

Sorting Algorithms

Sorting

Sorting. Given n elements, rearrange in ascending order.

Obvious sorting applications.

- List files in a directory.
- Organize an MP3 library.
- List names in a phone book.
- Display Google PageRank results.

Problems become easier once sorted.

- Find the median.
- Find the closest pair.
- Binary search in a database.
- Identify statistical outliers.
- Find duplicates in a mailing list.

Non-obvious sorting applications.

- Data compression.
- Computer graphics.
- Interval scheduling.
- Computational biology.
- Minimum spanning tree.
- Supply chain management.
- Simulate a system of particles.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.
- ...

1. Basic Algorithms: Bubble Sort

Bubble sort

Compare each element (except the last **one**) with its neighbor to the right

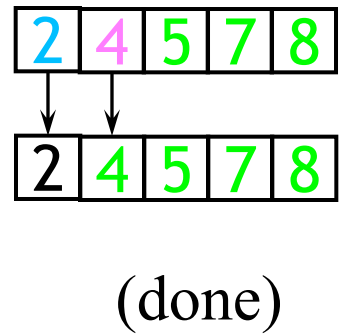
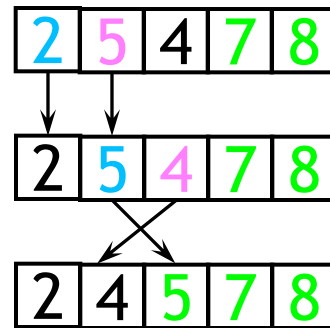
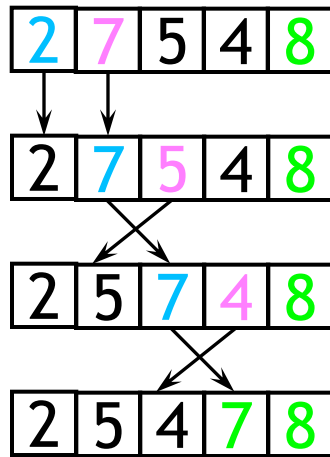
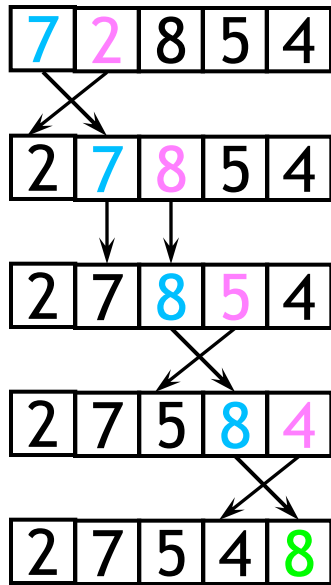
- If they are out of order, swap them
- This puts the largest element at the very end
- The last element is now in the correct and final place

Compare each element (except the last **two**) with its neighbor to the right

- If they are out of order, swap them
- This puts the second largest element next to last
- The last two elements are now in their correct and final places

Continue as above for n iterations

Example of bubble sort



Analysis of bubble sort

Claim: The time complexity $T(n)$ of Bubble Sort is $O(n^2)$

- n operations in the first scan
- $(n-1)$ operations in the second scan
- .
- .
- .
- $(n-(i-1))$ operations in i -th scan
- Adding the number of operations over all scans we have
 - $n+(n-1)+(n-2) + \dots + 1 = O(n^2)$

Analysis of bubble sort

Lower bound:

Bubble sort always spends $n+(n-1)+(n-2) + \dots + 1 = \Omega(n^2)$

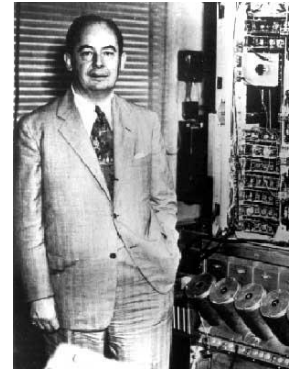
So Bubble Sort has time complexity $\theta(n^2)$

2. Mergesort

Mergesort

Mergesort.

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole (in linear time)



Jon von Neumann (1945)

A L G O R I T H M S

A L G O R

I T H M S

divide

A G L O R

H I M S T

sort $2T(n/2)$

A G H I L M O R S T

merge $O(n)$
(see last class)

A Useful Recurrence Relation

Mergesort recurrence.

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

solve left half solve right half merging

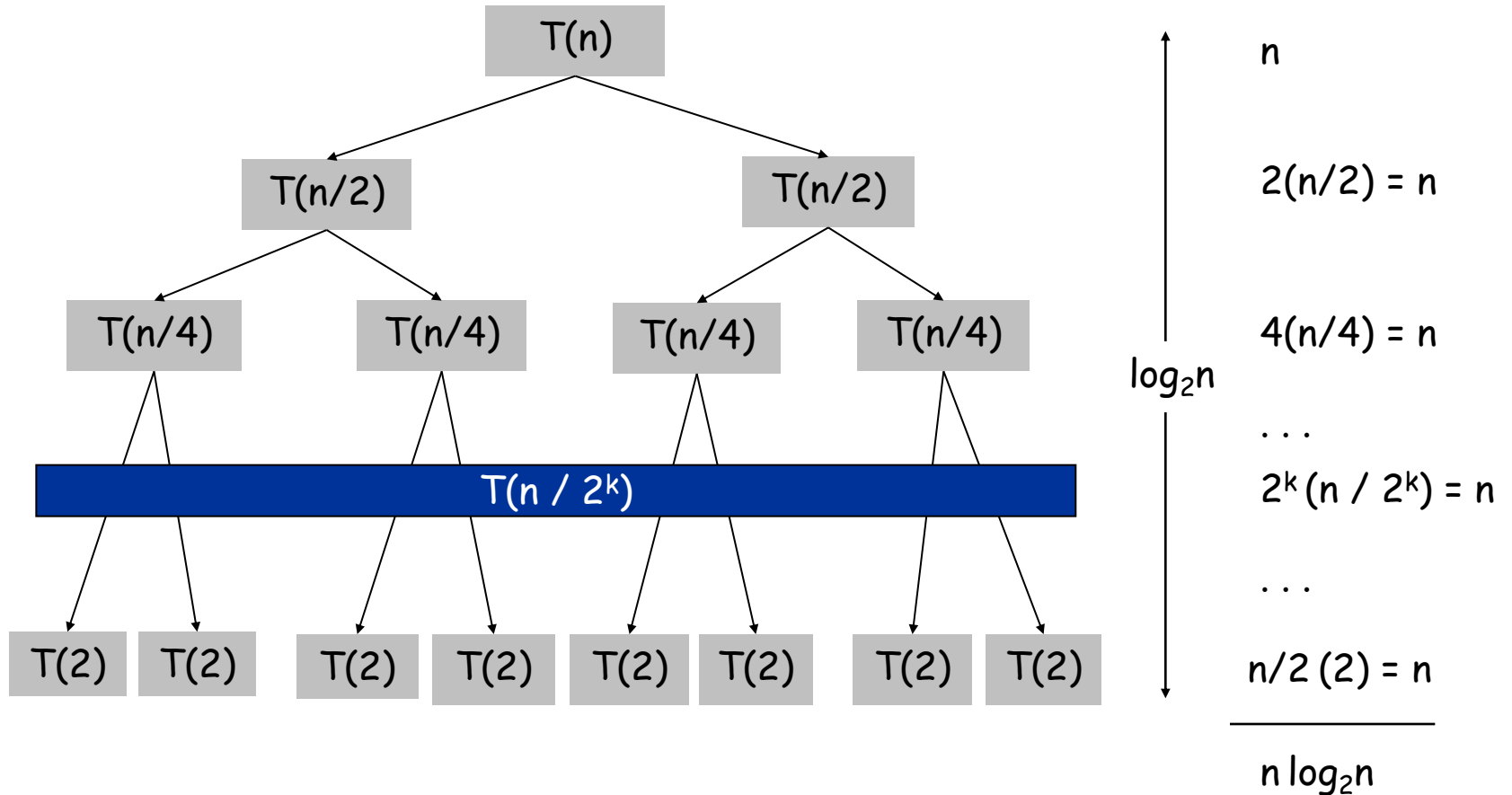
Solution. $T(n)$ is $O(n \log_2 n)$.

Proofs. We describe several ways to prove this recurrence. Initially we ignore floor/ceiling

Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

(sorting both halves)
(merging)



Proof by Induction

Claim. If $T(n)$ satisfies this recurrence, then $T(n) = n \log_2 n$.

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 2T(n/2) + n & \text{otherwise} \end{cases}$$

sorting both halves merging

Pf. (by induction on n)

- Base case: $n = 1$.
- Inductive hypothesis: for $n' \leq n-1$, $T(n') = n' \log_2 n'$.
- Goal: show that $T(n) = n \log_2 (n)$.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= (2n/2)(\log_2(n/2)) + n \\ &= (2n/2)(\log_2(n) - 1) + n \\ &= n \log_2(n) \end{aligned}$$

Proof Keeping the Floor/Ceiling

(This is just so you can see that everything works out if you keep floor/ceiling, do not worry about it)

Claim. If $T(n)$ satisfies the following recurrence, then $T(n) \leq n \lceil \lg n \rceil$.

$$T(n) \leq \begin{cases} 0 & \text{if } n=1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \{n & \text{otherwise} \\ \text{solve left half} & \text{solve right half} & \text{merging} \end{cases}$$

\uparrow
 $\log_2 n$

Pf. (by induction on n)

- Base case: $n = 1$.
- Define $n_1 = \lfloor n / 2 \rfloor$, $n_2 = \lceil n / 2 \rceil$.
- Induction step: assume true for $1, 2, \dots, n-1$.

$$\begin{aligned} T(n) &\leq T(n_1) + T(n_2) + n \\ &\leq n_1 \lceil \lg n_1 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &\leq n_1 \lceil \lg n_2 \rceil + n_2 \lceil \lg n_2 \rceil + n \\ &= n \lceil \lg n_2 \rceil + n \\ &\leq n(\lceil \lg n \rceil - 1) + n \\ &= n \lceil \lg n \rceil \end{aligned}$$

$$\begin{aligned} n_2 &= \lceil n/2 \rceil \\ &\leq \left\lceil 2^{\lceil \lg n \rceil} / 2 \right\rceil \\ &= 2^{\lceil \lg n \rceil} / 2 \\ \Rightarrow \lg n_2 &\leq \lceil \lg n \rceil - 1 \end{aligned}$$

Mergesort

In conclusion, Mergesort take time $O(n \log n)$

Interesting: Does not need to compare all pairs of items (that would be $\Omega(n^2)$)

2. Quicksort

Quicksort

- Sorts $O(n \lg n)$ in the **average** case (we will not prove)
- Sorts $\theta(n^2)$ in the **worst** case

So why would people use it instead of Mergesort?

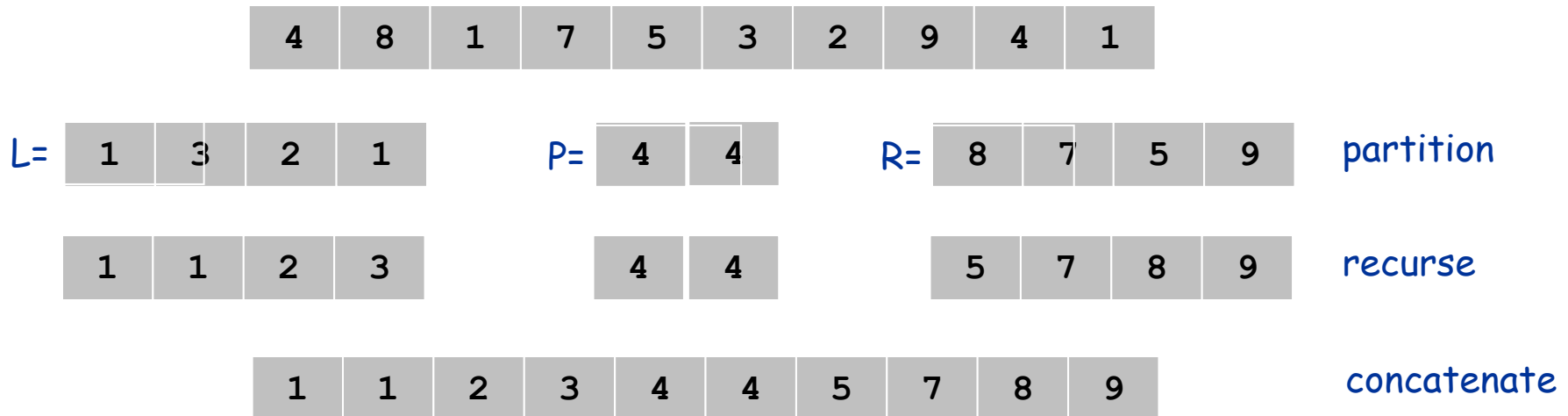
- Very simple to implement
- In **practice** is very fast (worst case is quite rare and constants are low)

Quicksort

Input: list of numbers $A[1], A[2], \dots, A[n]$

Quicksort algorithm:

- Choose a number *pivot* in the list; lets say we always choose $\text{pivot} = A[1]$
- **Partition** the list A into the list L containing all numbers $< \text{pivot}$, the list R containing all numbers $> \text{pivot}$, and list P containing all numbers $= \text{pivot}$
- Sort the list L and R **recursively** using Quicksort
- Concatenate L, P, R (in this order)



Quicksort

Input: list of numbers $A[1], A[2], \dots, A[n]$

Quicksort algorithm:

- Choose a number *pivot* in the list; lets say we always choose $\text{pivot} = A[1]$
- **Partition** the list A into the list L containing all numbers $< \text{pivot}$, the list R containing all numbers $> \text{pivot}$, and list P containing all numbers $= \text{pivot}$
- Sort the list L and R **recursively** using Quicksort
- Concatenate L, P, R (in this order)

```
Quicksort(A)
  if (A.len > 1)
    L, R, P = Partition(A);
    Quicksort(L);
    Quicksort(R);
  end if
  Return the concatenation of L,P,R
```

Partition

We can implement partition step in $O(n)$ using a auxiliary vectors

Can even do *in place* without any auxiliary memory

Analyzing Quicksort

Claim: The time complexity of Quicksort is $O(n^2)$

Proof:

- The height of execution tree of QuickSort is at most n (lists L and R are **strictly smaller** than A ; that's why we put pivot in P).
- At each level of the tree the algorithm spends $O(n)$ due to the partition procedure
- So the algorithm spends at most cn^2 operations

Analyzing Quicksort

Second proof (by induction): We have the recurrence

$$T(0) = c$$

$$T(1) = c$$

$$T(n) \leq \max_{i=0 \dots n-1} \{T(i) + T(n - i - 1) + cn\}$$

← We do not know
the size of the
subarrays

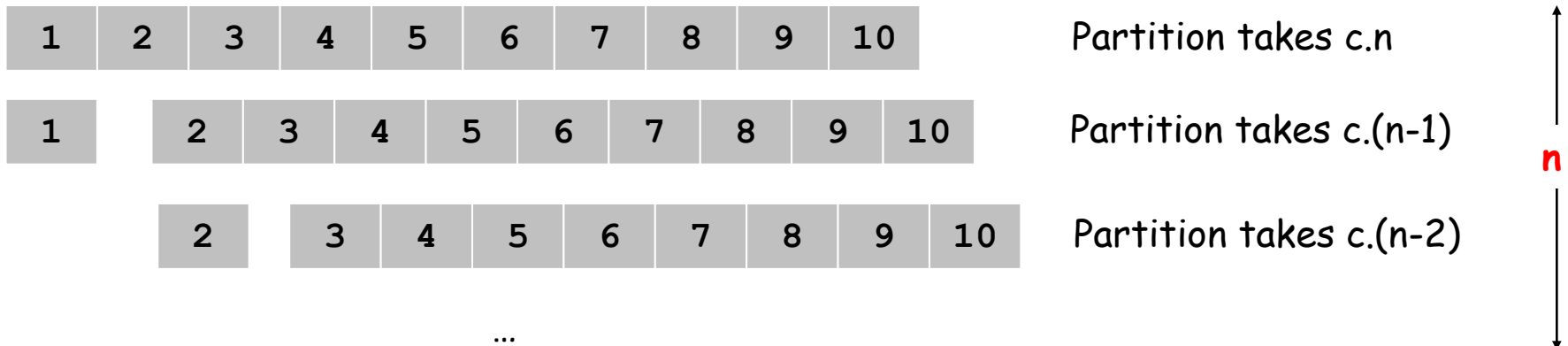
Proof by induction that $T(n) \leq cn^2$

- Base case $n=0,1$ is ok
- Let i^* be the value of i that maximizes the expression. By induction $T(n) \leq c(i^*)^2 + c(n-i^*-1)^2 + cn$
- The right-hand side is a parabola in i^* . So it takes maximum value with $i^*=0$ or $i^*=n-1$. Plugging in these values we get the induction hypothesis

Analyzing Quicksort

Is this the best analysis analysis we can do?

Yes: If the input is sorted, the partition is always **unbalanced**: height **n**



Algorithm takes $c.n + c.(n-1) + c.(n-2) + \dots + c = c.n.(n+1)/2 = \Omega(n^2)$

So Quicksort is $\Omega(n^2)$

So Quicksort is $\Theta(n^2)$

Analyzing Quicksort: Average Case

(Nao 'e cobrado)

$A(n)$: time complexity of Quicksort assuming instance is a random permutation

So partition generates splits (1:n-1, 2:n-2, 3:n-3, ... , n-2:2, n-1:1) each with probability $1/n$

So it is often balanced

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

We can solve the recurrence to get that $A(n)$ is $O(n \log n)$

Improving Quicksort

The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input

Book discusses two solutions:

- Randomize the input array, OR
- Pick a random pivot element

How will these solve the problem?

- By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ expected time

Exercícios

Exercício 1: Descreva um algoritmo com complexidade $O(n \log n)$ com a seguinte especificação:

Entrada: Uma lista de n números reais

Saida: SIM se existem números repetidos na lista e NÃO caso contrário

Exercícios

Solucao:

Ordene a lista L usando MergeSort

For i=1 to |L|-1

If L[i]=L[i+1]

 Return SIM

Return NÃO

Complexidade:

- A ordenação da lista L requer $O(n \log n)$ utilizando o MergeSort
 - O loop **For** requer tempo no maximo linear
- => Total: $O(n \log n)$

Exercícios

Exercício 2: Descreva um algoritmo com complexidade $O(n \log n)$ com a seguinte especificação:

Entrada: lista A de n números reais e um número real x

Saida: SIM se existem dois elementos em S com soma x e NÃO caso contrário

Exercícios

Solucao:

Ideia: Para cada elemento $A[i]$, procure por seu “par” $x - A[i]$ (ou seja, por um elemento tal que somado com $A[i]$ de igual a x)

```
Ordene o conjunto A usando MergeSort
For i = 1 to len(A)
    BinarySearch(A, A[i] - x)
    If busca binaria encontrou tal elemento
        Return SIM
Return NÃO
```

Complexidade:

- Ordenacao gasta $O(n \log n)$ operacoes
 - Cada iteracao do **For** gasta $O(\log n)$ operacoes (devido a busca bin.)
 - **For** gasta no total $O(n \log n)$
- => Total: $O(n \log n)$

Exercícios

Exercício 3:

(1.5 pts) Analise a complexidade do algoritmo abaixo, ou seja, se $T(n)$ é a complexidade para entradas com tamanho n , encontre uma função $f(n)$ tal que $T(n) = \Theta(f(n))$. Justifique **tanto a parte $O(f(n))$ quanto $\Omega(f(n))$** .

```
Algo2( $A$  : lista de números)
  If  $A$  vazia
    Return
  End if
  pivot  $\leftarrow A[1]$ 
   $B \leftarrow \emptyset$  //lista que conterà elementos menores que pivot
  For  $i = 1$  to  $|A|$ 
    If ( $A[i] < \text{pivot}$ ), adicione  $A[i]$  à lista  $B$ 
  End for
  Algo2( $B$ )
End algorithm
```

5.4 Sorting in linear time

Sorting In linear time

- As notas do vestibular são números entre 1 e 100
- 10.000 alunos prestaram vestibular. Como ordenar os alunos conforme sua classificação?
- Versão reduzida para exemplo
 - 10 alunos com notas {1,2,2,3,4,1,2,2,4,3}

Sorting In linear time

We will see **Counting sort**

No comparisons between elements!

But...depends on assumption about the numbers being sorted

- We assume numbers are **in the range $1..k$**

Counting Sort

Input: List of integers $A[1], A[2], \dots, A[n]$ with values between 1 and k

Main idea: Compute vector C where $C[i]$ is number of elements in the list at most i

How can we use this to sort?

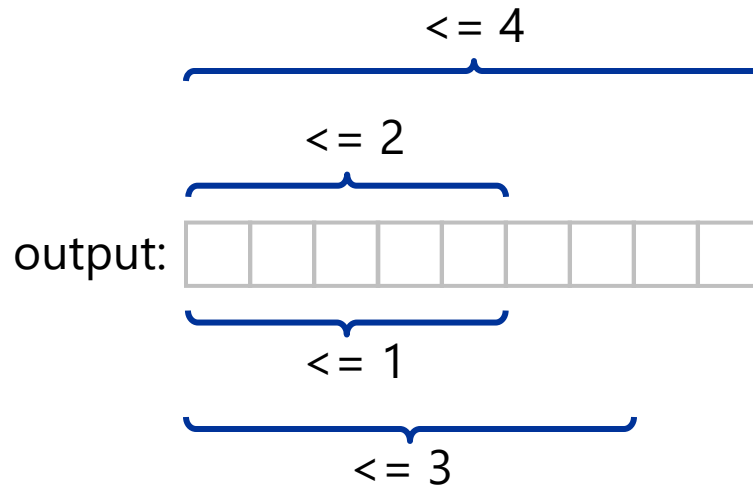
Ex 1: Consider distinct numbers $A = [3, 1, 5, 10, 7]$

- Counting vector is $C = [1, 1, 2, 2, 3, 3, 4, 4, 4, 5]$
- Put number "3" in position $C[3]$, number "1" in position $C[1]$, ...

Counting Sort

Ex 2: Consider non-distinct numbers $A = [1, 3, 3, 1, 1, 4, 1, 1, 4]$

- Counting vector is $C = [5, 5, 7, 9]$
- Put (say) second "3" in position $C[3]$, put first "3" in position $C[3] - 1, \dots$



Counting Sort

Q: How to compute counting vector C in time $O(n)$?
(recall: want $C[i]$ to be the number of elements with value at most i)

- First compute vector `equal`, where

$\text{equal}[i] = \text{\#elements with value **equal** to } i$

- Compute $C[1] = \text{equal}[1]$
 $C[2] = C[1] + \text{equal}[2]$
 $C[3] = C[2] + \text{equal}[3]$
...

Counting Sort

```
1 CountingSort(A, B, k) //A is input, B is output
2   for i=1 to k           ← Initialize
3       C[i]= 0, equal[i]=0;
4   for j=1 to n           ← stores the number
5       equal[A[j]] += 1;  of elements equal
6   for i=1 to k           ← C[i] stores the number
7       C[i] = equal[i]+C[i-1]; of elements smaller or
8   for j=n downto 1      equal to i, i = 1,...,k
9       B[C[A[j]]] = A[j];
10      C[A[j]] = C[A[j]] - 1; ← The position of element A[j]
                                in B is equal to the number of
                                integers that are at most A[j],
                                which is C[A[j]]
```

Ex: $A = \{1,2,2,3,4,1,2,2,4,3\}$

Counting Sort

Q: What is the running time of Counting Sort?

```
1 CountingSort(A, B, k) //A is input, B is output
2   for i=1 to k
3       C[i]= 0, equal[i]=0;
4   for j=1 to n
5       equal[A[j]] += 1;
6   for i=1 to k
7       C[i] = equal[i]+C[i-1];
8   for j=n downto 1
9       B[C[A[j]]] = A[j];
10      C[A[j]] = C[A[j]]- 1;
```

Takes time $O(k)$

Takes time $O(n)$

Counting Sort

- Total time: $O(n + k)$
 - In many cases, $k = O(n)$; when this happens, Counting sort is $O(n)$
- But sorting is $\Omega(n \lg n)$!
 - No contradiction--this is **not** a comparison sort (in fact, there are *no* comparisons at all!)
- Notice that this algorithm is *stable*
 - If x and y are two identical numbers and x is before y in the input vector then x is also before y in the output vector (Important for Radix sort)
 - That is the only reason why the last "For" is in reverse order

Counting Sort

Cool! *Why don't we always use counting sort?*

Because **complexity depends on range k** of elements



Very important

Could we use counting sort to sort 32 bit integers?

Answer: We **can** (bounded range) but **should not**: range is too large
($k = 2^{32} = 4,294,967,296$)

Counting Sort

Exercicio: Temos uma lista de tamanho n com numeros em $\{1,2,..,n\}$, possivelmente repetidos.

De um algoritmo em tempo $O(n)$ para retornar o numero que aparece mais vezes na lista.

Solucao 1: Monte o vetor contador do Counting Sort e encontre a posicao com maior contagem.

Solucao 2: Use Counting Sort e faça uma varredura na lista ordenada contando a maior sequencia de numeros iguais (como fazer isso em $O(n)$?)