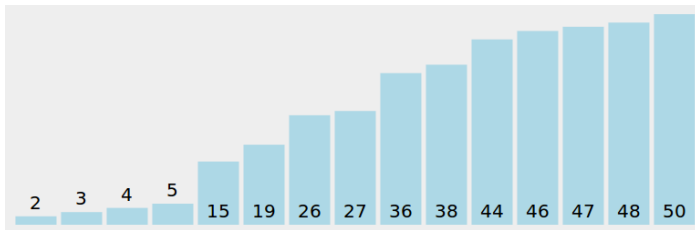


Sorting, Binary Search and Variants

Pedro Ribeiro

DCC/FCUP

2025/2026



- Sorting is an **initial step** to many other algorithms
 - ▶ Ex: finding the median
- When you don't know what to do... **sort!**
 - ▶ Ex: finding repeated elements is much easier after sorting
- **Different sorting types** might be more adequate to different scenarios
 - ▶ Ex: to less general cases, there might be $\mathcal{O}(n)$ algorithms
- It is important to know the sorting functions available on your language **libraries**
 - ▶ Ex: qsort (C), STL sort (C++), Arrays.sort (Java)

About sorting complexity

- What is the least possible complexity for a general sorting algorithm?
 $\Omega(n \log n)$... but only on the comparative model.
 - ▶ **Comparative model:** to distinguish elements I can only use comparisons ($<$, $>$, $=$, $\geq$, $\leq$). How many comparisons are needed?
- A sketch of the **proof** that comparative sorting is **$\Omega(n \log n)$**
 - ▶ Input of size n has **$n!$ possible permutations**
(only one is the desired ordering)
 - ▶ A comparison has **two possible results**
(it can distinguish between 2 different permutations)
 - ▶ Let $f(n)$ be the function that measures the **number of comparisons**
 - ▶ $f(n)$ comparisons: can **distinguish** between $2^{f(n)}$ permutations
 - ▶ We need that $2^{f(n)} \geq n!$, that is, **$f(n) \geq \log_2(n!)$**
 - ▶ Using **Stirling's approximation**, we know that **$f(n) \geq n \log_2 n$**

Some sorting algorithms

- **Comparative algorithms**

- ▶ **BubbleSort** (swap elements)
- ▶ **SelectionSort** (selected smallest/largest)
- ▶ **InsertionSort** (insert on correct position)
- ▶ **MergeSort** (divide in two, sort halves, merge sorted parts)
- ▶ **HeapSort** (create heap with all elements, remove one by one)
- ▶ **QuickSort** (divide according to a pivot and sort recursively)

- **Non Comparative Algorithms**

- ▶ **CountingSort** (count number of elements of each type)
- ▶ **RadixSort** (sort according to "digits")



<https://visualgo.net/en/sorting>

Dancing algorithms 

Non Comparative Algorithms

- To simplify, let's assume that the **elements to sort are integers**
- Idea can be **generalized** to other data types
- Suppose we have n elements to sort in **increasing order**, stored on an array v with indexes from 0 to $n - 1$

CountingSort

- **Key idea:** count the amount of numbers of each type

CountingSort

```
count[max_size] ← frequencies array
For i = 0 to n-1 do
    count[v[i]] ++ (one more v[i] element)
i = 0
For j = min_size to max_size do
    While count[j] > 0 do
        v[i] = j (put element on array)
        count[j] -- (one less element of that size)
        i ++ (increments first free position on the array)
```

You can check an animation at [VisuAlgo](https://www.visualgo.com/)

- Let k be the largest number
- This algorithm will take $\mathcal{O}(n + k)$

RadixSort

- **Key idea:** sort digit by digit

A possible RadixSort (starting on the least significant digit)

```
bucket[10] ← array of lists of numbers (one per digit)
For pos = 1 to max_number_digits do
  For i = 0 to n-1 do (for each number)
    Put v[i] in bucket[digit_position_pos(v[i])]
  For i = 0 to 9 do (for each possible digit)
    While size(bucket[i]) > 0 do
      Take first number of bucket[i] and add it to v[]
```

You can check an animation at [VisuAlgo](#)

- Let k be the largest quantity of digits in a single number
- This algorithm will take $\mathcal{O}(n \times k)$
(or $\mathcal{O}(n \times \log m)$ if m is the maximum number)

Some sorting algorithms

There are many more!

Exchange sorts	Bubble sort · Cocktail sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Stooge sort · Bogosort
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort · Strand sort
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burtsort · Counting sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort
Concurrent sorts	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network
Hybrid sorts	Block sort · Timsort · Introsort · Spreadsort · JSort
Other	Topological sorting · Pancake sorting · Spaghetti sort

(source of picture: http://en.wikipedia.org/wiki/Sorting_algorithm)

- There are **many** sorting algorithms
- The "**best**" algorithm depends on the use case
- It is possible to **combine** several algorithms (hybrid approaches)
 - ▶ Ex: RadixSort might have as internal step another algorithm, as long as it is a **stable sort** (keep initial order in case of a tie)
- In practice, on **real implementations**, this is what is done (to combine):
[Note: the exact implementation depends on compiler and version]
 - ▶ **Java:** uses **Timsort** (MergeSort + InsertionSort)
 - ▶ **C++ STL:** uses **IntroSort** (QuickSort + HeapSort) + InsertionSort

Example use cases of sorting

Repetitions

Problem: finding **repeated** elements

Input

9	21	27	38	34	53	19	38	43
51	1	9	10	39	50	6	26	44
5	32	16	20	50	22	41	30	39
3	32	30	31	40	50	56	13	19
46	32	56	26	20	57	32	27	31
17	32	54	61	34	22	14	54	9
34	30	38	10	30	5	37	61	44

Input

1	3	5	5	6	9	9	9	10
10	13	14	16	17	19	19	20	20
21	22	22	26	26	27	27	30	30
30	30	31	31	32	32	32	32	32
34	34	34	37	38	38	38	39	39
40	41	43	44	44	46	50	50	50
51	53	54	54	56	56	57	61	61

Equal elements are together when sorted!

Example use cases of sorting

Others

Problem: find the **frequency** of elements

(equal elements are in consecutive positions after being sorted)

Problem: find **closest** pair of points

(sort and see differences between consecutive numbers)

Problem: find the **k -th** number

(sort and seek position k)

Problem: find the **top- k**

(sort and seek first k numbers)

Problem: set **union**

(sort and "merge" - like in mergesort)

Problem: set **intersection**

(sort and traverse - similar to mergesort)

Example use cases of sorting

Anagrams

Problem: Finding anagrams
(words/sets of words that use the same letters)

Examples:

- **amor, ramo, mora and Roma** [amor]
- **Ricardo, criador and corrida** [acdiorr]
- **algorithm and logarithm** [aghilmort]
- **Tom Marvolo Riddle and I am Lord Voldemort** [addeillmmooorrtv]
- **Clint Eastwood and Old West action** [acdeilnoosttw]

Example use cases of sorting

Search

Problem: Searching for elements in sorted arrays

Binary search - $\Theta(\log n)$

Binary search

A definition

Binary search on a sorted array (bsearch)

Input:

- an array $v[]$ of n sorted number in increasing order
- a **key** to look for

Output:

- **Position** of *key* in array $v[]$ (if it exists)
- **-1** (if it is not found)

Example:

$v =$

2	5	6	8	9	12
---	---	---	---	---	----

$\text{bsearch}(v, 2) = 0$

$\text{bsearch}(v, 4) = -1$

$\text{bsearch}(v, 8) = 3$

$\text{bsearch}(v, 14) = -1$

Binary search

Algorithm

Binary search on a sorted array

```
bsearch(v, low, high, key)
    While ( $low \leq high$ ) do
         $middle = low + (high - low) / 2$ 
        If ( $key == v[middle]$ )      return( $middle$ )
        Else If ( $key < v[middle]$ )  $high = middle - 1$ 
        Else                       $low = middle + 1$ 
    return(-1)
```

$v =$

2	5	6	8	9	12
---	---	---	---	---	----

bsearch($v, 0, 5, 8$)

$low = 0, high = 5, middle = 2$

Since $8 > v[2]$: $low = 3, high = 5, middle = 4$

Since $8 < v[4]$: $low = 3, high = 3, middle = 3$

Since $8 = v[3]$: **return**(3)

Binary Search

A generalization

We can generalize **binary search** to cases where we have something like:

no	no	no	no	no	yes	yes	yes	yes	yes	yes
----	----	----	----	----	-----	-----	-----	-----	-----	-----

We want to find the **first** **yes** (or in some cases the **last** **no**)

Example:

- Searching for the least number bigger or equal than a certain *key* (**lower_bound** of C++)

2	5	6	8	9	12
no	no	no	yes	yes	yes

`lower_bound(7)` $\rightarrow$ condition: $v[i] \geq 7$

[the smallest number bigger than 7 in this array is 8]

Binary Search

A generalization

Binary search for smallest k such that $\text{condition}(k)$ is "yes"

```
bsearch(low, high, condition)
```

```
While (low < high) do
```

```
    middle = low + (high - low)/2
```

```
    If (condition(middle) == yes) high = middle
```

```
    Else low = middle + 1
```

```
If (condition(low) == no) return(-1)
```

```
return(low)
```

$v =$

2	5	6	8	9	12
no	no	no	yes	yes	yes

$\text{bsearch}(0, 5, \geq 7)$

$low = 0, high = 5, middle = 2$

Since $v[2] \geq 7$ is **no**: $low = 3, high = 5, middle = 4$

Since $v[4] \geq 7$ is **yes**: $low = 3, high = 4, middle = 3$

Since $v[3] \geq 7$ is **yes**: $low = 3, high = 3$ (exits while)

Since $v[3] \geq 7$ is **yes**: **return(3)**

Binary Search

A different example - Balanced Contiguous Partition

Balanced contiguous partition problem

Input: a sequence $\langle a_1, \dots, a_n \rangle$ of n positive integers and an integer k

Output: a way of partitioning the sequence into k contiguous subsequences, minimizing the sum of the biggest partition

Example:

7 9 3 8 2 2 9 4 3 4 7 9 9 $k = 4$ (4 partitions)

7 9 3 | 8 2 2 | 9 4 3 | 4 7 9 9 $\rightarrow 19 + 12 + 16 + 29$

7 9 3 8 | 2 2 9 | 4 3 4 7 | 9 9 $\rightarrow 27 + 13 + 18 + 18$

7 9 | 3 8 2 2 | 9 4 3 4 | 7 9 9 $\rightarrow 16 + 15 + 20 + 25$

...

Which one is the best (with the smallest maximum)?

Binary Search

A different example - Balanced Partition

- Exhaustive search would have to test all possible partitions!
(can you estimate how many are they?)
- This problem could also be solved with **dynamic programming**, but that is for another class
- Here we will discuss how to solve it with... **binary search!**

Binary Search

A different example - Balanced Partition

Let's think on a "similar" problem: **It is possible to create a partition where the sum of the largest partition is $\leq X$?**

"Greedy" idea: keep extending the partition while the sum is $\leq X$!

Examples:

Let $X = 21$ and $k = 4$

7 9 3 | 8 2 2 9 4 3 4 7 9 9

7 9 3 | 8 2 2 9 | 4 3 4 7 9 9

7 9 3 | 8 2 2 9 | 4 3 4 7 | 9 9

7 9 3 | 8 2 2 9 | 4 3 4 7 | 9 9 - OK!

Let $X = 20$ and $k = 4$

7 9 3 | 8 2 2 9 4 3 4 7 9 9

7 9 3 | 8 2 2 | 9 4 3 4 7 9 9

7 9 3 | 8 2 2 | 9 4 3 4 | 7 9 9

7 9 3 | 8 2 2 | 9 4 3 4 | 7 9 | 9 - Wrong! We would need more than 4 partitions

Binary Search

A different example - Balanced Partition

It is possible to create a partition where the sum of the largest partition is $\leq X$?

If we think about the X for which the answer is **yes**, we have a search space where:

no	no	no...	no	no	yes	yes	yes	...	yes	yes
----	----	-------	----	----	-----	-----	-----	-----	-----	-----

We can apply **binary search on X** !

- Let s be the sum of all numbers
- X will be at least 1 (or in alternative the largest a_i)
- X will be at most s
- Verify answer for a certain X : $\Theta(n)$
- Binary search on X : $\Theta(\log s)$
- Global time: $\Theta(n \log s)$

Binary Search

A different example - Balanced Partition

Example: 7 9 3 8 2 2 9 4 3 4 7 9 9 $k = 4$ (4 partitions)

low = 1, high = 76, middle = 38 → possible(38)? Yes

low = 1, high = 38, middle = 19 → possible(19)? No

low = 20, high = 38, middle = 29 → possible(29)? Yes

low = 20, high = 29, middle = 24 → possible(24)? Yes

low = 20, high = 24, middle = 22 → possible(22)? Yes

low = 20, high = 22, middle = 21 → possible(21)? Yes

low = 20, high = 21, middle = 20 → possible(20)? No

low = 21, high = 21

Exits the cycle and verifies that **possible(21)** is true, and 21 is the answer!

7 9 3 | 8 2 2 9 | 4 3 4 7 | 9 9 → 19 + 21 + 18 + 18

Binary Search

A different example - Balanced Partition

2nd Example: 7 9 3 8 2 2 9 4 3 4 7 9 9 $k = 3$ (3 partitions)

low = 1, high = 76, middle = 38 → possible(38)? Yes

low = 1, high = 38, middle = 19 → possible(19)? No

low = 20, high = 38, middle = 29 → possible(29)? Yes

low = 20, high = 29, middle = 24 → possible(24)? No

low = 25, high = 29, middle = 27 → possible(27)? Yes

low = 25, high = 27, middle = 26 → possible(26)? No

low = 27, high = 27

Exits the cycle and verifies that **possible(27)** is true, and 27 is the answer!

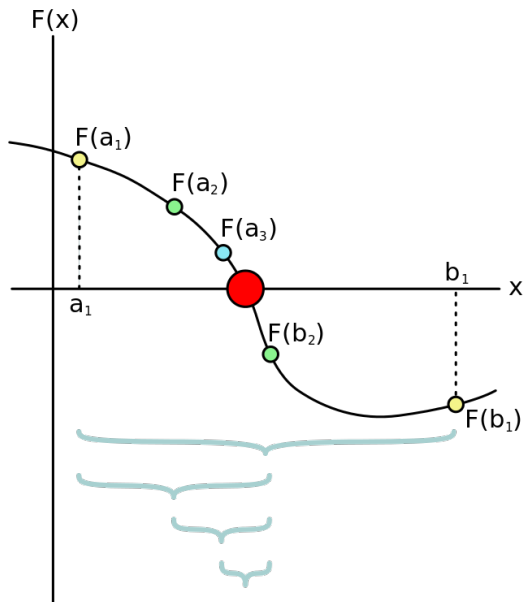
7 9 3 8 | 2 2 9 4 3 4 | 7 9 9 → 27 + 24 + 25

Bisection Method

A similar idea to binary search can be used to find the **root of a function**

- Let $f(n)$ be a **continuous** function defined on an interval $[a, b]$ and where $f(a)$ and $f(b)$ have **opposite signs**
- $f(n)$ must have **at least one root** on the interval $[a, b]$
- Starting in $[a, b]$, look at **middle point** c and according to $f(c)$ **reduce the interval** to $[a, c]$ or $[c, b]$

Bisection Method



(image: Wikipedia)

Bisection Method

Example: $f(x) = x^3 - x - 2$

(1) Find a and b with opposite signals:

$$f(1) = 1^3 - 1 - 2 = -2 \qquad f(2) = 2^3 - 2 - 2 = 4$$

(2) Make successive divisions:

#	a	b	c	f(c)
1	1.0	2.0	1.5	-0.125
2	1.5	2.0	1.75	1.6093750
3	1.5	1.75	1.625	0.6660156
4	1.5	1.625	1.5625	0.2521973
5	1.5	1.5625	1.5312500	0.0591125
6	1.5	1.5312500	1.5156250	-0.0340538
7	1.5156250	1.5312500	1.5234375	0.0122504
8	1.5156250	1.5234375	1.5195313	-0.0109712
9	1.5195313	1.5234375	1.5214844	0.0006222
10	1.5195313	1.5214844	1.5205078	-0.0051789
11	1.5205078	1.5214844	1.5209961	-0.0022794
12	1.5209961	1.5214844	1.5212402	-0.0008289
13	1.5212402	1.5214844	1.5213623	-0.0001034

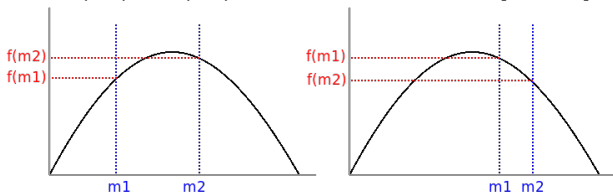
Bisection Method

- Stop when you have the **required precision**
or
- Stop when you reach your **desired number of iterations**
- There are other methods that **converge more rapidly**
 - ▶ Newton's method
 - ▶ Secant method

Ternary Search

Another similar idea can be used to find the **maximum** (or minimum) of an **unimodal** function (*that is, with a "single peak"*)

- Let $f(n)$ be a **unimodal** function defined on an interval $[a, b]$
- Take any two points m_1 and m_2 such that $a < m_1 < m_2 < b$. Then:
 - $f(m_1) < f(m_2)$ then max cannot be in $[a, m_1]$. Continue in $[m_1, b]$
 - $f(m_1) > f(m_2)$ then max cannot be in $[m_2, b]$. Continue in $[a, m_2]$
 - $f(m_1) = f(m_2)$ then max should be in $[m_1, m_2]$.



- We can choose m_1 and m_2 to be $1/3$ and $2/3$ of $[a, b]$
- With each iteration we will eliminate at least $1/3$ of the search space!
Runtime: $T(n) = T(2n/3) + \Theta(1) = \Theta(\log n)$

Binary Search

- Binary search is very **useful** and **flexible**
- It can be used on a **vast number of applications**
- There are many other **variations** on it (besides the ones we already described)
 - ▶ Interpolated (binary) search
(instead of going into the middle, estimate position)
 - ▶ Exponential (binary) search
(Start by fixing interval in $low = 2^a$ and $high = 2^{a+1}$)
 - ▶ ...