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

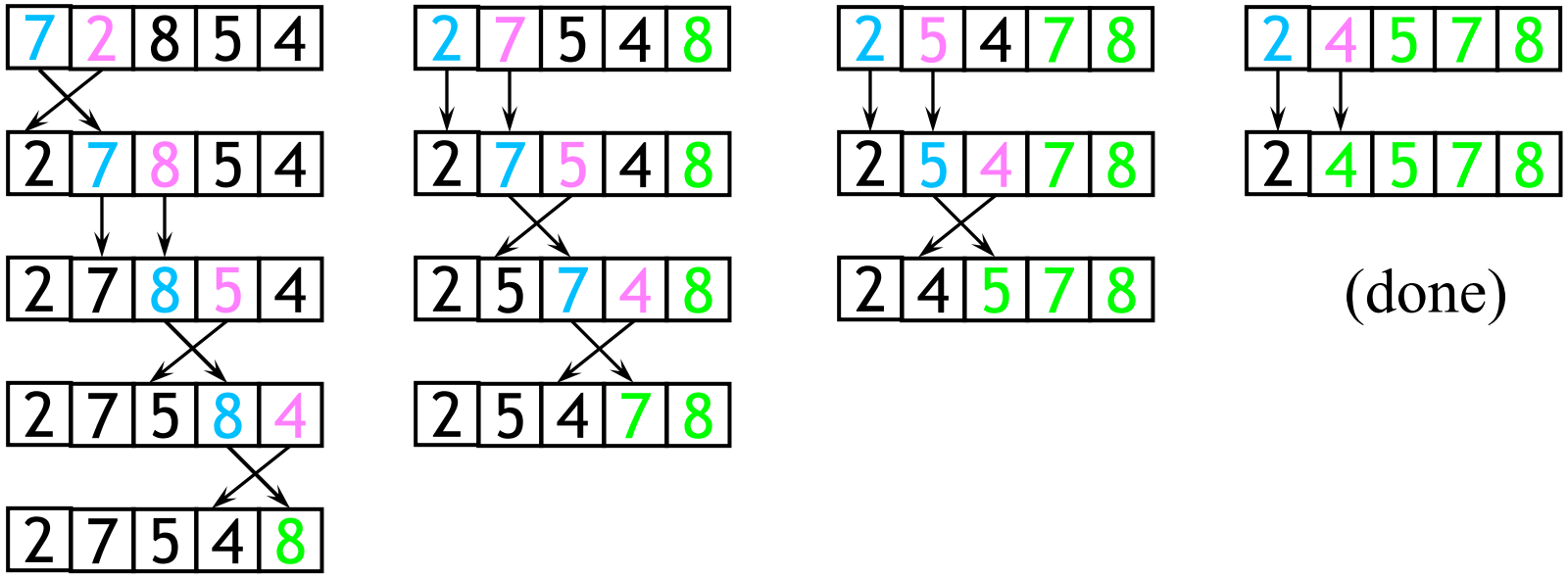
- If they are out of order, swap them
- This puts the largest element at the very end
- The last element is now in the correct and final place

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

$$n+(n-1)+(n-2) + \ldots + 1 = O(n^2)$$

Analysis of bubble sort

Lower bound:

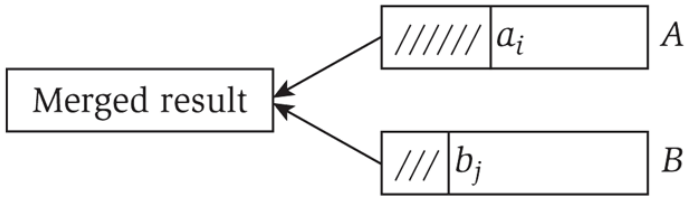
Bubble sort always spends $n+(n-1)+(n-2) + \ldots + 1 = \Omega(n^2)$

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

2. Mergesort

Building block: Merge

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ (increasing order) into sorted whole.




```
i = 1, j = 1
while (i <= |A| and j <= |B|) {
   if (a_i ≤ b_j) append a_i to output list and increment i
   else           append b_j to output list and increment j
}
append remainder of nonempty list to output list
```

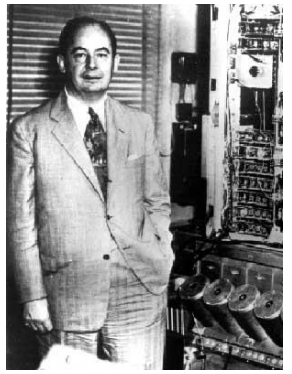
Claim. Merging two lists of size n takes O(n) time

Pf. After at most 2n iterations, both pointers i,j reach n, can't go any further

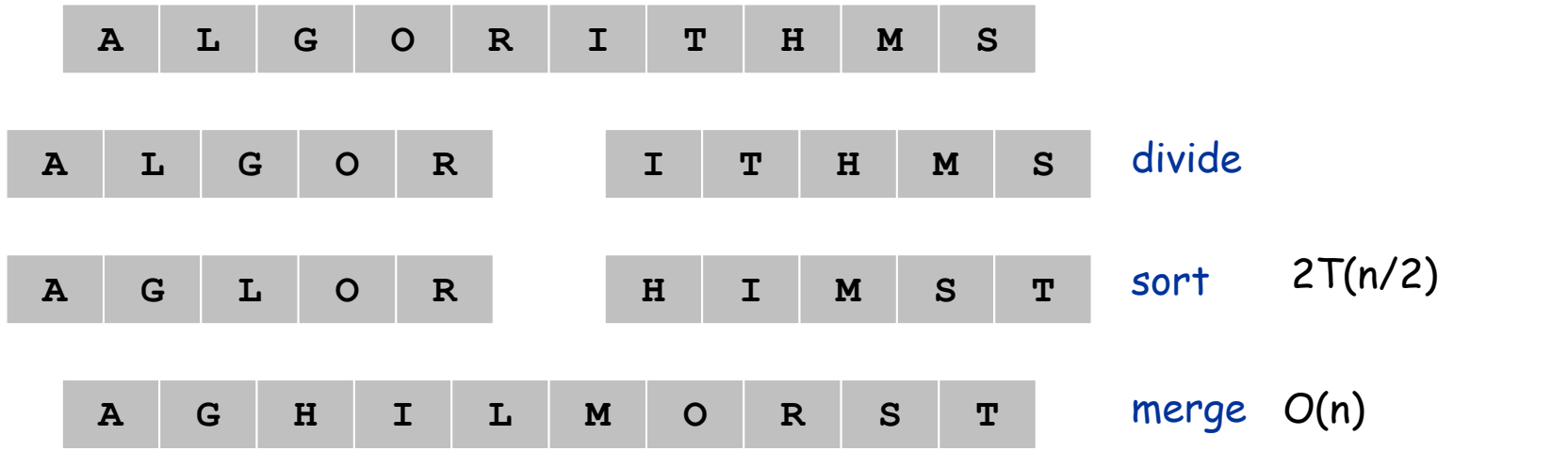
Mergesort

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 Useful Recurrence Relation

Mergesort recurrence.

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

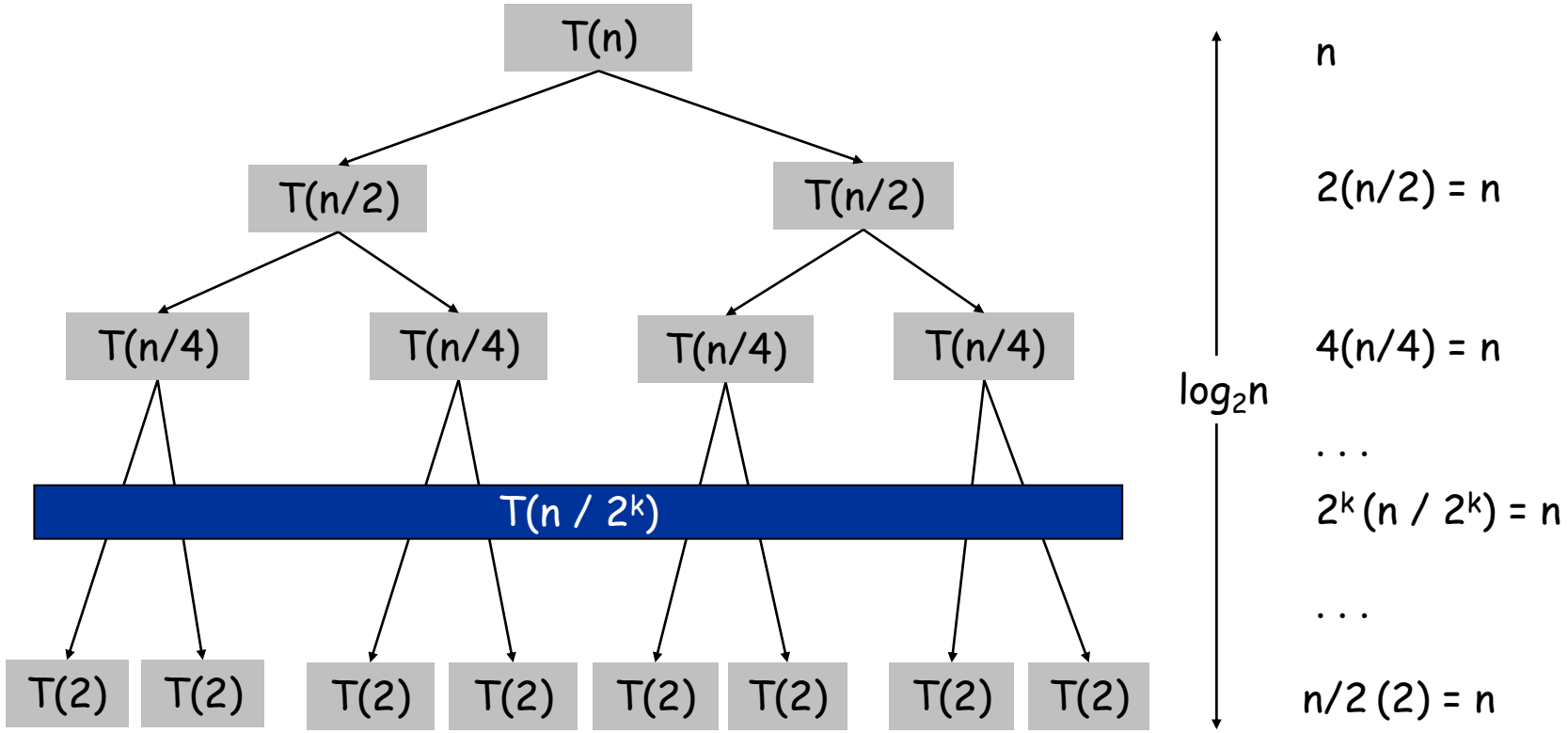
Solution. T(n) is $O(n \log_2 n)$.

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

Proof by Recursion Tree

Proof by Recursion Tree

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Proof by Induction

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

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

Pf. (by induction on n)

- Base case: n = 1.
- Inductive hypothesis: for n′ <= 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 $T(n) \le n \lceil \lg n \rceil$.

$\lg n$ = $\log_2 n$

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

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, ... , n–1.

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

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

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?

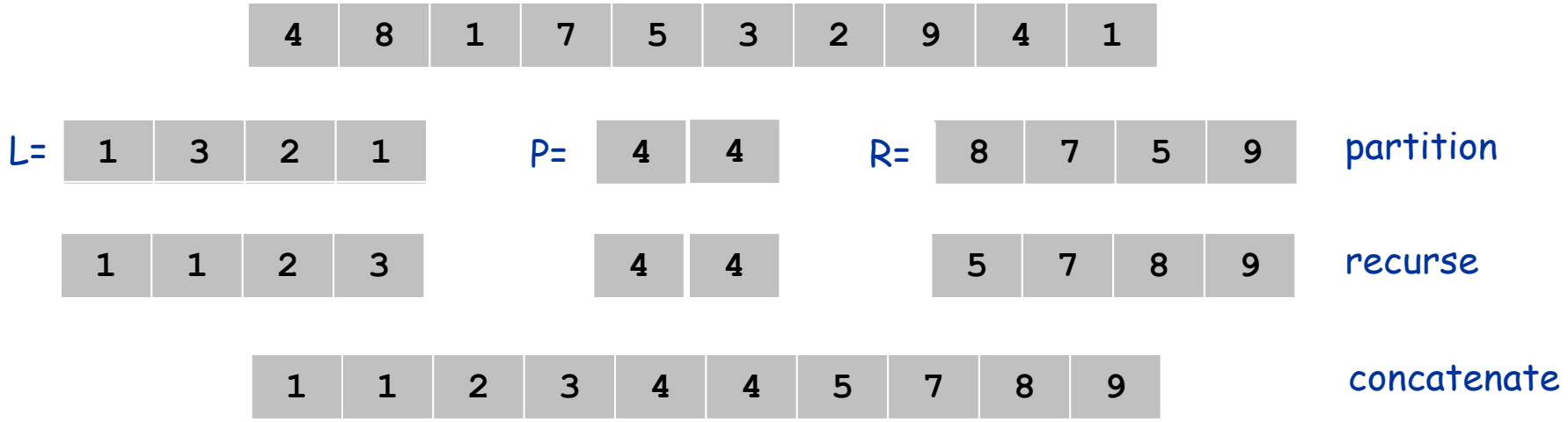
- Very simple to implement
- In practice is very fast (worst case is quite rare and constants are low)

Quicksort

Input: list of numbers A[1], A[2], …, A[n]

Quicksort algorithm:

- Choose a number *pivot* in the list; lets say we always choose *pivot* = A[1]
- Partition the list A into the list L containing all numbers $< pivot$, the list R containing all numbers $> pivot$, and list P containing all numbers $= pivot$
- 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], …, A[n]

Quicksort algorithm:

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

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

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 stricly 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 precedure

- So the algorithm spends at most cn^2 operations

Analyzing Quicksort

Second proof (by induction): We have the recurrence

$$T(0) = c$$
$$T(1) = c$$
$$T(n) <= \max_{i=0\ldots n-1} \{T(i) + T(n - i - 1) + cn\}$$

We do not know the size of the subarrays

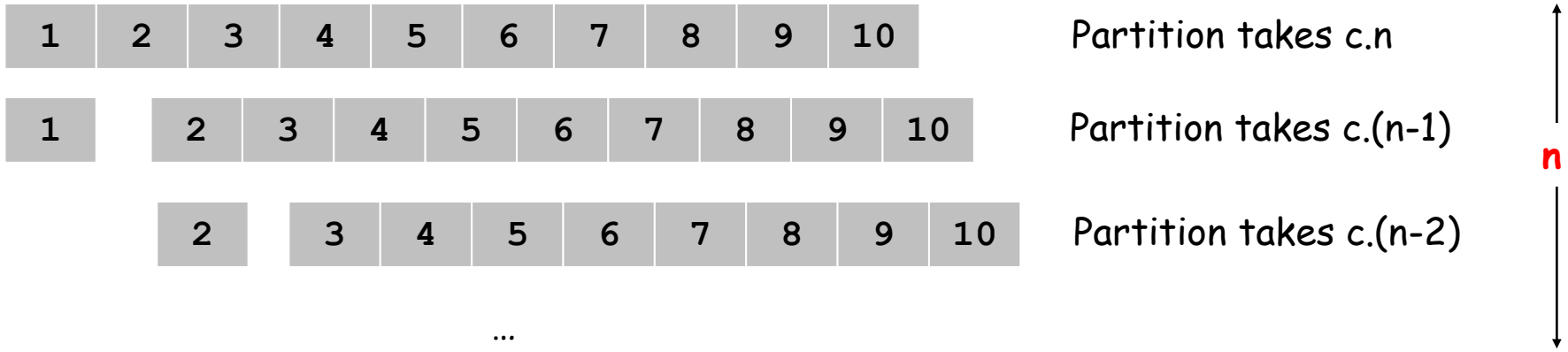
Proof by induction that $T(n) <= cn^2$

- Base case n=0,1 is ok
- Let i* be the value of i that maximizes the expression. By induction $T(n) <= c(i^*)^2 + c(n-i^*-1)^2 + cn$
- 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) + ... + c = c.n.(n+1)/2 = $\Omega(n^2)$

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

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

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

Exercicio 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

Exercicio 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

Exercicio 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 ← A[1]
    B ← ∅                    //lista que conterá elementos menores que pivot
    For i = 1 to |A|
        If (A[i] < 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 ordernar os alunos conforme sua classificação?

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], …, 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 =
- C[3] = 2 ⇔

- Put number "3" in position C[3], number "1" in position C[1], …

output:

	3			

Counting Sort

Input: List of integers A[1], A[2], …, 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]
- C[3] = 2 ⇔ 2 elements in the input with value at most "3"
  ⇔ "3" is the second smallest element
  ⇔ should put "3" on the **second position** of output!

- Put number "3" in position C[3], number "1" in position C[1], …

output:

	3			

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]
- Since C[1] = 5
  - The first "1" we find, we put in **position 5** of the output (it is still the fifth smallest element)
  - The next "1" we put in the **previous position**
  - …
- Put first "1" in position C[1], put second "1" in position C[1] – 1,…

output:

Counting Sort

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

- First compute vector equal, where

  equal[i] = #elements with value **equal** to i

- Compute C[1] = equal[1]
  C[2] = C[1] + equal[2]
  C[3] = C[2] + equal[3]
  …

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;
```

Initialize

stores the number of elements **equal** to i,i =1,…,k

C[i] stores the number of elements smaller or equal to i,i =1,…,k

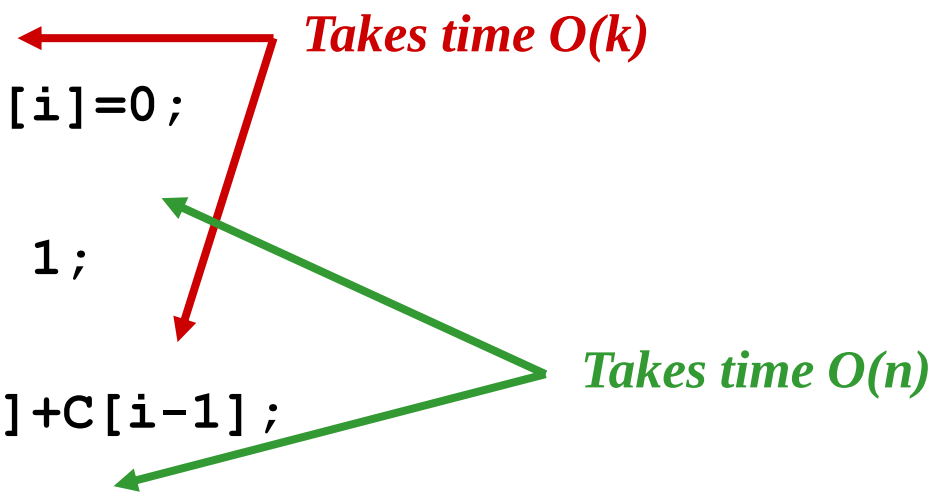
The position of element A[j] in B is equal to the number of integers that are at most A[j], which is C[A[j]]

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;
```



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.

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 equal do Counting Sort e encontre a posicao com maior contagem.

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