

# 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 and Selection Sort

# Bubble sort

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

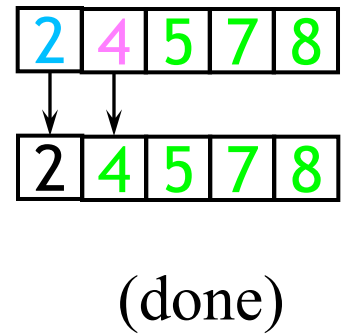
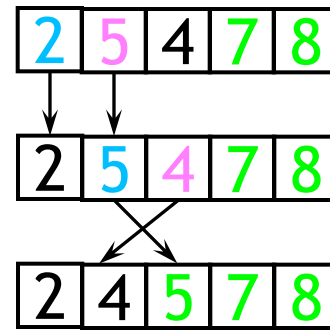
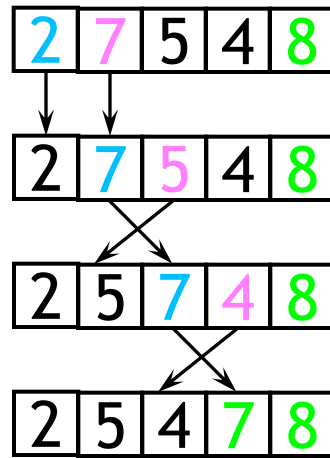
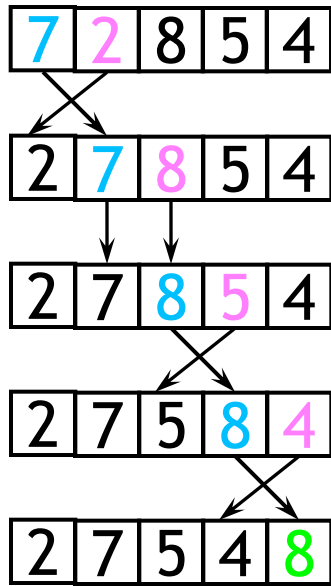
- If they are out of order, swap them
- This puts the largest element at the very end
- The last element is now in the correct and final place

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 until no elements are swapped during a scan

# 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:** In a list sorted from the largest to the smallest element we spend  $n+(n-1)+(n-2) + \dots + 1 = \Omega(n^2)$

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

**Obs:** In the best case (sorted list) we just scan the list once:  $O(n)$  time

## Selection sort

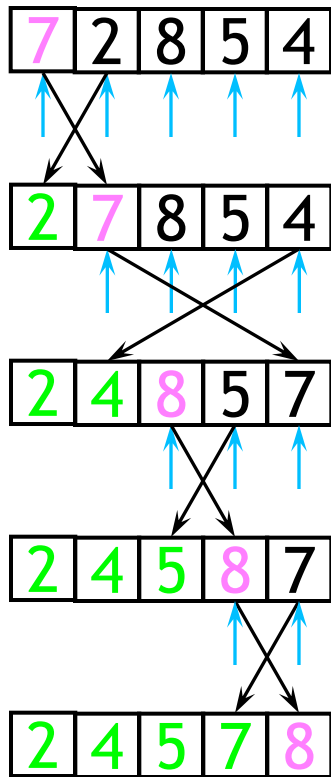
Given an array of length  $n$ ,

- Search elements **1** through  $n$  and select the smallest
  - Swap it with the element in location **1**
- Search elements **2** through  $n$  and select the smallest
  - Swap it with the element in location **2**
- Search elements **3** through  $n$  and select the smallest
  - Swap it with the element in location **3**
- Continue in this fashion until there's nothing left to search



## Example and analysis of selection sort

### Analysis:



- The outer loop executes  **$n-1$**  times
- The  $i$ -th internal loop executes  $(n-i)$  operations
- Work done in the inner loop is constant (swap two array elements)
- Time complexity:  $O(n^2)$
- Also  $\Omega(n^2)$  (for every instance)
- So it is  $\Theta(n^2)$

## 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.

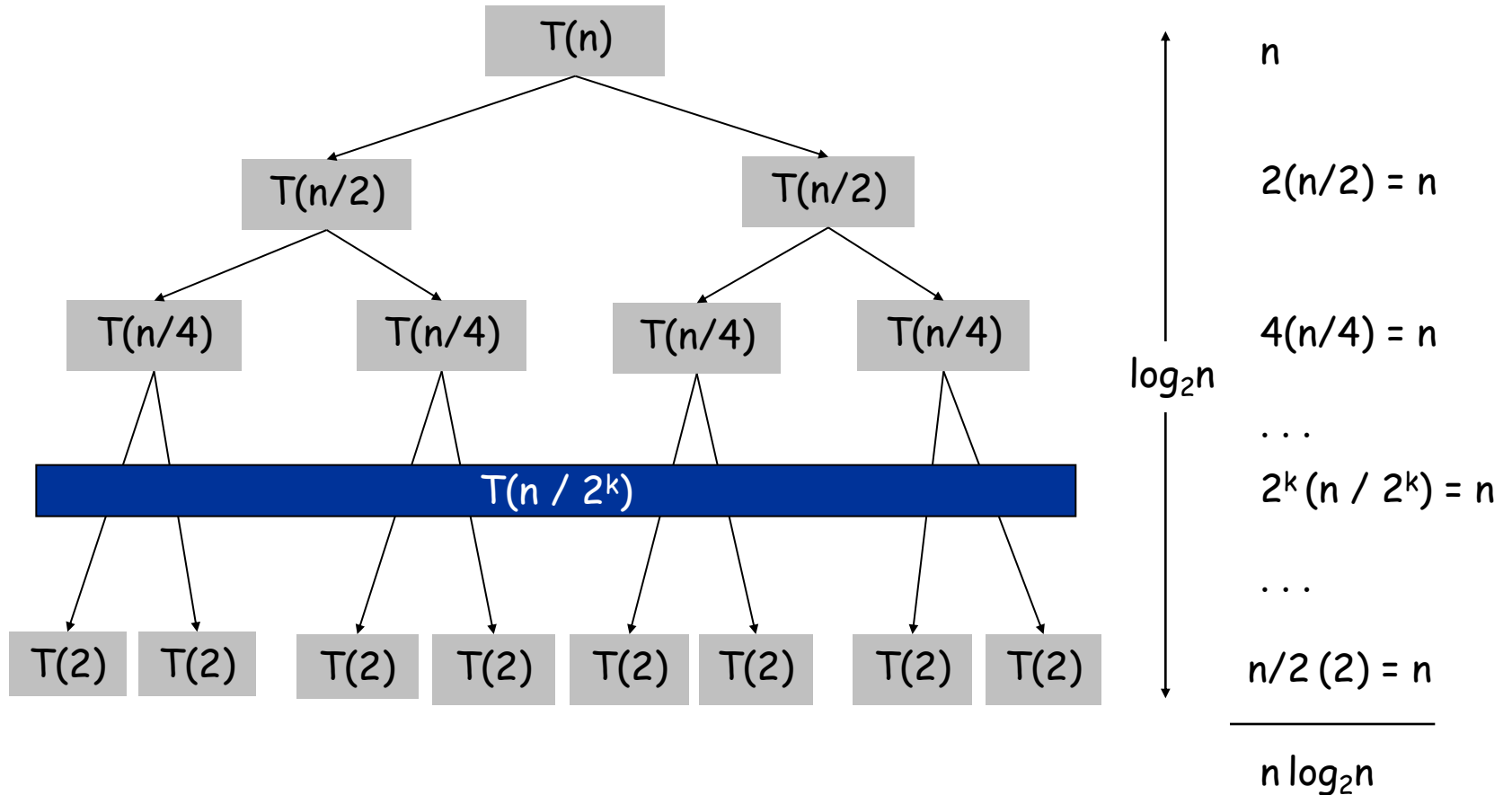
$$T(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ \underbrace{T(\lceil n/2 \rceil)}_{\text{solve left half}} + \underbrace{T(\lfloor n/2 \rfloor)}_{\text{solve right half}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

**Solution.**  $T(n) = O(n \log_2 n)$ .

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

# Proof by Recursion Tree

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



# 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  $n' \leq n-1$ ,  $T(n') = n' \log_2 n'$ .
- 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  $T(n) \leq n \lceil \lg n \rceil$ .

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

$\uparrow$   
 $\log_2 n$

**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?*

*In practice is very fast (worst case is quite rare and constants are low)*

# Quicksort

A divide-and-conquer algorithm

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 *pivot* =  $A[1]$
- Move number around so that all number in  $A[1..q-1]$  are  $\leq$  pivot,  $A[q]$  = pivot, and all elements in  $A[q+1..n]$  are  $>$  pivot (called *partition* step)
- Sort these subarrays  $A[1..q - 1]$  and  $A[q+1..n]$  recursively using Quicksort

```
Quicksort(A, p, r)
  if (p < r)
    q = Partition(A, p, r);
    Quicksort(A, p, q-1);
    Quicksort(A, q+1, r);
  end if
```

# 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  $n$ .
- 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

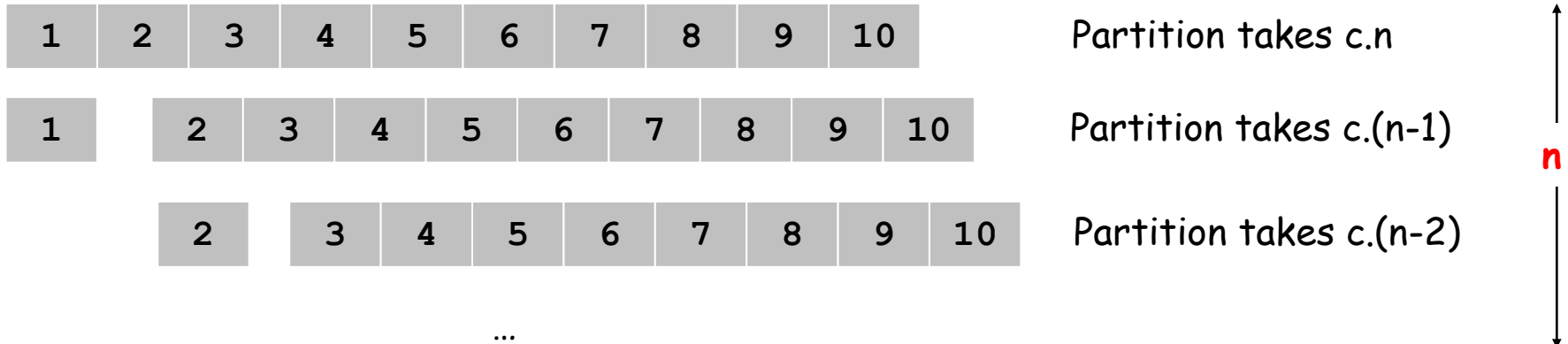
Proof by induction that  $T(n) \leq cn^2$

- 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

## Exercício

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

## Exercício - Solução

Ordene a lista lista L

Para  $i=1$  até  $|L|-1$

Se  $L[i]=L[i+1]$  Devolva SIM

Fim Para

Devolva NÃO

- ANÁLISE

- A ordenação da lista L requer  $O(n \log n)$  utilizando o MergeSort
- O loop Para requer tempo linear

## Exercício

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

Entrada: conjunto  $S$  de  $n$  números reais e um número real  $x$

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

## Exercício - Solução

$L \leftarrow$  conjunto  $S$  em ordem crescente

**Enquanto** a lista  $L$  tiver mais que um elemento **faça**

Some o menor e o maior elemento de  $L$

Se a soma é  $x$

**Devolva SIM**

Se a soma é maior que  $x$

Retire o maior elemento de  $L$

Se a soma é menor que  $x$

Retire o menor elemento de  $L$

**Fim Enquanto**

**Devolva NÃO**

## Exercício - Solução

L <- conjunto S em ordem crescente

**Enquanto** a lista L tiver mais que um elemento **faça**

Some o menor e o maior elemento de L

Se a soma é x

**Devolva SIM**

Se a soma é maior que x

Retire o maior elemento de L

Se a soma é menor que x

Retire o menor elemento de L

**Fim Enquanto**

**Devolva NÃO**

### ▪ ANÁLISE

- A ordenação do conjunto S requer  $O(n \log n)$  utilizando o MergeSort
- O loop enquanto requer tempo linear

3 How fast can we sort?

# How Fast Can We Sort?

Can we sort a list of  $n$  numbers in **constant time** (independent of  $n$ )?

How can we prove that **no algorithm** can sort in constant time?

- This is very hard, research question. We don't know how to do it....

All of the sorting algorithms so far are *comparison sorts*

- The only operation used to gain ordering information about a sequence is the pairwise comparison of two elements

**Easier question:** How many **comparisons** are necessary for any **comparison sort** algorithm?

- Just counting comparisons, reordering of items is free



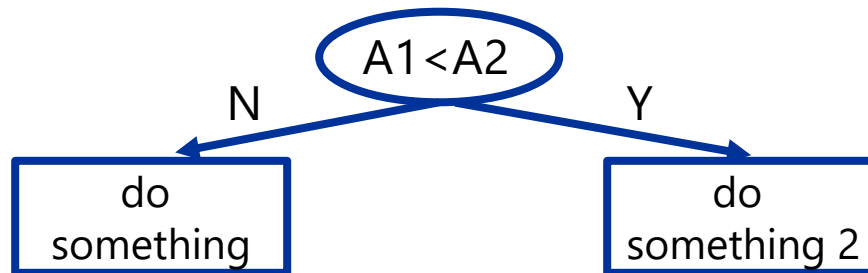
## How Fast Can We Sort?

Ex: Can we sort 3 numbers with just 1 comparison?

A = 

?	?	?
---	---	---

Answer: No: Suppose just compare A[1] with A[2]



But consider A = 

2	3	4
---	---	---

 and A = 

2	3	1
---	---	---

 : both execute "do something 2"

- If "do something 2" **moves** A[3] then it makes a mistake in instance 2,3,4
- If "do something 2" **does not** move A[3] then it makes a mistake in instance 2,3,1

*Need to make enough comparisons to find out order of the instance*

## How Fast Can We Sort?

We will show that **no** comparison algorithm can beat MergeSort

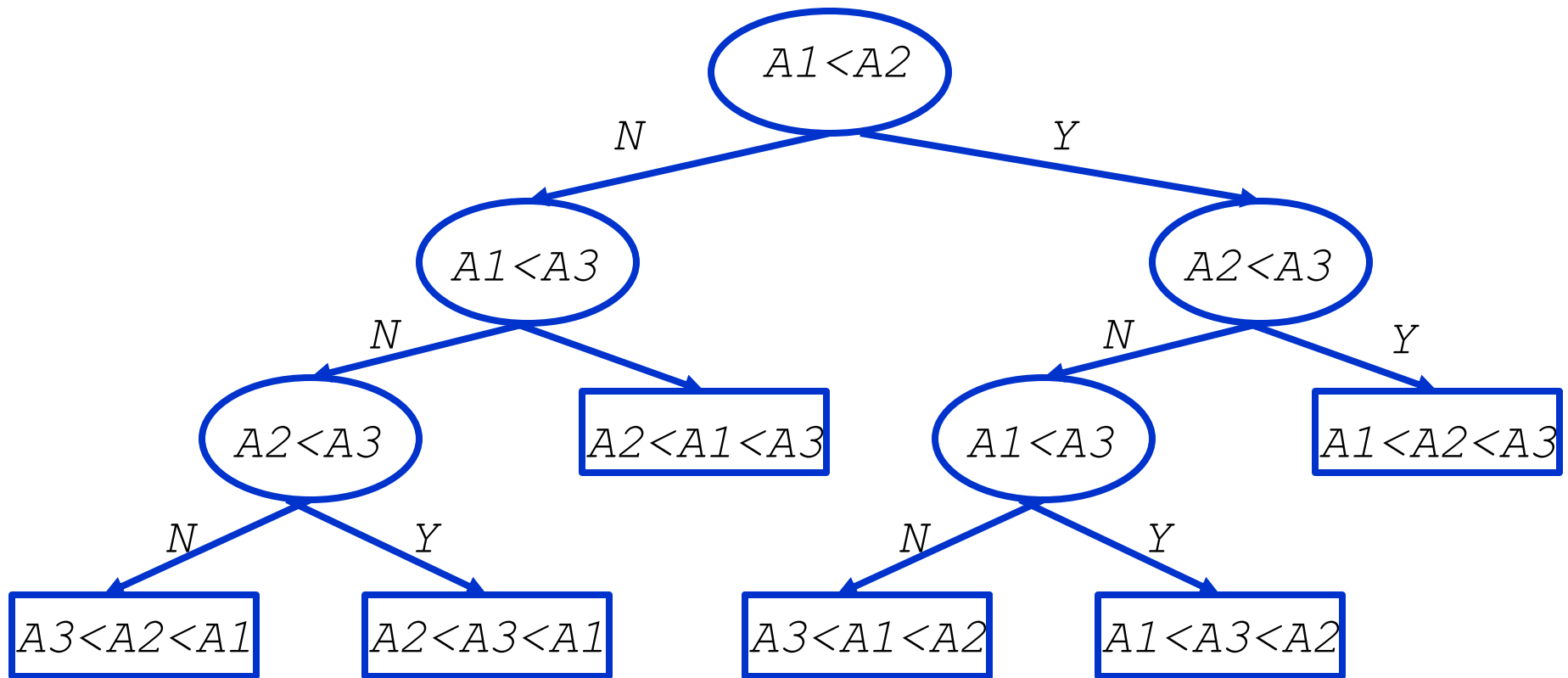
**Theorem:** Any comparison based sorting algorithm needs at least  $\Omega(n \log n)$  comparisons

# Decision Trees

- Decision trees provide an abstraction of comparison sorts
  - A decision tree represents the comparisons made by a comparison sort. Every thing else is ignored
  - Each node corresponds to a comparison
  - Each branch corresponds to an outcome of the comparison
  - There is one leaf for each possible ordering of the numbers

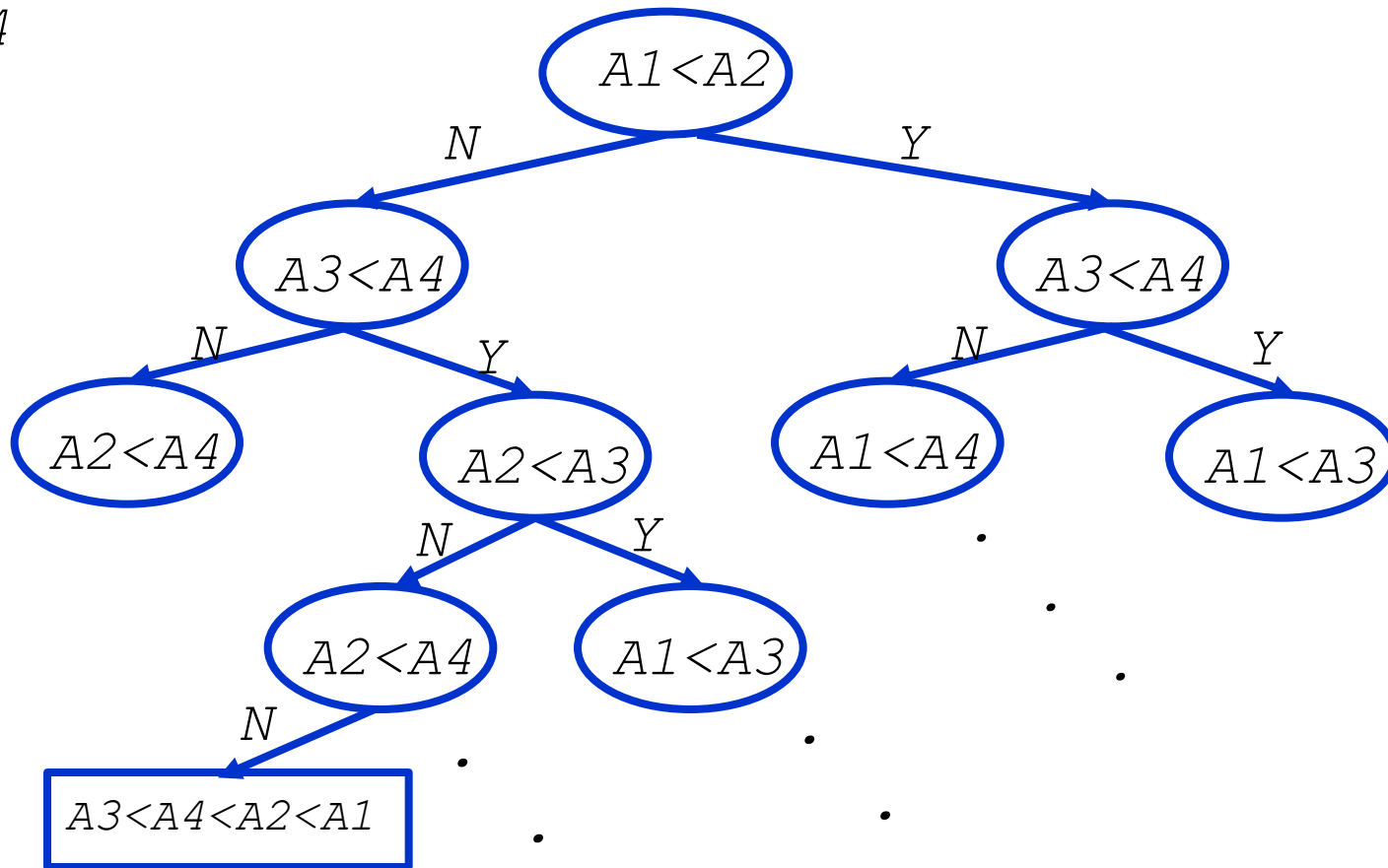
# Decision tree for BubbleSort

Bubble sort for 3 numbers A1, A2, A3



# Decision tree for MergeSort

$n=4$



# Decision Trees

- Decision trees provide an abstraction of comparison sorts

Q: How many leaves does a decision tree have?

Q: What does the height correspond to?

# Decision Trees

- Decision trees provide an abstraction of comparison sorts

Q: How many leaves does a decision tree have?

A:  $n!$

Q: What does the height correspond to?

A: Number of comparisons for worst case instance

What can we say about the height of any decision tree?

## Lower bound for comparison sorting

**Theorem:** The height of any decision tree that sorts  $n$  numbers is at least  $\Omega(n \log n)$

**Proof:**

- Any decision tree has  $n!$  leaves
- The height of a decision tree with  $n!$  leaves is at least  $\log(n!)$  because the decision tree is a binary tree
- $\log(n!) \geq (n/2) \log n - (n/2)$ :

$$\begin{aligned}\log(n!) &= \log(n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1) = \log n + \log(n-1) + \dots + \log 2 + \log 1 \\ &\geq \log n + \log(n-1) + \dots + \log(n/2) \geq (n/2) \log(n/2) \\ &= (n/2) \log n - (n/2)\end{aligned}$$



## Lower bound for comparison sorting

**Theorem:** The height of any decision tree that sorts  $n$  numbers is at least  $\Omega(n \log n)$

**Corollary:** No **comparison sort** can beat MergeSort asymptotically

## 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?

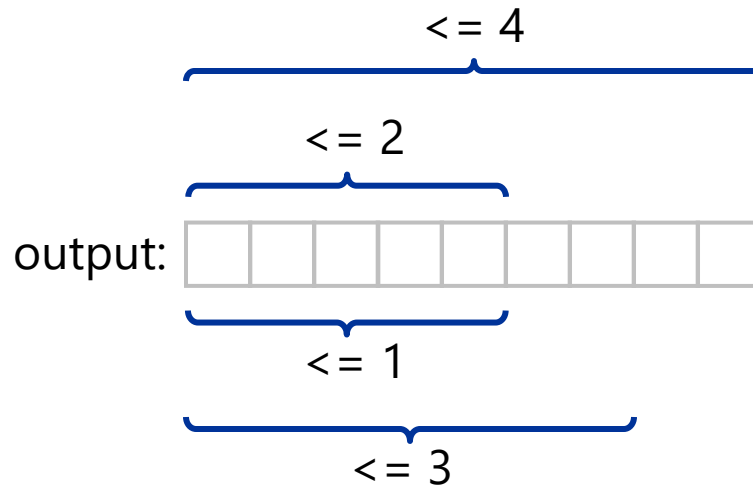
Ex 1: Consider distinct numbers  $A = [3, 1, 5, 10, 7]$

- 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

```
1 CountingSort(A, B, k)
```

```
2   for i=1 to k
```

← Initialize

```
3       C[i] = 0;
```

```
4   for j=1 to n
```

←

C[i] stores the number of elements equal to i, i = 1, ..., k

```
5       C[A[j]] += 1;
```

```
6   for i=2 to k
```

```
7       C[i] = C[i] + C[i-1];
```

←

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

```
8   for j=n downto 1
```

```
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)
2      for i=1 to k
3          C[i] = 0;
4      for j=1 to n
5          C[A[j]] += 1;
6      for i=2 to k
7          C[i] = C[i] + C[i-1];
8      for j=n downto 1
9          B[C[A[j]]] = A[j];
10         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 it depends on range  $k$  of elements

*Could we use counting sort to sort 32 bit integers? Why or why not?*

Answer: no,  $k$  too large ( $2^{32} = 4,294,967,296$ )

# Radix Sort

You have a list of numbers to sort

One idea is to sort them based on most significant digit, then the second m.s.d., etc.

Q: Does it work?

A: Not directly: consider input 91, 19, 55, 54

- Sorting by most significant digit: 19, 55, 54, 91
- Then sorting by second most significant: 91, 54, 44, 91 **wrong order!**

Key idea: sort the **least** significant digit first

```
RadixSort(A, d)
```

```
  for i=1 to d
```

```
    Use a stable sorting algorithm  
    to sort based on digit i
```

# Radix Sort

Ex: (329,457,657,839,436,720)

Step 1: third digit

(720, 436, 457,657, 329,839)

Step 2: second digit

(720, 329,436, 839,457,657)

Step 3: first digit

(329,436,457,657, 720, 839)



That is why we need a stable sorting algorithm

# Radix Sort

- *Can we prove it will work?*
- Sketch of an inductive argument (induction on the number of passes):
  - Assume lower-order digits  $\{j: j < i\}$  are sorted
  - Show that sorting next digit  $i$  leaves array correctly sorted
    - ◆ If two digits at position  $i$  are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - ◆ If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

# Radix Sort

*What sort will we use to sort on digits?*

Counting sort is obvious choice:

- Sort  $n$  numbers on digits that range from 0..9
- Time:  $O(n)$

Each pass over  $n$  numbers with  $d$  digits takes time  $O(n)$ , so total time  $O(dn)$

**Obs:** Does not need to use digits (writing numbers in base 10), can use any other base

# Radix Sort

In general, radix sort based on counting sort is:

- Fast
- Good worst-case guarantee  $O(nd)$
- Simple to code
- A good choice