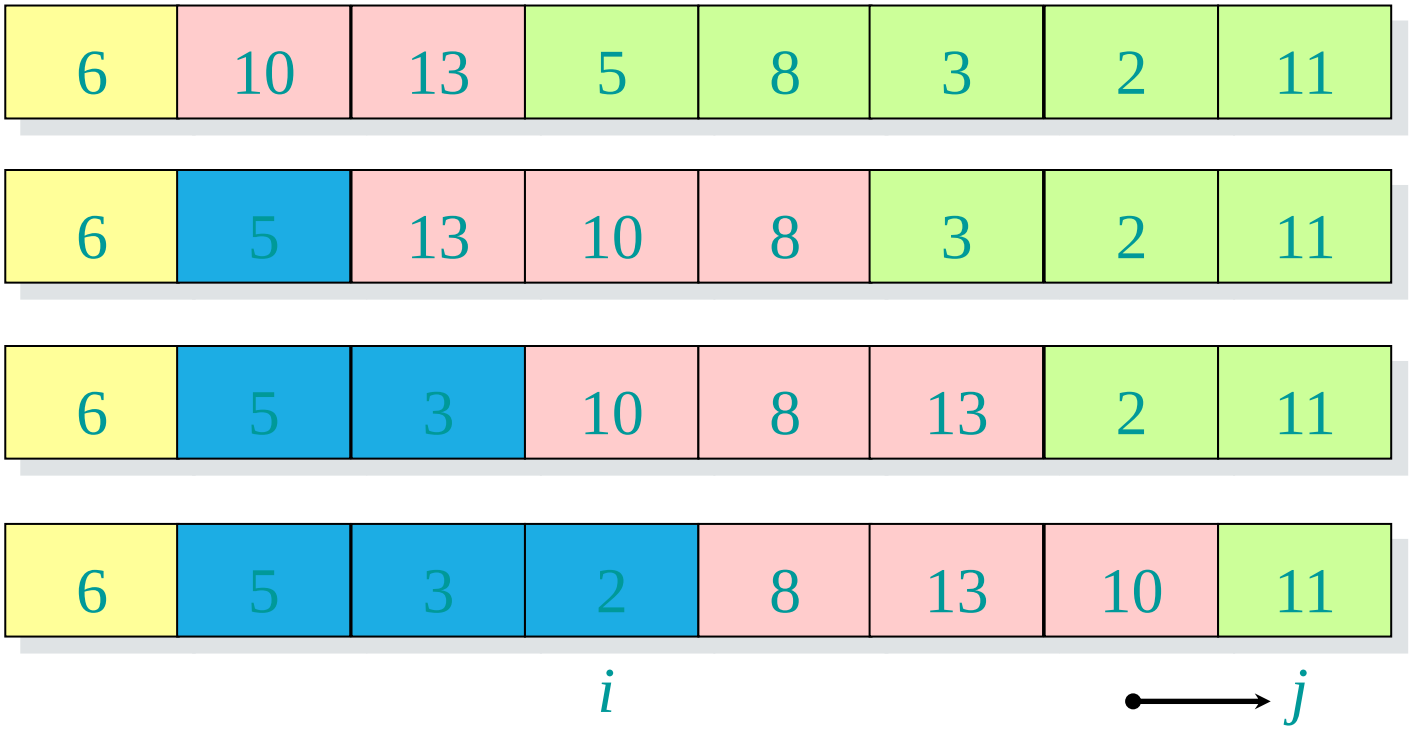
EXAMPLE OF PARTITIONING



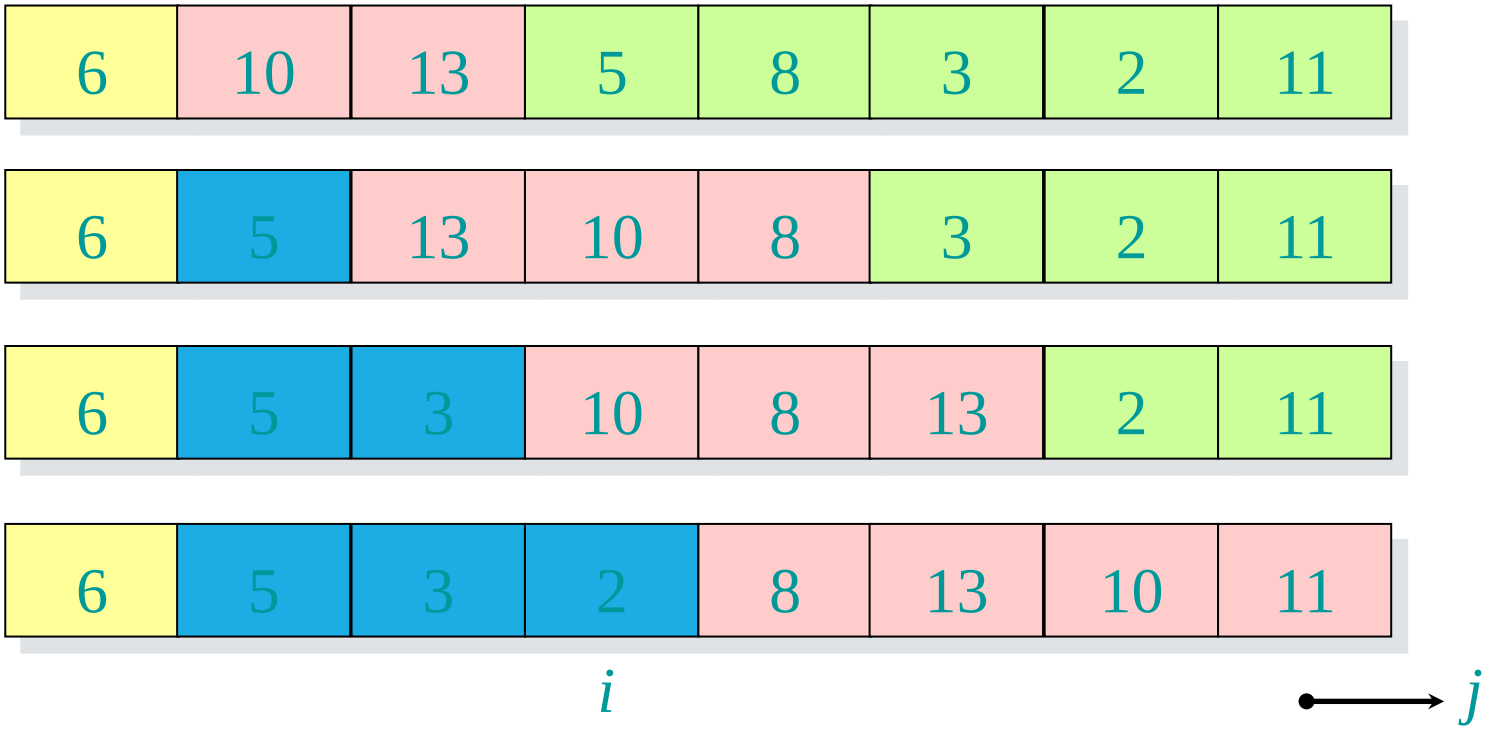
EXAMPLE OF PARTITIONING



EXAMPLE OF PARTITIONING

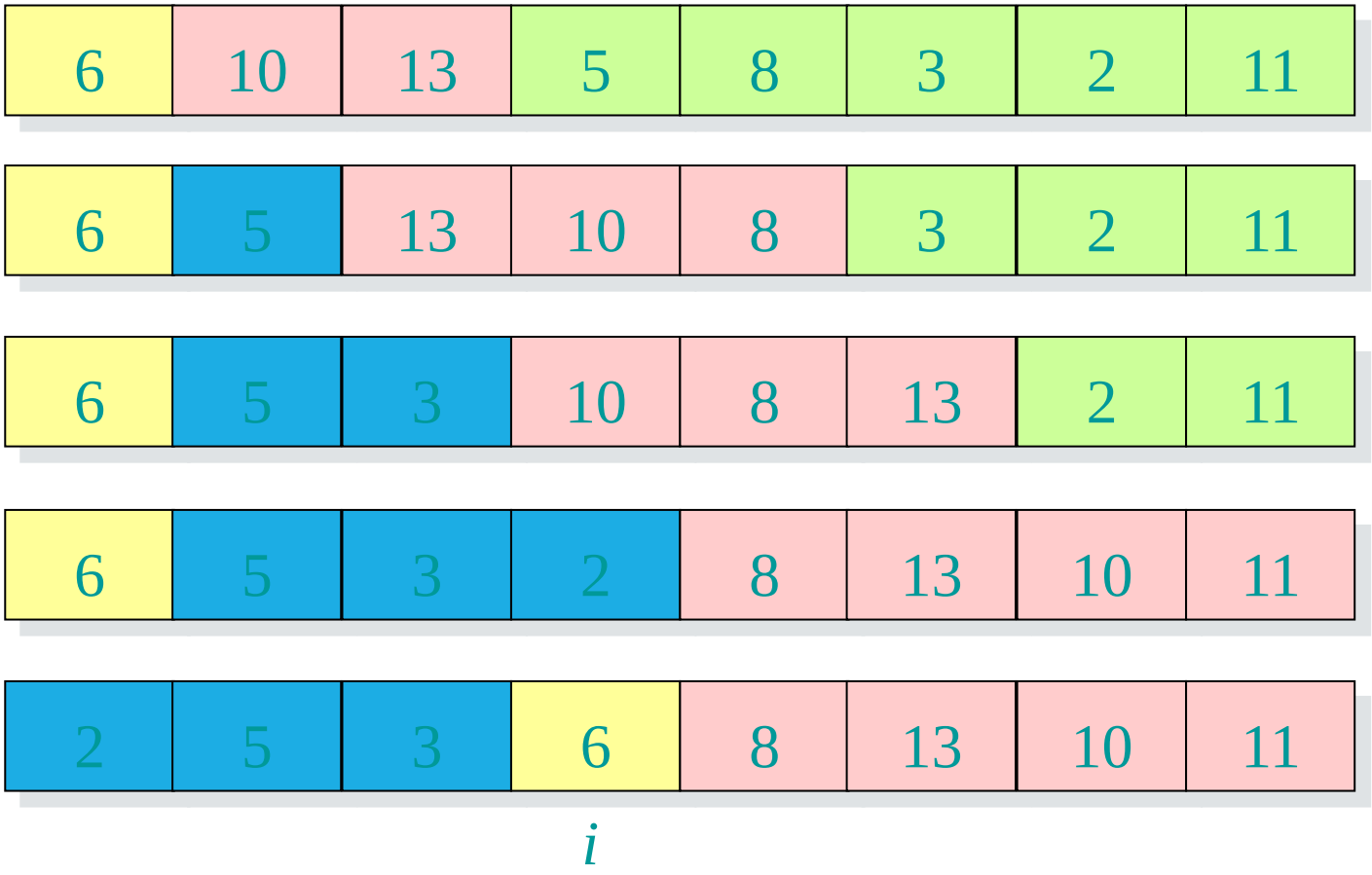


EXAMPLE OF PARTITIONING



ora facciamo A[p] <-> A[i]

EXAMPLE OF PARTITIONING



PSEUDOCODE FOR 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 ed r sono gli estremi dell'array

Initial call: QUICKSORT(*A*, 1, *n*)

ANALYSIS OF QUICKSORT

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.

WORST-CASE OF QUICKSORT

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

for the partitioning phase

perchè gli elementi sono già ordinati

$$
\begin{aligned}
T(n) &= T(0) + T(n-1) + \Theta(n) \\
&= \Theta(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2)
\end{aligned}
$$

(arithmetic series)

WORST-CASE RECURSION TREE

$T(n) = T(0) + T(n–1) + cn$

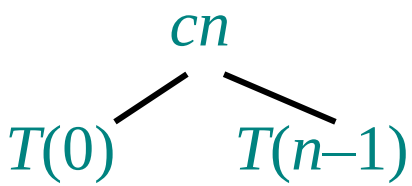
WORST-CASE RECURSION TREE

$T(n) = T(0) + T(n–1) + cn$

$T(n)$

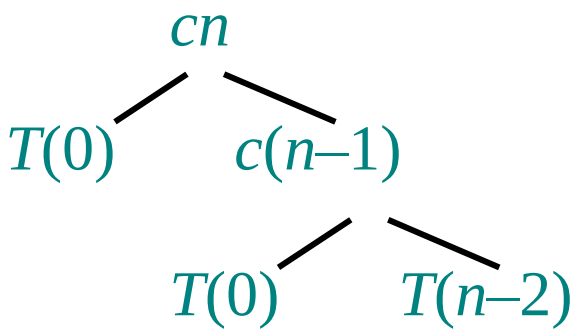
WORST-CASE RECURSION TREE

$T(n) = T(0) + T(n–1) + cn$



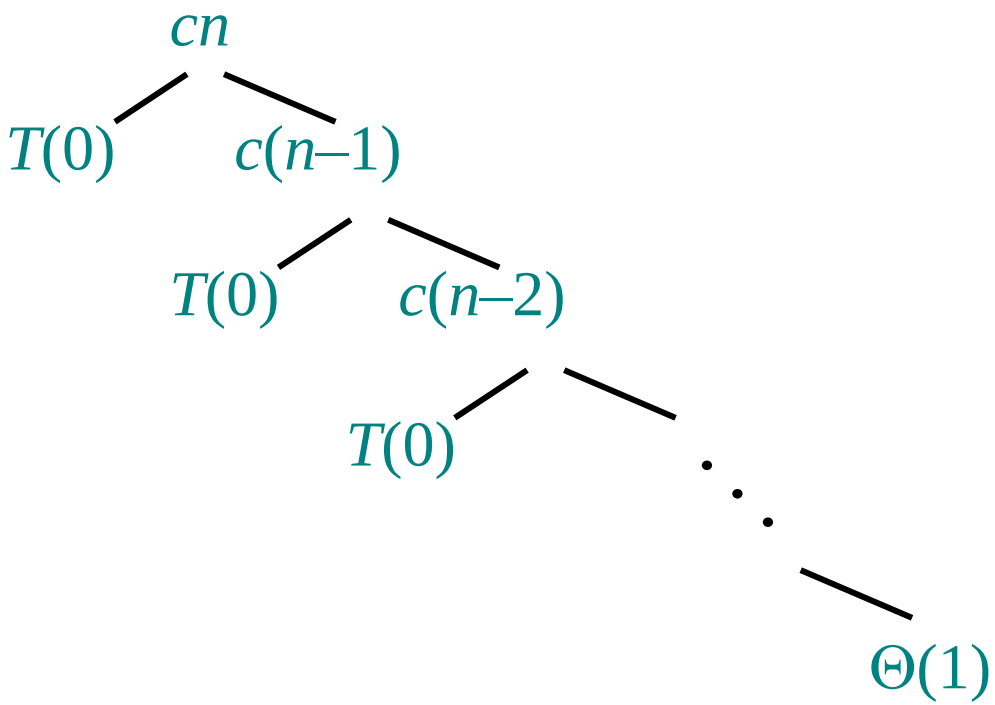
WORST-CASE RECURSION TREE

$T(n) = T(0) + T(n–1) + cn$



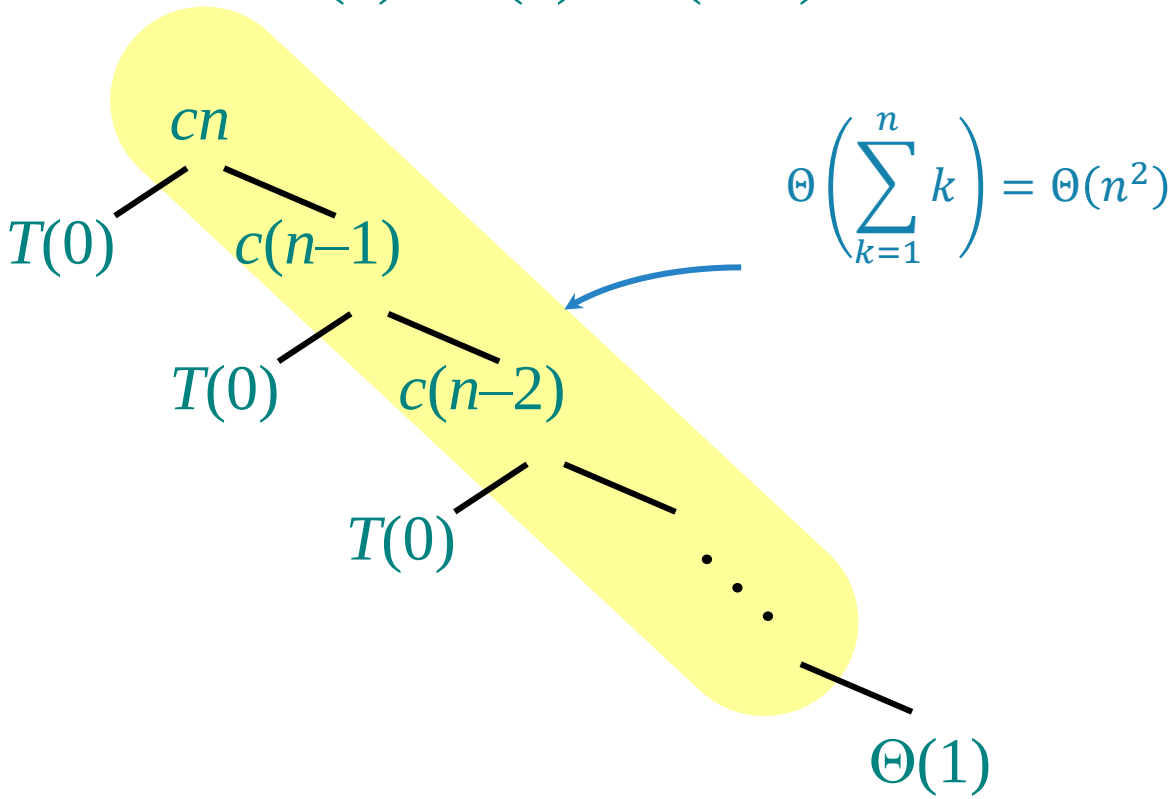
WORST-CASE RECURSION TREE

$T(n) = T(0) + T(n–1) + cn$



WORST-CASE RECURSION TREE

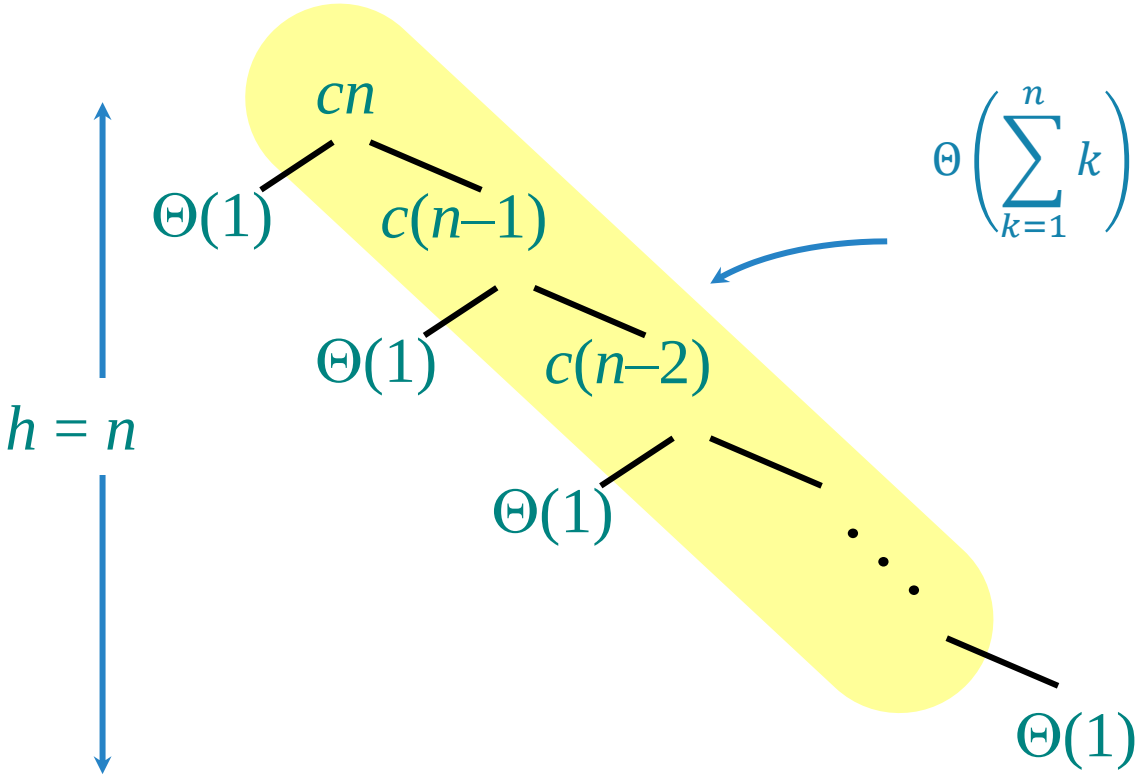
$T(n) = T(0) + T(n–1) + cn$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

WORST-CASE RECURSION TREE

$T(n) = T(0) + T(n–1) + cn$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

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

BEST-CASE ANALYSIS (*For intuition only!*)

If we're lucky, PARTITION splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \lg n)$$

complexity of the partition

deriva sempre dal master theorem questa cosa di ottenere nlogn

(same as merge sort)

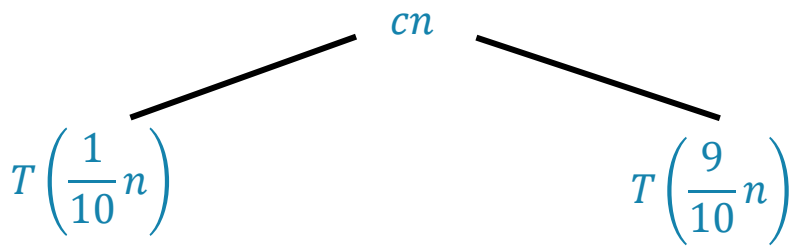
What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

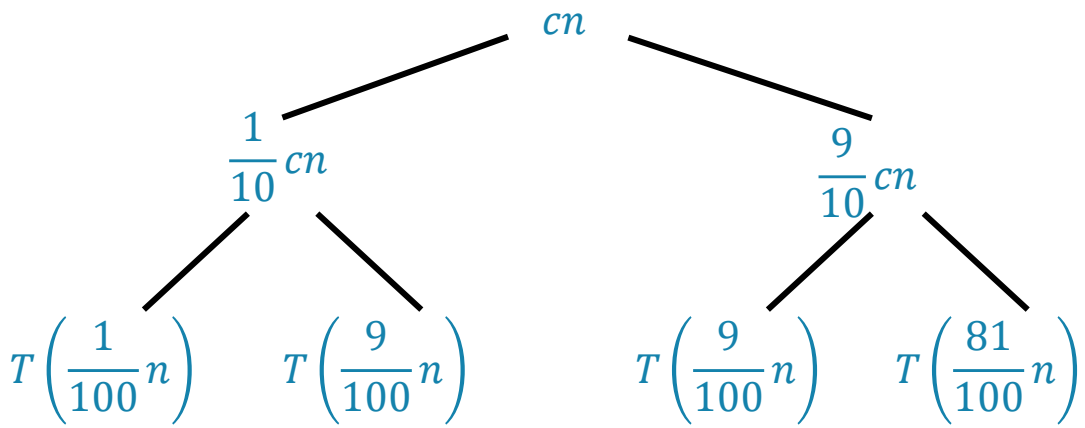
What is the solution to this recurrence?

ANALYSIS OF “ALMOST-BEST” CASE

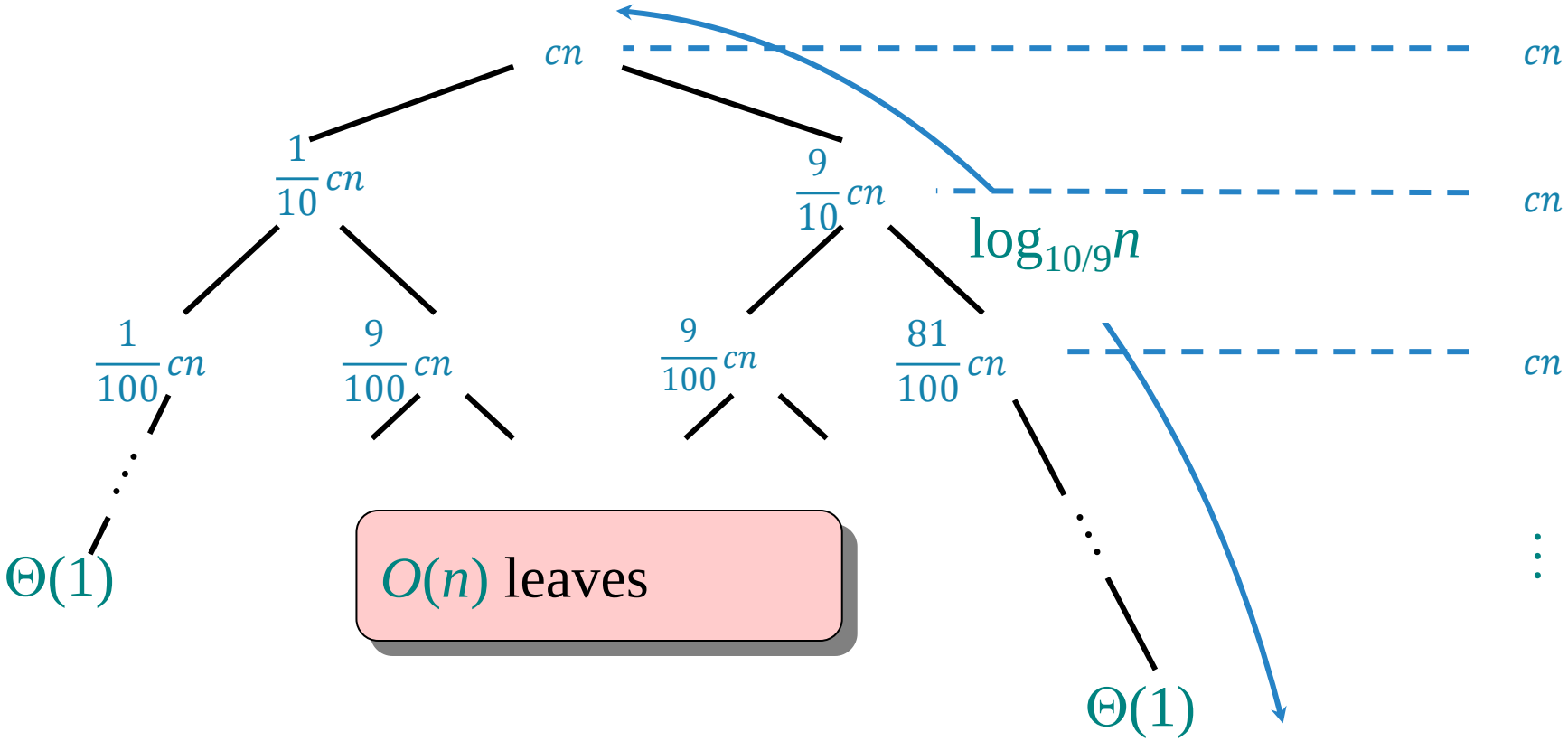
ANALYSIS OF "ALMOST-BEST" CASE



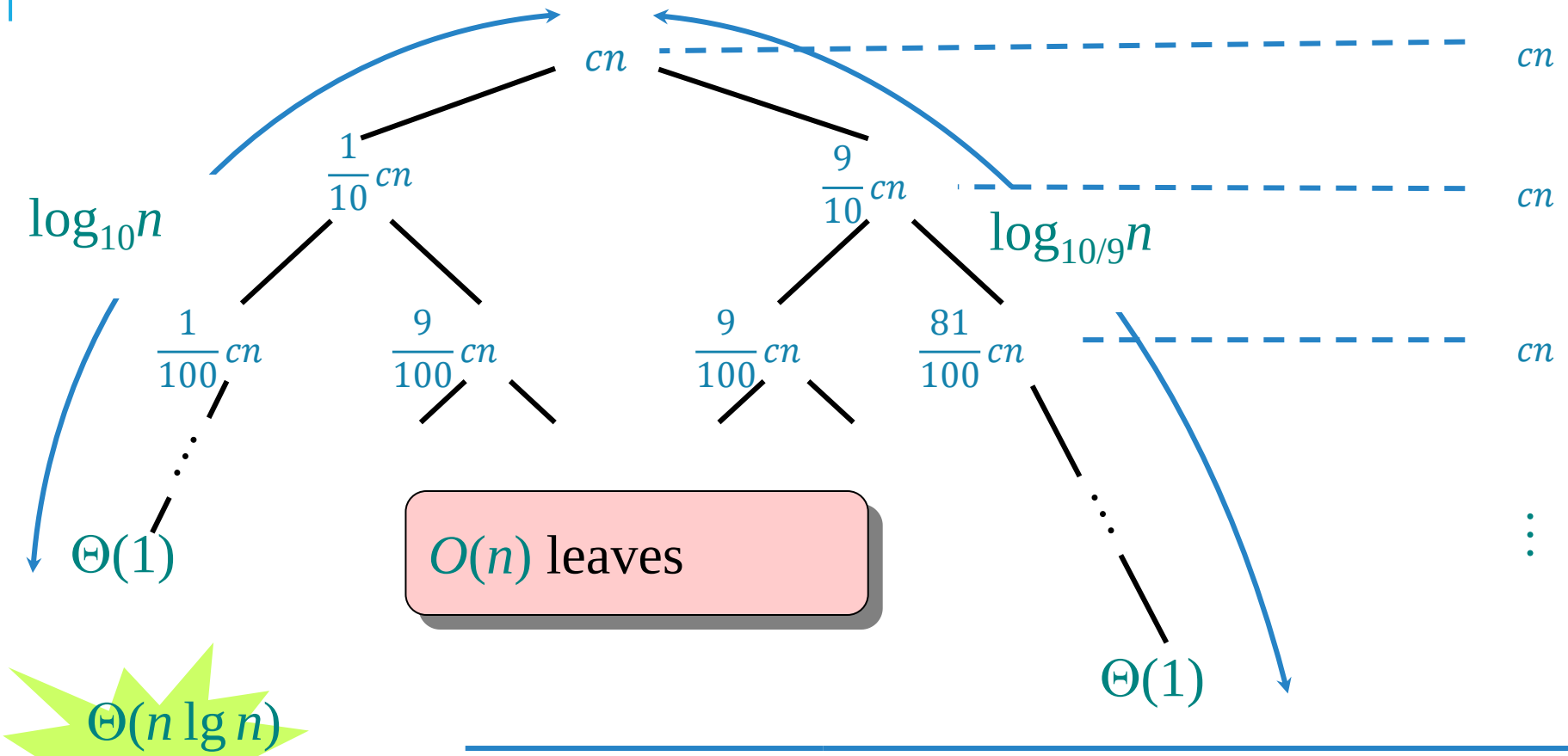
ANALYSIS OF "ALMOST-BEST" CASE



ANALYSIS OF "ALMOST-BEST" CASE



ANALYSIS OF "ALMOST-BEST" CASE



MORE INTUITION

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ….

$$L(n) = 2U(n/2) + \Theta(n) \quad \textit{lucky}$$
$$U(n) = L(n-1) + \Theta(n) \quad \textit{unlucky}$$

Solving:

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

How can we make sure we are usually lucky?

RANDOMIZED QUICKSORT

IDEA: Partition around a *random* element.

- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

so due to the fact we're considering a random algorithm, generally we obtain an average case which costs nlogn

RANDOMIZED QUICKSORT ANALYSIS

Let $T(n) =$ the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the *indicator random variable*

this is the random variable

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

sulla base dell'algoritmo randomico potrei dividere l'array in diverse porzioni, e poichè tutte queste porzioni possono uscire in modo equo allora la prob è proprio 1/n

ANALYSIS (CONTINUED)

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split}, \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split}, \\ \quad\vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split}, \end{cases}$$

tutti i possibili casi in cui avviene lo split

$$= \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))$$

CALCULATING EXPECTATION

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right]$$

Take expectations of both sides.

CALCULATING EXPECTATION

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right]$$

$$= \sum_{k=0}^{n-1} E[X_k(T(k) + T(n-k-1) + \Theta(n))]$$

Linearity of expectation.

CALCULATING EXPECTATION

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right]$$

$$= \sum_{k=0}^{n-1} E[X_k(T(k) + T(n-k-1) + \Theta(n))]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

Independence of X_k from other random choices.

CALCULATING EXPECTATION

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right]$$

$$= \sum_{k=0}^{n-1} E[X_k(T(k) + T(n-k-1) + \Theta(n))]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

Linearity of expectation; $E[X_k] = 1/n$.

CALCULATING EXPECTATION

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E[X_k(T(k) + T(n-k-1) + \Theta(n))] \\
&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n) \\
&= \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)
\end{aligned}
$$

Summations have identical terms.

HAIRY RECURRENCE

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Prove: $E[T(n)] \le a\,n \lg n$ for constant $a > 0$.

- Choose a large enough so that $a\,n \lg n$ dominates $E[T(n)]$ for sufficiently small $n \ge 2$.

Use fact: $\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$ (exercise).

SUBSTITUTION METHOD

$$E[T(n)] \leq \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

Substitute inductive hypothesis.

SUBSTITUTION METHOD

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$
$$\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

Use fact.

SUBSTITUTION METHOD

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\
&\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an \lg n - \left(\frac{an}{4} - \Theta(n)\right)
\end{aligned}
$$

Express as *desired – residual*.

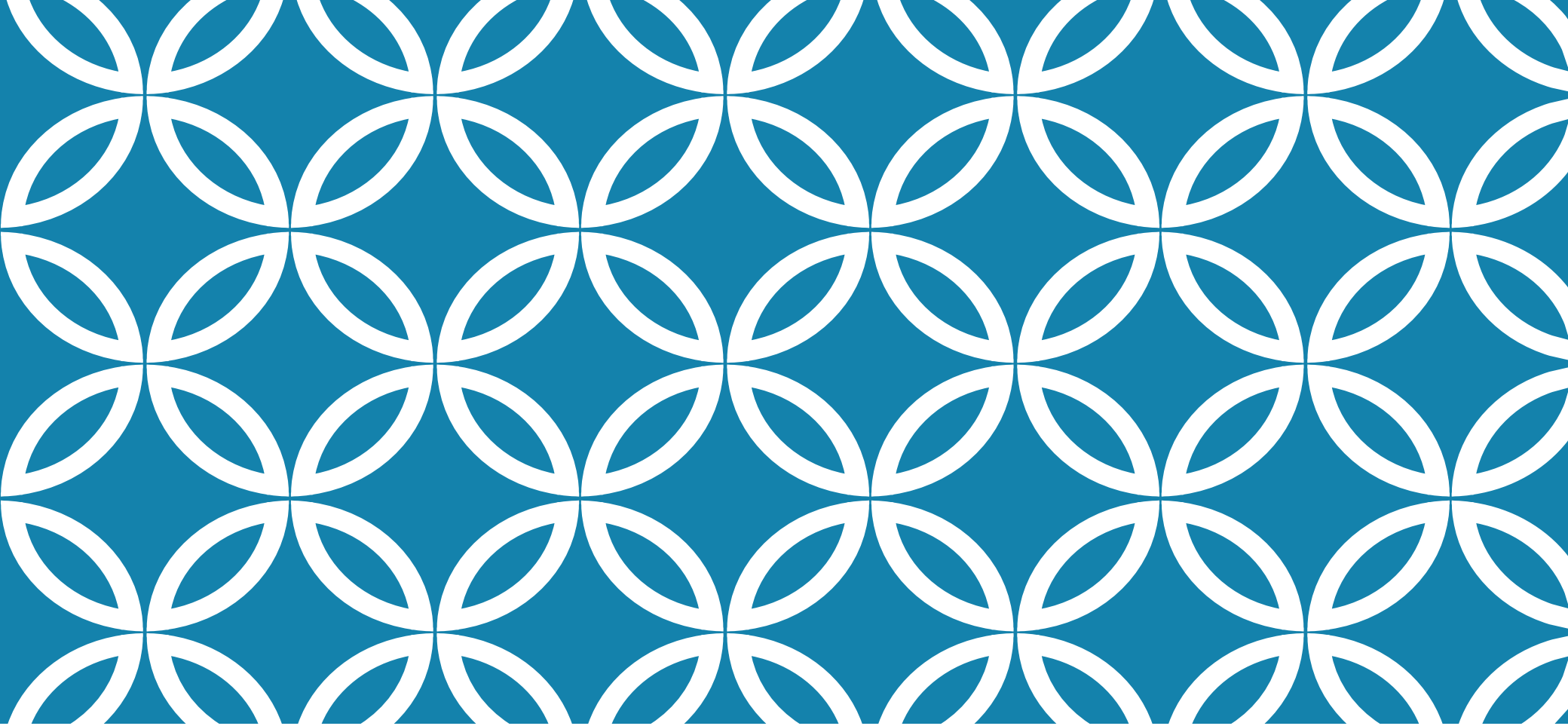
SUBSTITUTION METHOD

$$
\begin{aligned}
E[T(n)] &\le \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\
&= \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an \lg n - \left(\frac{an}{4} - \Theta(n)\right) \\
&\le an \lg n
\end{aligned}
$$

if a is chosen large enough so that $an/4$ dominates the $\Theta(n)$.

QUICKSORT IN PRACTICE

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from *code tuning*.
- Quicksort behaves well even with caching and virtual memory.



SORTING LOWER BOUNDS

DECISION TREES

LINEAR-TIME SORTING

COUNTING SORT

RADIX SORT

APPENDIX: PUNCHED CARDS

HOW FAST CAN WE SORT?

All the sorting algorithms we have seen so far are *comparison sorts*: only use comparisons to determine the relative order of elements.

- *E.g.*, insertion sort, merge sort, quicksort, heapsort.

The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

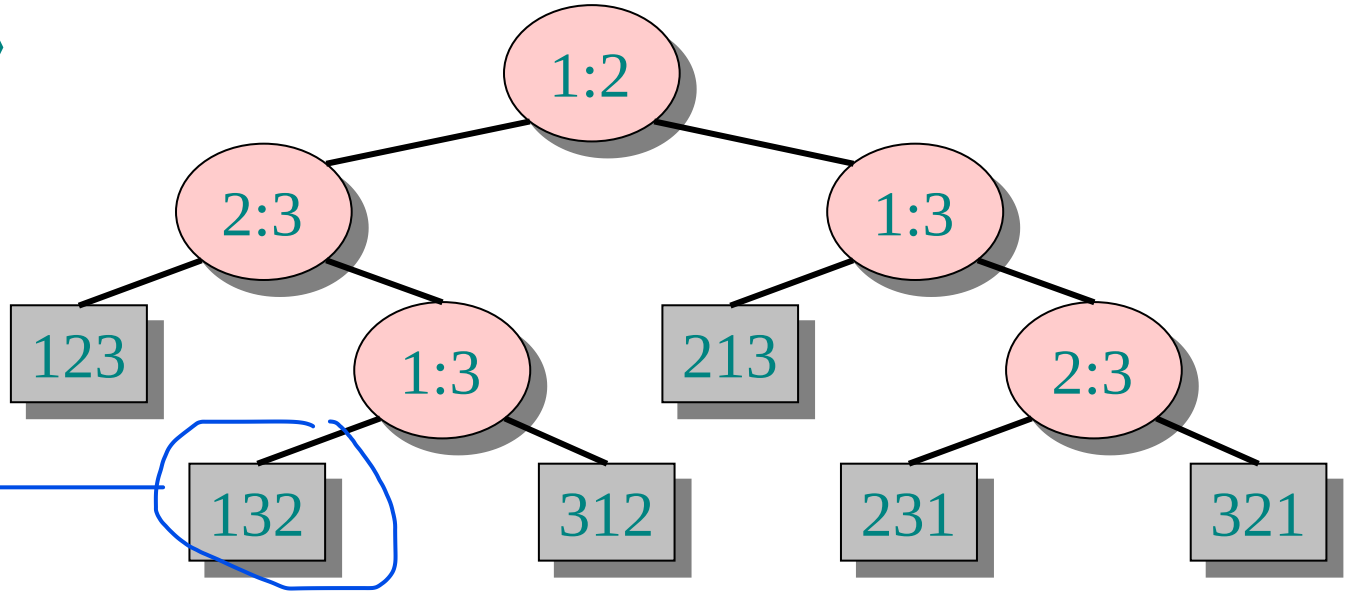
Is $O(n \lg n)$ the best we can do?

Decision trees can help us answer this question.

questo è un caso generale di sorting. -> prendo due elementi nell'array e se il primo è minore del secondo scendo nel ramo di sx altrimenti in quello di sx fino ad arrivare alle foglie

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, \ldots, a_n \rangle$



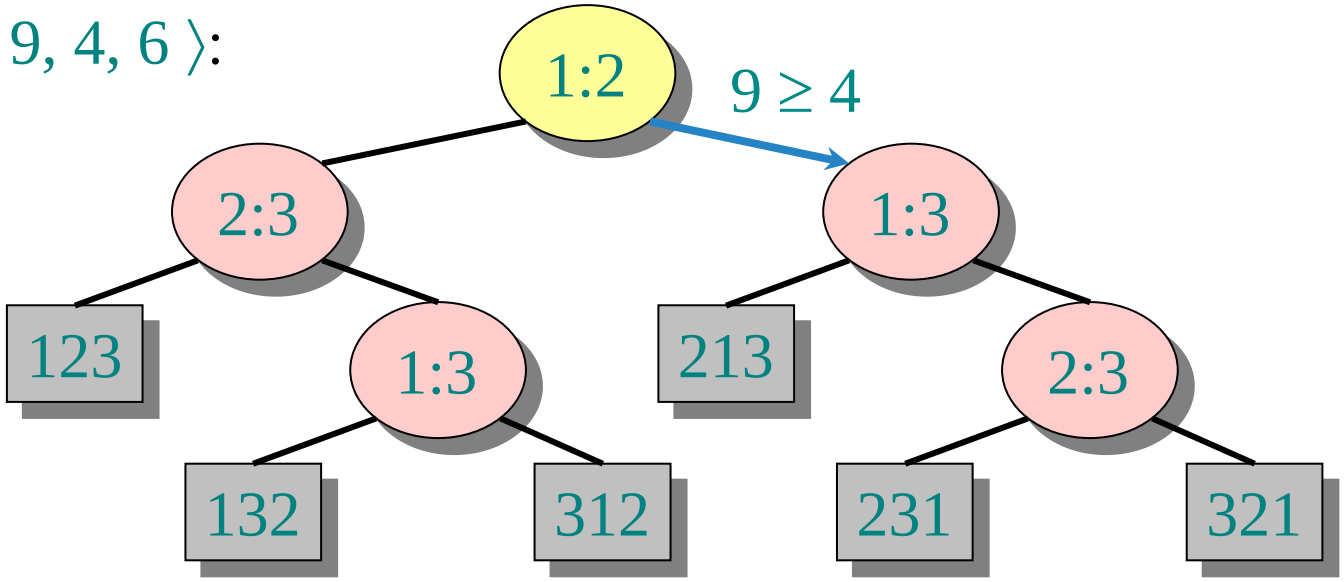
in questo percorso otteniamo un ordinamento che è diverso da quello standard

Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle\ 9, 4, 6\ \rangle$:

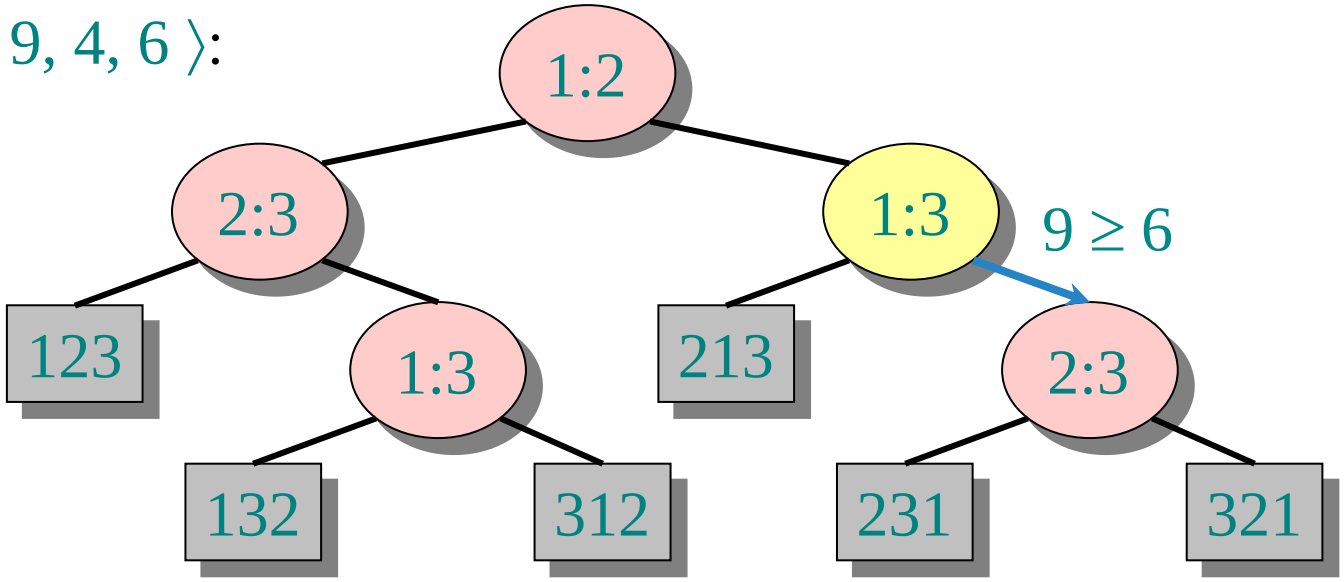


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle\ 9, 4, 6\ \rangle$:

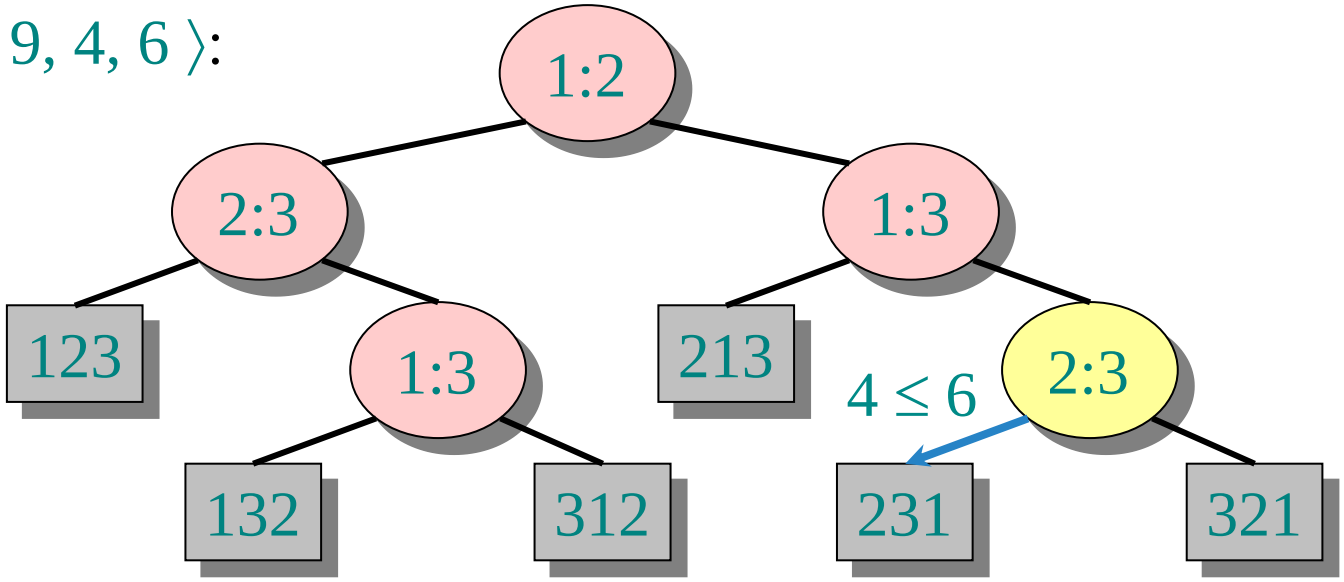


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle\ 9, 4, 6\ \rangle$:

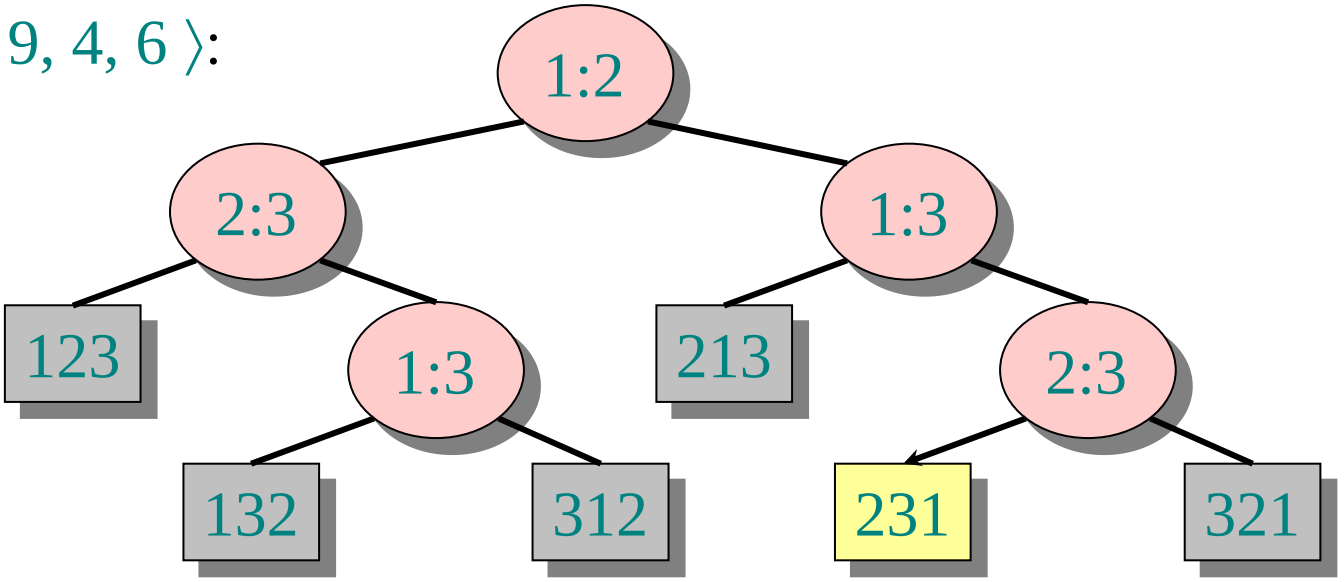


Each internal node is labeled $i{:}j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

DECISION-TREE EXAMPLE

Sort $\langle a_1, a_2, a_3 \rangle = \langle\ 9, 4, 6\ \rangle$:



Each leaf contains a permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ to indicate that the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$ has been established.

DECISION-TREE MODEL

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

LOWER BOUND FOR DECISION-TREE SORTING

Theorem. Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Proof. The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$$
\begin{aligned}
\therefore\ h &\geq \lg(n!) && (\lg \text{ is mono. increasing}) \\
&\geq \lg\left((n/e)^n\right) && (\text{Stirling's formula}) \\
&= n \lg n - n \lg e \\
&= \Omega(n \lg n)\,. \quad \square
\end{aligned}
$$

LOWER BOUND FOR COMPARISON SORTING

Corollary. Heapsort and merge sort are asymptotically optimal comparison sorting algorithms. 

SORTING IN LINEAR TIME

Counting sort: No comparisons between elements.

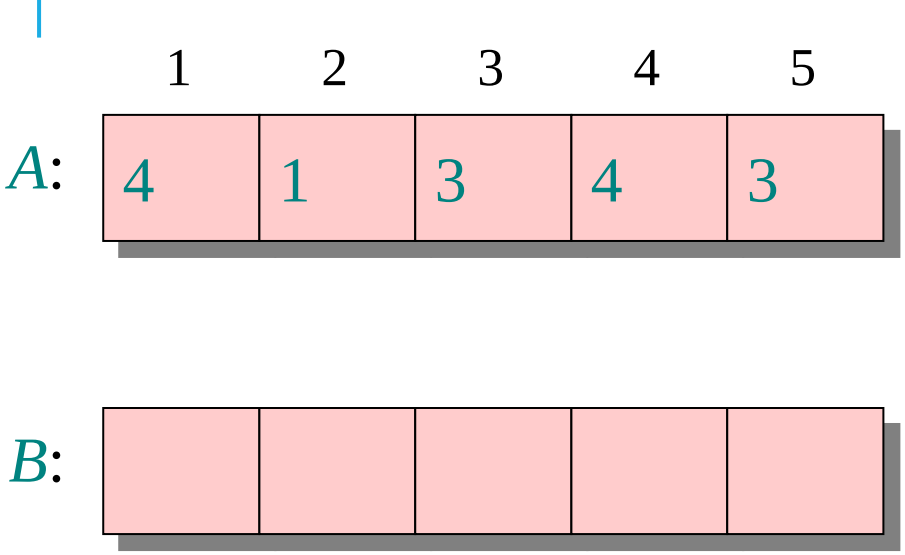
- *Input*: $A[1 \ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- *Output*: $B[1 \ldots n]$, sorted.
- *Auxiliary storage*: $C[1 \ldots k]$.

this is an assumption -> i wont store all the value in the auxiliary mem

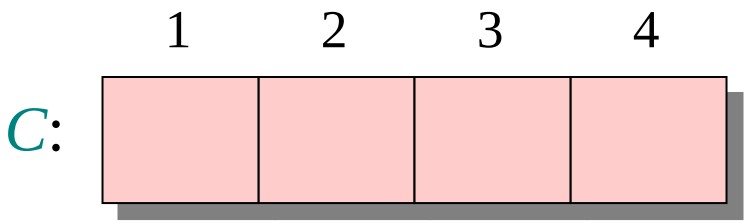
COUNTING SORT

```
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1      ▷ C[i] = |{key = i}|
for i ← 2 to k
    do C[i] ← C[i] + C[i–1]       ▷ C[i] = |{key ≤ i}|
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] – 1
```

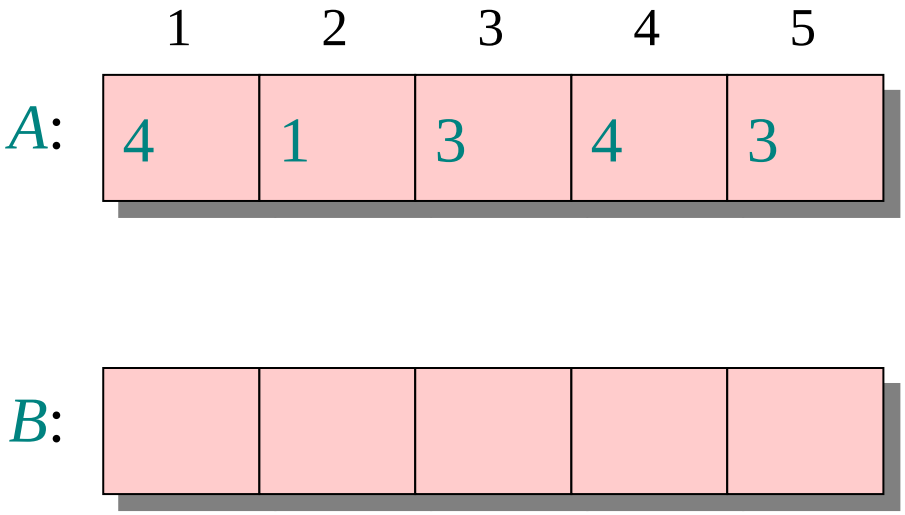
COUNTING-SORT EXAMPLE

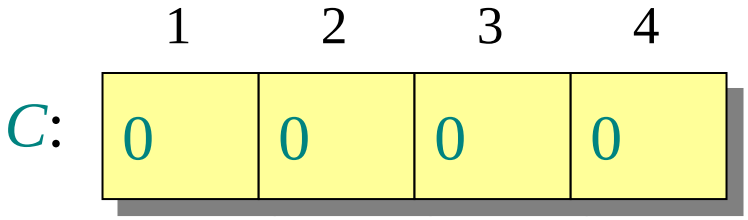


auxiliary array



LOOP 1



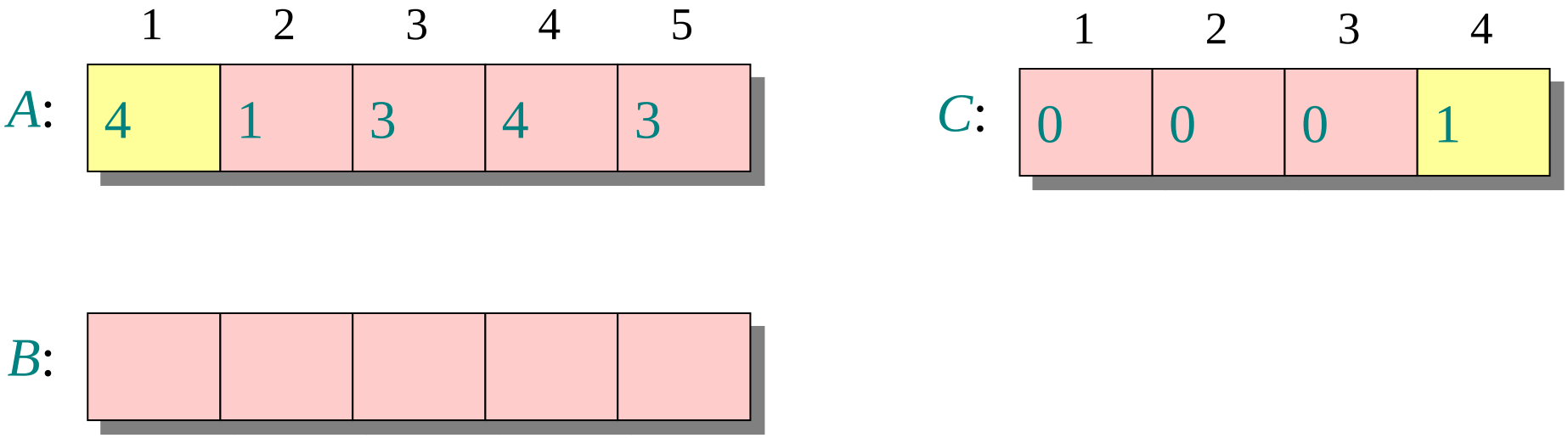


for $i \leftarrow 1$ to k

do $C[i] \leftarrow 0$

LOOP 2

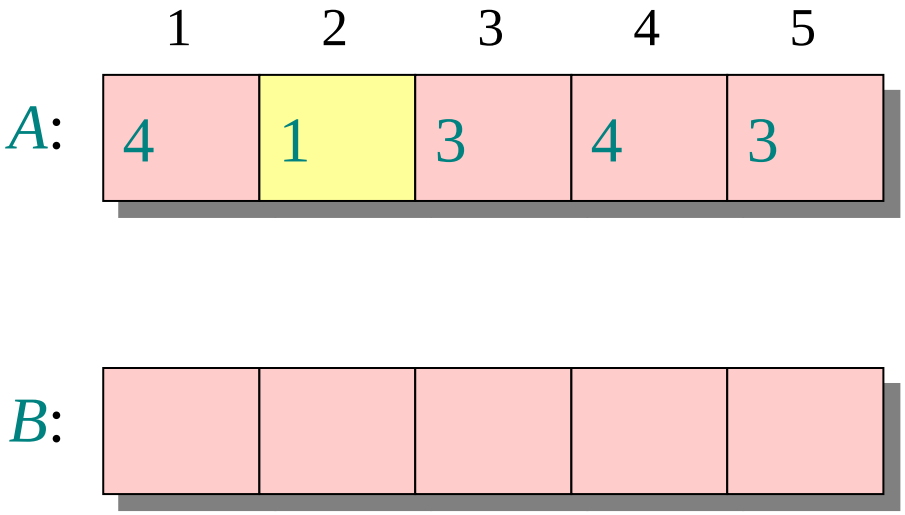
in this loop we're counting how many 4 are there and we store the counter in position 4

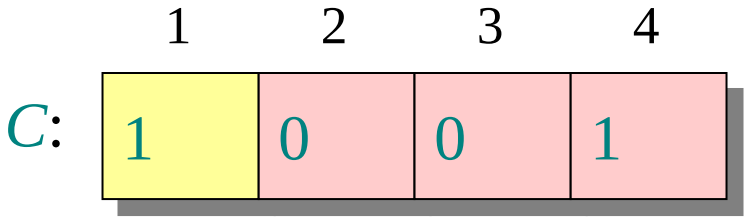


for $j \leftarrow 1$ to n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

LOOP 2

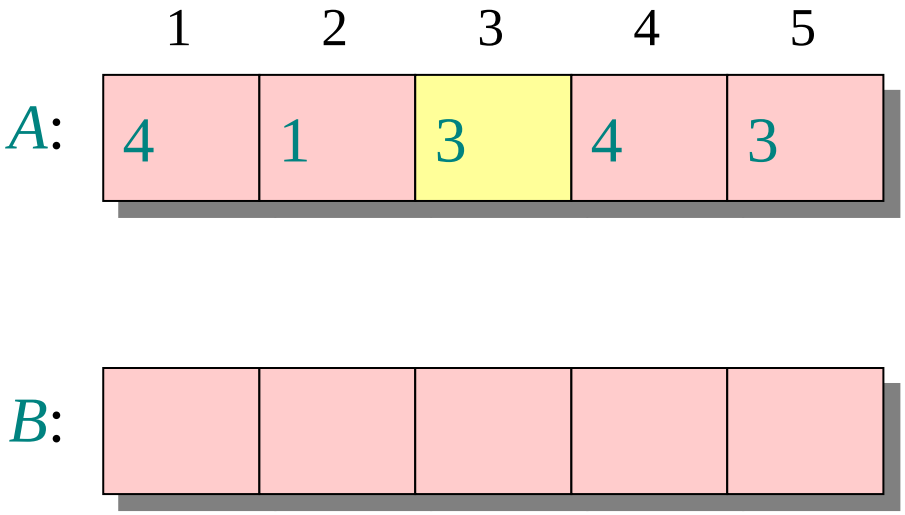


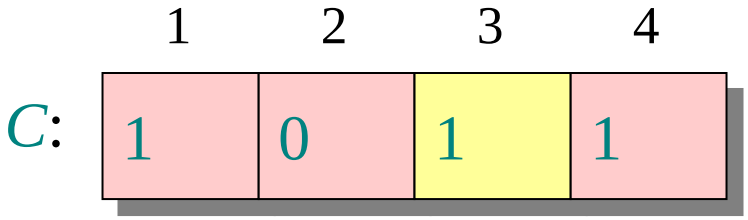


for $j \leftarrow 1$ to n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

LOOP 2

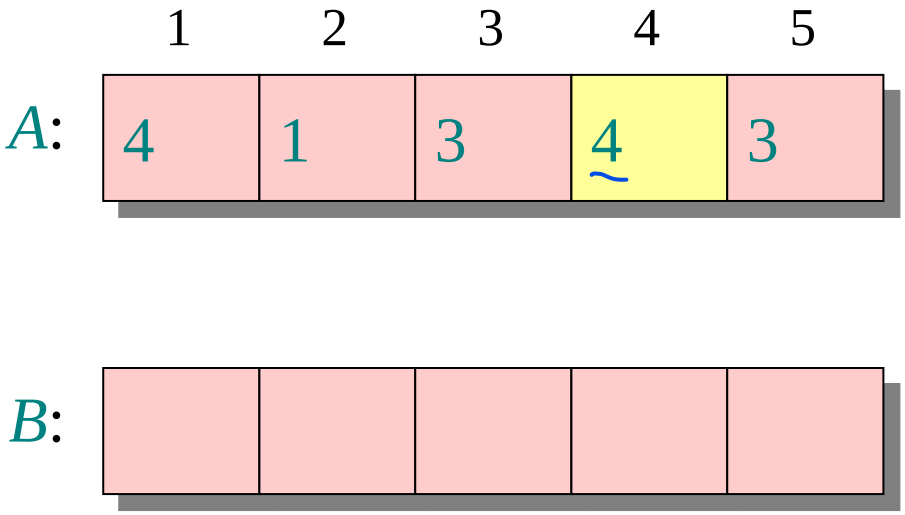


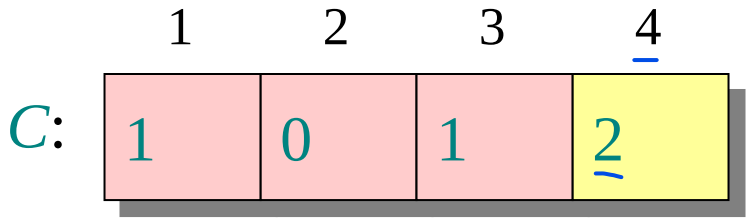


for $j \leftarrow 1$ to n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

LOOP 2

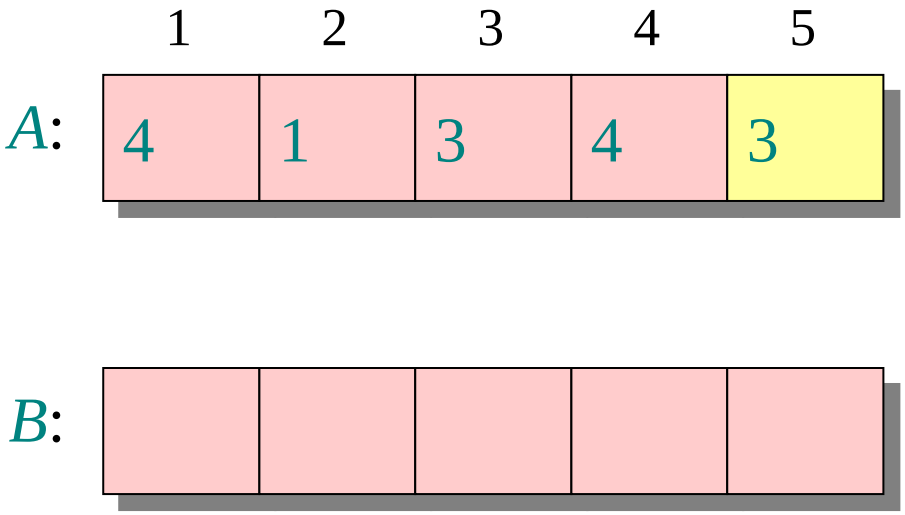


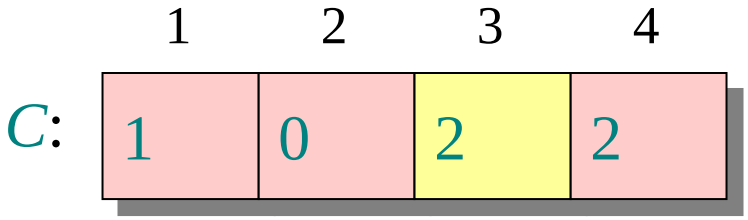


for $j \leftarrow 1$ to n

do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

LOOP 2

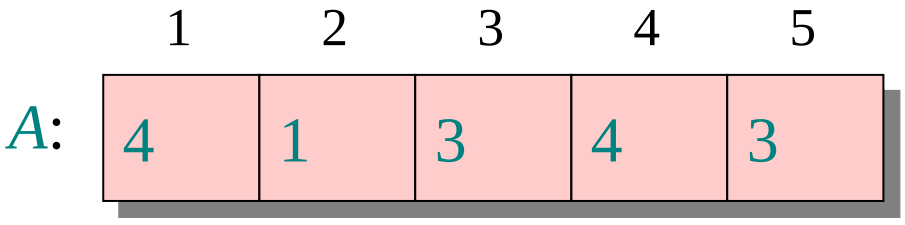


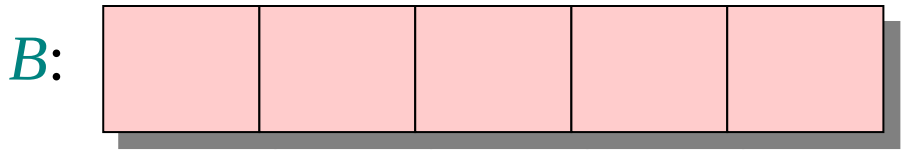


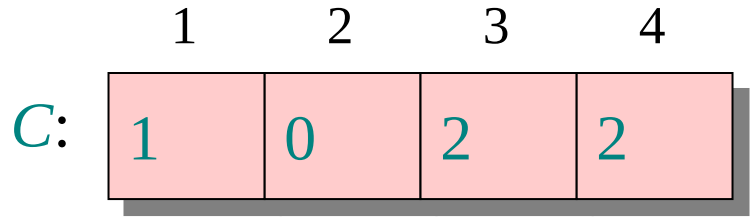
for $j \leftarrow 1$ to n

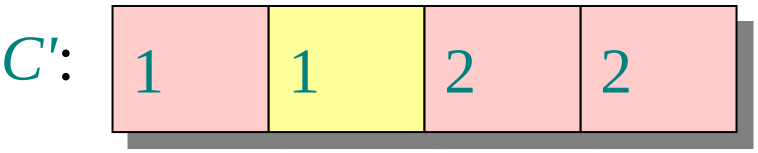
do $C[A[j]] \leftarrow C[A[j]] + 1$ ▷ $C[i] = |\{\text{key} = i\}|$

LOOP 3







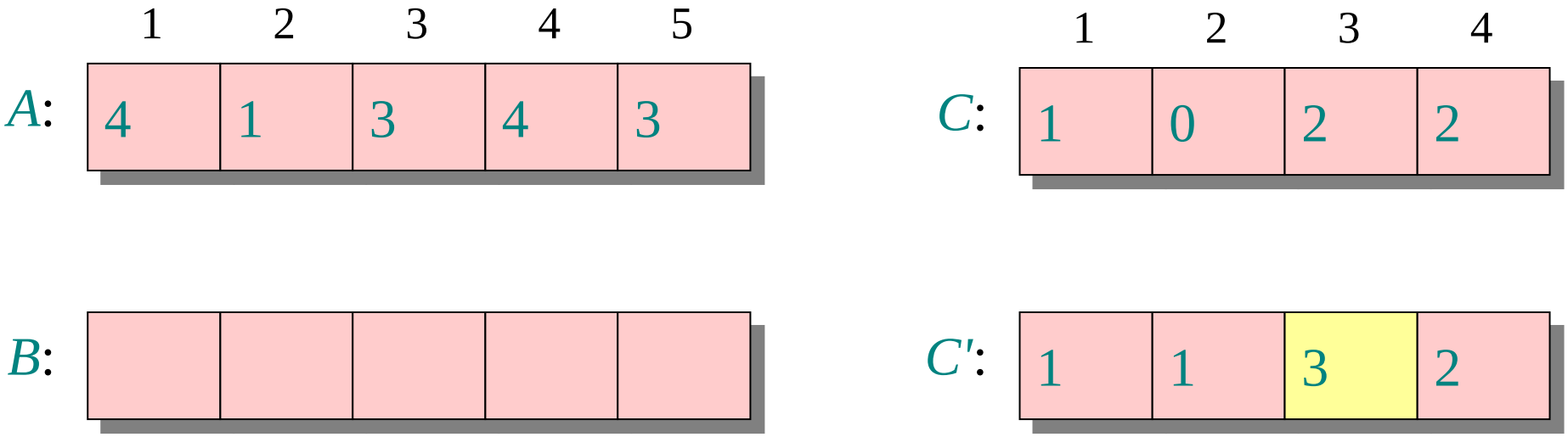


now we're doing the prefix scan -> starts from the seocnd value and we're doing the sum between that value and the previous one

for $i \leftarrow 2$ to k

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

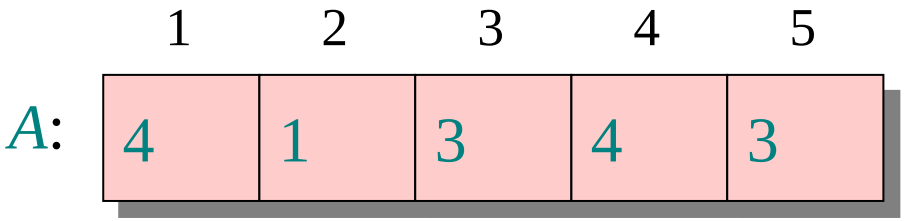
LOOP 3

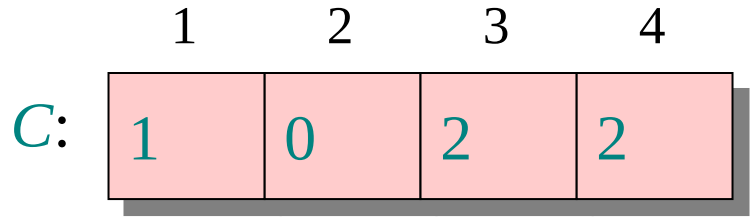


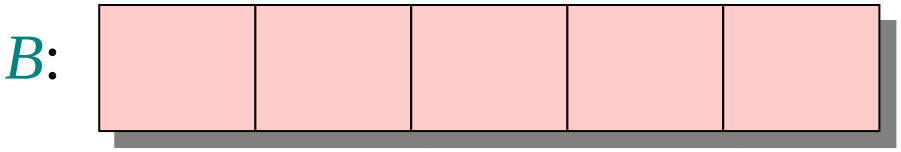
for $i \leftarrow 2$ to k

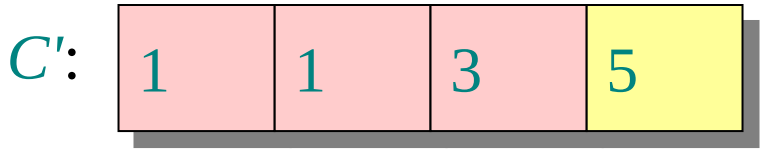
do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

LOOP 3







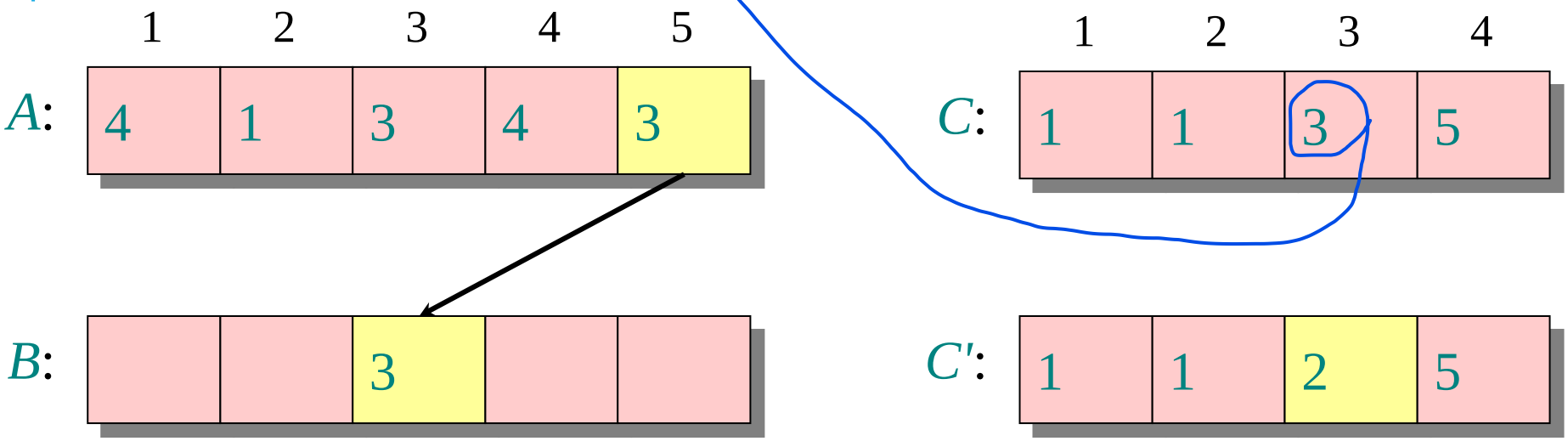


for $i \leftarrow 2$ to k

do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

LOOP 4

we start by looking to the final item of the array A -> then we look to the auxiliary array and we see where we have to put the value = 3 -> exactly in position 3



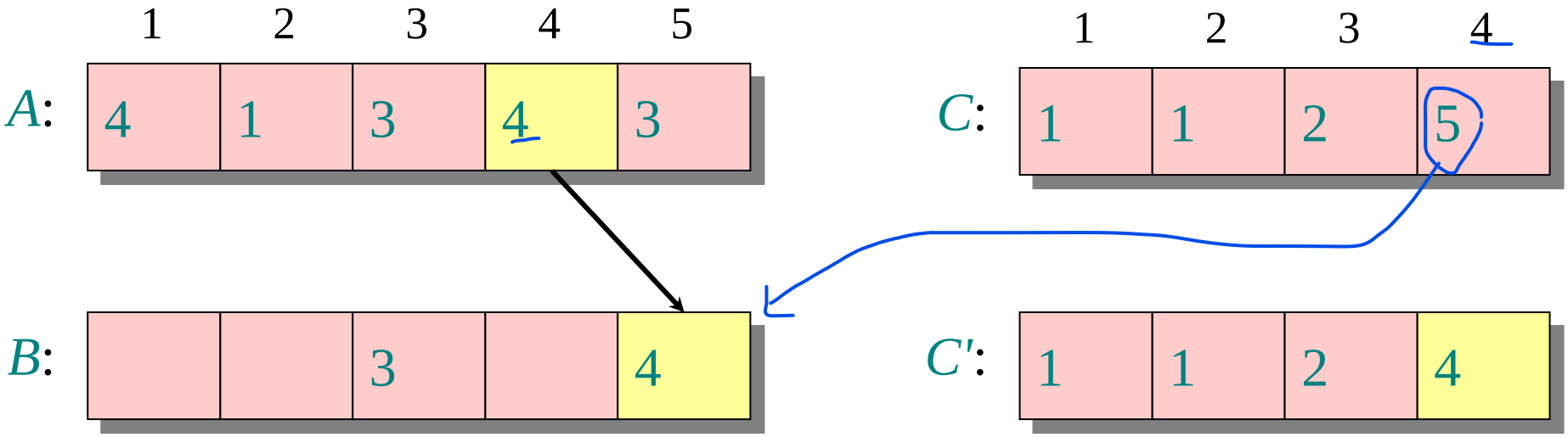
and we're decrementing here

for $j \leftarrow n$ downto 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

LOOP 4

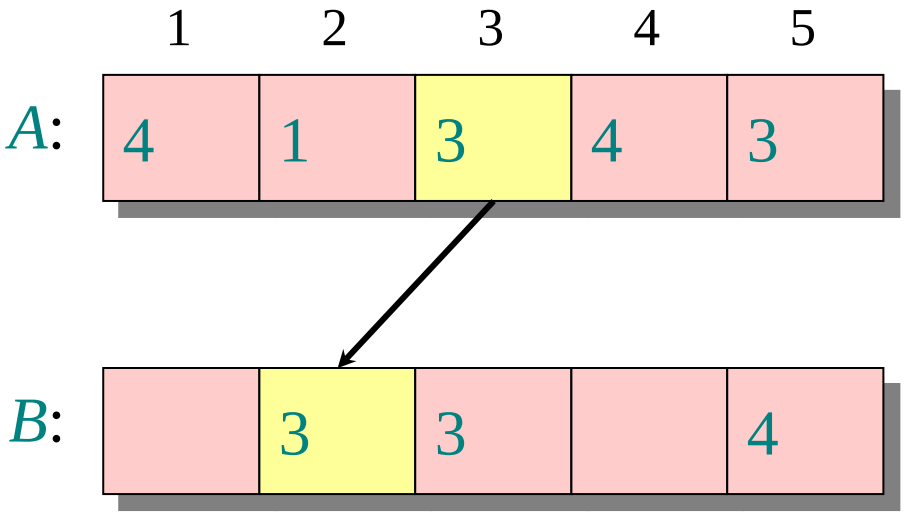


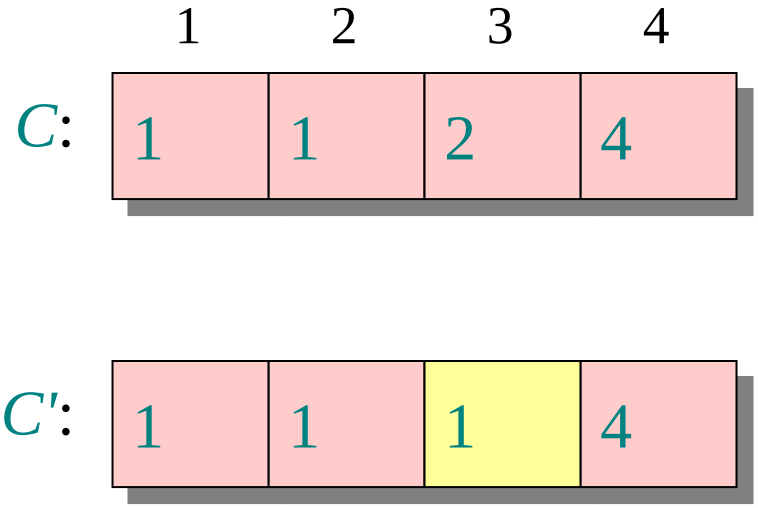
for $j \leftarrow n$ downto 1

do $B[C[A[j]]] \leftarrow A[j]$

$C[A[j]] \leftarrow C[A[j]] - 1$

LOOP 4



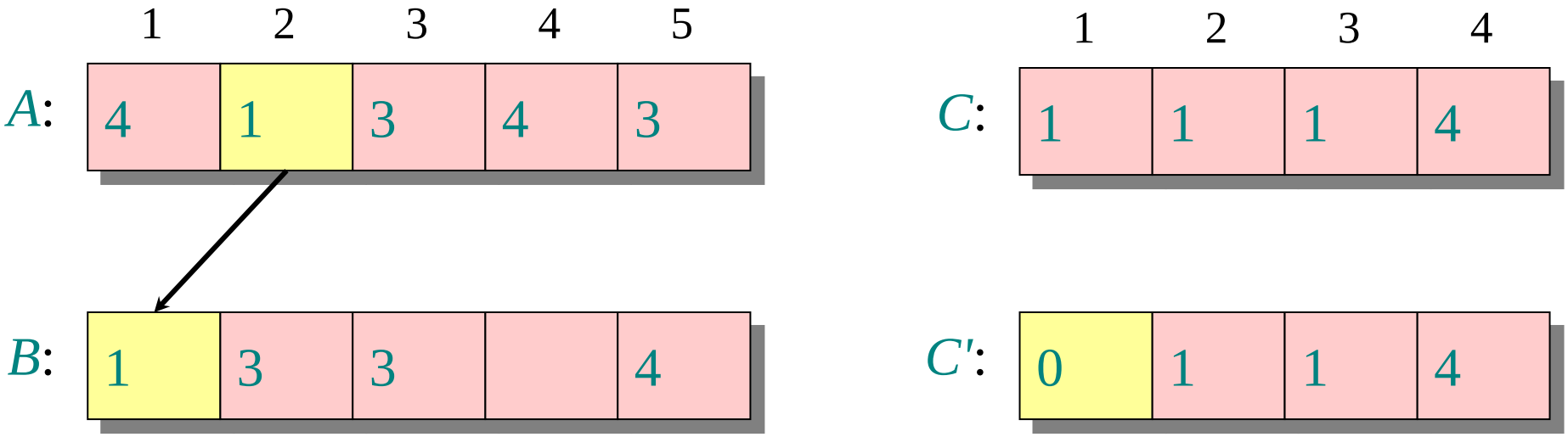


for $j \leftarrow n$ downto 1

do $B[C[A[j]]] \leftarrow A[j]$

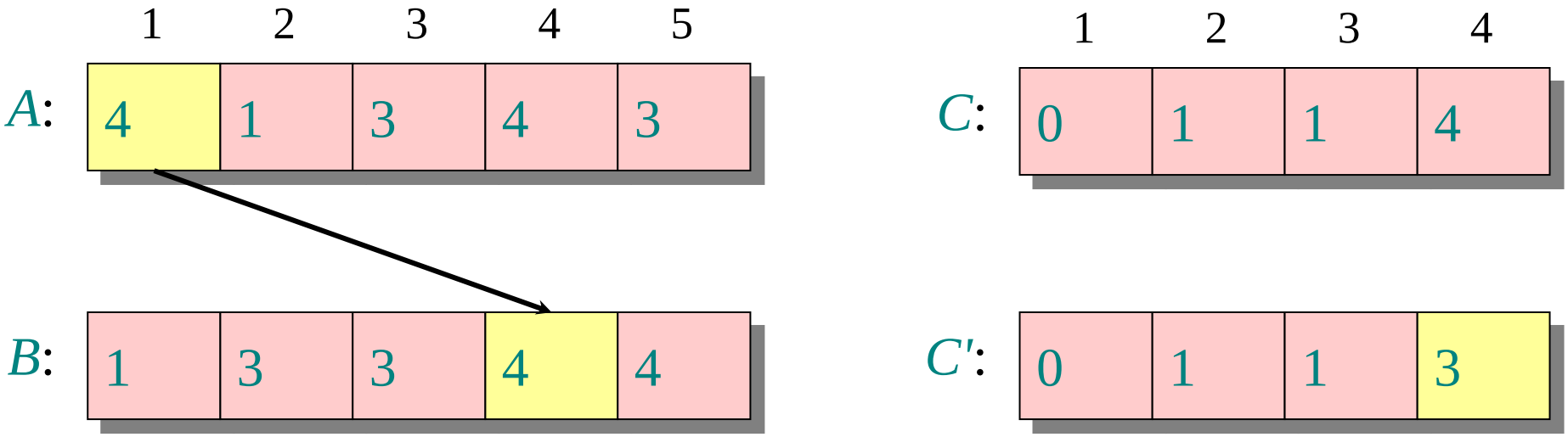
$C[A[j]] \leftarrow C[A[j]] - 1$

LOOP 4



for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$

LOOP 4



for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$

ANALYSIS

$\Theta(k)$ for $i \leftarrow 1$ to k
do $C[i] \leftarrow 0$

$\Theta(n)$ for $j \leftarrow 1$ to n
do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ for $i \leftarrow 2$ to k
do $C[i] \leftarrow C[i] + C[i–1]$

$\Theta(n)$ for $j \leftarrow n$ downto 1
do $B[C[A[j]]] \leftarrow A[j]$
$C[A[j]] \leftarrow C[A[j]] – 1$

$\Theta(n + k)$

in this algorithm there is no decision tree. It is just counting -> but the main problem is that this algorithm is using a lot of memory to storing data in auxiliary arrays

RUNNING TIME

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

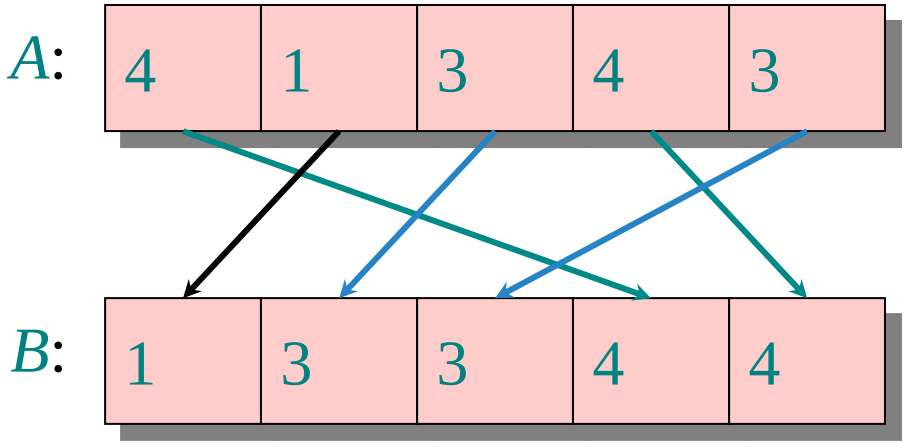
- But, sorting takes $\Omega(n \lg n)$ time!
- Where's the fallacy?

Answer:

- *Comparison sorting* takes $\Omega(n \lg n)$ time.
- Counting sort is not a *comparison sort*. because it doesnt use any comparison
- In fact, not a single comparison between elements occurs!

STABLE SORTING

Counting sort is a *stable* sort: it preserves the input order among equal elements.



quicksort is not very stable

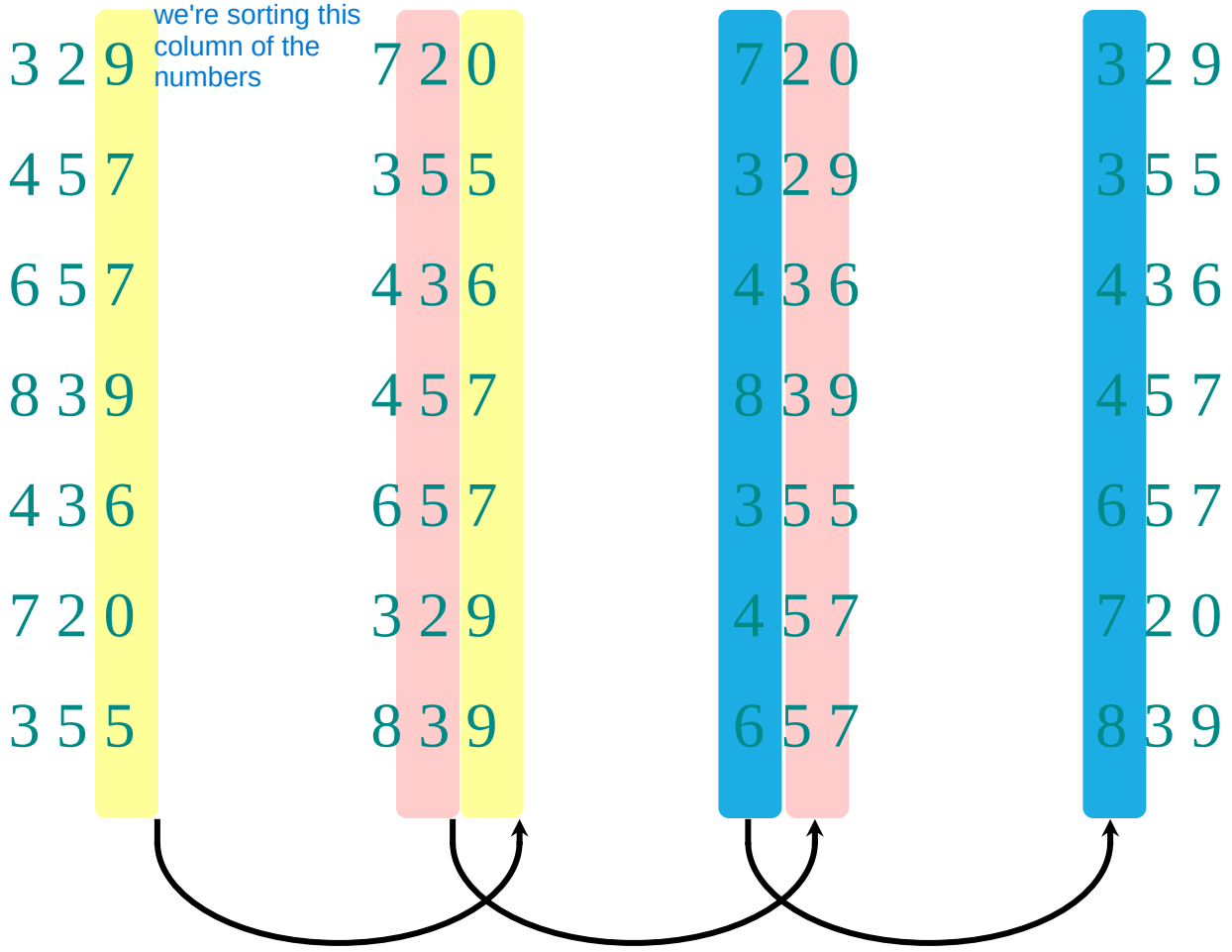
Exercise: What other sorts have this property?

RADIX SORT

- *Origin*: Herman Hollerith's card-sorting machine for the 1890 U.S. Census. (See Appendix )
- Digit-by-digit sort.
- Hollerith's original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on *least-significant digit first* with auxiliary *stable* sort.

in orizzontale abbiamo i numeri che devono essere ordinati -> li ordiniamo prendendo in considerazione le loro ultime cifre in colonna e le ordiniamo e cosi via.

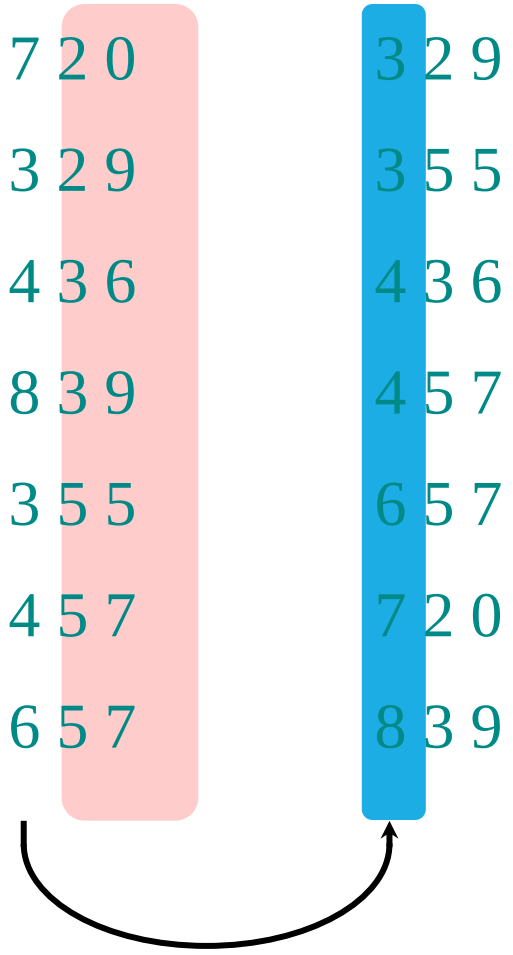
OPERATION OF RADIX SORT



CORRECTNESS OF RADIX SORT

Induction on digit position

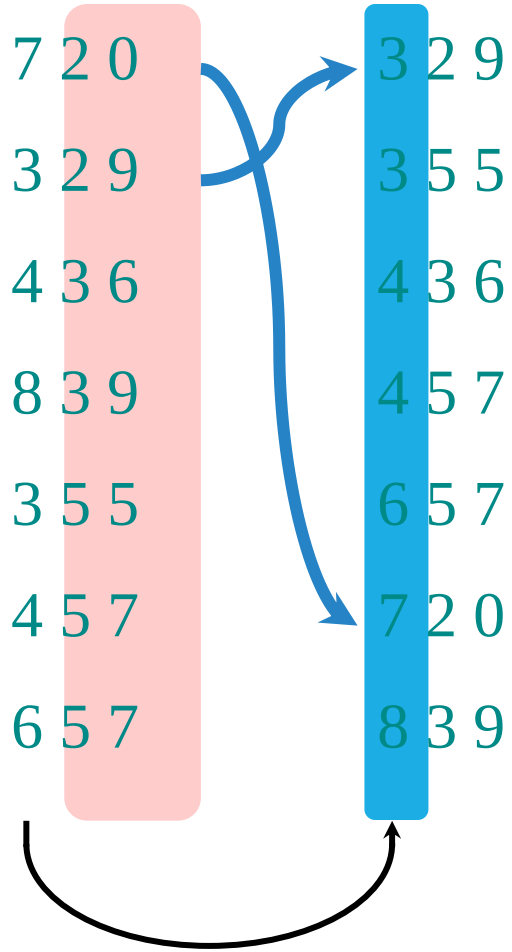
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t



CORRECTNESS OF RADIX SORT

Induction on digit position

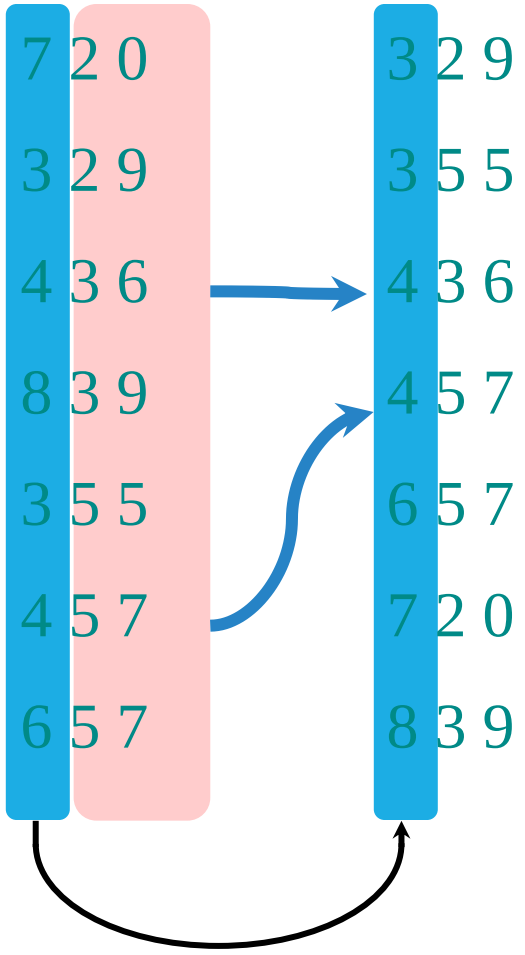
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.



CORRECTNESS OF RADIX SORT

Induction on digit position

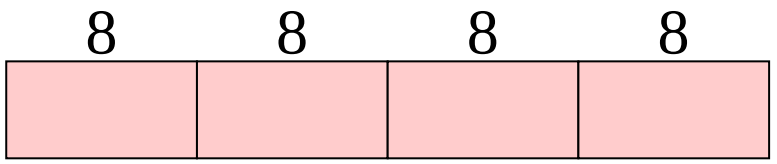
- Assume that the numbers are sorted by their low-order $t - 1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input $\Rightarrow$ correct order.



ANALYSIS OF RADIX SORT

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: 32-bit word



$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

ANALYSIS (CONTINUED)

Recall: Counting sort takes $\Theta(n + k)$ time to sort n numbers in the range from 0 to $k - 1$.

If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have

T(n,b) = theta((b/r)(n+2^r))

$$T(n,b) = \Theta\left(\frac{b}{r}(n+2^r)\right).$$

Choose r to minimize $T(n, b)$:

- Increasing r means fewer passes, but as $r > \lg n$, the time grows exponentially.

CHOOSING r

$$T(n,b) = \Theta\left(\frac{b}{r}(n + 2^r)\right)$$

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don't want $2^r > n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.

Choosing $r = \lg n$ implies $T(n, b) = \Theta(bn/\lg n)$.

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(dn)$ time.

CONCLUSIONS

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

Example (32-bit numbers):

- At most 3 passes when sorting ≥ 2000 numbers.
- Merge sort and quicksort do at least $\lceil \lg 2000 \rceil = 11$ passes.

Downside: Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

APPENDIX: PUNCHED-CARD TECHNOLOGY

- Herman Hollerith (1860-1929)
- Punched cards
- Hollerith's tabulating system
- Operation of the sorter
- Origin of radix sort
- "Modern" IBM card
- Web resources on punched-card technology

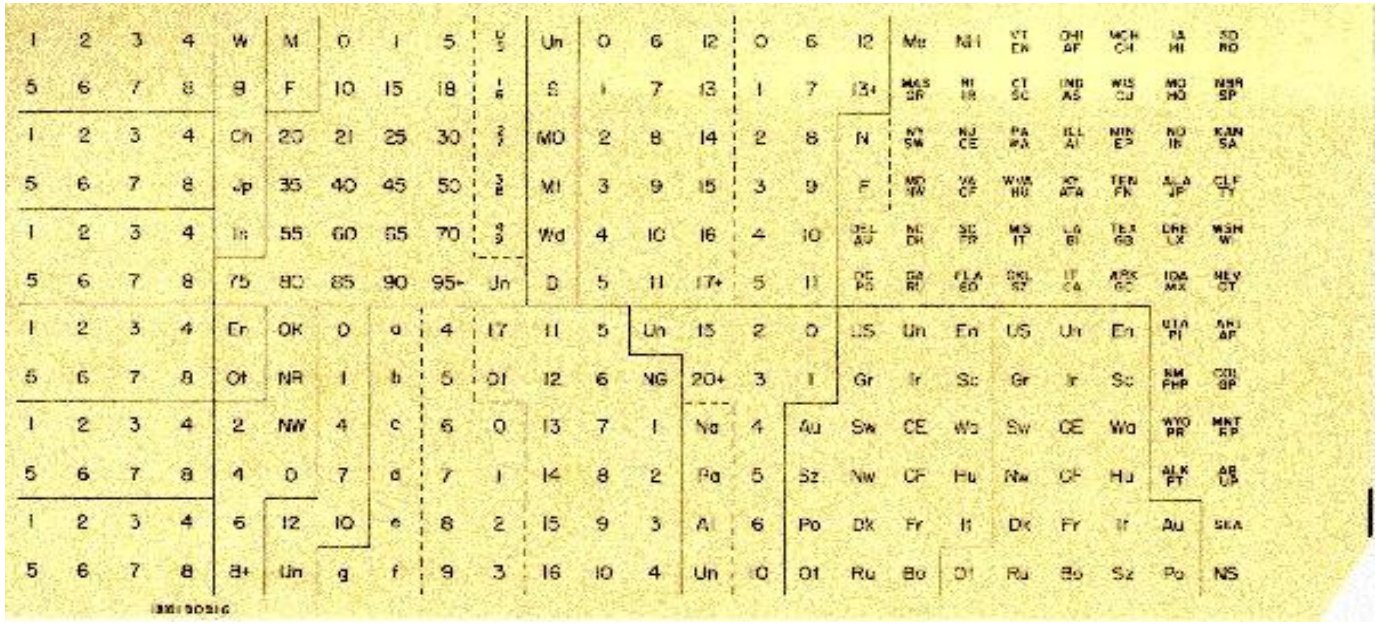
Return to last slide viewed.

HERMAN HOLLERITH (1860-1929)

- The 1880 U.S. Census took almost 10 years to process.
- While a lecturer at MIT, Hollerith prototyped punched-card technology.
- His machines, including a "card sorter," allowed the 1890 census total to be reported in 6 weeks.
- He founded the Tabulating Machine Company in 1911, which merged with other companies in 1924 to form International Business Machines.

PUNCHED CARDS

- Punched card = data record.
- Hole = value.
- Algorithm = machine + human operator.



Replica of punch card from the 1900 U.S. census. [Howells 2000]

HOLLERITH'S TABULATING SYSTEM

- Pantograph card punch
- Hand-press reader
- Dial counters
- Sorting box

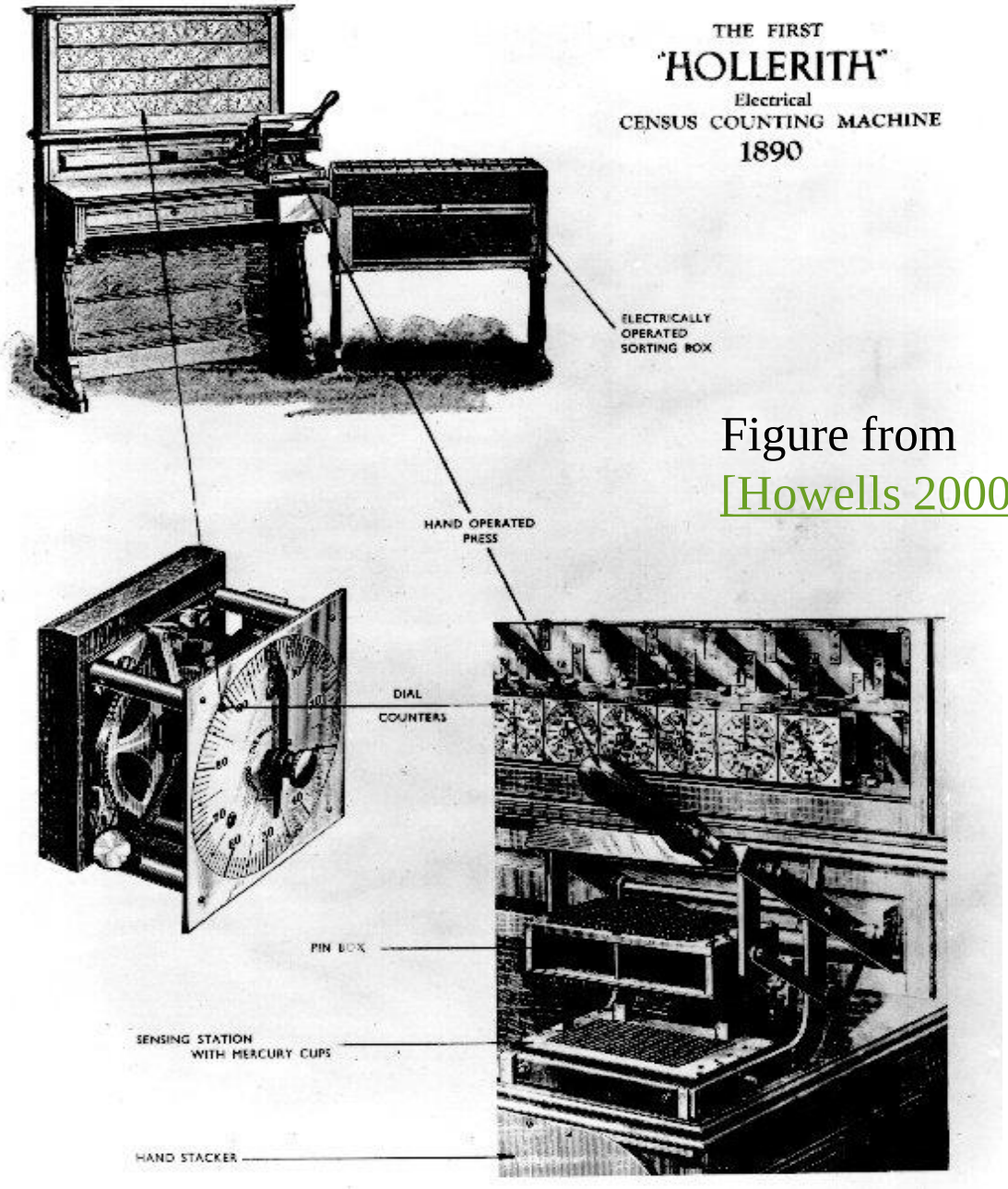
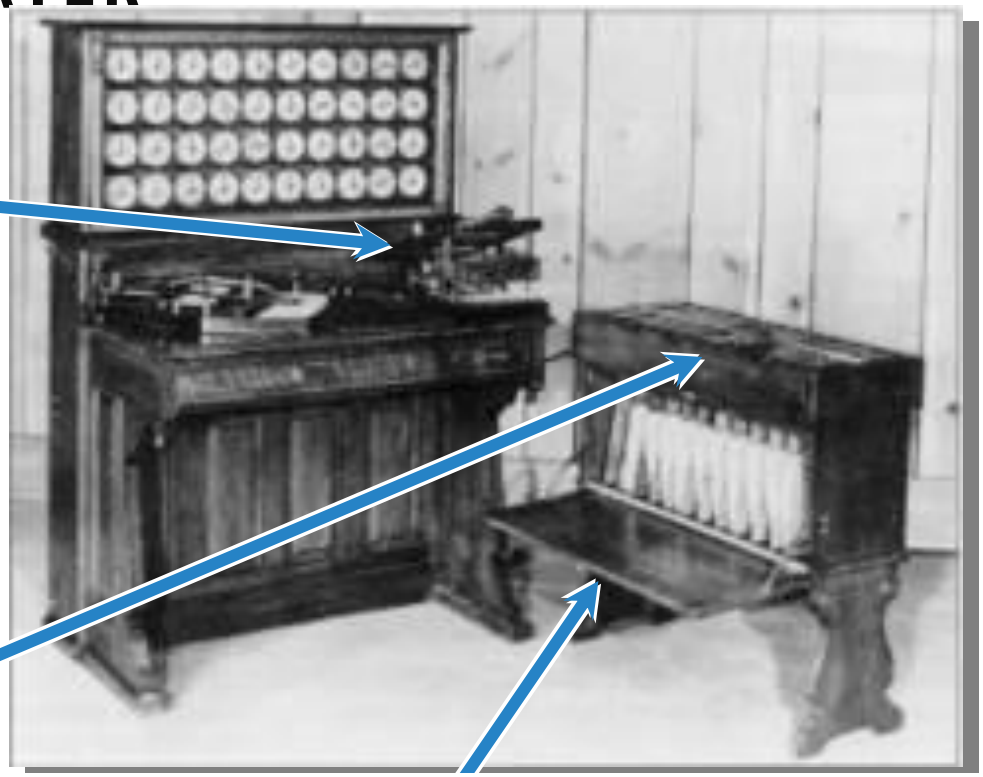


Figure from [Howells 2000].

OPERATION OF THE SORTER

- An operator inserts a card into the press.
- Pins on the press reach through the punched holes to make electrical contact with mercury-filled cups beneath the card.
- Whenever a particular digit value is punched, the lid of the corresponding sorting bin lifts.
- The operator deposits the card into the bin and closes the lid.
- When all cards have been processed, the front panel is opened, and the cards are collected in order, yielding one pass of a stable sort.



Hollerith Tabulator, Pantograph, Press, and Sorter

ORIGIN OF RADIX SORT

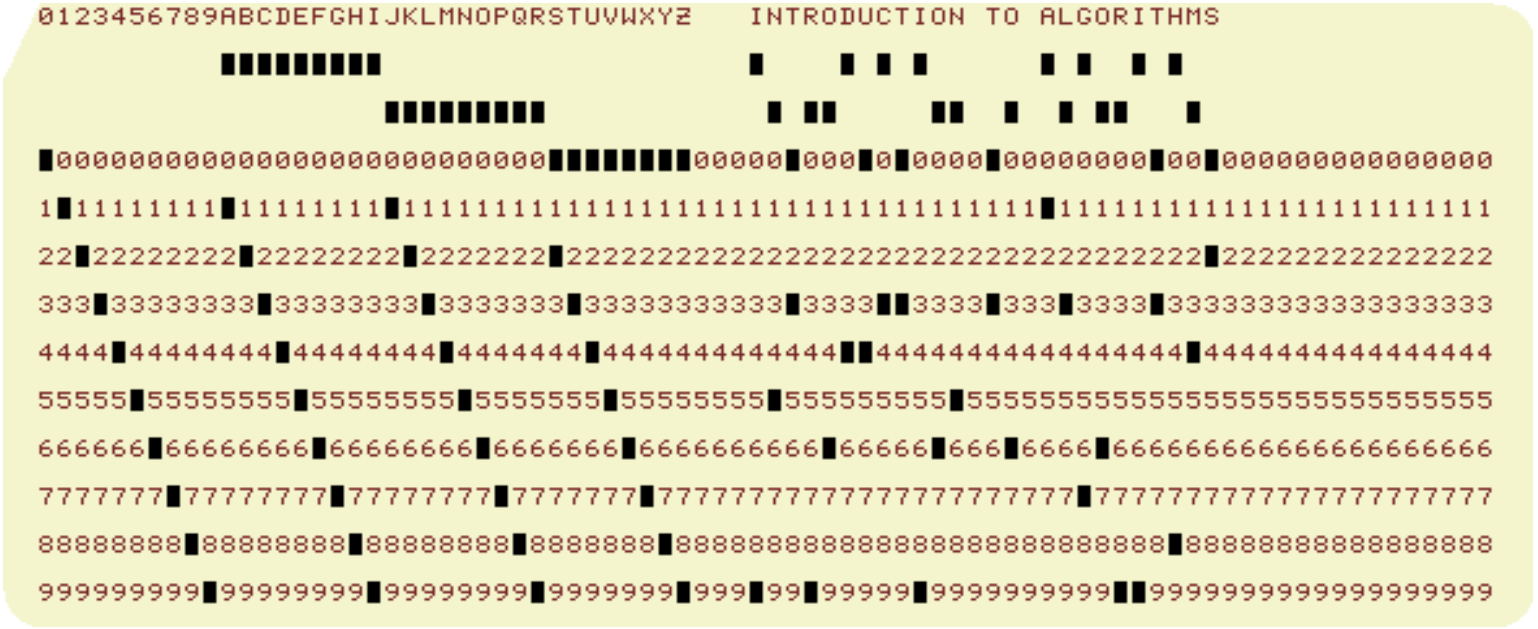
Hollerith's original 1889 patent alludes to a most-significant-digit-first radix sort:

> *"The most complicated combinations can readily be counted with comparatively few counters or relays by first assorting the cards according to the first items entering into the combinations, then reassorting each group according to the second item entering into the combination, and so on, and finally counting on a few counters the last item of the combination for each group of cards."*

Least-significant-digit-first radix sort seems to be a folk invention originated by machine operators.

"MODERN" IBM CARD

- One character per column.



Produced by the WWW Virtual Punch-Card Server.

So, that's why text windows have 80 columns!

WEB RESOURCES ON PUNCHED-CARD TECHNOLOGY

- Doug Jones's punched card index
- Biography of Herman Hollerith
- The 1890 U.S. Census
- Early history of IBM
- Pictures of Hollerith's inventions
- Hollerith's patent application (borrowed from Gordon Bell's CyberMuseum)
- Impact of punched cards on U.S. history