CSE 101

Computer Science Principle

Lecture 05: Search & Sort

April. 2023
Prof. Francois Rameau
francois.rameau@sunykorea.ac.kr

SUNY Korea
The State University of New York

Attendance time!

Scan the QR code and select your name in the list

What are we studying today?

This week we will discover very useful algorithms to **search and sort** data stored in a data structure!

Search

Sort

Searching and sorting algorithms allow efficient processing and organization of large amounts of data.

Linear search

Searching algorithms

Searching is a common operation in many different situations

Finding a file

Spotlight on MacOS, the search box in Windows Explorer

Online dictionary

Looking for the definition of a word

Web search

Web search engines (e.g., Google) use searching algorithms

E-Commerce

Help customers find products based on keywords

Database management

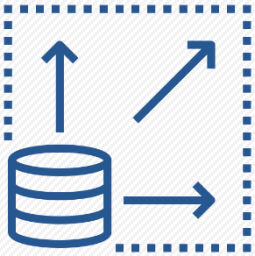
Quickly retrieve information from large datasets.

Text editor

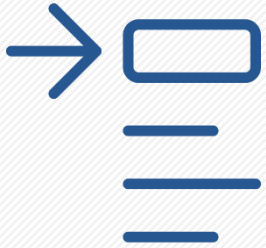
Find specific words or phrases within a document

Searching algorithms

What do these searching problems have in common?



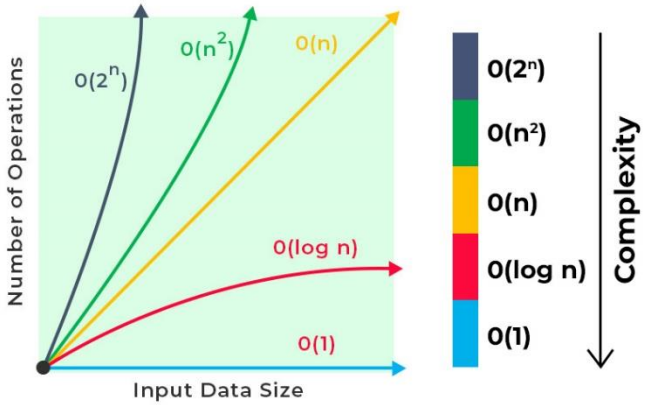
We have a potentially large collection of items



We need to search the collection to find a single item that matches a certain condition
(e.g., name of book/name of a person)

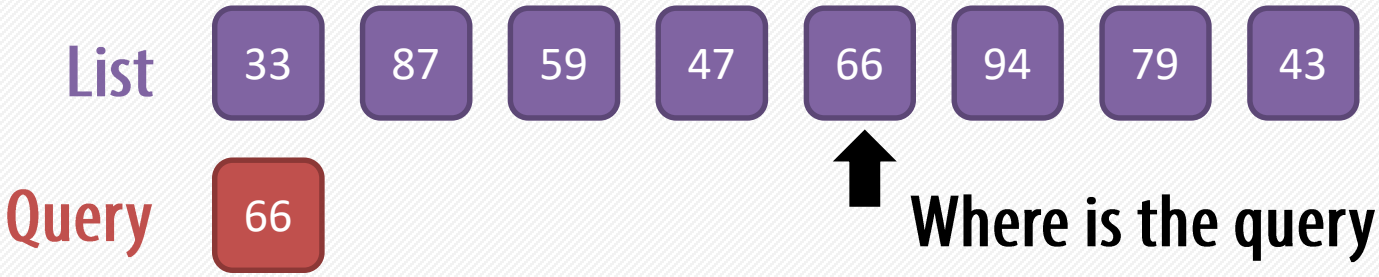
Notice that the number of "operations" grows as the database size grows

- Can you guess what kind of dependence there is?
- Why does this matter?

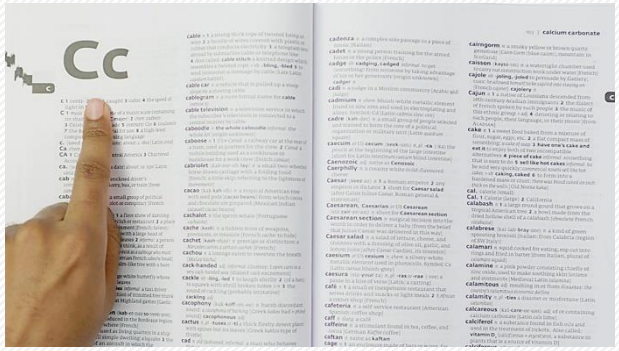


Searching concrete example

Find a number in a list



Find a word in a data structure



As you may guess, searching algorithms are iterative and rely heavily on loops

Linear search

Linear search is the simplest, most straightforward search strategy

Concept of linear search

The idea is to start at the beginning of a collection and compare items one after another until it finds the desired item

- Also called **sequential search** because the elements of the collection are examined sequentially

An intuitive technique

Imagine I give you a shuffled deck of 52 cards, and I ask you to find the five of Spade!

You will look at all the cards one after another until you find the card I asked for! If the card is the last one, you will have to first look at every single other card.

Linear search

Recall the index method for list?

This method returns the position of an element in a list

Example:

```
notes = ['do', 're', 'me', 'fa', 'sol', 'la', 'ti']
notes.index('sol')    # will return the value 4
```

The **index** method is performing a search

Another reminder

Also recall the `in` operator, which tells us if an element is present in a list
This operator is necessary because the **index** method will cause our program to crash if the element we want is missing

```
if 'sol' in notes:      # notes.index('sol') could crash
    print('Present in list...')
else:
    print('Not present in list...')
```

Implementing linear search

Now that you understand the concept, let's get our hands dirty and let's code it!

Requirements

The linear search function (**`isearch`**) we'll implement is similar to Python's **`index`** method

- Pass it a list and an item to search for
- If the item is in the list, the function returns the location where it was found
- If the item is not in the list, the function returns **None**

Usage examples

```
notes = ['do', 're', 'me', 'fa', 'sol', 'la', 'ti']
print(isearch(notes, 'ti'))     # returns 6
print(isearch(notes, 'ba')) # returns None
```

Implementing linear search

One way to write our **isearch** is to use a for loop with a range expression

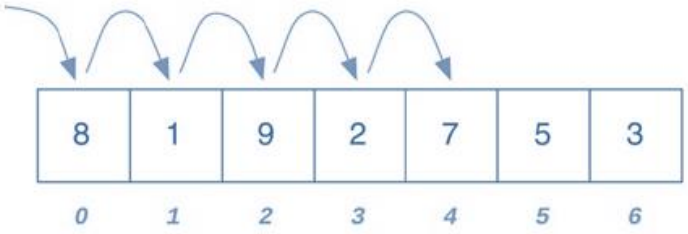
```
def isearch(a, x):
    for i in range(len(a)):
        if a[i] == x:
            return i
    return None

my_list = [54, 26, 89, 10, 3, 7]
query = 89
isearch(my_list, query)
```

Variable	Value
i	2
a[i]	89

- Note the **i** variable acts as an index into **a**
- We do this so that we can return the position (index) of the target element, **x**, in the list

Implementing linear search

Practice time



Practice

modify **`isearch()`** so that it performs the search only within the given range:

- **`isearch(lst, query, begin, end)`** : only search from index 'begin' up to index 'end'
- 'end' is exclusive

Implementing linear search

We can also implement the linear search algorithm using a **`while`** loop

```
def isearch(a, x):
    i = 0
    while i < len(a):
        if a[i] == x:
            return i
        i = i + 1
    return None
```

Advise

unless there is a good reason to use a while loop, use a for loop

- Programs will be shorter, simpler and less likely to contain errors
- Why is this the case?

The single line of code **`for i in range(len(a))`** would require three lines if written as a while loop!

Linear search: Performance

The linear search algorithm looks for an item in a list

- Start at the beginning (**a[0]**, or "the left")
- Compare each item, moving systematically to the right (**i = i + 1**)

How many comparisons will the linear search algorithm make as it searches through a list with n items?

- **Another way to phrase it**: how many iterations will our Python function make in its while loop?
- It depends on whether the search is successful or not

Unsuccessful

- Visit every item before returning None
- i.e., make **n** comparisons (n = total number of items)

Successful

For a **successful** search, anywhere from 1 and n iterations are required

- Search may be lucky and find the item in the first location
- At the other extreme, the item might be in the last location
- Expect, **on average, n/2 comparisons**

Time complexity
-The big O notation-

Big O notation

In 2009, in South Africa, they did an experiment. They sent data through the internet and also using a . . . **pigeon**

Constant time beats linear time IF data is sufficiently large. . .

Office 1

Homing pigeon

50 miles

Internet

Office 2

Internet Transfer

1Gb → 30 min
2GB → 60 min
3Gb → 90min
1000Gb → 500hours

LINEAR TIME

$O(n)$

Pigeon Transfer (50 miles)

1Gb → 60 min
2GB → 60 min
3Gb → 60min
1000Gb → 60min

CONSTANT TIME

$O(1)$

Very good example taken from HackerRank (12) Big O Notation - YouTube

Big O notation

! Unformal terminology alert !

Big O

"How code slows as data grows"

"How time scale with respect to some input variables"

Key elements of big O notation

1. Describes the performance of an algorithm as the amount of data increases
2. Machine independent (# of steps to completion)
3. Ignore smaller operations $O(n+1) \rightarrow O(n)$

Examples:

$O(1)$

$O(n)$

$O(\log n)$

$O(n^2)$

(where n is the number of steps):

Slide credit to BroCode

SUNY Korea

Big O notation

Let's have a look to a concrete example

$O(n)$ Linear time

```
def addUp1(n):
    my_sum = 0
    for i in range(1,n+1):
        my_sum += i
    return my_sum
```

$n = 10000$
~ 10000 steps

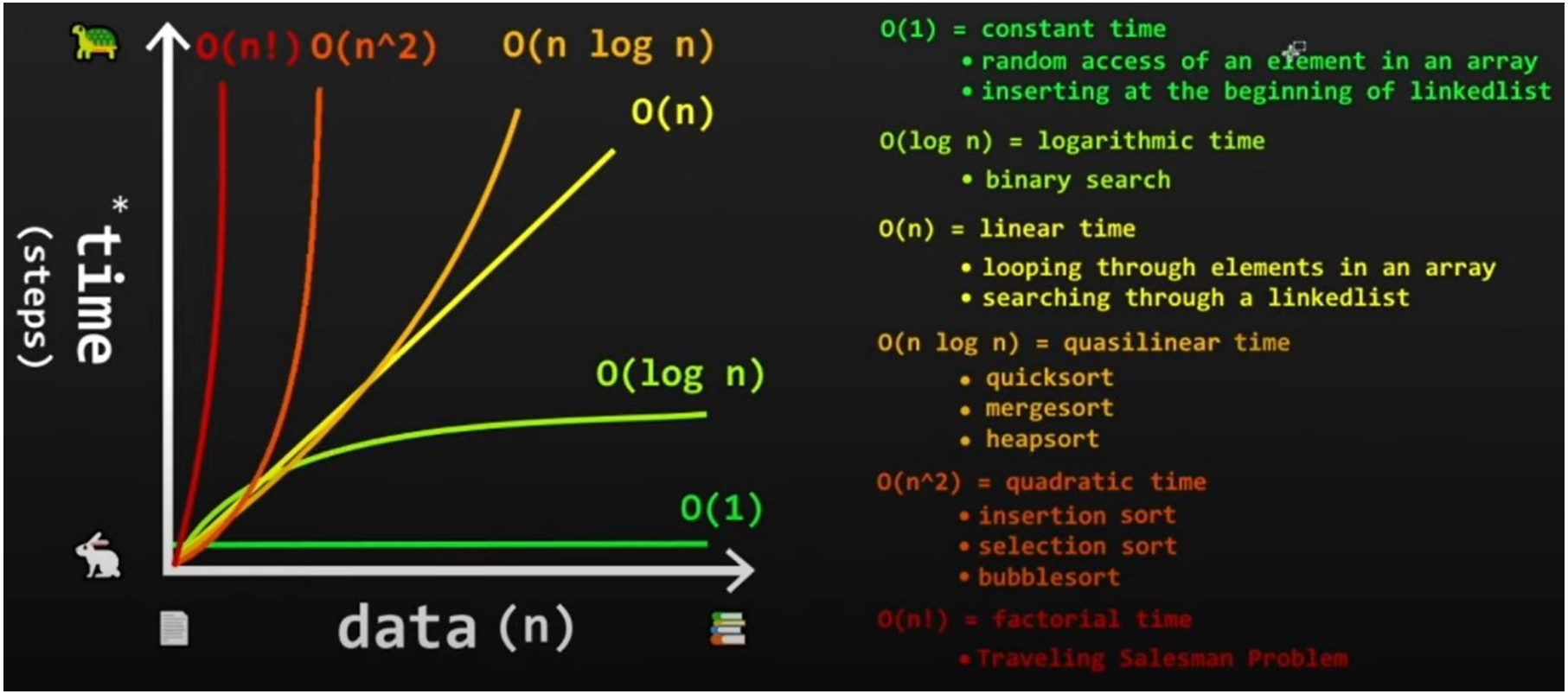
$O(1)$ Constant time

```
def addUp2(n):
    my_sum = n * (n + 1)/2
    return int(my_sum)
```

$n = 10000$
~ 3 steps

Slide credit to BroCode

Big O notation



Additional resources:
[What Is Big O Notation]
[Big O Notation]

Slide credit to [BroCode]

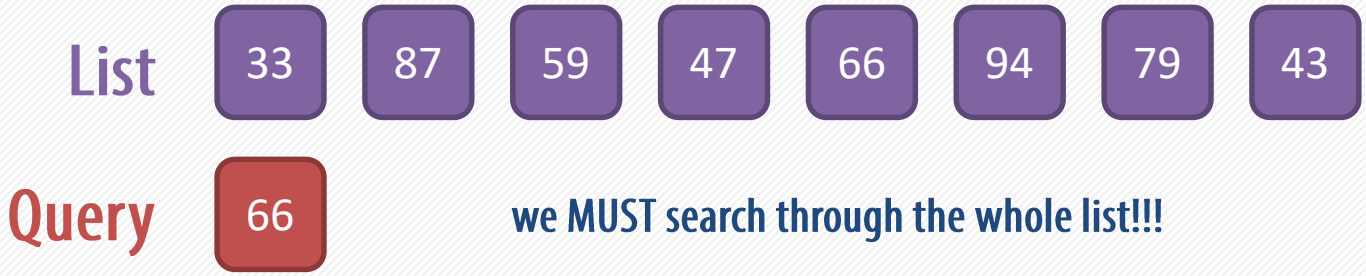
Binary search

Binary search

Can we do better than linear search?

Not really (why?) …

If it is a shuffled stack, there is no shortcut



But what is the data is more "structured"?

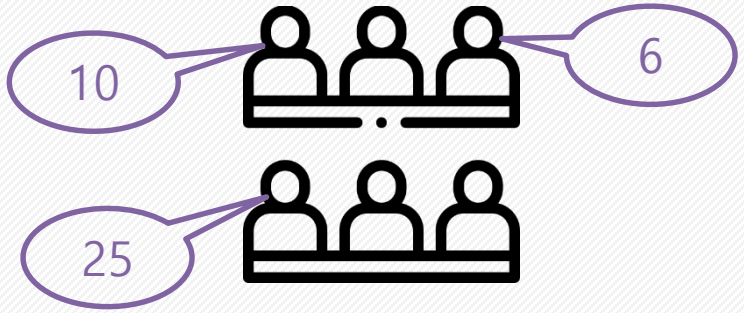
For instance what is the list is sorted? How would it help?

Binary search

Let's play a game!

I will think about a number between 0 -100

You will try to guess it!



I will reply only by YES/NO
You have 10 chances only!

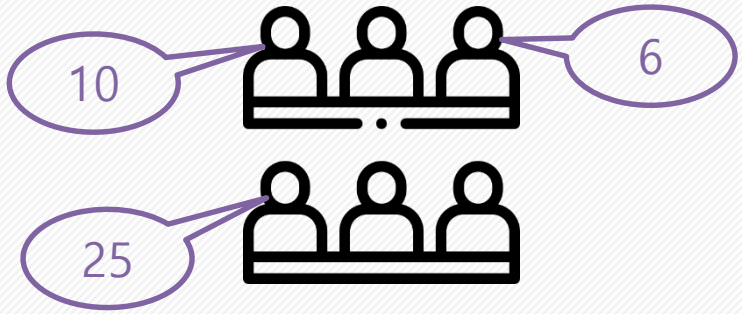
Binary search

Well . . . the outcome was likely unsuccessful

What question should we ask instead?

I will think about a number between 0 -100

You will try to guess it!



This time I will reply MORE/LESS

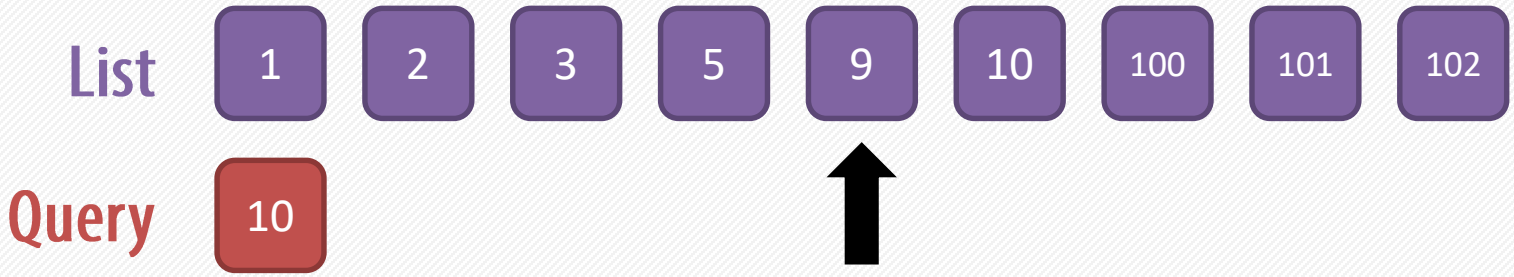
You have 10 chances only!

Binary search

I guess it is was much effective!

How to we optimize this strategy?

Let's consider this list



Where is the good place to start asking?

The middle! Why?

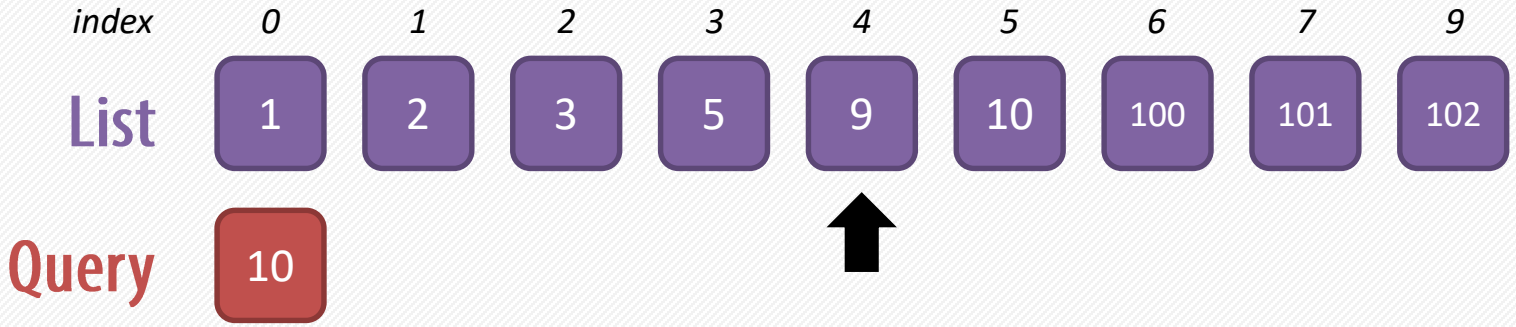
Binary search uses the **mid-point** to split the data because it ensures that the search space is divided into two equal parts, making it possible to eliminate half of the search space with each comparison, thus achieving a logarithmic time complexity.

Binary search

I guess it is was much effective!

How to we optimize this strategy?

Let's consider this list



Three possibilities:

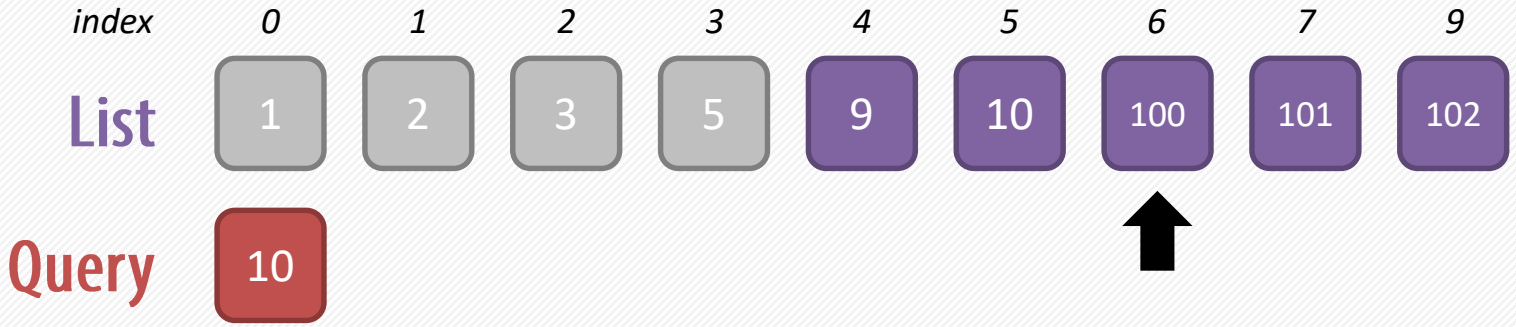
- If `List[4] == Query` → You are done!
- If `List[4] > Query` → Remove the second half of the list
- If `List[4] < Query` → Remove the first half of the list

Binary search

I guess it is was much effective!

How to we optimize this strategy?

Let's consider this list



Recompute the midpoint on the remaining list

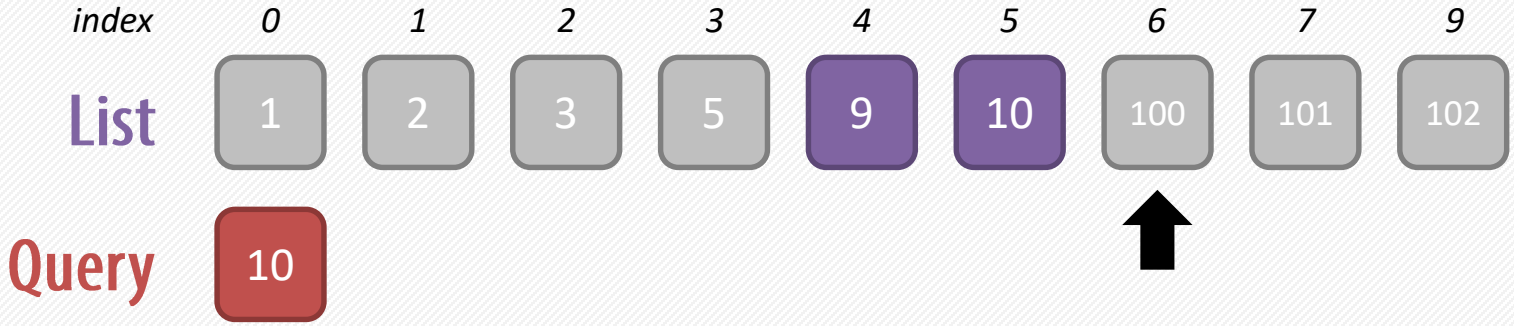
And split again!

Binary search

I guess it is was much effective!

How to we optimize this strategy?

Let's consider this list



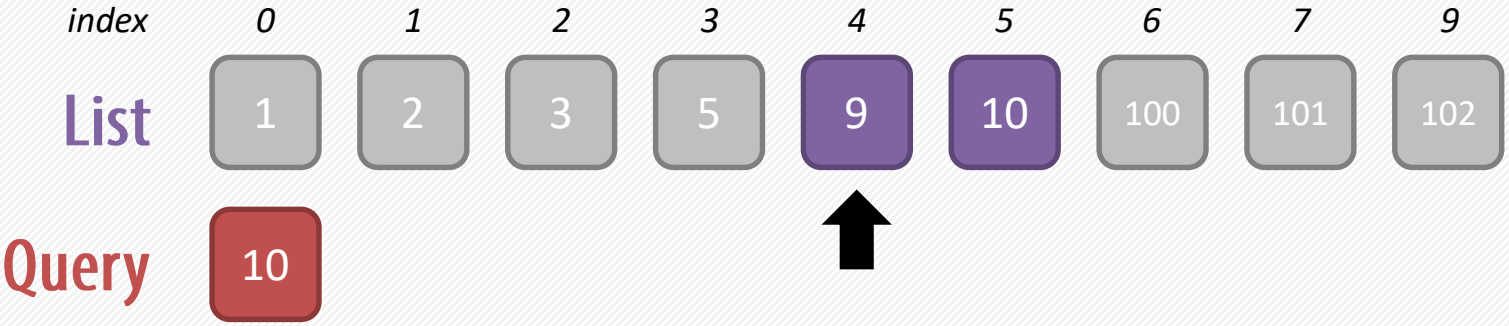
Split again!

Binary search

I guess it is was much effective!

How to we optimize this strategy?

Let's consider this list



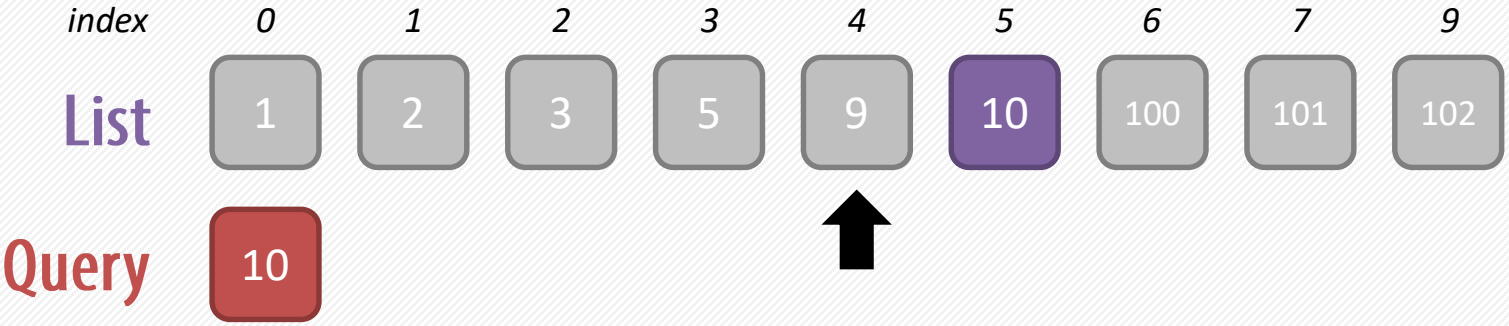
Keep going!

Binary search

I guess it is was much effective!

How to we optimize this strategy?

Let's consider this list



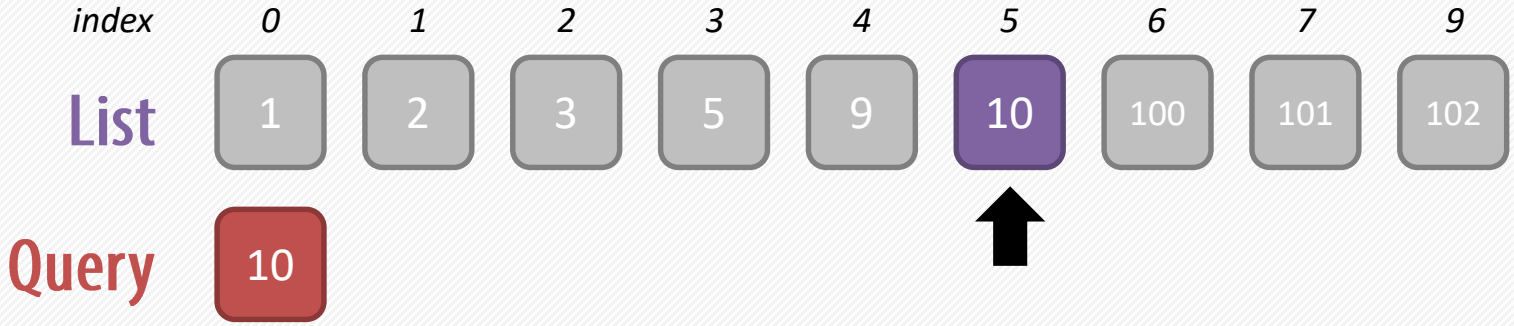
Keep going!

Binary search

I guess it is was much effective!

How to we optimize this strategy?

Let's consider this list



DONE!

Binary search time complexity

Every time we make a comparison, we either find the right element of remove half of the range

If we have 40 elements $40 \rightarrow 20 \rightarrow 10 \rightarrow \cdots \rightarrow 1$

We can generalize with n elements $n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \rightarrow \cdots \rightarrow 1$

It starts from n and ends with one element

It can be rewritten $n \rightarrow \frac{n}{2} \rightarrow \frac{n}{2^2} \rightarrow \frac{n}{2^3} \rightarrow \cdots \rightarrow \frac{n}{2^x} \approx 1$

x being the number of iteration that reach a single last element

Thus

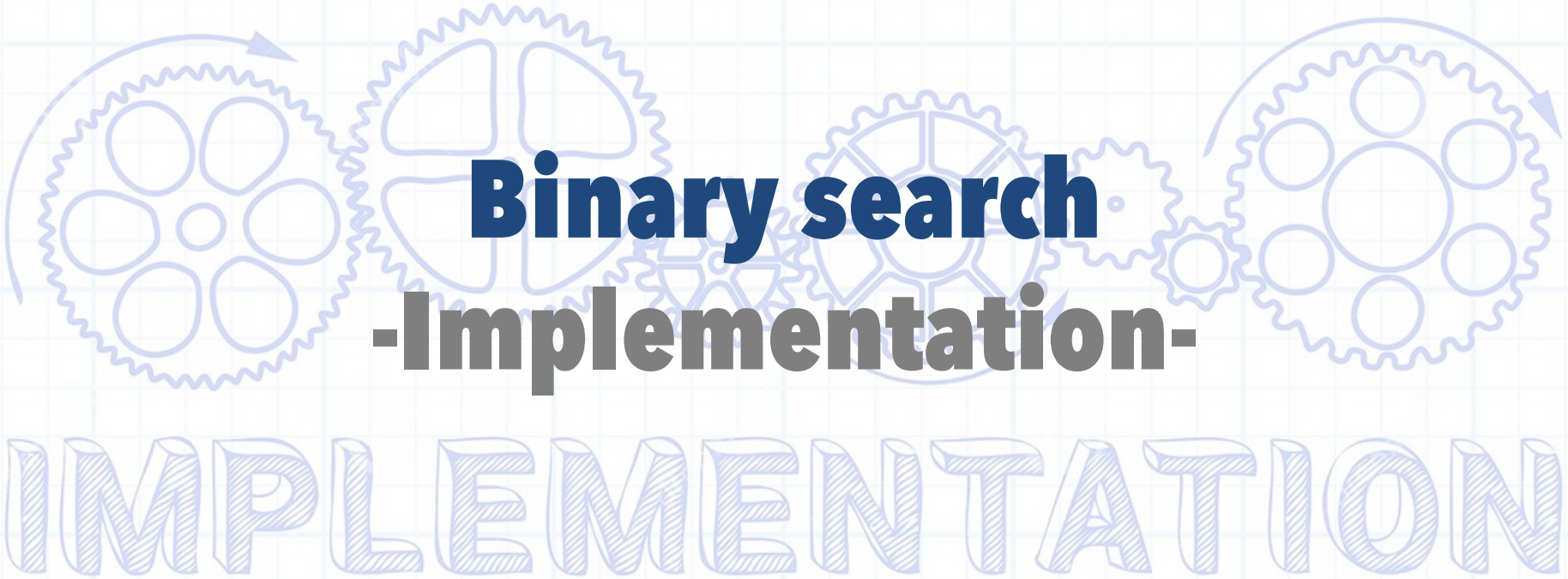
$$\frac{n}{2^x} \approx 1 \rightarrow n = 2^x$$

$$\rightarrow \boldsymbol{x = \log_2(n)}$$

$$\boldsymbol{O(\log(n))}$$

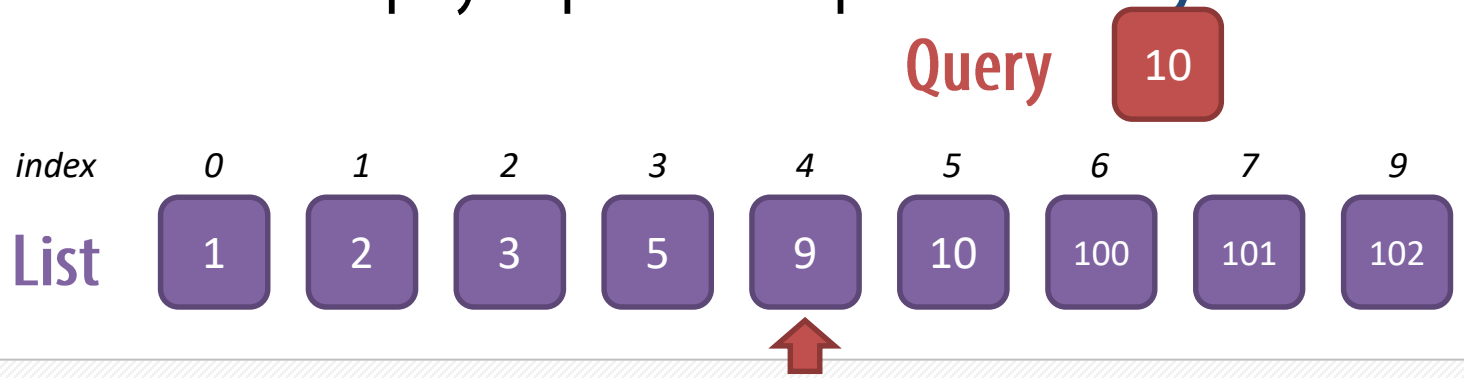
Imagine if you have a list of 10,000,000 elements in which you look for one element, with binary search it would take only $\boldsymbol{\log_2(10,000,000) = 24}$ checks

Binary search
-Implementation-



Binary search implementation

Let's see step by step how to implement **binary search**



How do we find the middle (mid-point)

- `mid_point = (max_index + min_index) / 2` ← `/!\ it should be an int`
- For the very first iteration: `min_index = 0` and what will be the **max_index**?
- But remember, we have to pick the middle element many times! Thus we will maintain two variables `range_begin` and `range_end`, denoting the range search for each iteration.

```
def bsearch(lst, query):
    range_begin = 0
    range_end = len(lst) - 1
    mid_point = (range_end + range_begin) // 2
    …
```

In our example:

`mid_point = (9 + 0)/2 = 4`

Binary search

Let's see step by step how to implement **binary search**

Time to enter the loop!

- We now need to iterate the split until the item is found or when no more item have to be checked
- How do we know when the item is not found?
  - When `range_begin > range_end` Do you see why?
- What kind of loop should be used? **`While loop`** since we do not know how many iterations are needed before the search is finished.

```
def bsearch(lst, query):
    range_begin = 0
    range_end = len(lst) - 1
    mid = (range_end + range_begin) // 2
    while range_begin <= range_end:
    …
```

Binary search

Let's see step by step how to implement **binary search**

Inside the loop

- Inside the loop, always look at the middle element
  - If `lst[mid]` is `query`, then problem solved!
  - If `lst[mid]` is greater than `query`, what do we do? (case 1)
  - If `lst[mid]` is less than `query`, what do we do? (case 2)

- For each of the inequalities, we adjust the '`range_begin`' and '`range_end`'
  - Case 1: that must mean '`query`' is in the lower half of the list
    → restrict the search to lower half → '`range_end`' should be one less than '`mid`'
  - Case 2: by symmetric logic, '`range_begin`' should be one more than '`mid`'

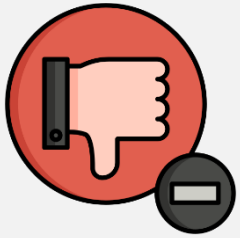
Binary search: The code

```
def bsearch(lst, query):
    range_begin = 0
    range_end = len(lst) - 1
    mid = (range_end + range_begin) // 2
    while range_begin <= range_end:
        if lst[mid] == query:
            return mid
        elif lst[mid] > query:
            range_end = mid - 1
        else:
            range_begin = mid + 1
        mid = (range_end + range_begin) // 2
    return None
```

Note that we can slightly rewrite the code to remove one line.

Do you see where?

Linear vs binary search

	Linear	Binary
Pros 	•Simple and easy to implement •Suitable for small arrays •No additional memory space needed	•Efficient for large sorted arrays •Time complexity is O(log n) •Best for repetitive searches
Cons 	•Time complexity is O(n) •Inefficient for large arrays •Not suitable for repetitive searches	•Array must be sorted •Requires additional memory space for the index •Not suitable for unsorted arrays

In conclusion, binary search is more efficient for **larger arrays and repetitive searches**, but requires a sorted array and **additional memory space**. Linear search is simple and suitable for small arrays, but becomes inefficient for larger arrays and repetitive searches. Choosing the appropriate search method depends on the specific requirements of your application.

Sorting

A classical problem: sorting

Using a sorted list, we can drastically speed up the search of an element!

You should now wonder:

How to sort a list?

How effective is it?

The **linear search** is an **iterative** algorithm

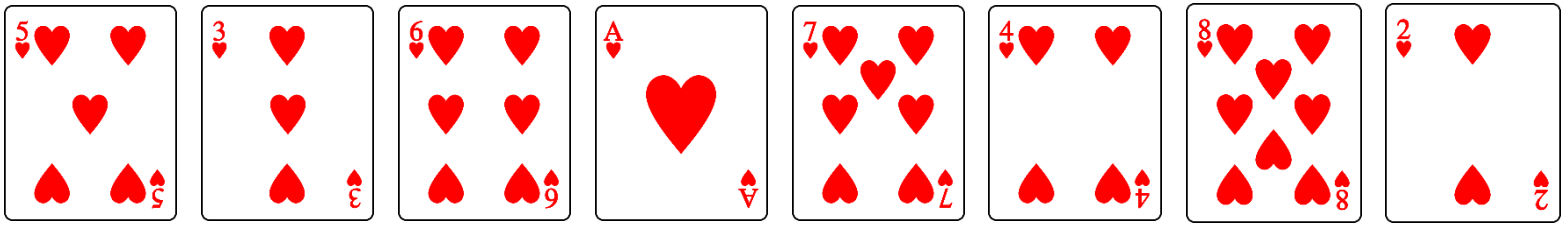
- Starts at the beginning of a collection and systematically progress, all the way to the end, if necessary.
- We will see a similar strategy for **sorting** item in a list

Sorting

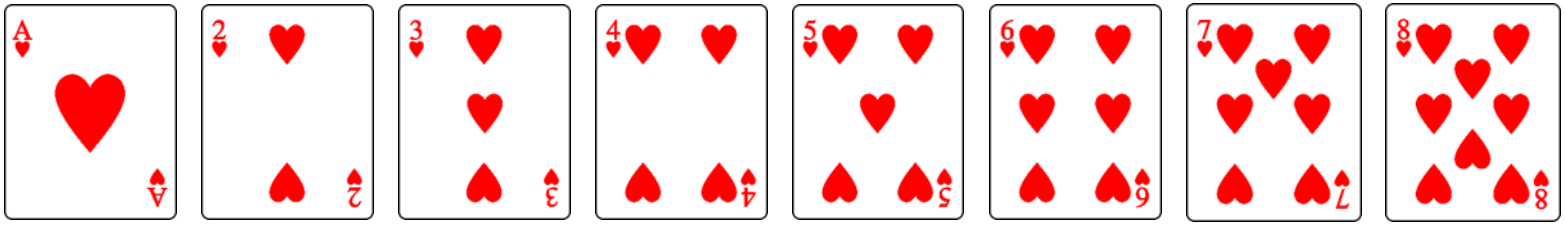
Sorting: an important problem – arises frequently in computer science

- Suppose there is a deck of cards to be put in order
- Example: We have the Ace up to 8 of Hearts

Given this:



You want to obtain the following:

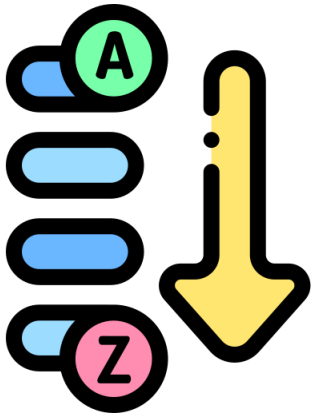


Insertion sort

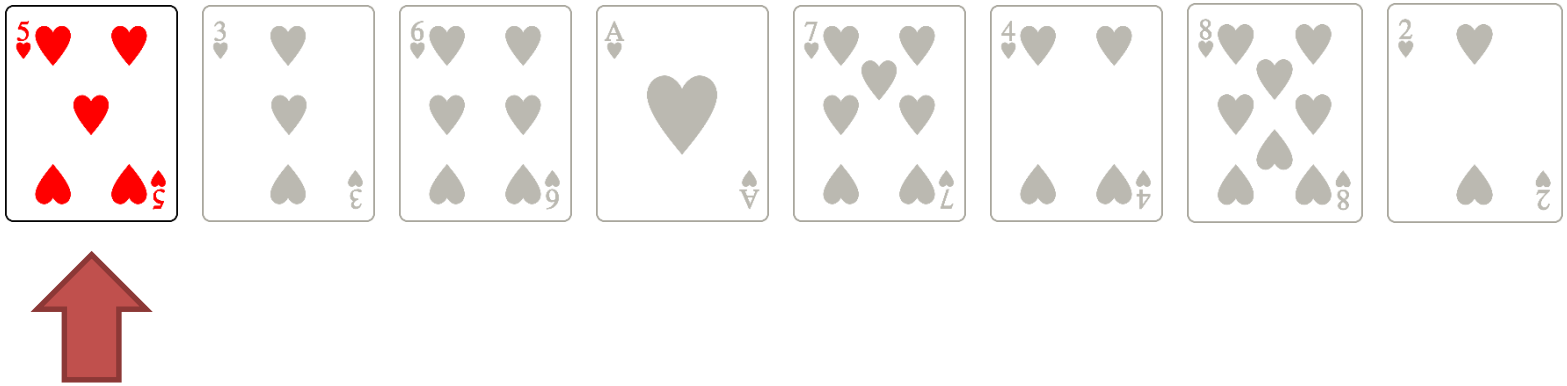
One technique called **Insertion sort**
It repeatedly searches for and swaps cards in the list

The basic idea is:

- Pick up an item, find the place it belongs, insert it back into the list
- Move to the next item and repeat

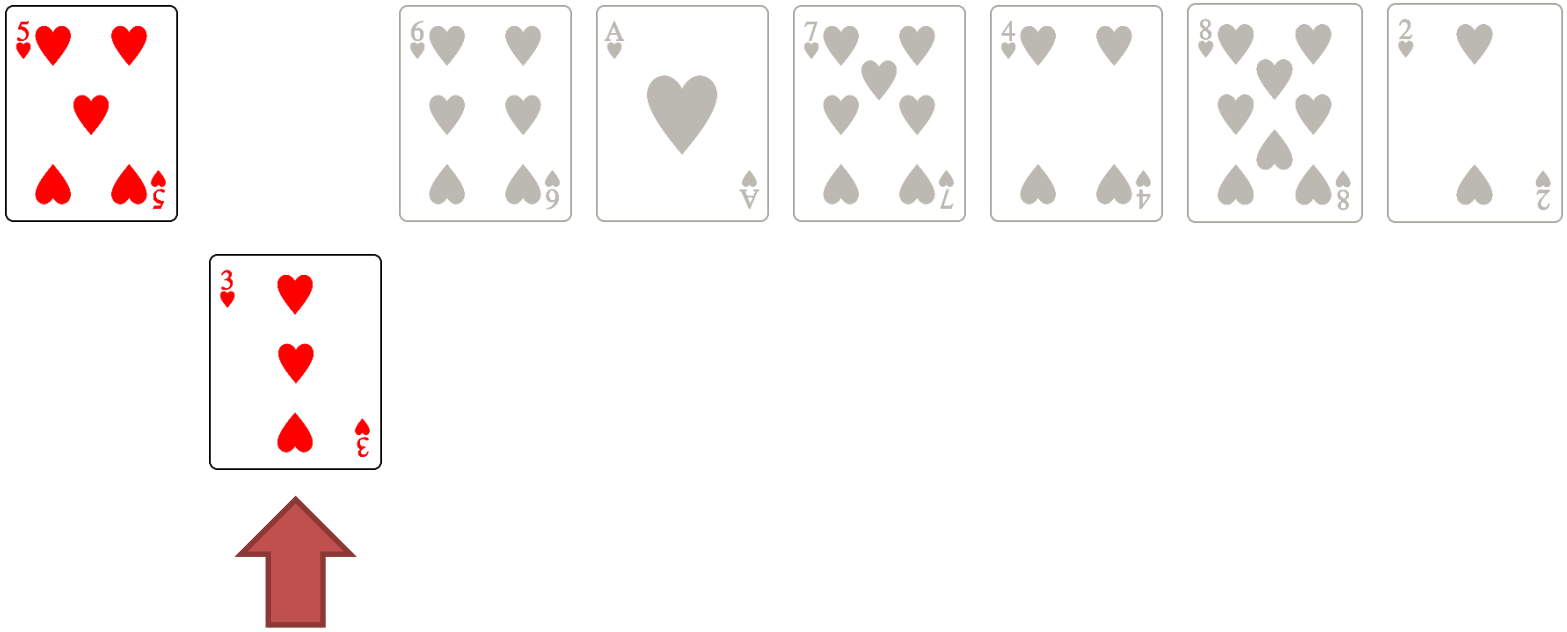


Insertion sort



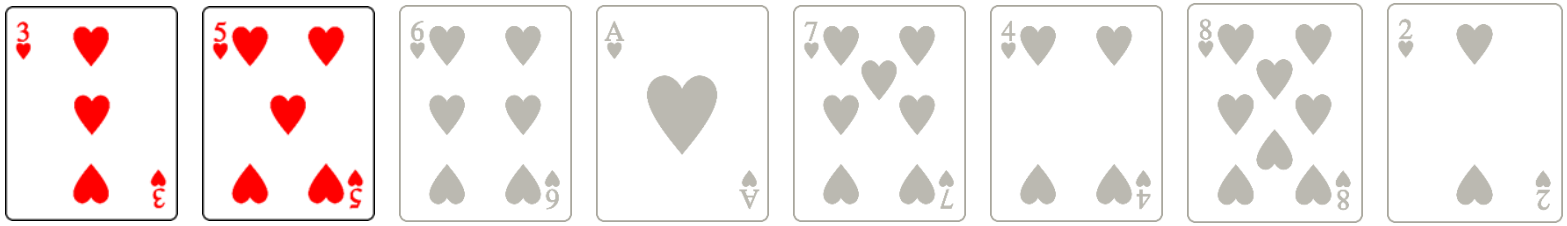
The first element has no left neighbor so will start directly by the second element

Insertion sort

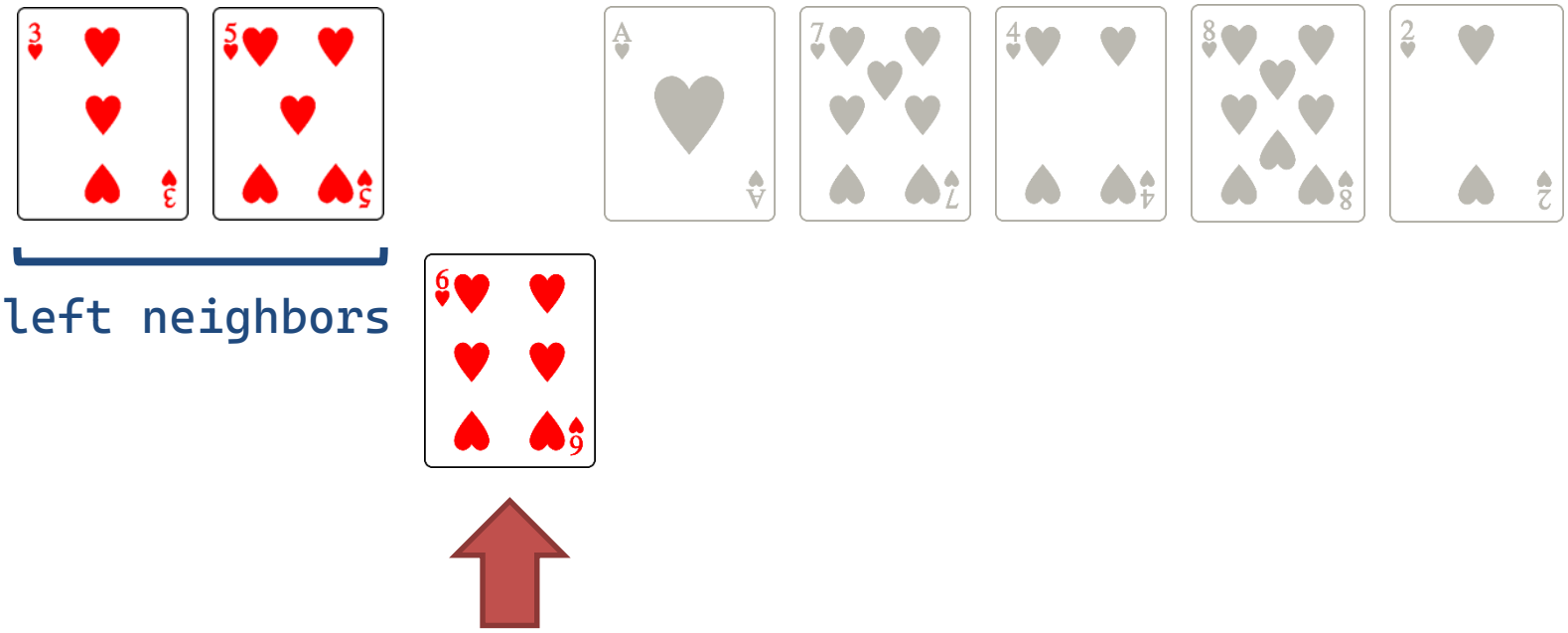


We pick the second card and we check if the active list is well sorted. In that case, we need to insert the card of interest in the first position!

Insertion sort

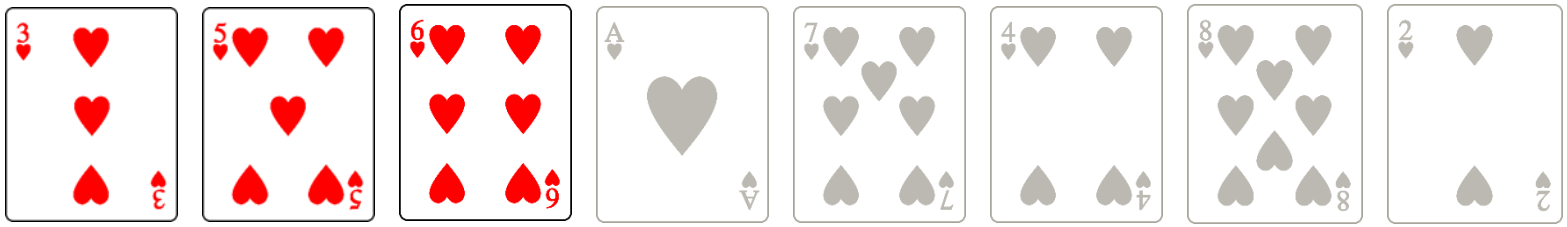


Insertion sort

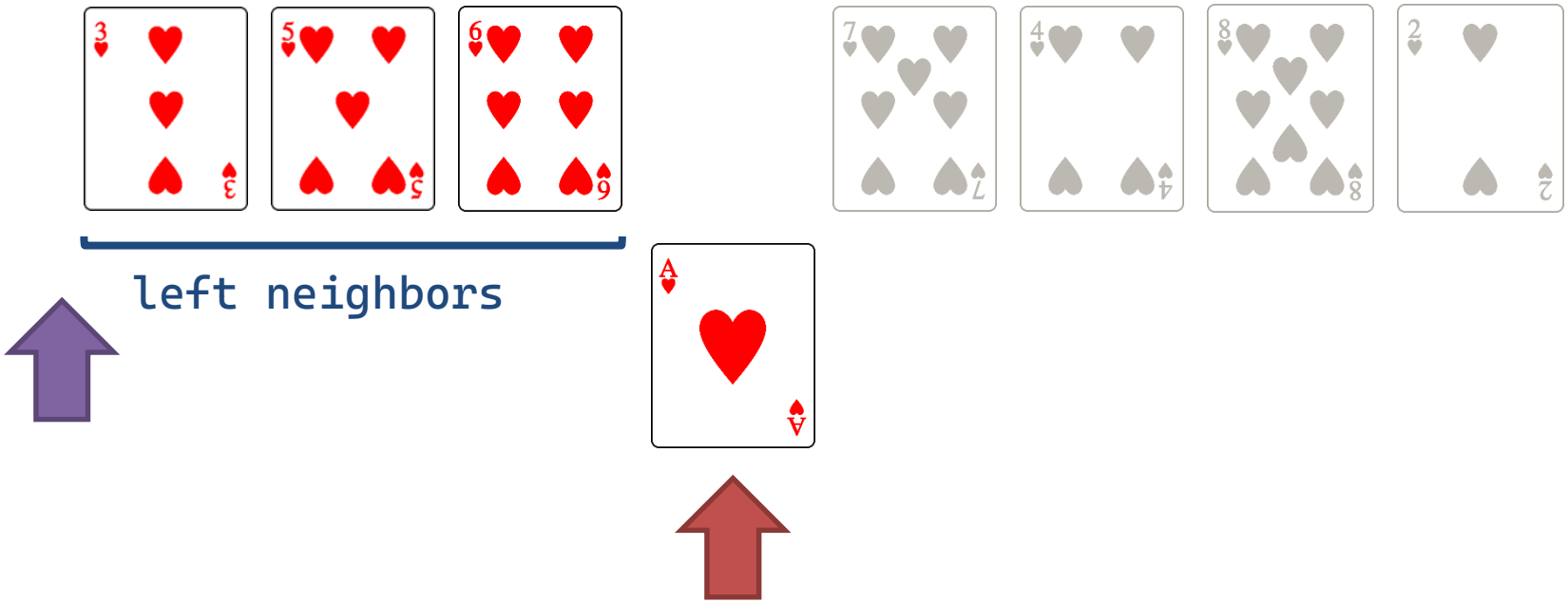


We now remove the third element and try to find where to insert it back among its left neighbors → we **do not** need to move anything (the list is well sorted so far)

Insertion sort

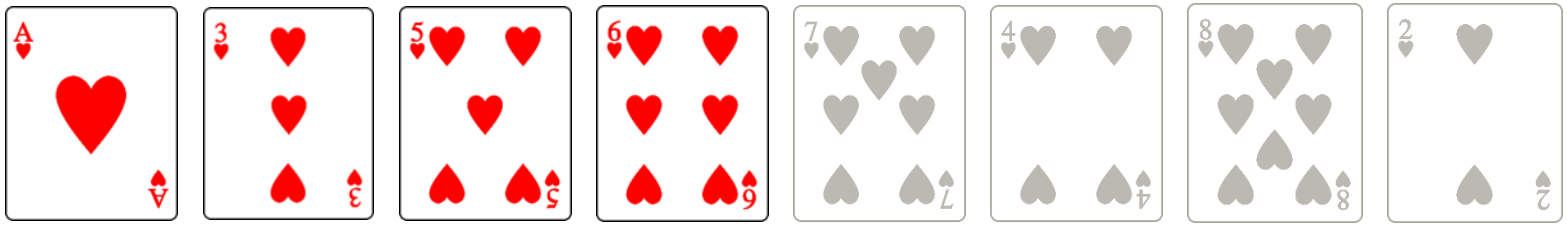


Insertion sort

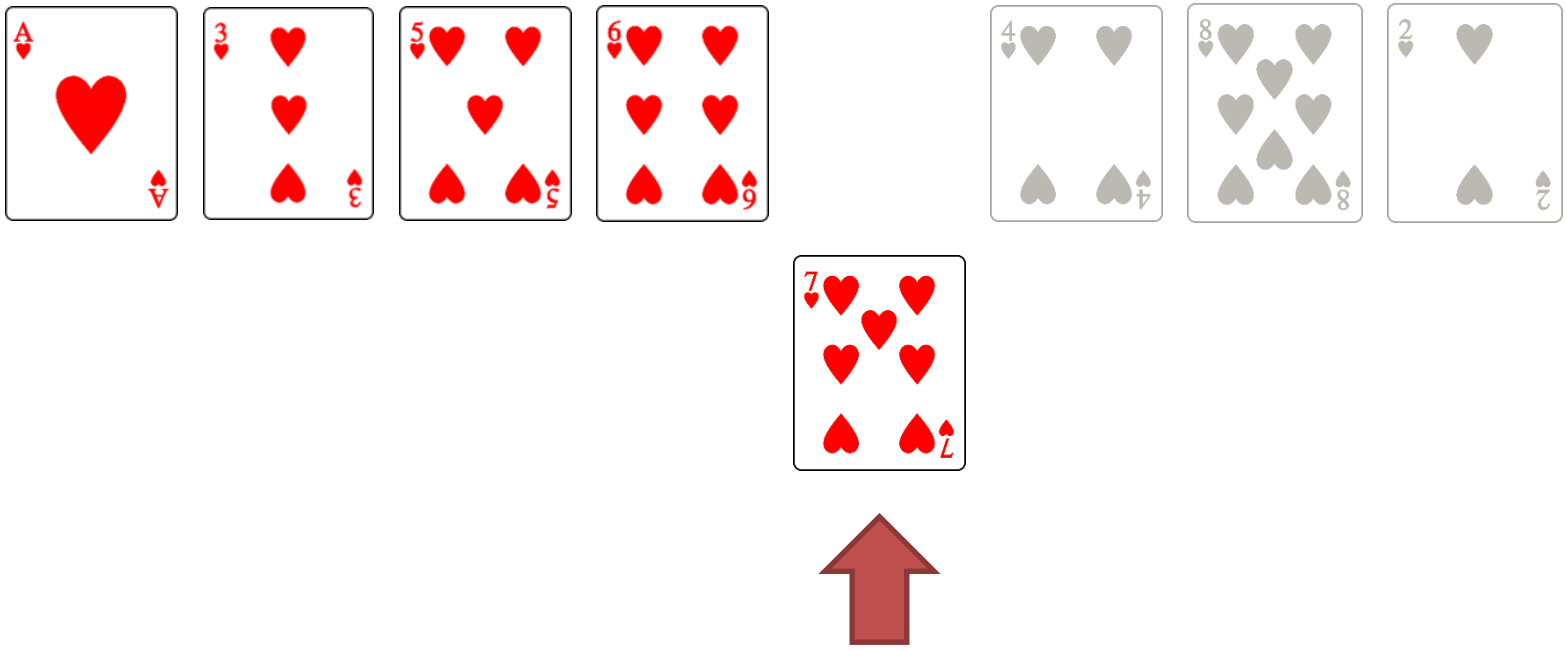


We take the fourth element and we try to find where we have to insert it back such that the active list is sorted → Insert in first position!!

Insertion sort

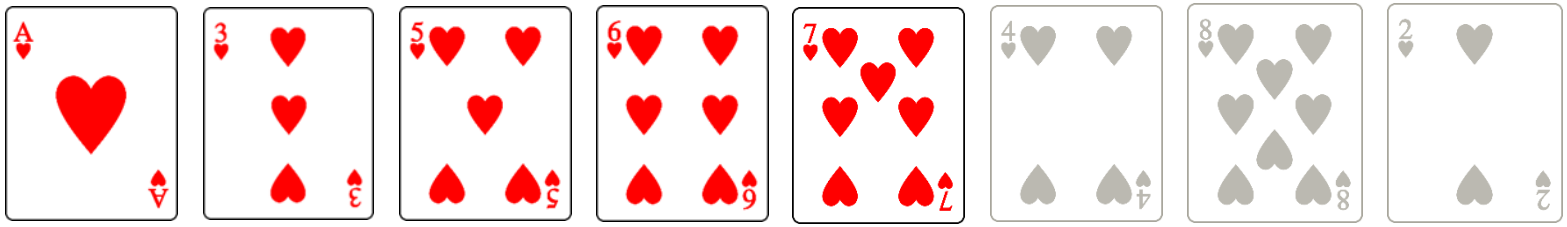


Insertion sort



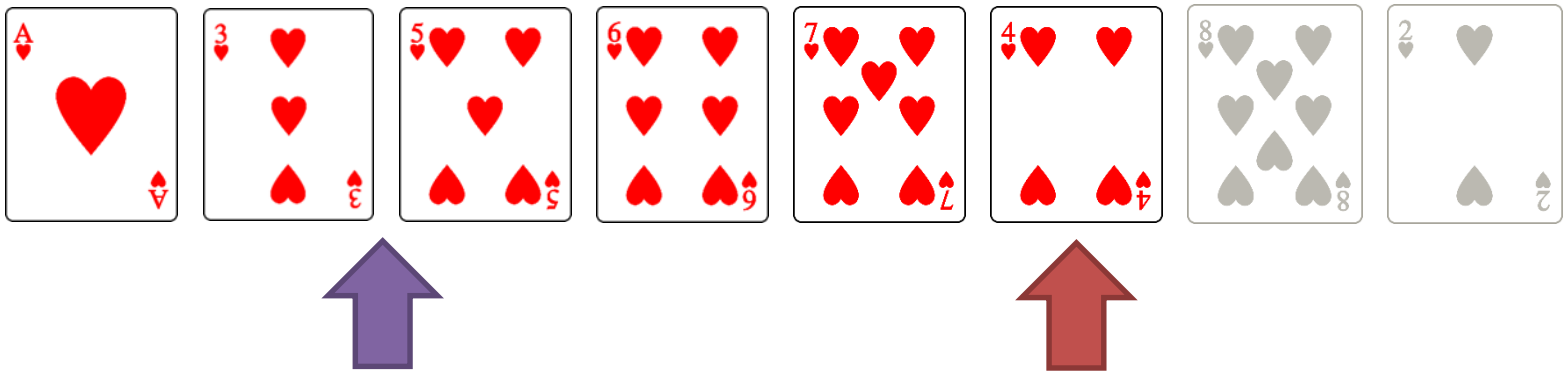
We move to the next element, and we repeat

Insertion sort



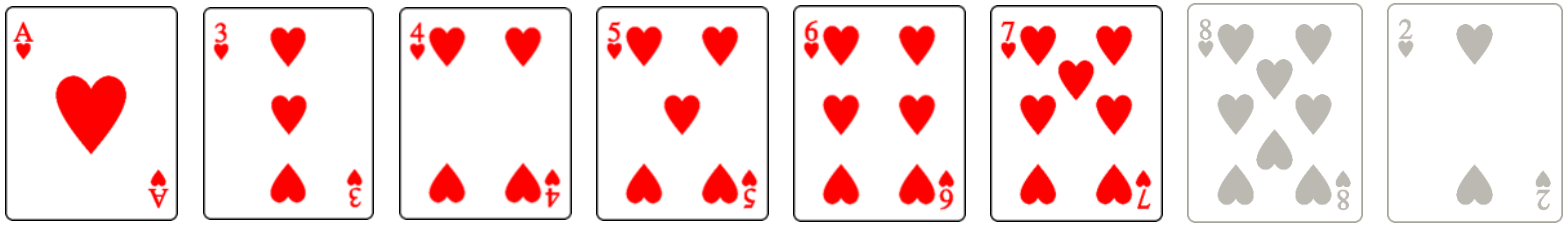
In this case the 7 was at the right location

Insertion sort



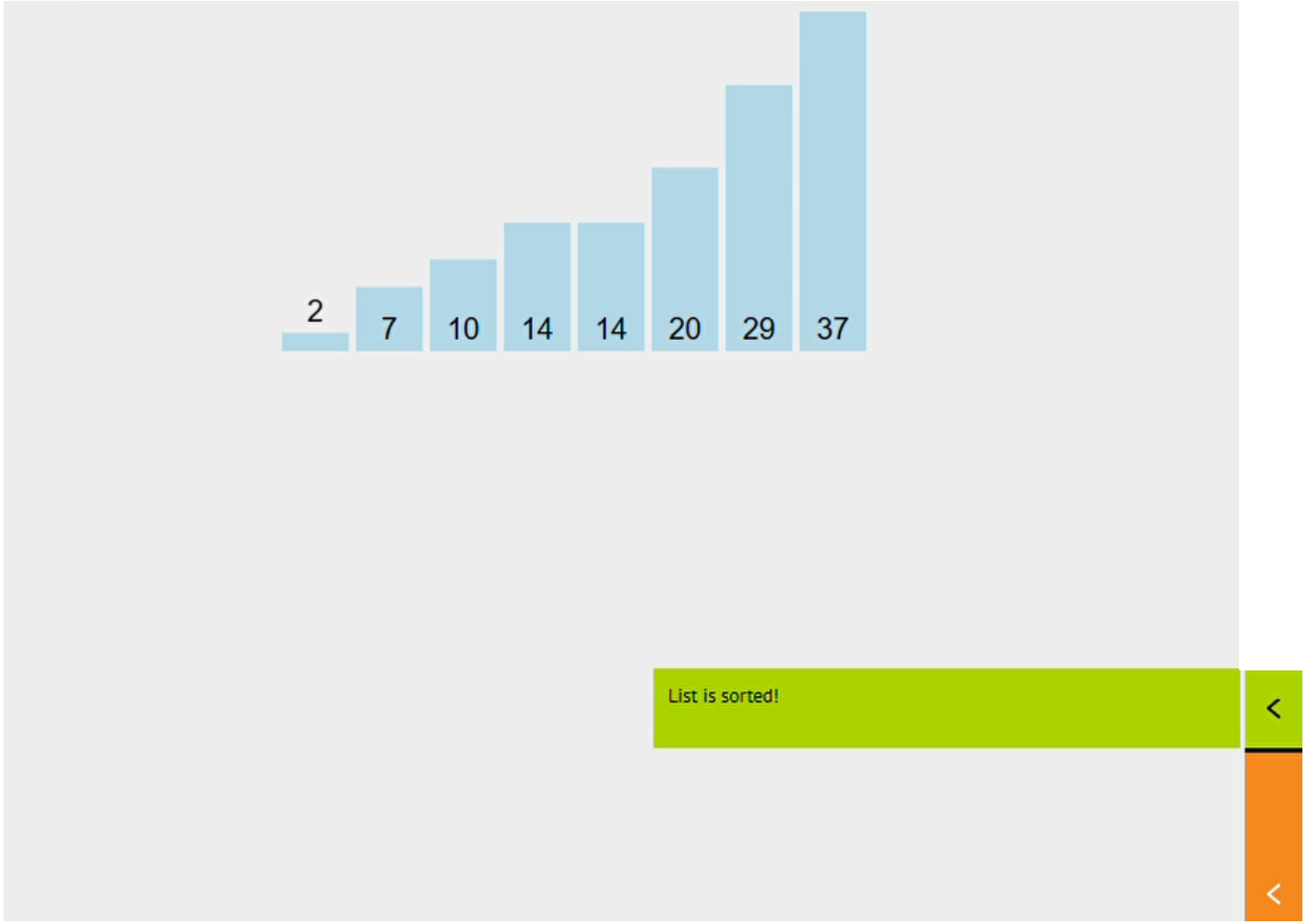
And we will insert the element at the right place by successively comparing it with its left neighbors

Insertion sort



If you do the same for all the card, your list will be sorted!

Insertion sort



Another illustration from VisuAlgo

Insertion sort

Let's get our hands dirty: Time for implementation!

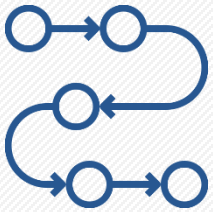
As always, we will **split** the process into small, easy to understand steps!

Insertion sort

Something noteworthy:

At any point in insertion sort, the **left-hand part of the list is sorted** while the right-hand part is not

Insertion sort step-by-step



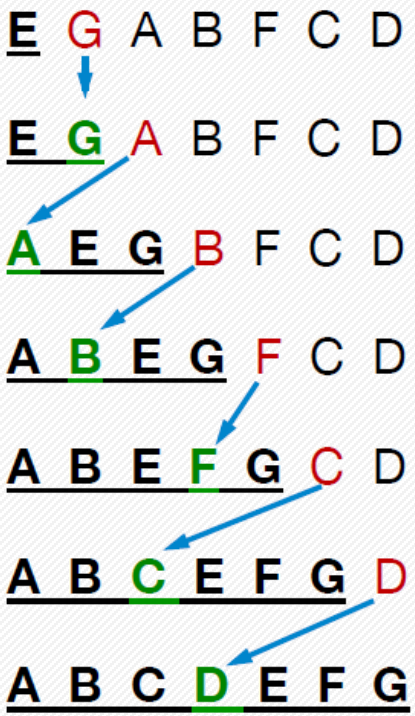
1. The initial item to work on is at index 1
2. Pick up the current item
3. Scan the left-hand part backwards from that index until we find an item lower than the current item or we arrive at the front of the list, whichever comes first
4. Insert the current item back into the list at this location
5. The next item to work on is to the right of the original location of the item
6. Go back to step 2

Insertion sort

One more illustration?

This example illustrates the general idea

- The underlined letters constitute the sorted part of the array
- Initially, the leftmost item is considered to be in a sorted sub-list by itself, and all the other items are in an adjoining unsorted sub-list
- The leftmost item in the unsorted sub-list is selected and inserted into its correct position in the sorted sub-list



Insertion sort in Python (almost)

When you are unsure about how to implement something, try to think of a rough pseudo-code

```
def isort(lst):
    i = 1
    while i < len(lst):
        current_item = lst[i]
        remove current_item from lst
        j = location for current_item
        insert current_item at lst[j+1]
     i+=1
```

We have a couple of gaps in this pseudocode, we have replaced the difficult parts with some statements. We just need to figure out how to implement them in Python!

Insertion sort: the core

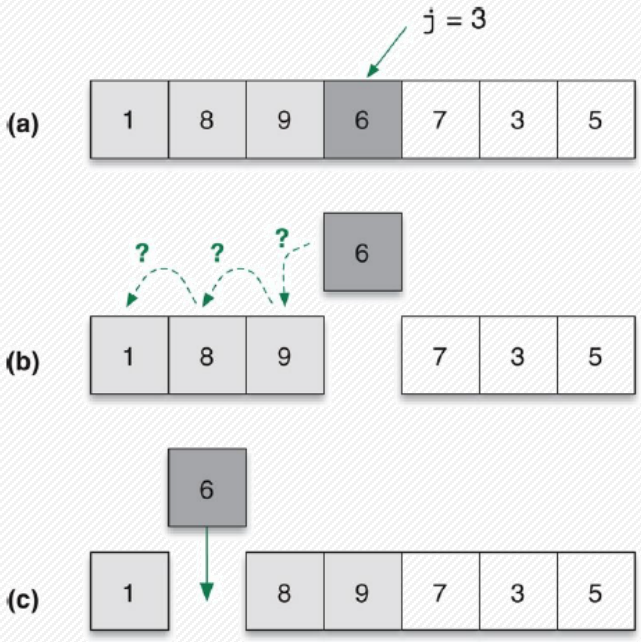
Let's try to understand the core part of insertion sort!

Core implementation steps

Steps:

- Select the next item (step (a))
- Remove the item from the list (step (b))
- Determine at which index the item should go (step (b) also)
- Insert the item at that index (step (c))

- When we're working on the item at index 3, the values to the left (indexes 0 through 2) have been already sorted
- The statements in the body of the while loop find the new location for this item and insert it back into the list

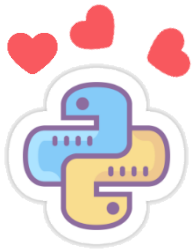


Insertion sort: the core

Now, we know that we need two main functionalities:

- **Removing** an item from the middle of a list
- **Inserting** an item into a list

Thankfully, Python is our best friends and these operations are already implemented as methods of the **list** class



Remove an element: pop()!

Call **`my_list.pop(i)`** to delete the item at location **`i`** in list **`my_list`**

Note: The method returns the item that was deleted

Insert an element: insert()!

Call **`my_list.insert(i, x)`** to insert item **`x`** into the list **`my_list`** at location **`i`**

Let's learn how to use those in the next slide

List methods: pop() & insert()

An example of pop and insert

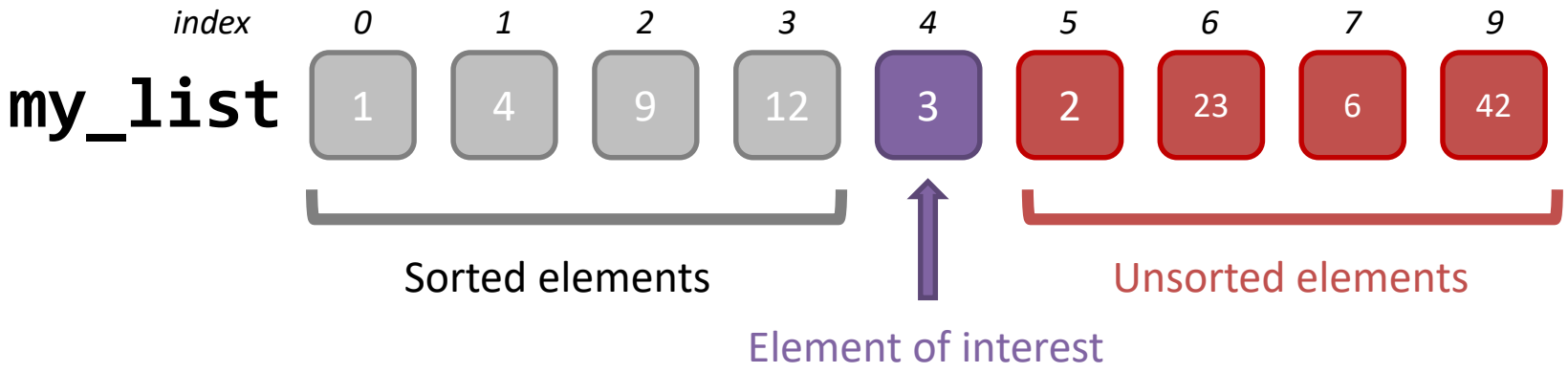
```
#Suppose we had a random list of seven chemical elements
# #(e.g., oxygen, hydrogen, etc.):
chem_elem = ['Co', 'Tm', 'U', 'Hs', 'F', 'Rn', 'Y']
#Now let's remove the item at index 4 and save it in x:
x = chem_elem.pop(4)    # x will contain 'F'
#The list a will become:
→ ['Co', 'Tm', 'U', 'Hs', 'Rn', 'Y']
#Let's insert the element at index 2:
chem_elem.insert(2, x)
#The list a will become:
→ ['Co', 'Tm', 'F', 'U', 'Hs', 'Rn', 'Y']
```

Insertion sort: finding where the item belongs

Before the can remove and re-insert the item of interest, **we first need to know where to place it in the list!!**

Let's take this example

- Here the list has been sorted until index 3
- Our item of interest is stored at the index 4
- The right hand-side of the list is still unsorted

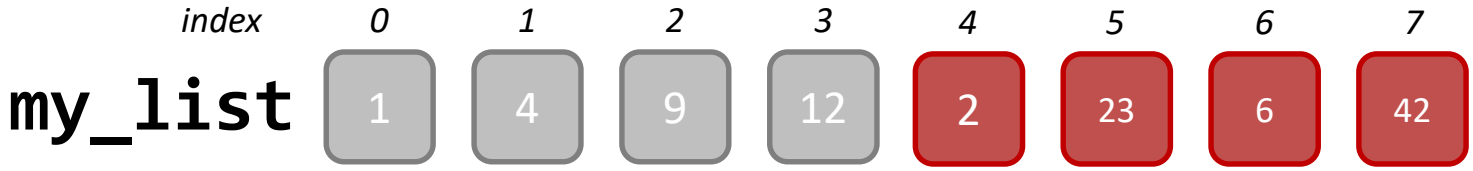


Insertion sort: finding where the item belongs

Before the can remove and re-insert the item of interest, **we first need to know where to place it in the list!!**

Let's take this example

- Here the list has been sorted until index 3
- Our item of interest is stored at the index 4
- The right hand-side of the list is still unsorted



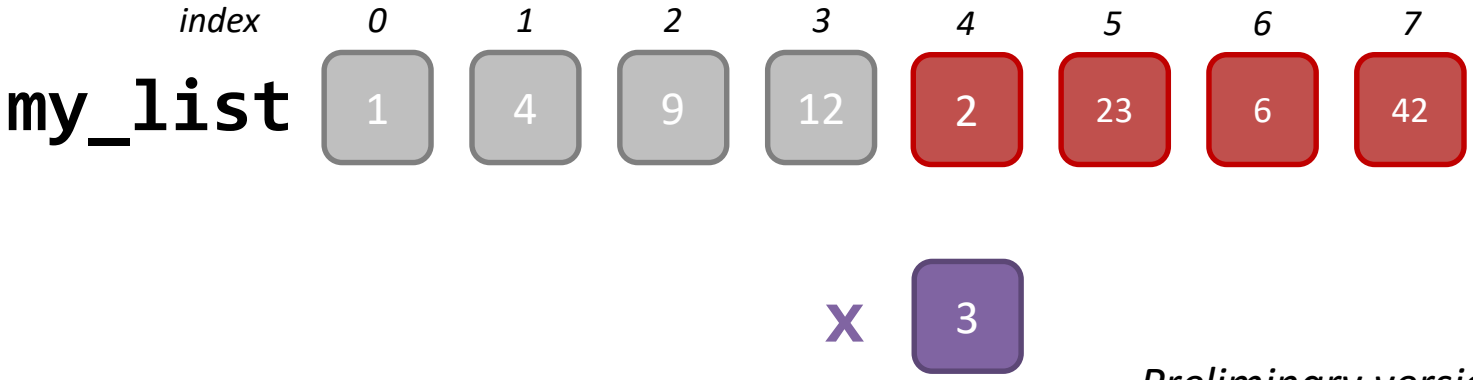
3

Where should we move it?
Well … For sure on the left side!!
How do we "check" the ellements on the left?

Insertion sort: finding where the item belongs

Let's assume our item of interest is located at index `j` we would like to scan the left-hand side of the list to find where to insert it back

- Subtracting 1 from `j` will tell Python to move "left" during its search
- We can use a while loop that keeps subtracting 1 from `j` until it finds the place where item x belongs



Preliminary version of the loop

We can use a while loop that keeps subtracting 1 from `j` until it finds the place where item x belongs

```
while my_list[j-1] > x:
    j = j - 1
```

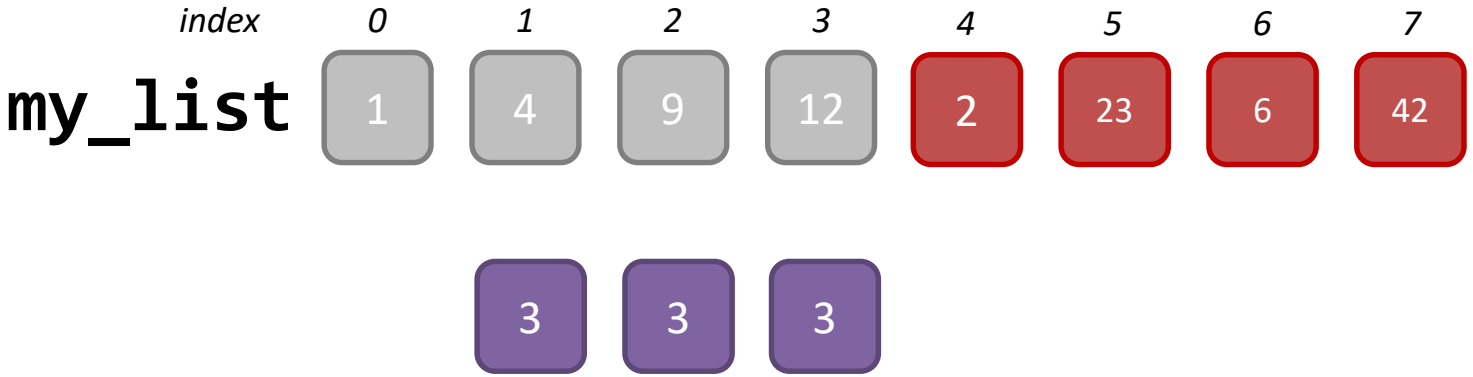
After this loop j will have the value of where we insert the item

Insertion sort: finding where the item belongs

Let's try this!

```
while my_list[j-1] > x:
    j = j - 1
```

Variable	Value
j	1
my_list[j-1]	1
x	3



3>1 then the loop stops and the index j is 1 which is the index where we want to insert our item!!

Insertion sort: finding where the item belongs

index	0	1	2	3	4	5	6	7	8
my_list	1	3	4	9	12	2	23	6	42

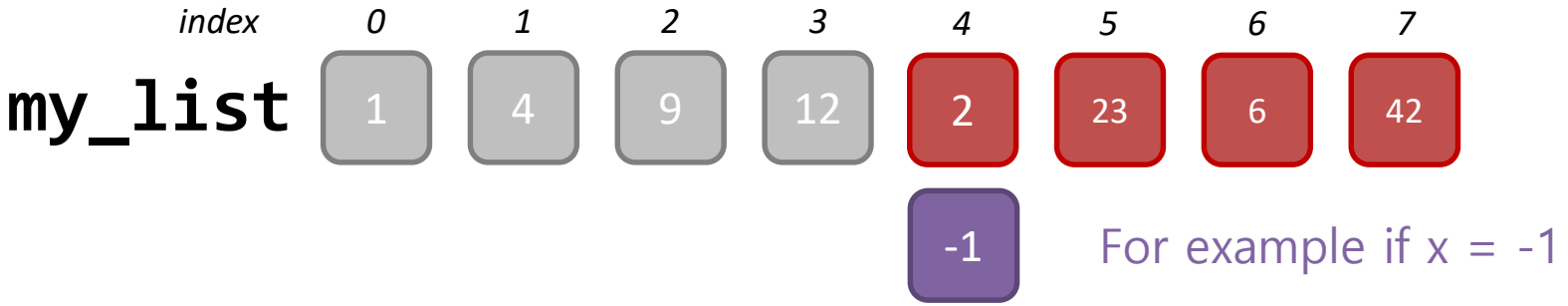
Congrats! Done! We can go home now!

Hooooo!!! WAIT A MINUTE!

Insertion sort: finding where the item belongs

What is happening if x is smaller than the value at index 0??

- The loop will reach a point where `j = 0` and there is nothing remaining to compare
- It will try to compare `x` to `my_list[-1]` – this compares to the last element in the list.
  - This causes unexpected program behavior and eventually an index out of range error



The solution is to keep iterating only if `j > 0` *and* the item to the left is greater than `x`

Final version of the loop

```
while j > 0 and a[j-1] > x:
        j = j - 1
```

move_left()

We can now write a helper function named **`move_left()`** that inserts an item where it needs to go

```
def move_left(my_list, j):
    x = my_list.pop(j)
    while j > 0 and my_list[j-1] > x:
        j -= 1
    my_list.insert(j, x)
```

A call to **`move_left(my_list, j)`** will remove the item at **`my_list[j]`** and insert it back into **`my_list`** where it belongs

Example

- Example: let **a = [1, 3, 4, 6, 2, 7, 5]**
- After **moveLeft(a, 4)**, **a** is **[1, 2, 3, 4, 6, 7, 5]** **# Notice the 2 moved**

Note: A helper function is a function that performs part of the computation of another function.

Completed isort()

With a helper function to move items, writing
`isort` is easy

Implementation

- Use a for loop where an index variable **`i`** marks the start of the unsorted region
  - Initially **i** will be 1 (the single item at **`my_list[0]`** is a sorted region of size 1)
  - In the body of the loop, just call **`move_left`** to move the item at location **i** to its proper position

```
def isort(my_list):
    for i in range(1, len(my_list)):
        move_left(my_list, i)
```

Why is it okay to use a 'for' loop instead of a 'while' loop?

Completed isort()

```
def isort(my_list):
    for i in range(1, len(my_list)):
        move_left(my_list, i)
```

An example of how to use the **isort** function:

```
nums = [5, 2, 9, 10, -2, 6, 7, 1]
isort(nums)   # nums is now sorted
```

Isort(): Performance

We saw earlier that for a list with n items we can expect, on average, to do n/2 comparisons during a linear search

Can we have a similar estimation for the insertion sort?

Let's think about that!

- At first glance, it might seem that insertion sort is a "linear" algorithm like linear search
  - It has a for loop that progresses through the list from left to right
- But remember that **`moveLeft`** also contains a loop
  - The step that finds the proper location for the current item is also a loop
  - It scans left from location i, going all the way back to 0 if necessary

Isort(): Performance

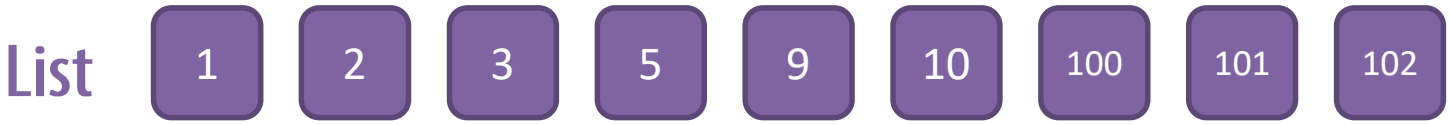
If we write **`isort`** without the **`move_left`** helper function, we can see that one loop is inside another loop

```
def isort(my_list):
    for i in range(1, len(my_list)):        # Loop 1
        j = i
        x = my_list.pop(j)
        while j > 0 and my_list[j-1] > x:  # Loop 2
            j = j - 1
        my_list.insert(j, x)
```

Isort(): Performance

Let's study the best case scenario!

Our list is already sorted



The outer loop iterates from 1 to n-1

The function move_left will never perform any action

→ Linear complexity $O(n)$

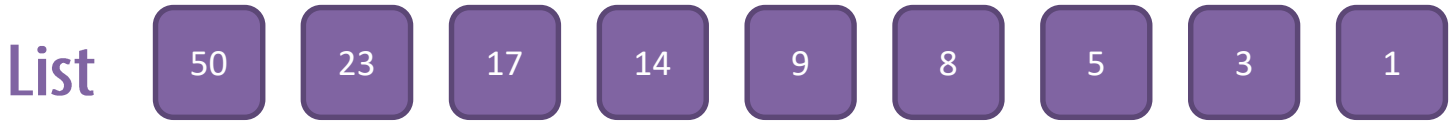
```
def isort(my_list):
    for i in range(1, len(my_list)):
        move_left(my_list, i)
```

But when we compute the time complexity, we only care about the worst case scenario!

What is the worst case?

Isort(): Performance

Here is one occurrence of the worst case scenario!



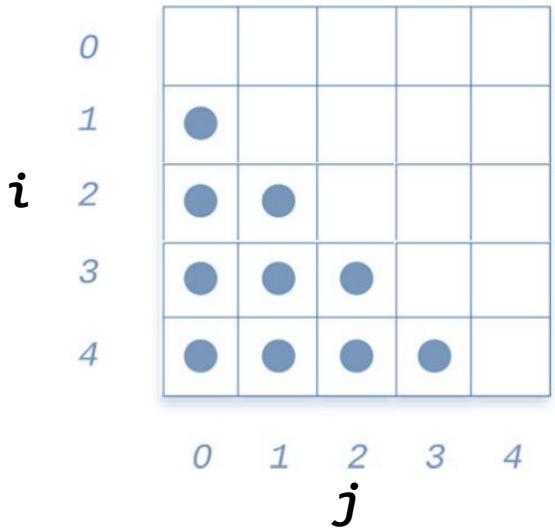
Nested loops

- The outer loop has the same structure as the iteration in linear search
  - A for loop checking the index **i** from 1 up to n-1
- Note that the items to the left of **i** are always sorted
- The inner loop moves **my_list[i]** to its proper location in the sorted region
  - Thus the size of the sorted region grows on each iteration
- An algorithm like **isort** that has one loop inside another is said to have **nested loops**
  - Need to understand the loops better to analyze the code

```
def isort(my_list):
    for i in range(1, len(my_list)):      # Loop 1
        j = i
        x = my_list.pop(j)
        while j > 0 and my_list[j-1] > x:  # Loop 2
            j = j - 1
        my_list.insert(j, x)
```

Isort(): Nested loop

In the figure, a dot in a square indicates a potential comparison when sort a list of 5 items



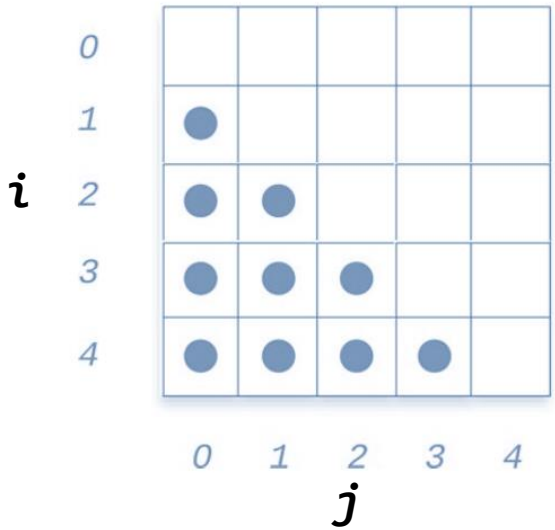
*The row and column numbers are **i** and **j** from our code*

For any value of **i**, the inner loop might have to compare values from `my_list[i-1]` all the way down to `my_list[0]`

```
def isort(a):
    for i in range(1, len(a)):
        j = i
        x = a.pop(j)
        while j > 0 and a[j-1] > x:
            j = j - 1
        a.insert(j, x)
```

Isort(): Nested loop

In the figure, a dot in a square indicates a potential comparison when sort a list of 5 items



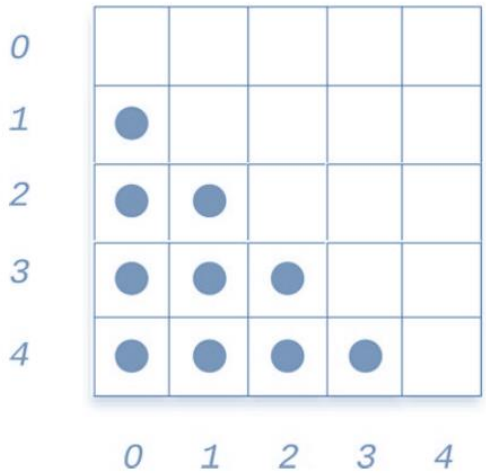
*The row and column numbers are **i** and **j** from our code*

- So, as **i** increases, the potential number of comparisons also increases
- Note the label next to a row is the value of **i** passed to the **move_left** function

```
def isort(a):
    for i in range(1, len(a)):
        j = i
        x = a.pop(j)
        while j > 0 and a[j-1] > x:
            j = j - 1
        a.insert(j, x)
```

Isort(): Nested loop

In the figure, a dot in a square indicates a potential comparison when sort a list of 5 items



- There are 4+3+2+1 = 10 dots total
  - This shows there we be at most 10 comparisons
- For a list of n=6 items, it could have 5 additional comparisons
  - Then at most 15 comparisons = (5+4+3+2+1)

In general, for a list with n items, the potential number of comparisons is $\mathbf{n(n-1)/2 \approx n^2/2}$
We say that in the worst case, the sorting algorithm will make **approximately n^2 / 2** comparisons

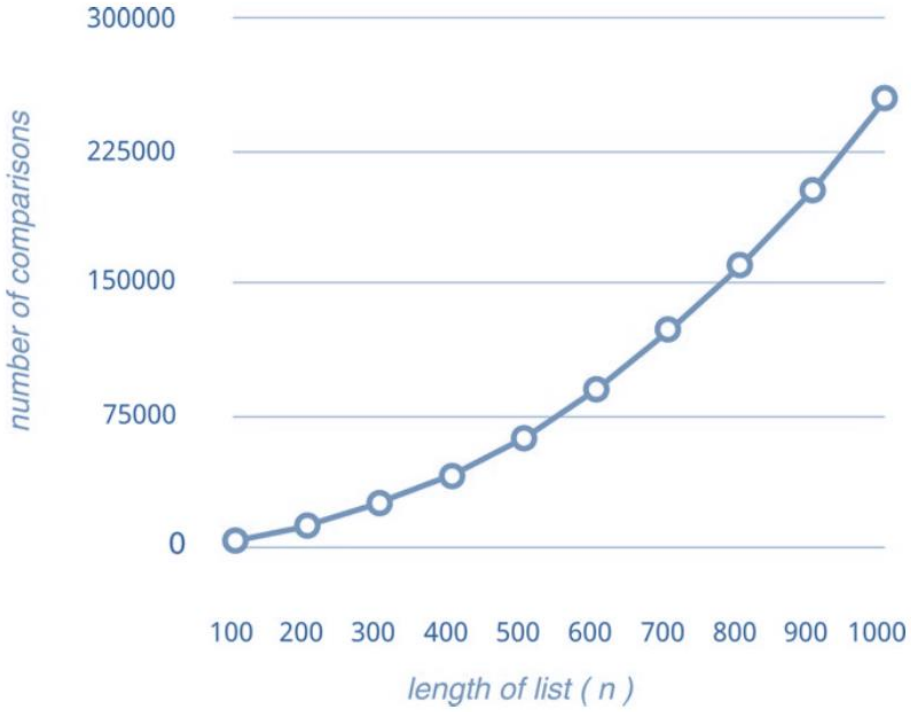
$$O(n^2/2)$$

Insertion sort complexity

- The formula for the worst case number of comparisons in isort is:
  - $n(n-1)/2 \approx (n^2 - n)/2$
- For small lists, we can compute the exact answer (e.g. n = 5):
  - $(5^2-5)/2 = 20/2 = 10$
- For larger lists, the **–n** term doesn't affect the result much because n^2 will be much larger than –n
- We say that the n^2 term **dominates** the expression
- Therefore, we can get a good estimate by computing only $n^2 / 2$

Insertion sort complexity

This graph gives a sense of how much "work" the insertion sort algorithm does on average, based on the length of the input list



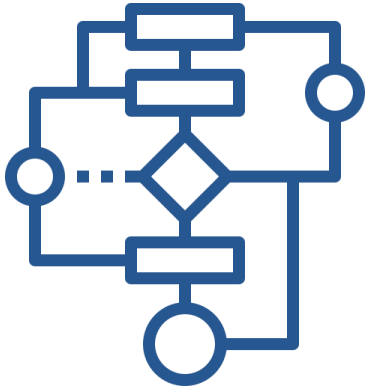
Selection Sort

Selection sort

The selection sort algorithm is another $O(n^2)$ iteration-based algorithm for sorting a list of values

Algorithm

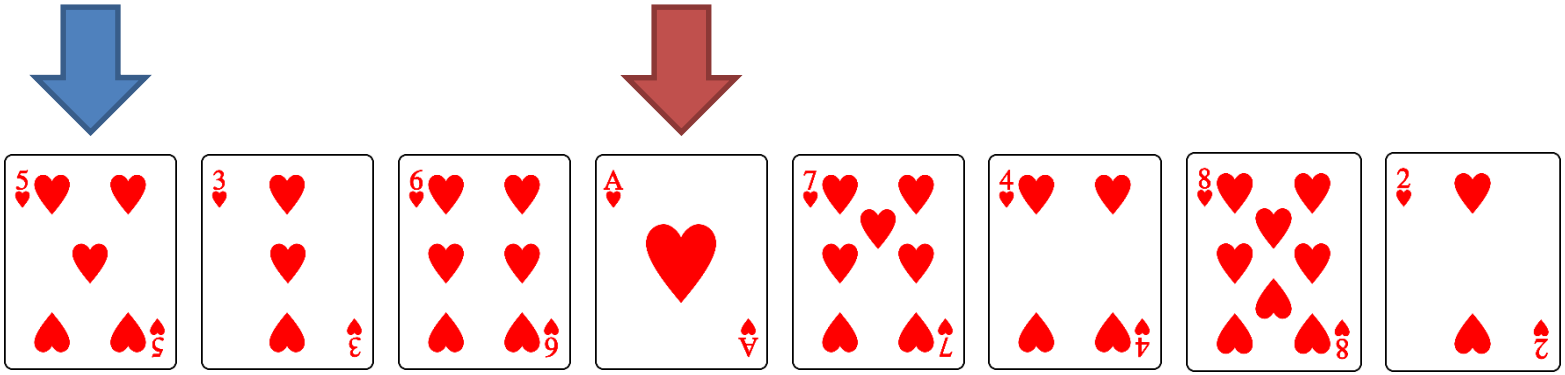
1. Find the smallest value. Swap it (exchange it) with the first value in the list.
2. Find the second-smallest value. Swap it with the second value in the list.
3. Find the third-smallest value. Swap it with the third value in the list.
4. Repeat finding the next-smallest value and swapping it into the correct position until the list is sorted.



Selection sort

One technique called **Selection sort**

It repeatedly searches for and swaps cards in the list

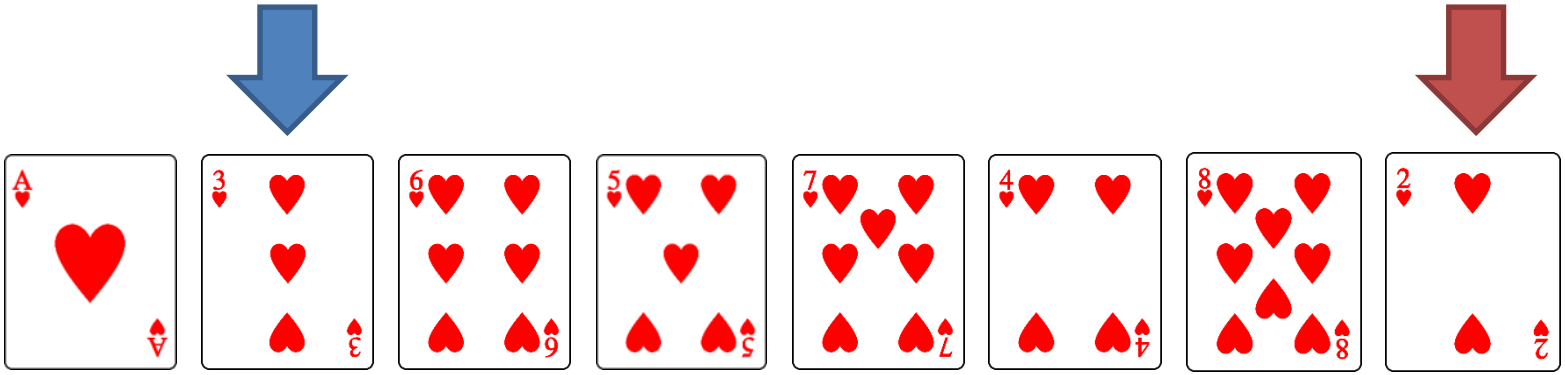


1. First, find the smallest item and exchange it with the card in the first position

Selection sort

One technique called **Selection sort**

It repeatedly searches for and swaps cards in the list

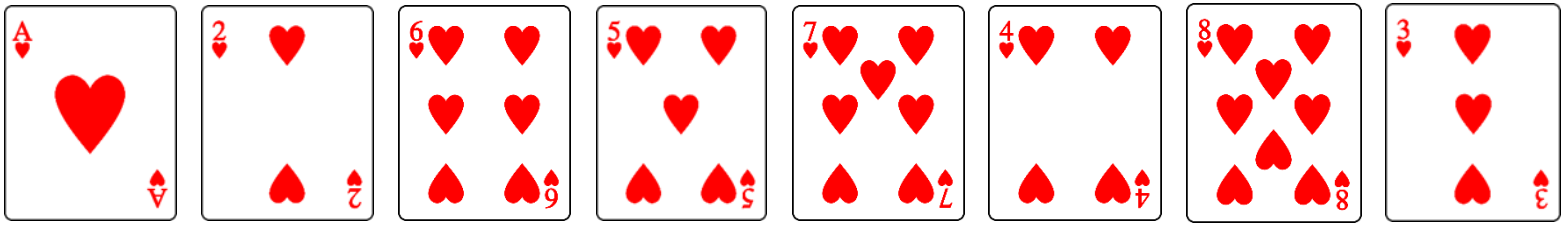


1. First, find the smallest item and exchange it with the card in the first position
2. Select the second smallest item and exchange it with the card in the second position

Selection sort

One technique called **Selection sort**

It repeatedly searches for and swaps cards in the list

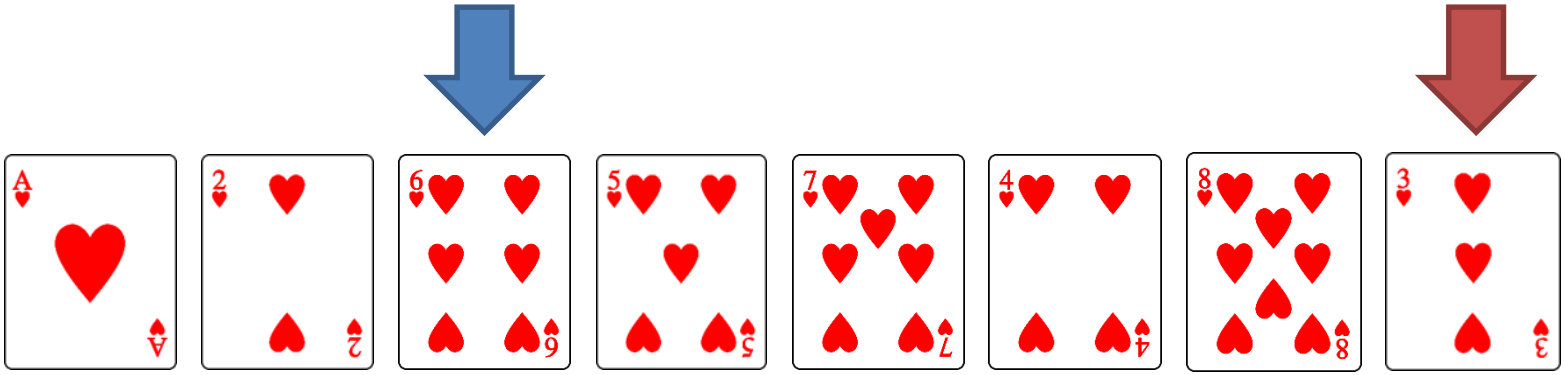


1. First, find the smallest item and exchange it with the card in the first position
2. Select the second smallest item and exchange it with the card in the second position

Selection sort

One technique called **Selection sort**

It repeatedly searches for and swaps cards in the list

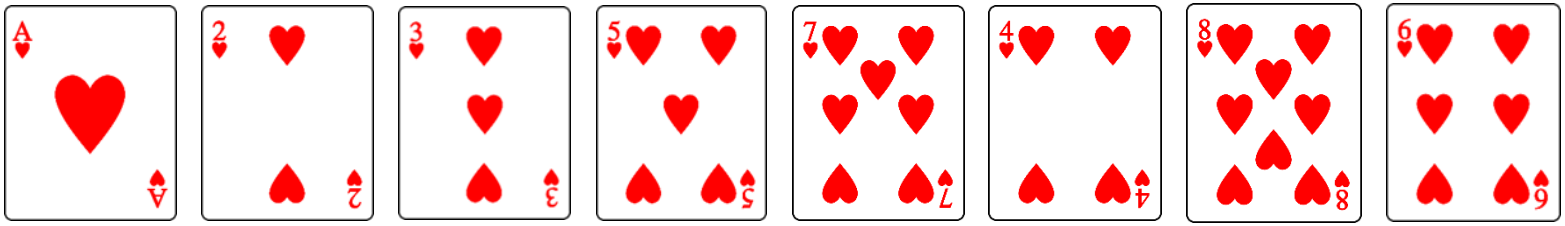


1. First, find the smallest item and exchange it with the card in the first position
2. Select the second smallest item and exchange it with the card in the second position
3. Select the third smallest item and exchange it with the card in the third position

Selection sort

One technique called **Selection sort**

It repeatedly searches for and swaps cards in the list

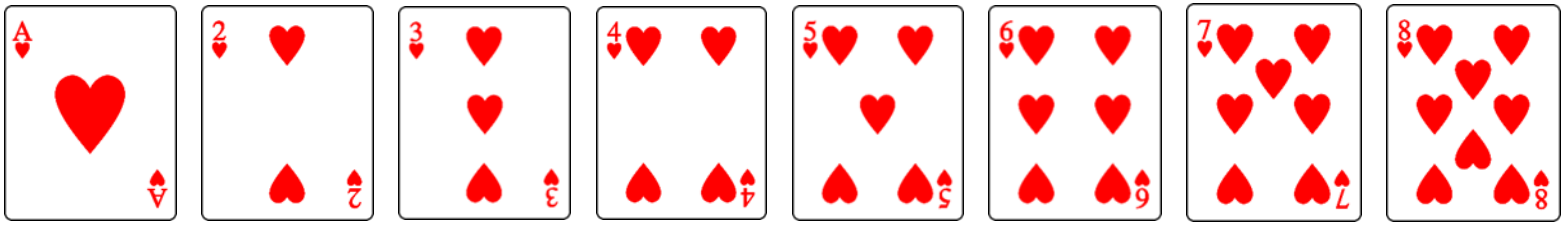


1. First, find the smallest item and exchange it with the card in the first position
2. Select the second smallest item and exchange it with the card in the second position
3. Select the third smallest item and exchange it with the card in the third position

Selection sort

One technique called **Selection sort**

It repeatedly searches for and swaps cards in the list



1. First, find the smallest item and exchange it with the card in the first position
2. Select the second smallest item and exchange it with the card in the second position
3. Select the third smallest item and exchange it with the card in the third position
4. Keep going!

Selection sort

Another illustration from visualgo.net/en/sorting

Selection sort: Example 1

```
7 1 6 9 5 4
1 7 6 9 5 4     swapped 1 and 7
1 4 6 9 5 7     swapped 4 and 7
1 4 5 9 6 7     swapped 5 and 6
1 4 5 6 9 7     swapped 6 and 9
1 4 5 6 7 9     swapped 7 and 9
```

Eventually, only the largest value will remain. But, it will be in the rightmost position, so we don't need to do anything with it

Selection sort

```
8 4 6 7 5 3 2 9 1
1 4 6 7 5 3 2 9 8     swapped 1 and 8
1 2 6 7 5 3 4 9 8     swapped 2 and 4
1 2 3 7 5 6 4 9 8     swapped 3 and 6
1 2 3 4 5 6 7 9 8     swapped 4 and 7
1 2 3 4 5 6 7 9 8     swapped 5 and 5 (?)
1 2 3 4 5 6 7 9 8     swapped 6 and 6 (?)
1 2 3 4 5 6 7 9 8     swapped 7 and 7 (?)
1 2 3 4 5 6 7 8 9     swapped 8 and 9
```

Note that sometimes the algorithm does no useful work, like "swapping" 5 with itself

Selection sort

- Perhaps you noticed that during execution of the algorithm, the list is divided into two parts:
  - the sorted part (green)
  - the yet-to-be-sorted part (black)
- Also you may have noticed that the largest element winds up in the rightmost spot without any additional work
- Think about that for a moment. Suppose we have 10 elements in our list.
  - Once we have moved the 9 smallest elements into their final positions, the 10th (largest) value must be in the rightmost position
  - This has a small implication in the implementation

Selection sort: Swap elements

Python makes it very easy to **swap** (exchange) the values stored in two variables

Swap two variables

To exchange the contents of two variables x and y, all we need to type is this:

```
x, y = y, x
```

This swapping notation also works with elements of a list

- Suppose **i** and **j** are valid indices of list **my_list**
- We can type this to swap the contents of `my_list[i]` and `my_list[j]`:

```
my_list[i], my_list[j] = my_list[j], my_list[i]
```

With the pseudocode from earlier and this syntax for swapping list elements, we can implement selection sort

Out of the box sorting in python

Details and examples from: How to Use sorted() and sort() in Python – Real Python

Sorting data in Python: sorted()

While implementing these functions yourself is a very good exercise, Python already contains very effective sorting algorithms!

Sorting a list in one line

The function `sorted` will sort an input container and return its sorted version

```
numbers = [6, 9, 3, 1]
numbers_sorted = sorted(numbers)
print(numbers_sorted)
```

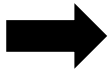
→ [1, 3, 6, 9]

`sorted()` will not modify the original list but return a new sorted one

Sorting data in Python: sorted()

The function `sorted()` can also work on other container such as tuples

```
numbers = (6, 9, 3, 1)
numbers_sorted = sorted(numbers)
print(numbers_sorted)
print(numbers)
```



```
[1, 3, 6, 9]
(6, 9, 3, 1)
```

A few notes on `sorted`:

- Regardless the type of container it returns a list
- The function `sorted()` did not have to be defined. It's a built-in function that is available in a standard installation of Python.
- `sorted()`, with no additional arguments or parameters, is ordering the values in numbers in an ascending order, meaning smallest to largest.
- The original numbers variable is unchanged because `sorted()` provides sorted output and does not change the original value in place.

Sorting data in Python: sorted()

The function `sorted()` can also sort strings!

```
string_number_value = '34521'
string_value = 'I like to sort'
sorted_string_number = sorted(string_number_value)
sorted_string = sorted(string_value)
print(sorted_string_number)
print(sorted_string)
```

```
['1', '2', '3', '4', '5']
[' ', ' ', ' ', 'I', 'e', 'i', 'k', 'l', 'o', 'o', 'r', 's', 't', 't']
```

Sorting data in Python: sorted()

You can also sort list of words by their first letter!

```
string_value = 'I like to sort'
sorted_string = sorted(string_value.split())
print(sorted_string)
print(' '.join(sorted_string))
```

```
['I', 'like', 'sort', 'to']
I like sort to
```

Note
When you are sorting string, upper/lower cases matter!

Sorting data in Python: sorted()

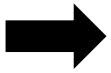
If the first letter is the same, then sorted() will use the second character to determine order, and the third character if that is the same, and so on, all the way to the end of the string:

```
very_similar_strs = ['hhhhhd', 'hhhhha', 'hhhhhc','hhhhhb']
print(sorted(very_similar_strs))
```

```
['hhhhha', 'hhhhhb', 'hhhhhc', 'hhhhhd']
```

Be careful, the function sorted() does not work with mixed inputs

```
mixed_numbers = [5, "1", 100, "34"]
sorted(mixed_numbers)
```



```
TypeError: '<' not supported between instances of 'str' and 'int'
```

Sorting data in Python: sorted()

You can also sort in a descending order using the flag "reverse"

```
names_with_case = ['harry', 'Suzy', 'al', 'Mark']
sorted_names = sorted(names_with_case, reverse=True)
print(sorted_names)
```

['harry', 'al', 'Suzy', 'Mark']

Sorting data in Python: sorted()

Using `sorted()` you can also sort according to other criterion using the flag "key"

Sort by length:

```
words = ['banana', 'pie', 'Washington', 'book']
print(sorted(words, key=len))
```

['pie', 'book', 'banana', 'Washington']

Sort by lower/upper cases:

```
names_with_case = ['harry', 'Suzy', 'al', 'Mark']
print(sorted(names_with_case, key=str.lower))
```

['al', 'harry', 'Mark', 'Suzy']

Sorting data in Python: sorted()

You can create your own key functions

```
def reverse_word(word):
    return word[::-1]

words = ['banana', 'pie', 'Washington', 'book']
print(sorted(words, key=reverse_word))
```

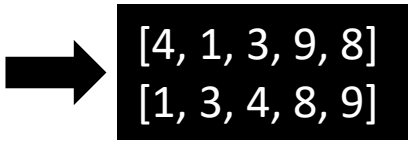
➡ ['banana', 'pie', 'book', 'Washington']

Note
Your function should be void and modify directly the object

Sorting data in Python: .sort()

Instead of using the function `sorted()`, you can use the list method `.sort()`

```
lst = [4, 1, 3, 9, 8]
print(lst)
lst.sort()
print(lst)
```



.sort() can also take keys like reverse or any inline functions but it modifies the original list directly without returning anything

Nearest Neighbors

Context

- Given a query data you want to find its most similar data (nearest neighbor) in a database.

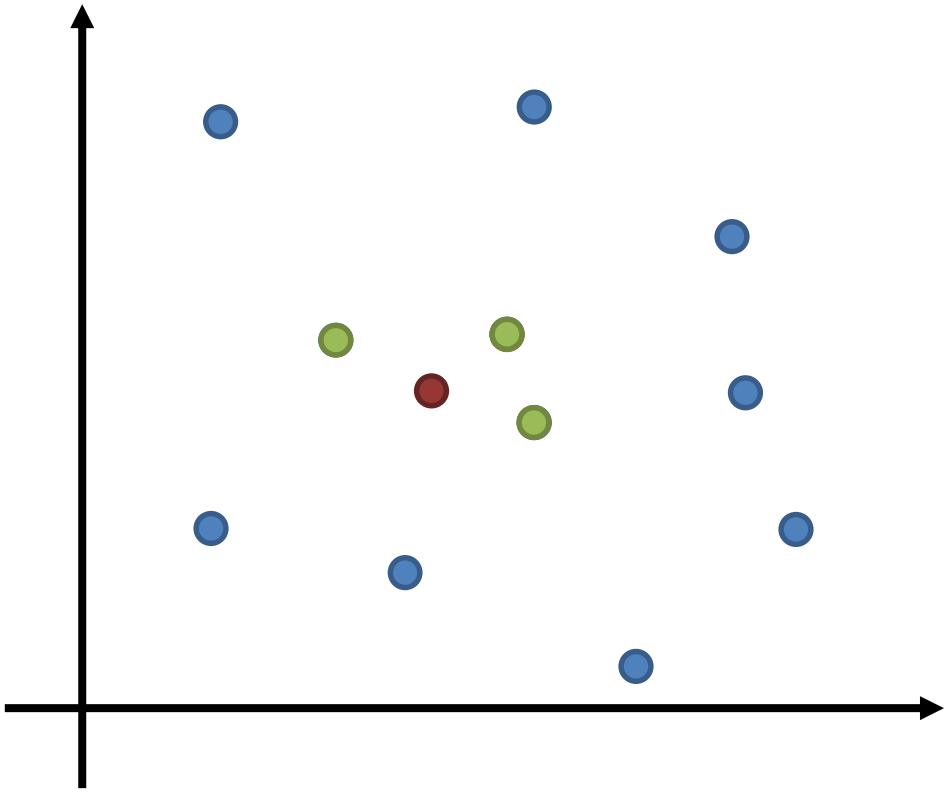
- A practical example is to find a fingerprint in a very large dataset

Query

Database

Nearest neighbor in Hollywood

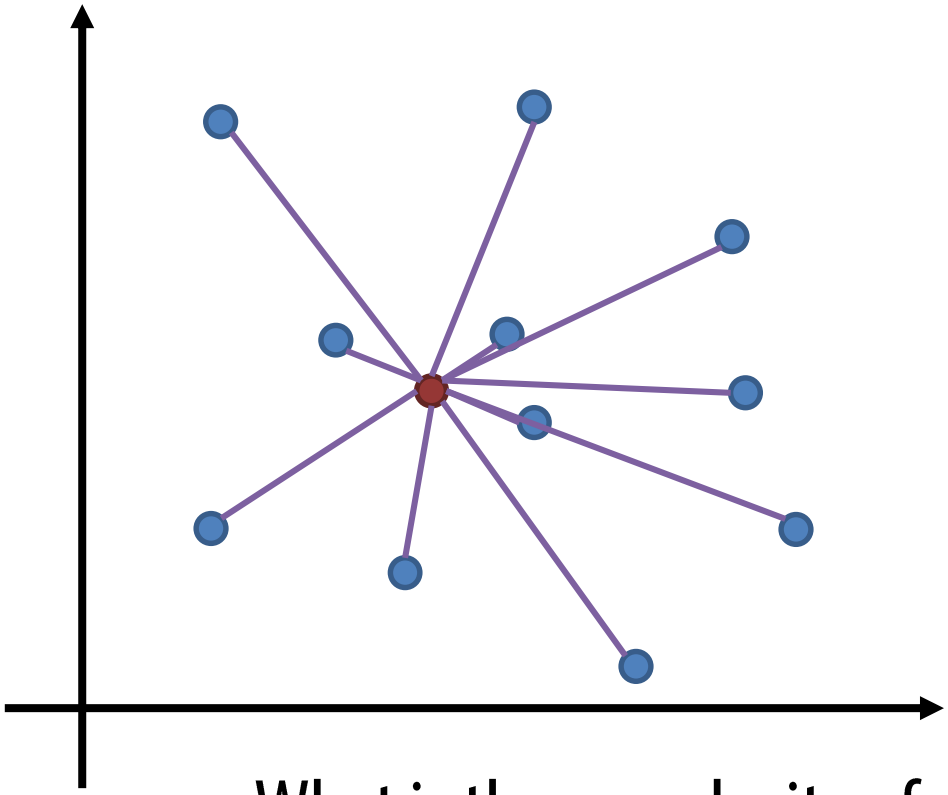
A 2D nearest neighbors example



- Point in dataset
- Query point
- Nearest neighbor
- *k*-NN (*k*=3)

How to return the *k* Nearest Neighbors (*k*-NN)?

A 2D nearest neighbors example



A naïve approach?

Compute the distance to all datapoints and keep the n closest ones

Distance metric?

Euclidean, Manhattan . . . To be defined depending upon the type of data

What is the complexity of such approach?

O(N)

What if *N* is very large? What if *d* is very large?

Example with MNIST

- How to find a number with most similarity in MNIST?
- What dimensionality? 28x28 = 784

Original image

Vectorized image

- L2 photometric distance:

$$d(a,b) = \sum_{1 \le i,j \le 28} \sqrt{(a_{ij} - b_{ij})^2}$$

$d($  $,$  $) = 115.7$

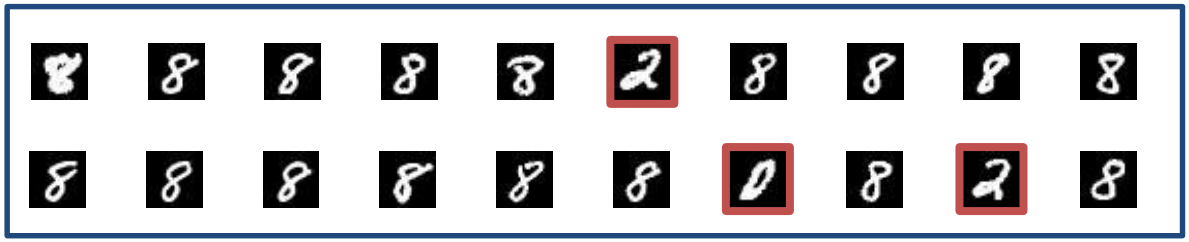
Example proposed by: Erik Berhardsson

Example with MNIST

- Database 1000 digits from MNIST

- Query image:

- Top-20 Nearest Neighbors:



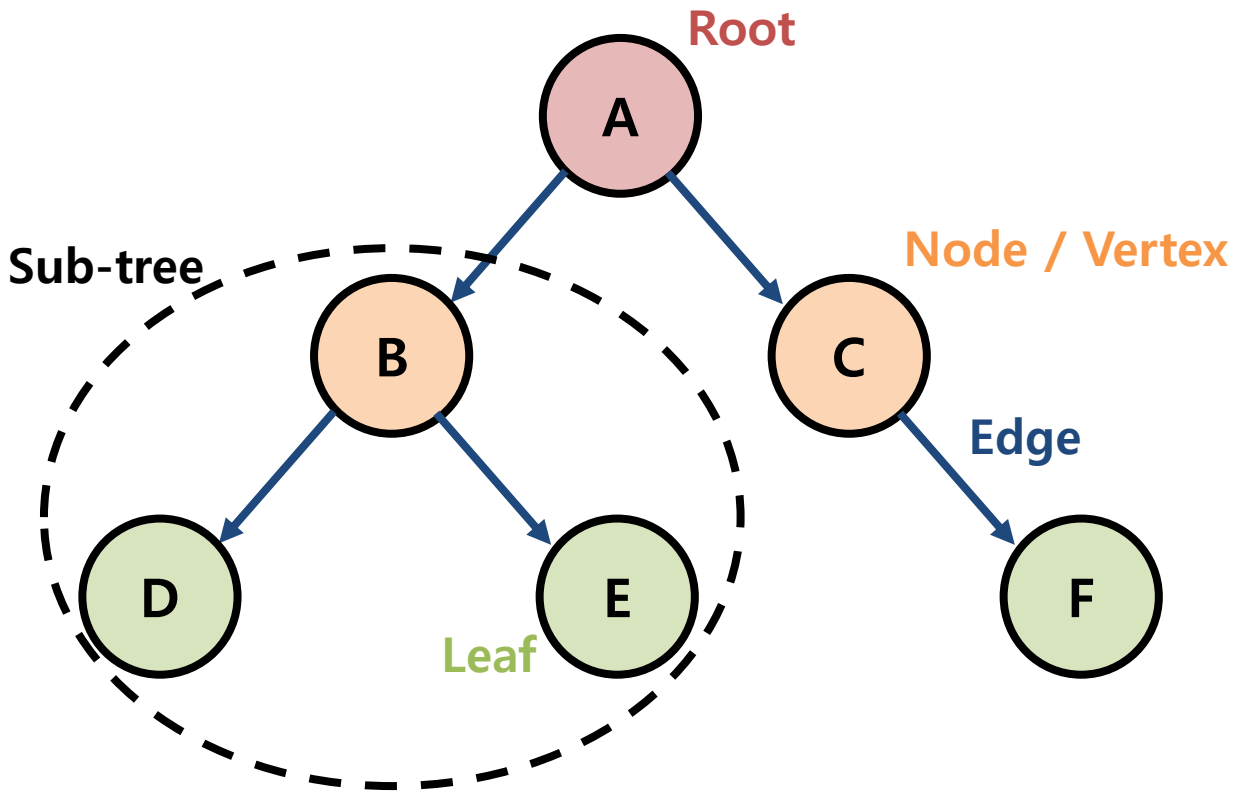
Kd-tree

- A "smarter", elegant and efficient approach to perform a NN search is to use appropriate data structures such as: **kd-tree**
- Proposed by Jon Louis Bentley in 1975 (Stanford)
- Kd-tree is a Binary Search Tree (BST)
- Idea: kd-tree is a space-partitioning data structure for organizing points in a k-dimensional space

Example with MNIST

- **Brute force** search (*O(N)*):
  - It takes ~7ms to compare the query image and one image in the dataset
  - With a dataset N=1000, search time → 1000*7 ~ 7 seconds
- **Kd-Tree** query (*O(log(N))*) for 20NN ~ 25ms
- Creation of the tree ~31ms
- Keep in mind that using the direct photometric distance is **not** an efficient manner to compare images (high dimensionality, not robust …) and is used just as a simple example.

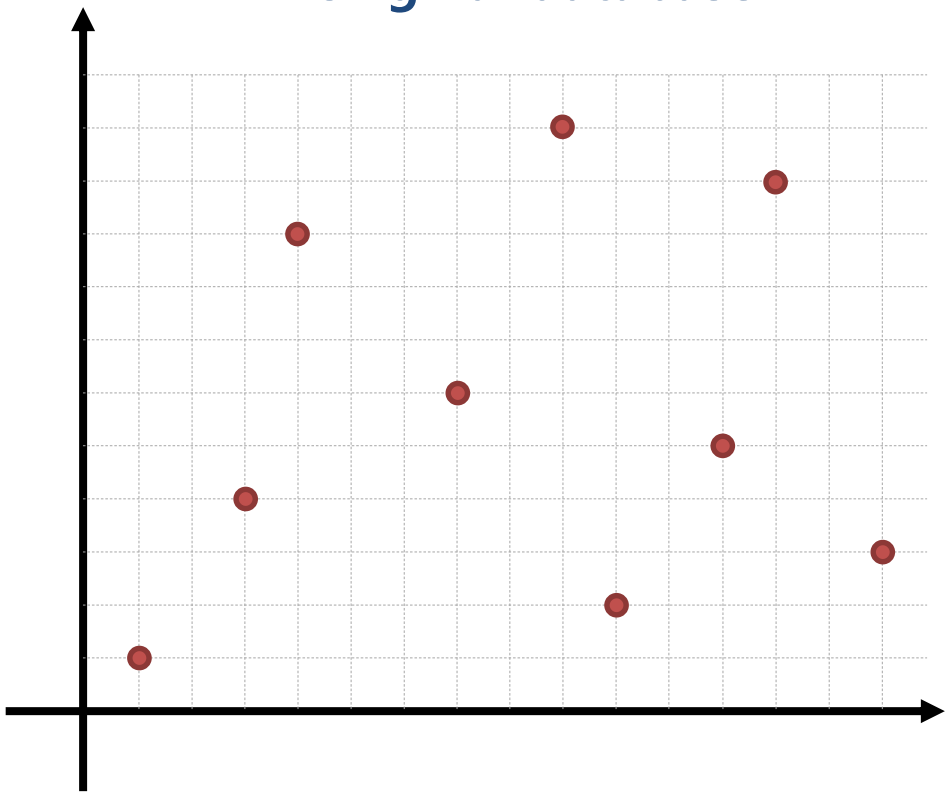
Binary Search Tree: Terminology



Binary tree: A node has at most 2 children

Kd-tree creation

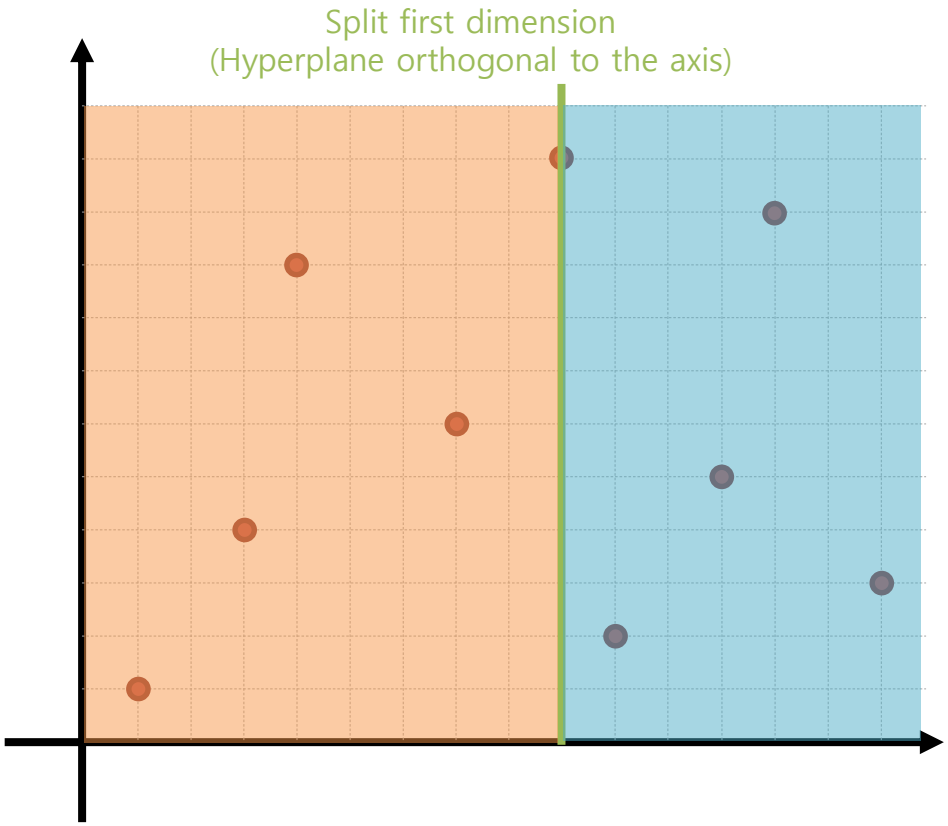
Original database



Points list

x	y
1	1
3	4
4	9
7	6
9	11
10	2
12	5
13	10
15	3

Kd-tree creation



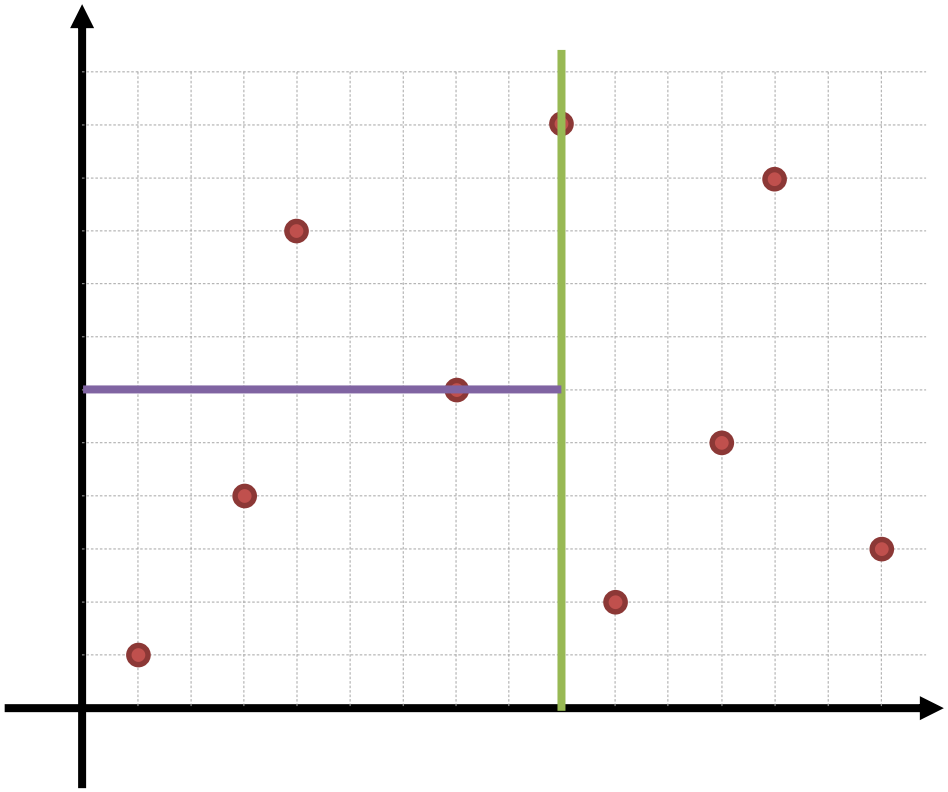
(1,1), (3,4), (4,9), (7,6), (9,11), (10,2), (12,5), (13,10), (15,3)

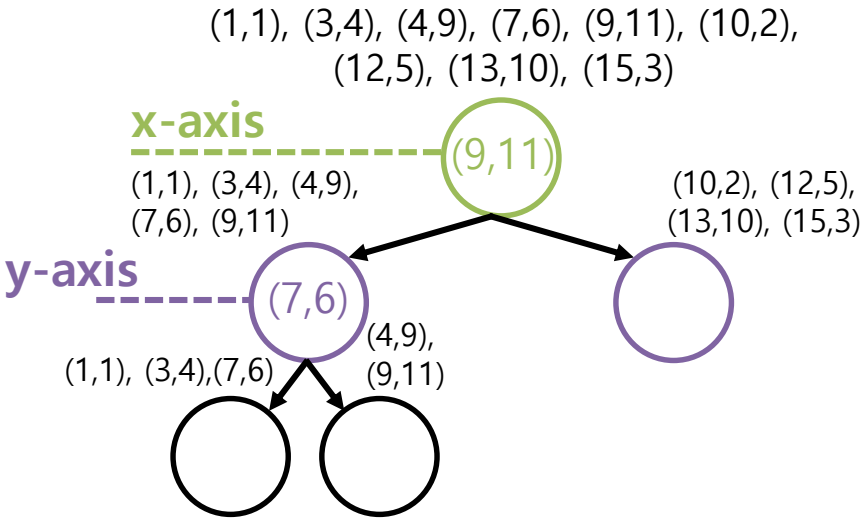
Gives two children

The first split is performed along the x-axis using the median value along this dimension!

Other splits are possible (e.g. mean) but the median value ensures a better balanced tree

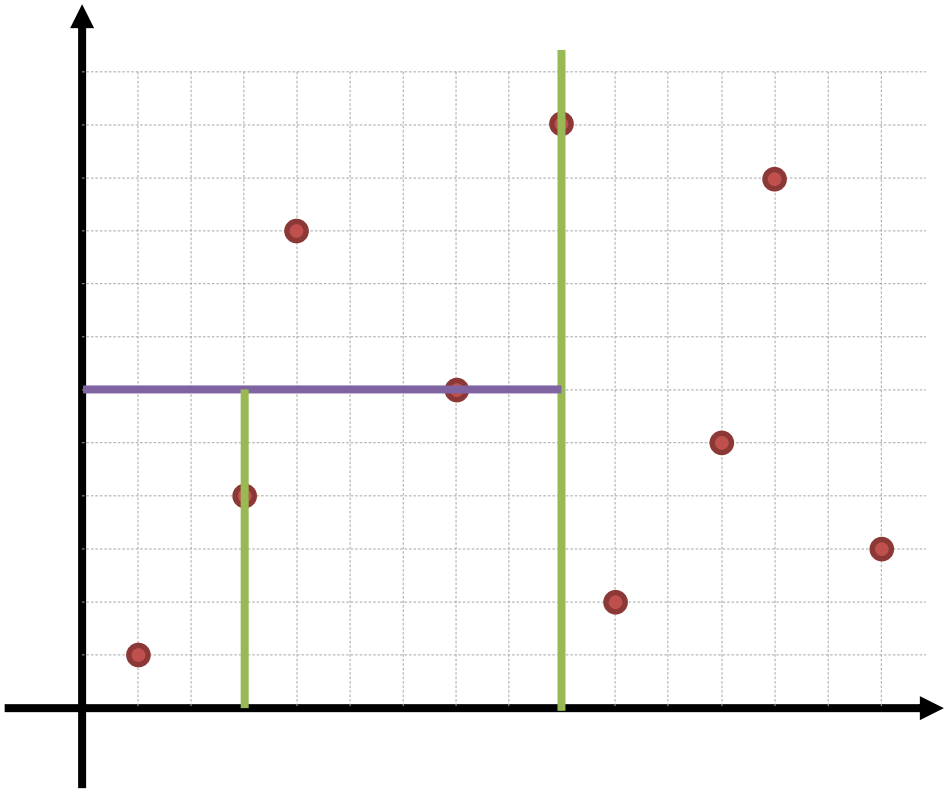
Kd-tree creation

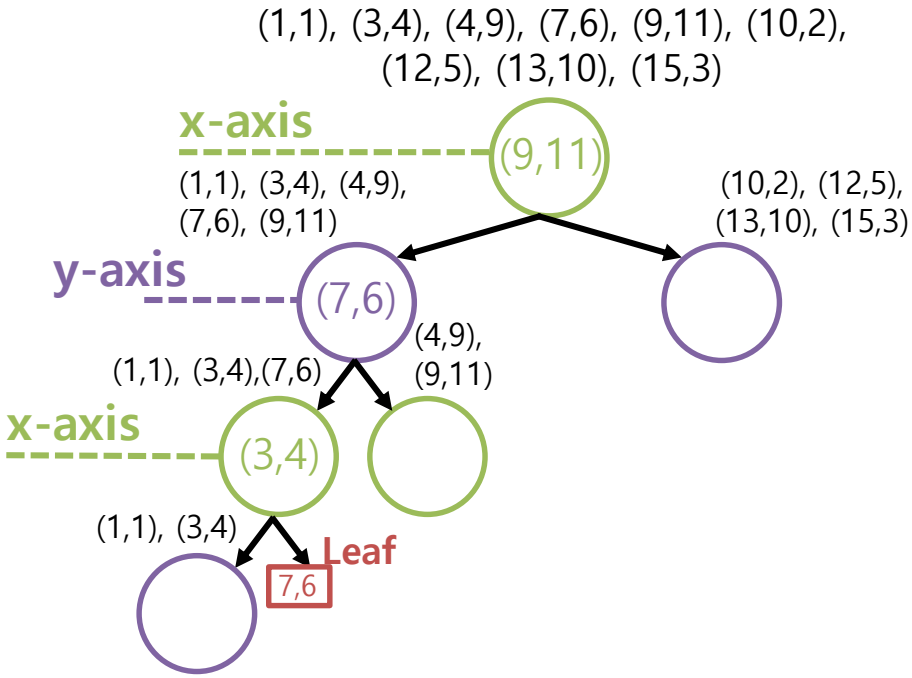




The second split is performed along the next dimension: y-axis

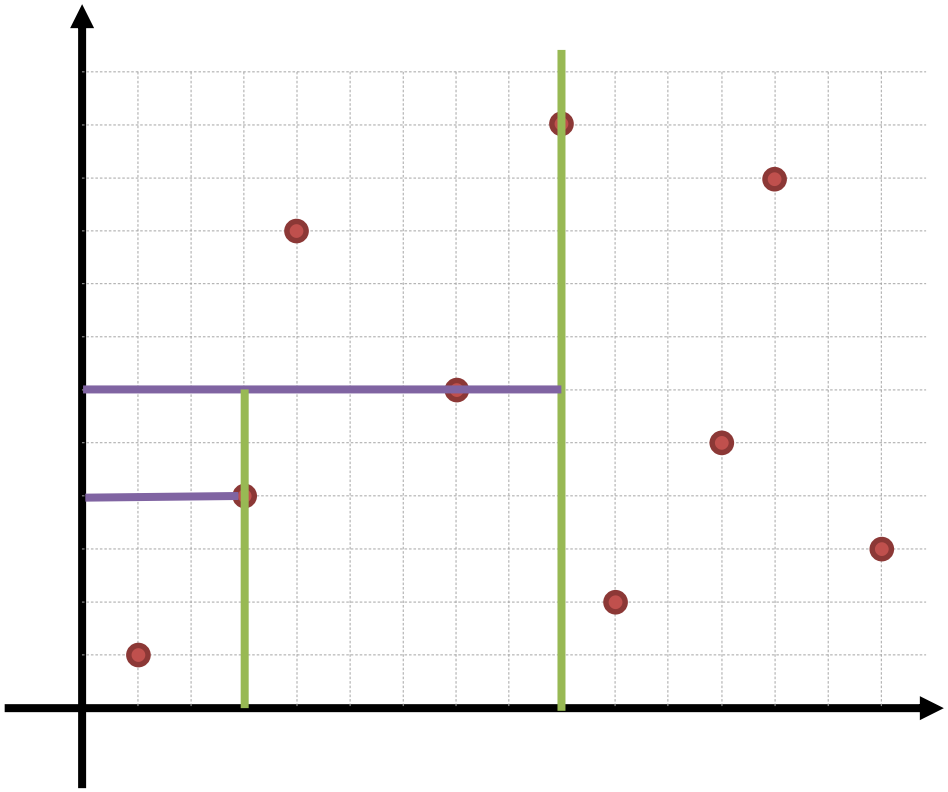
Kd-tree creation

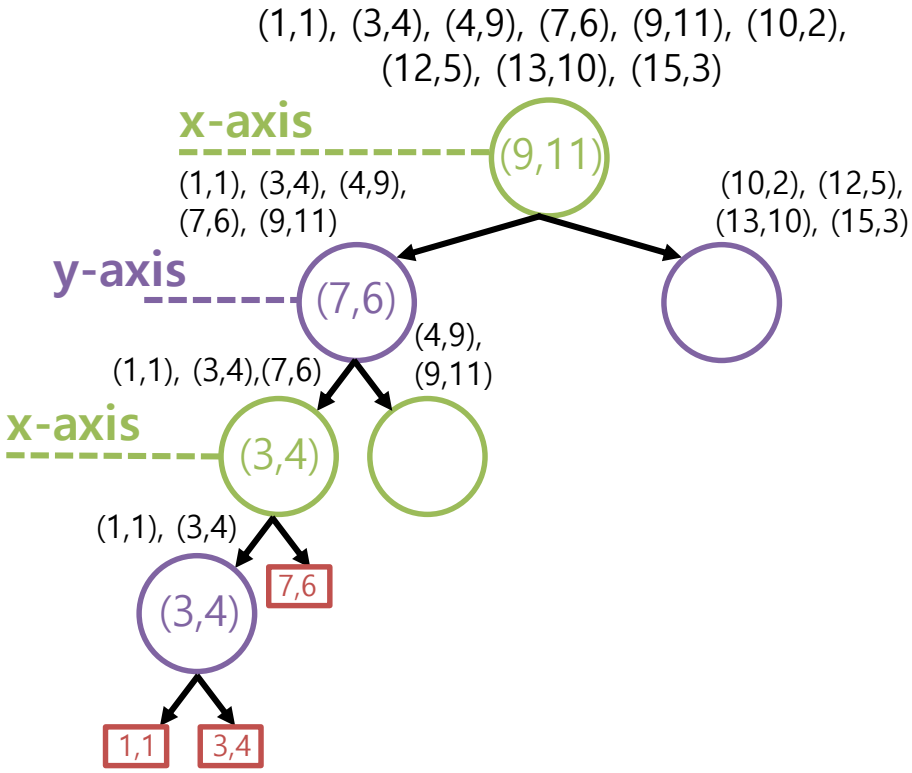




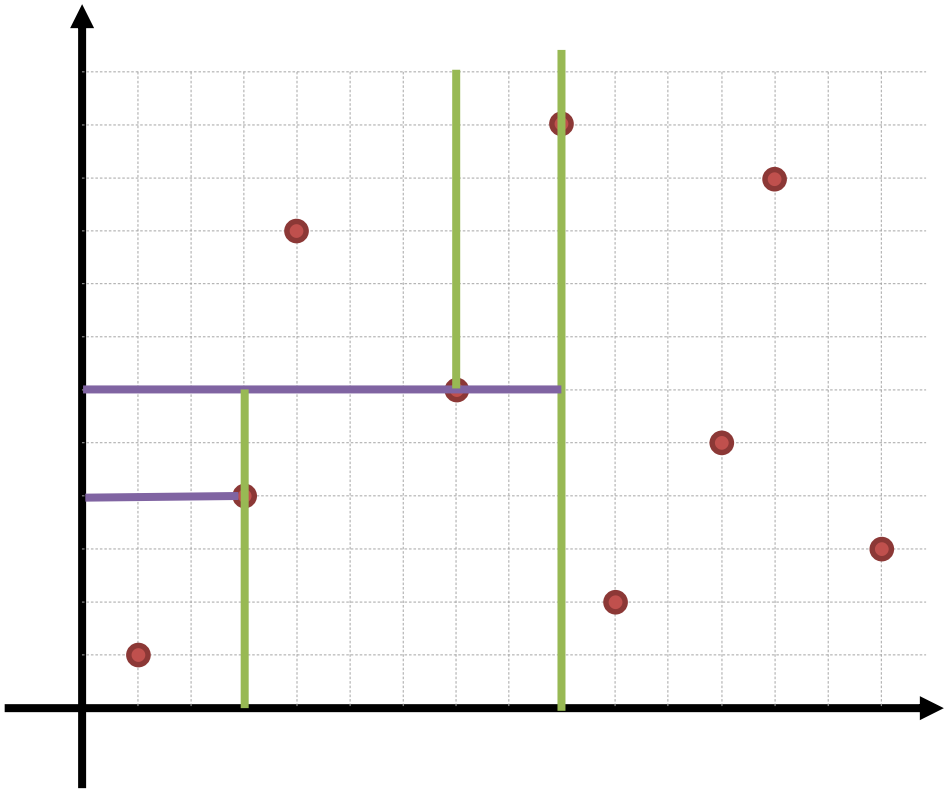
The resulting children are split along the x-axis

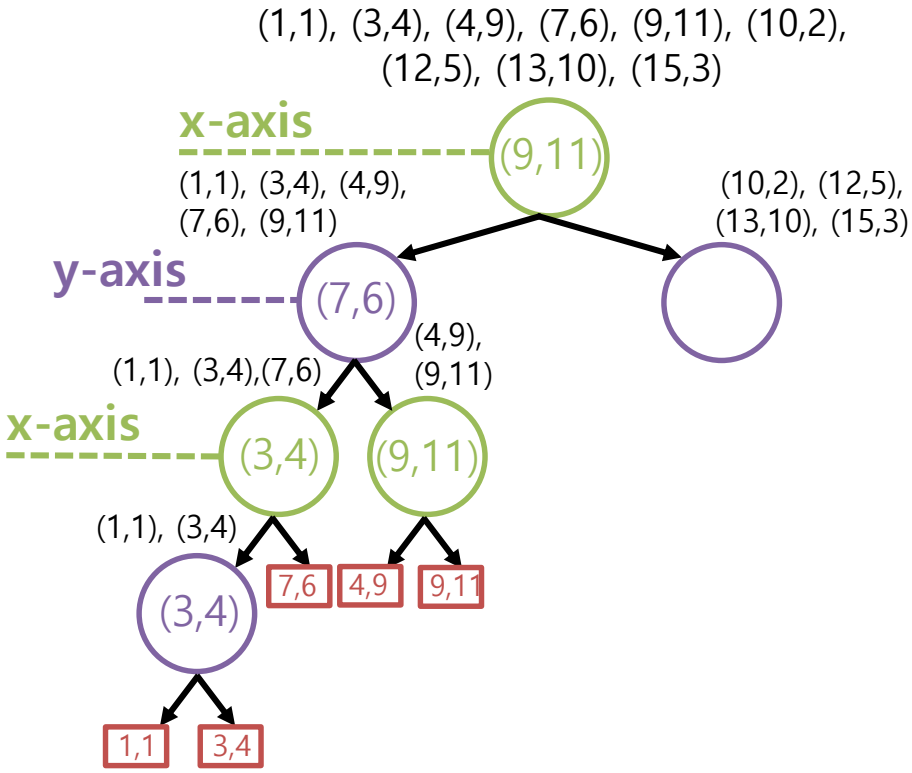
Kd-tree creation



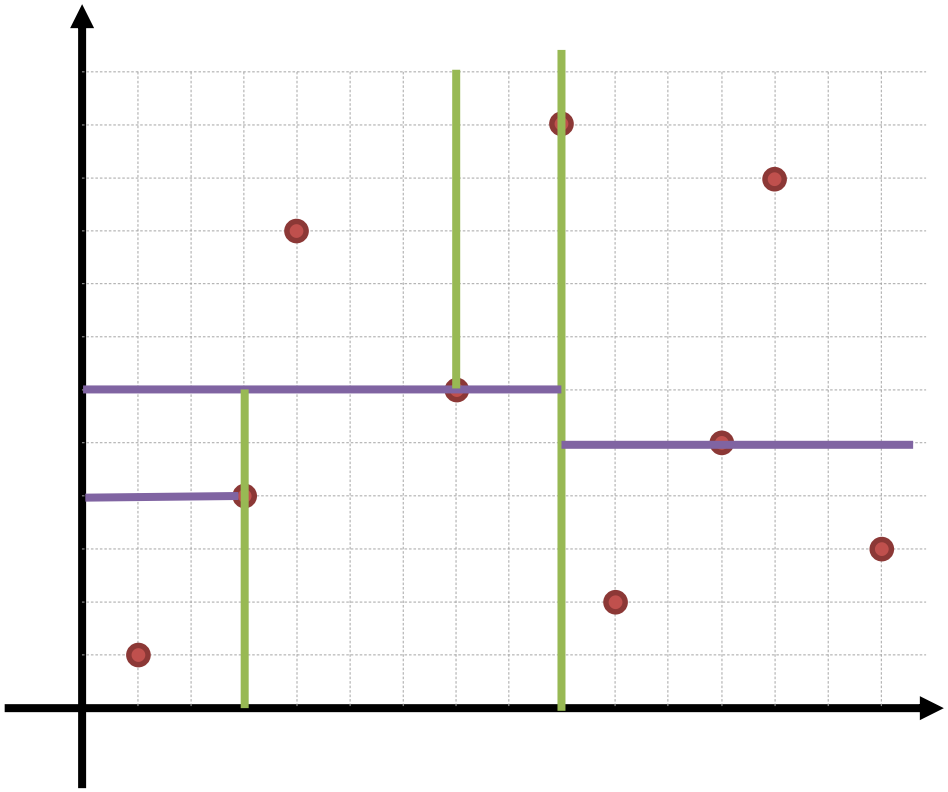


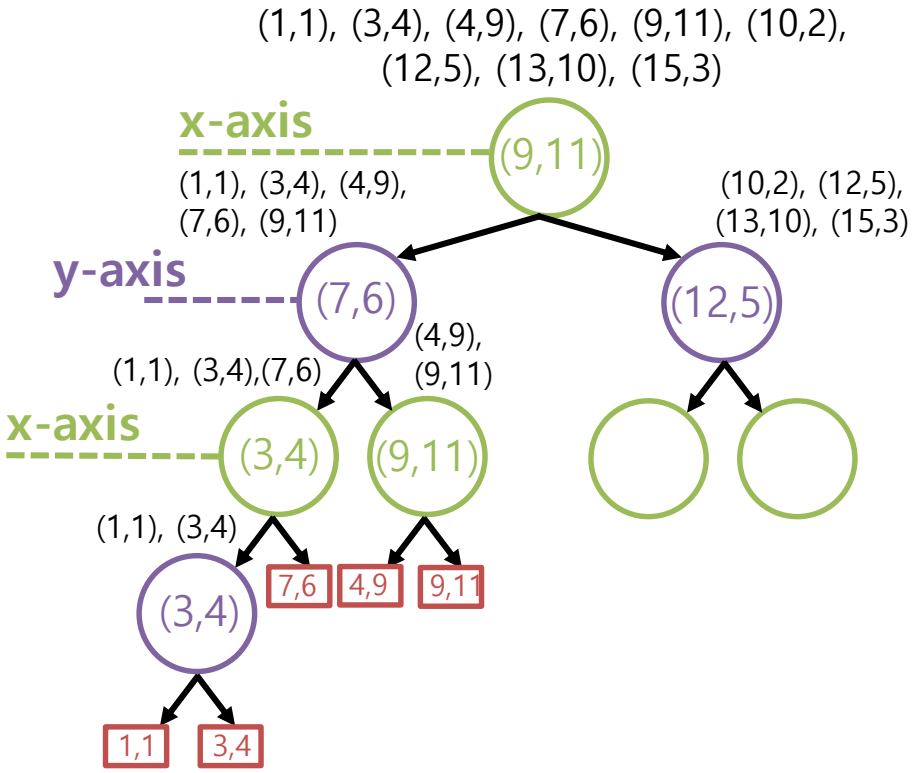
Kd-tree creation



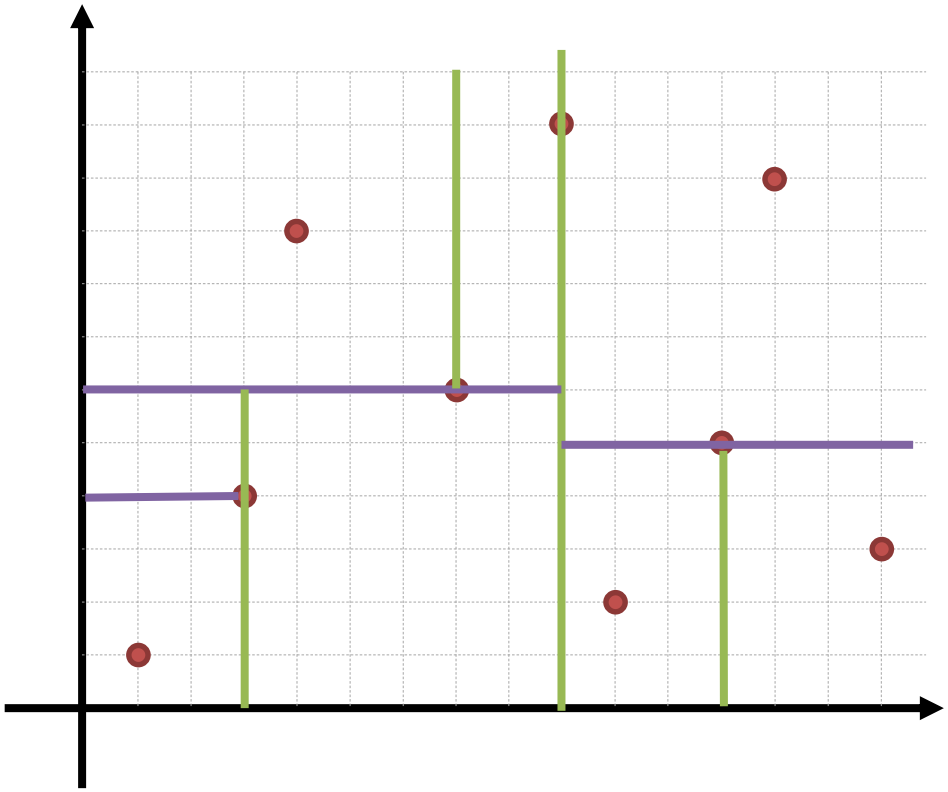


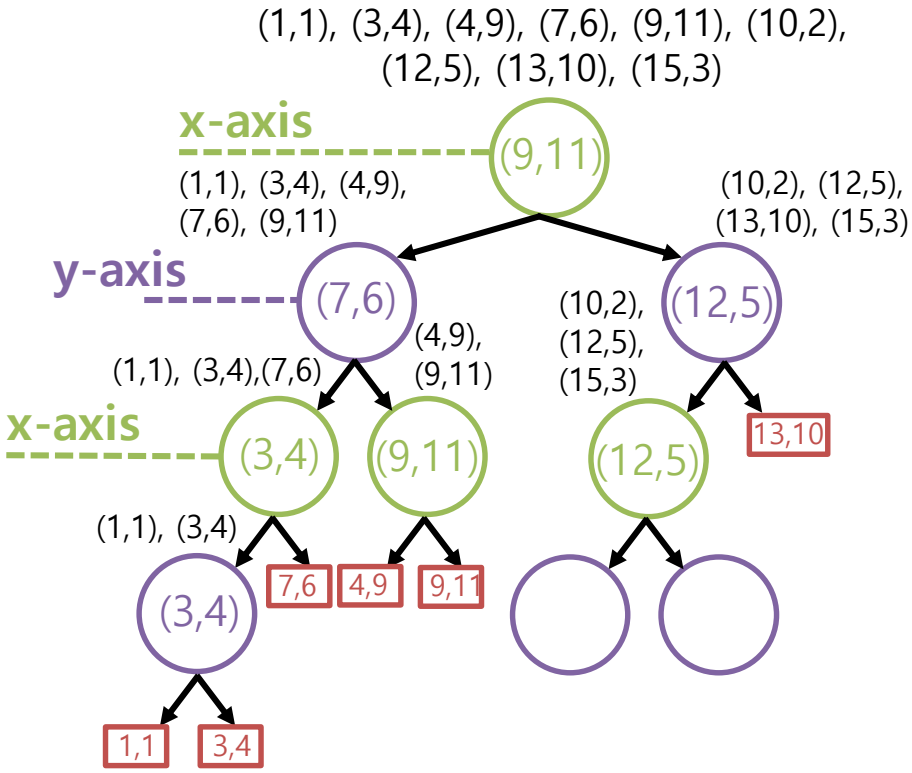
Kd-tree creation



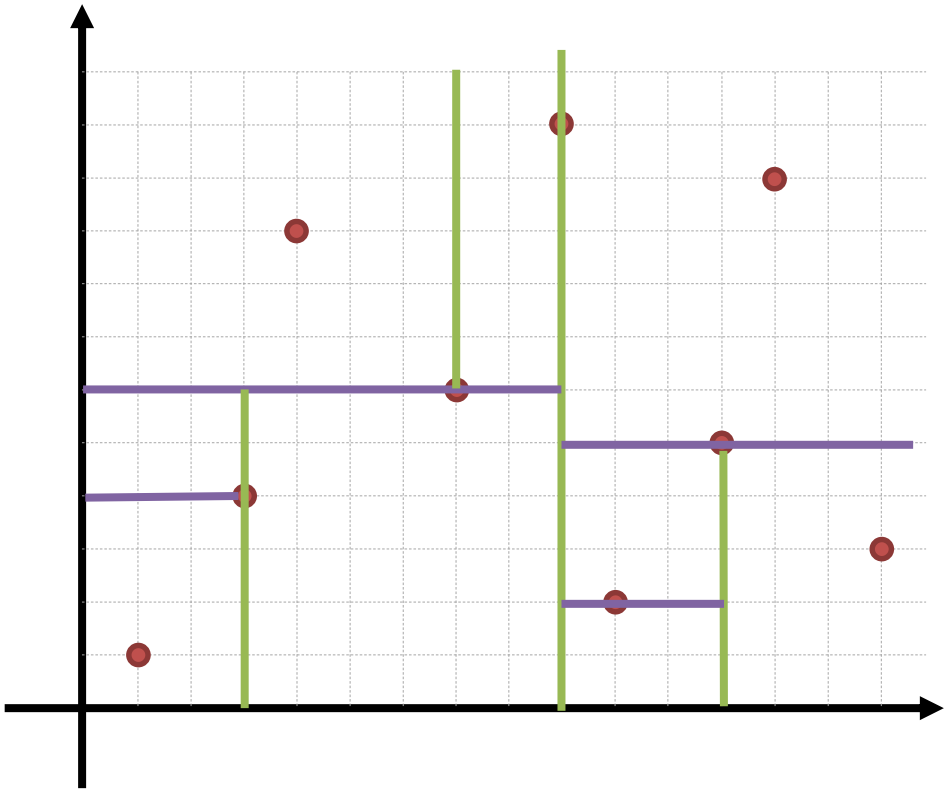


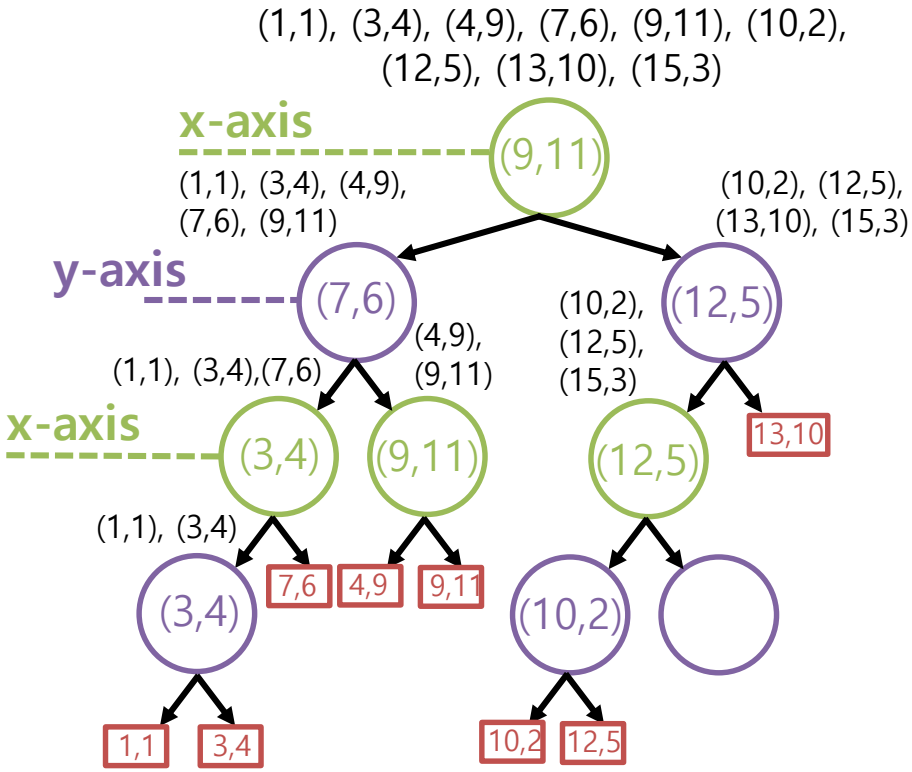
Kd-tree creation



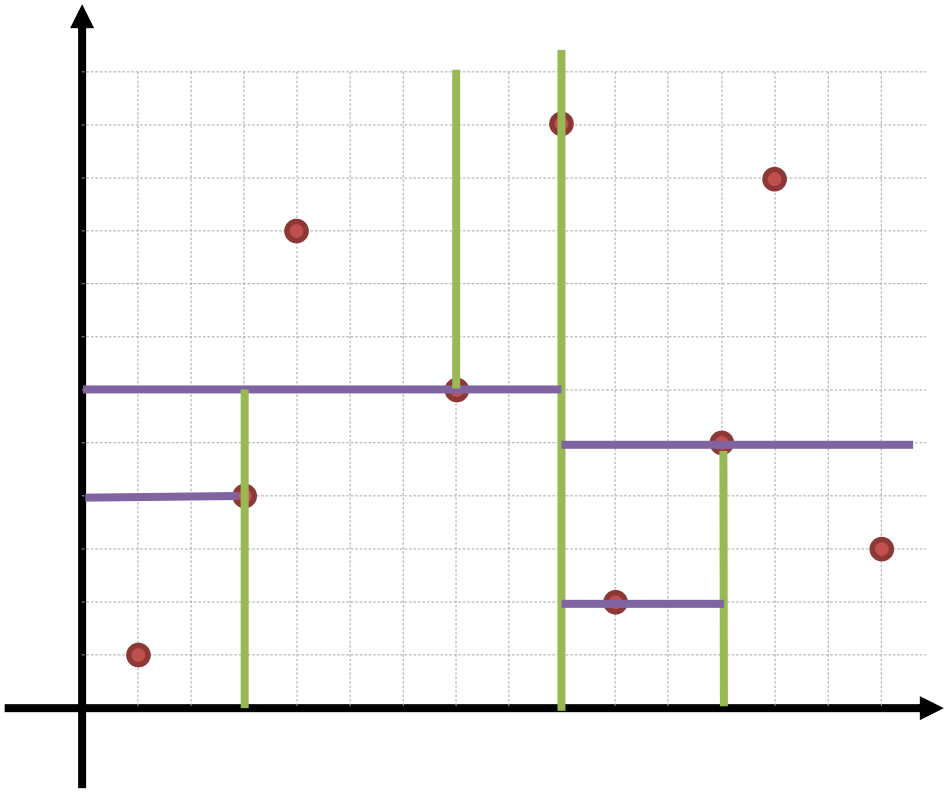


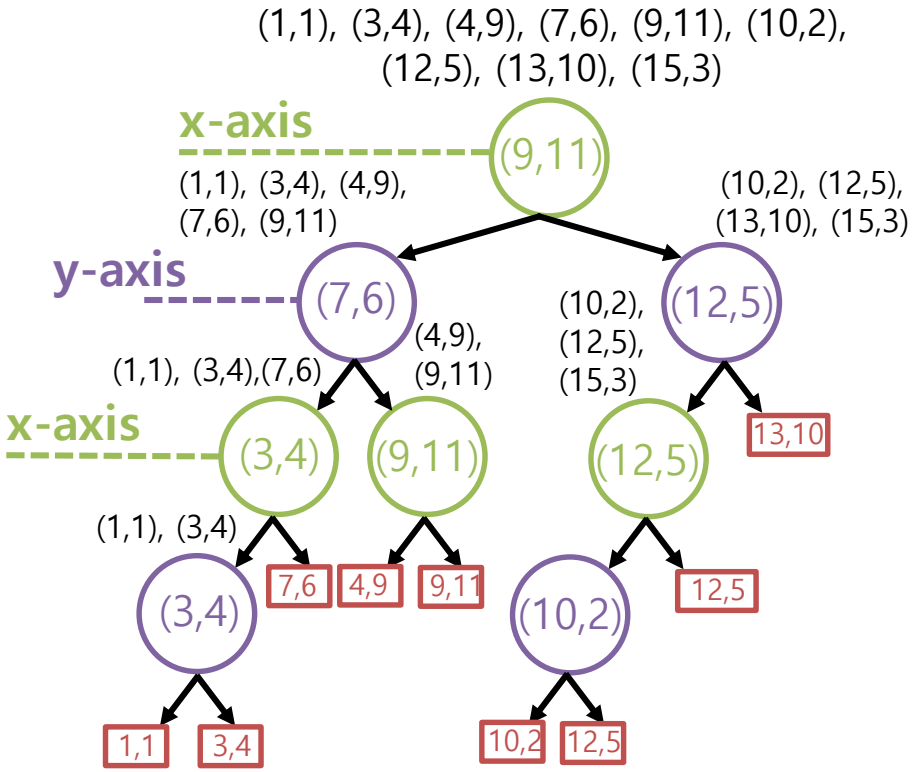
Kd-tree creation





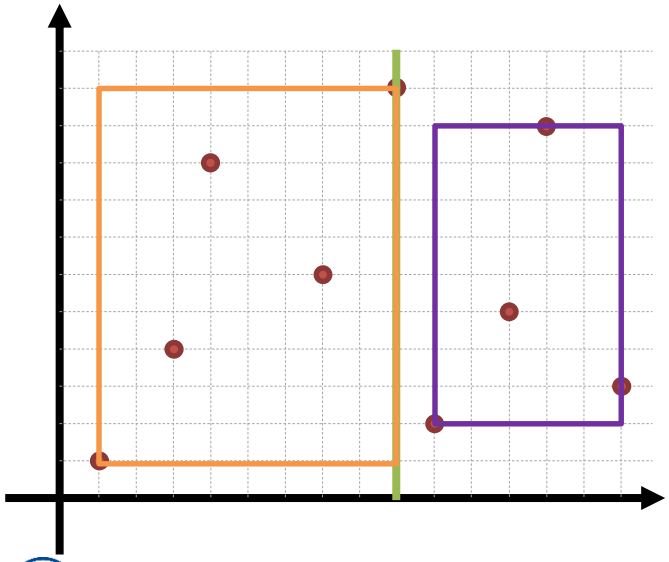
Kd-tree creation

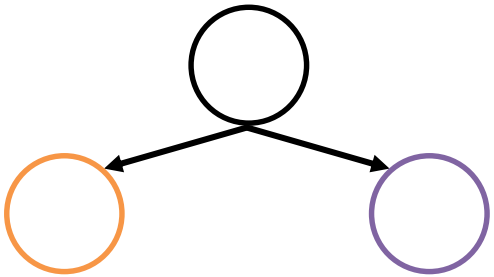


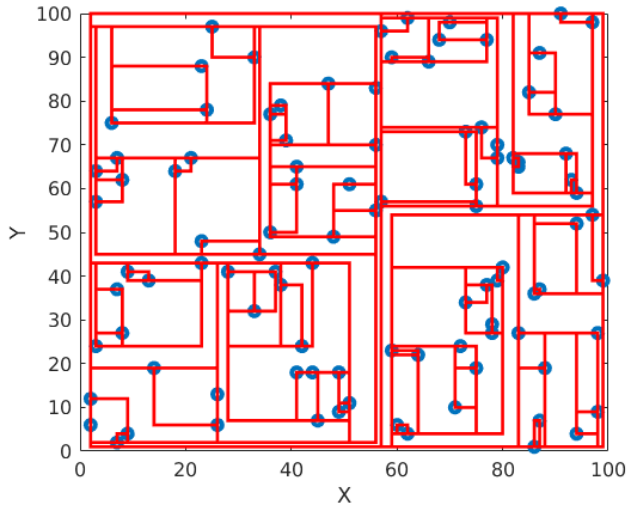


Kd-tree creation

- To perform NN search in the tree a certain number of information are needed
- For each node, we store:
  - The split dimension
  - The split value
  - The tightest bounding box (essential for fast pruning)





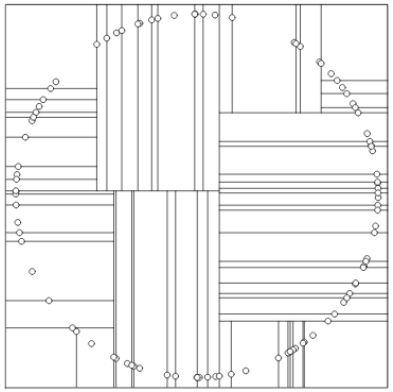


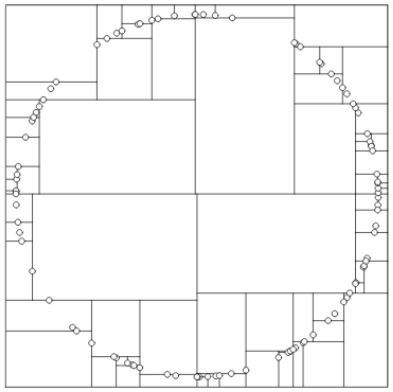
Kd-tree creation

- When to stop splitting?
  - The leaf can contain more than one points
  - The splitting can stop when reaching a minimum number of points
  - It can also be stop when the points are close to each others (minimum BB size)
- How to split?
  - Using the median value is an effective manner to split but other strategies are possible. For instance, the center of the splitting dimension can be used.

ANDREW, W. M. An Introductory Tutorial on KD-trees [R]. *Technical Report*, 1991, 209.

Median splitting

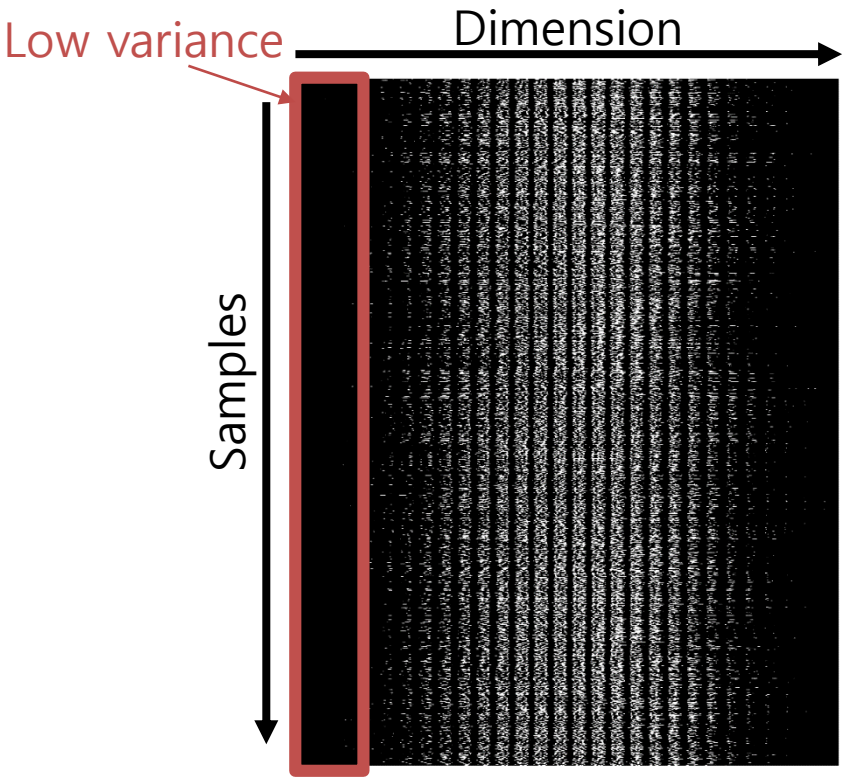




Center splitting

Kd-tree creation

- On what dimension do we split?
  - Splitting each dimension one after an other is not an effective solution:
    - Start from the widest to narrowest dimension
    - Sort the dimensions per variance



1000 samples from MNIST

Kd-tree creation

- Pseudo code (recursive):

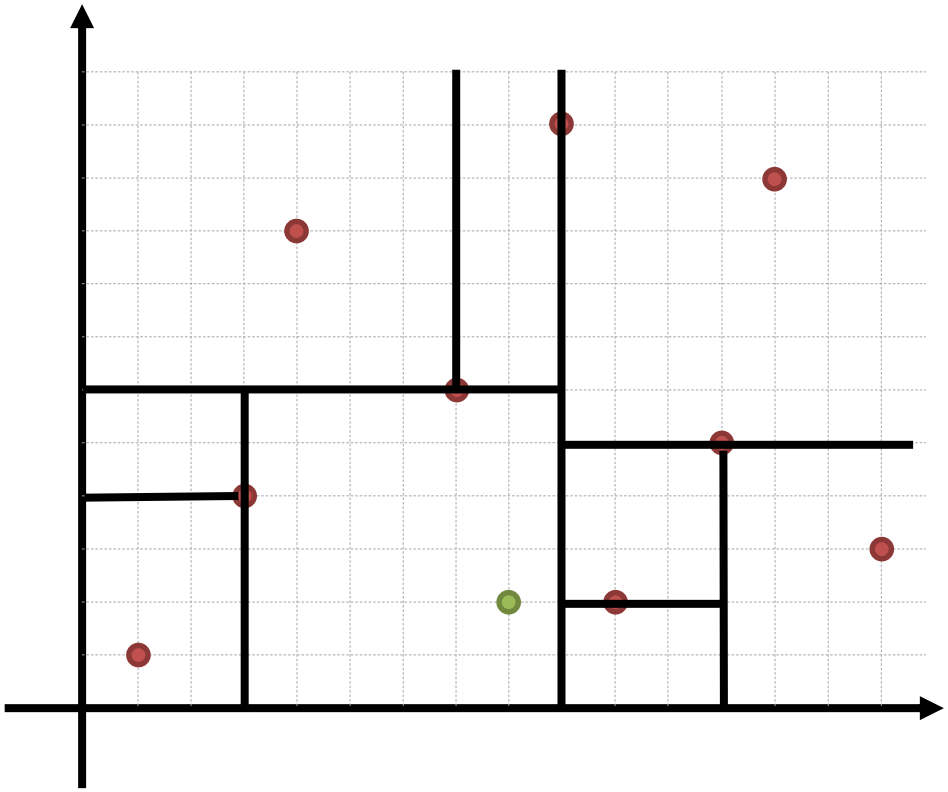
```
function kdtree (list of points pointList, int depth)
{
    // Select axis based on depth so that axis cycles through all valid values
    var int axis := depth mod k;

    // Sort point list and choose median as pivot element
    select median by axis from pointList;

    // Create node and construct subtree
    node.location := median;
    node.leftChild := kdtree(points in pointList before median, depth+1);
    node.rightChild := kdtree(points in pointList after median, depth+1);
    return node;
}
```

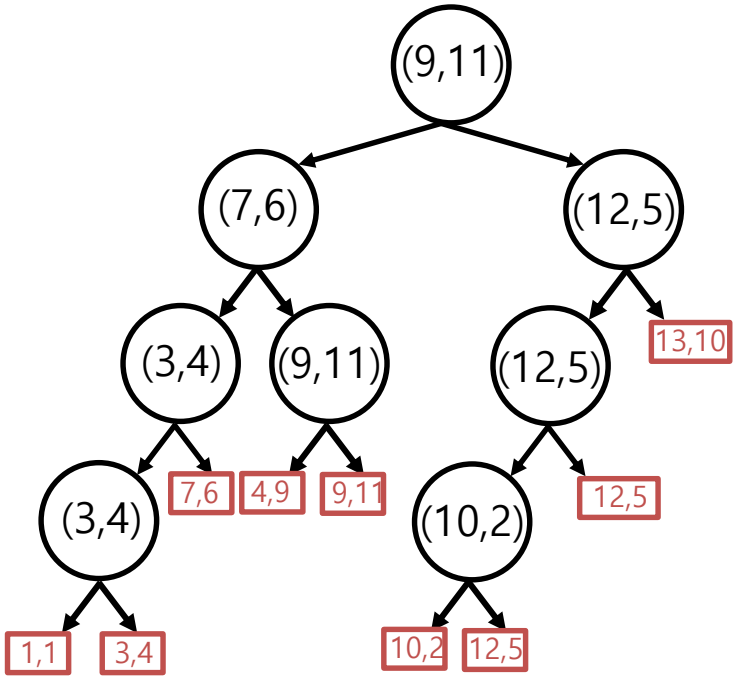
- The operation of insertion and deletion will unbalance the tree and rotation operations will be needed!

Wikipedia: https://en.wikipedia.org/wiki/K-d_tree

NN search

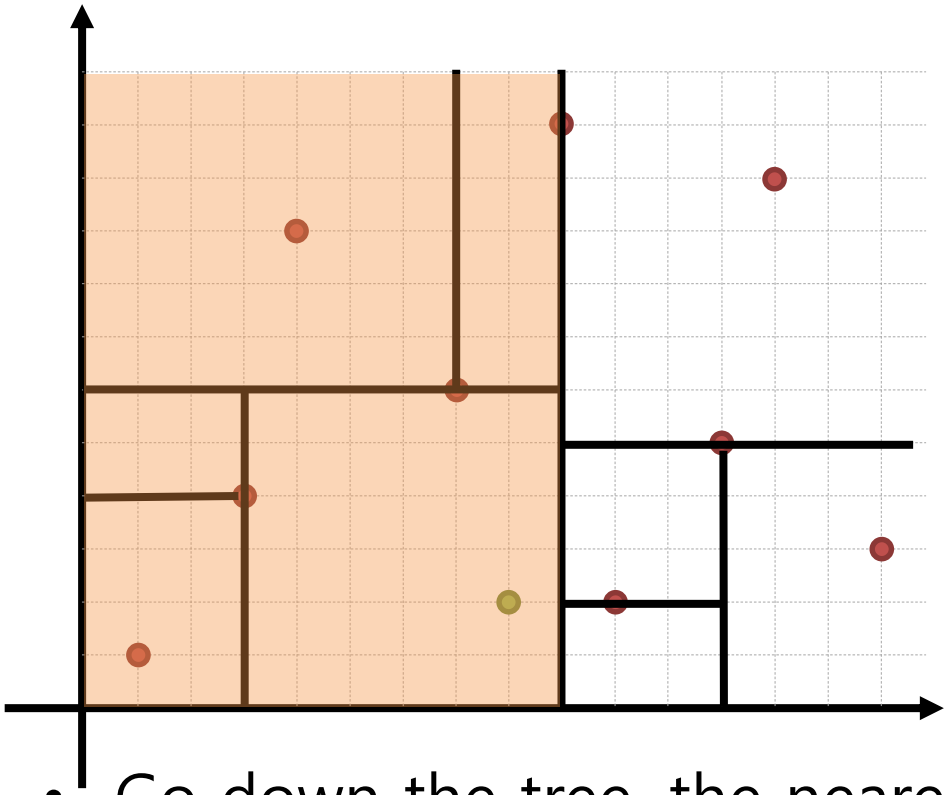
- Query point: (8,2)

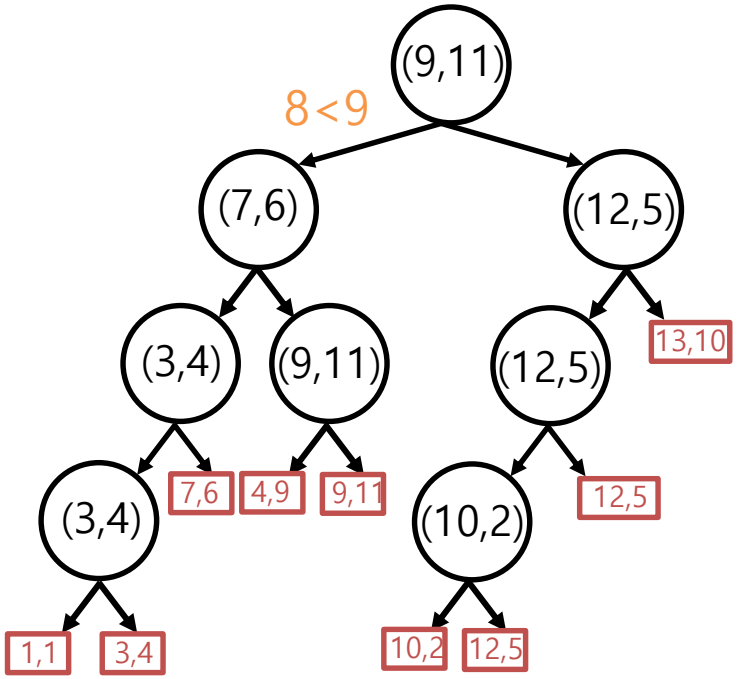




NN search

- Query point: (8,2)

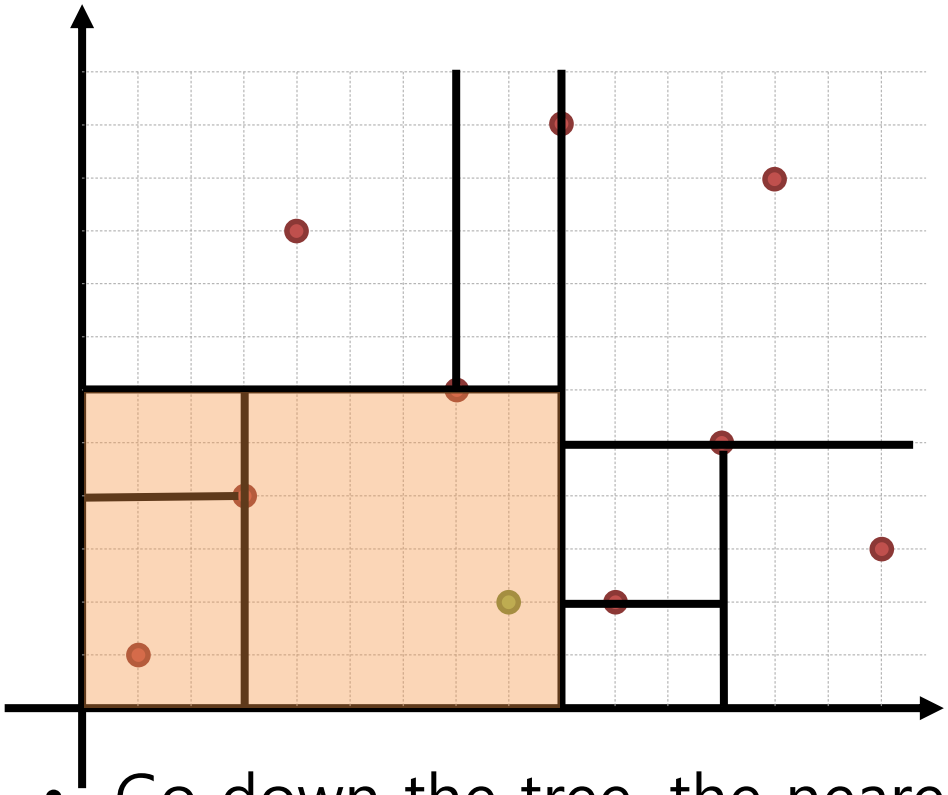


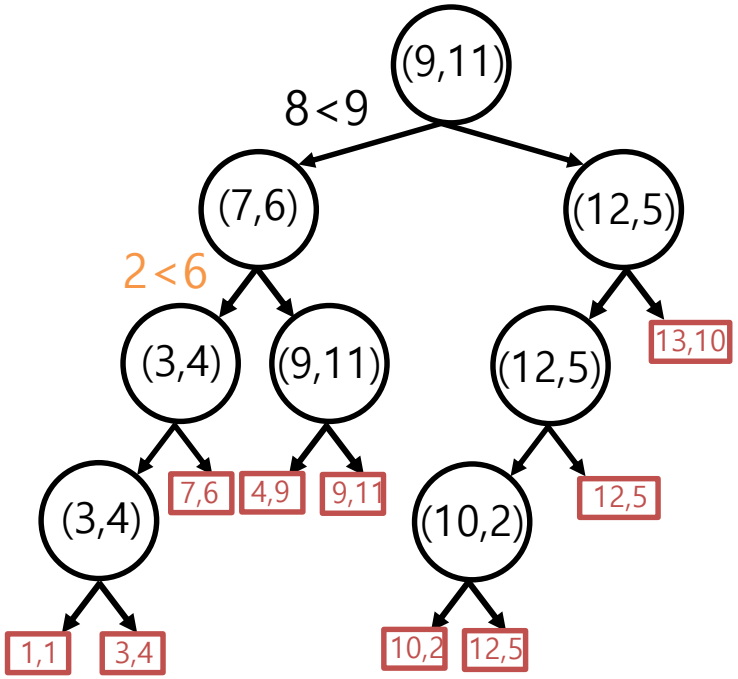


- Go down the tree, the nearest neighbor might be in the in the leaf it belongs to!

NN search

- Query point: (8,2)

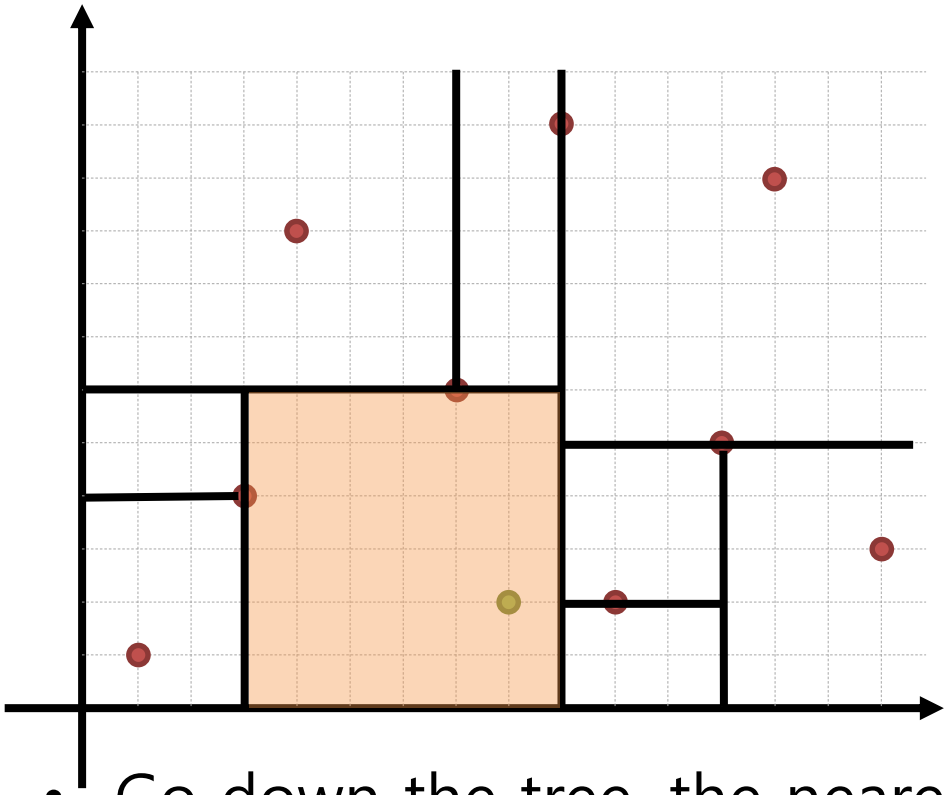


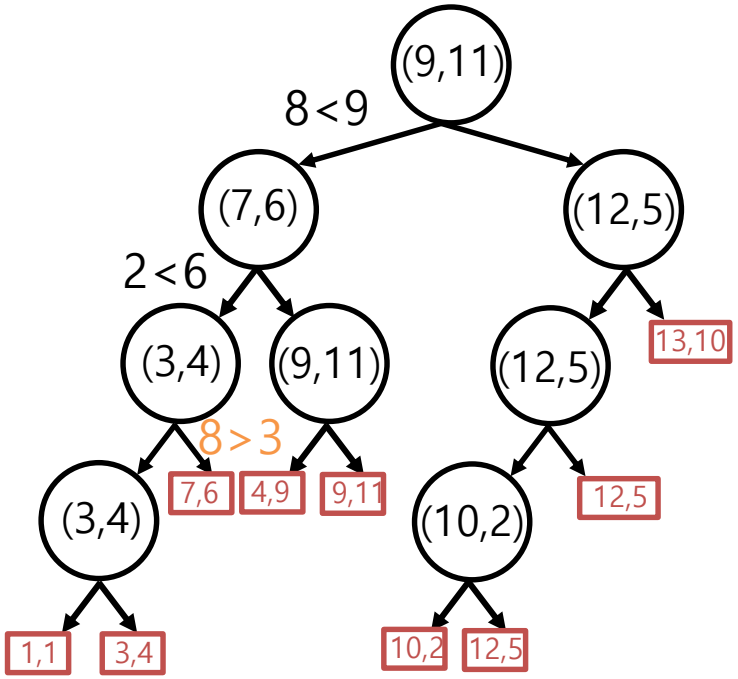


- Go down the tree, the nearest neighbor might be in the in the leaf it belongs to!

NN search

- Query point: (8,2)

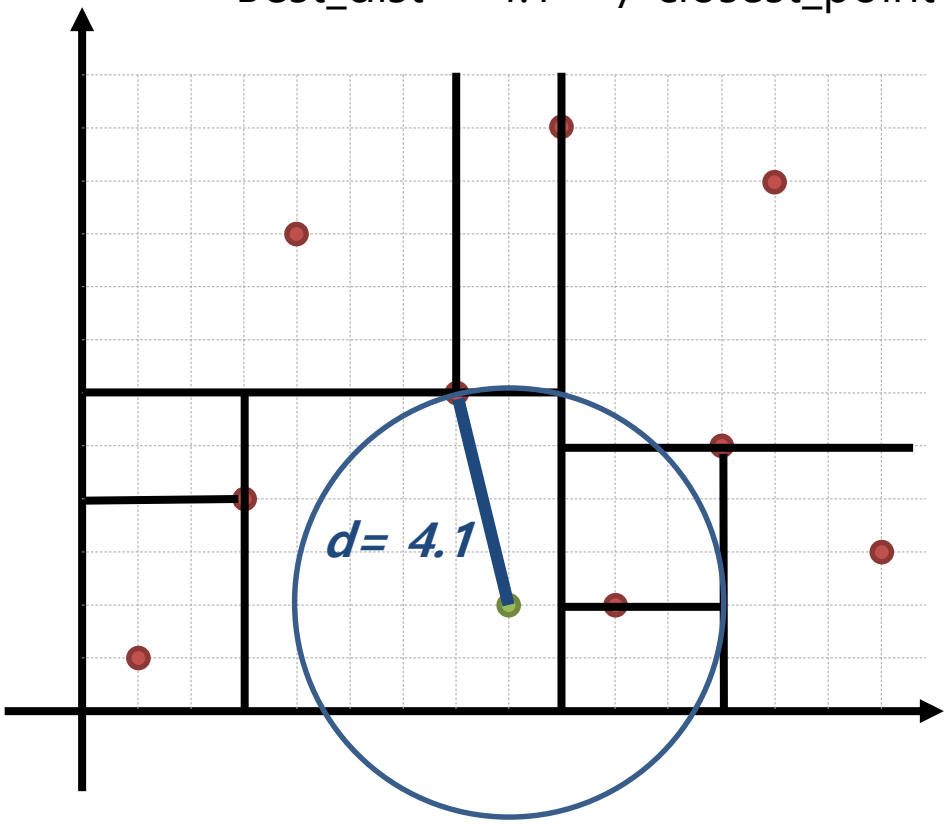


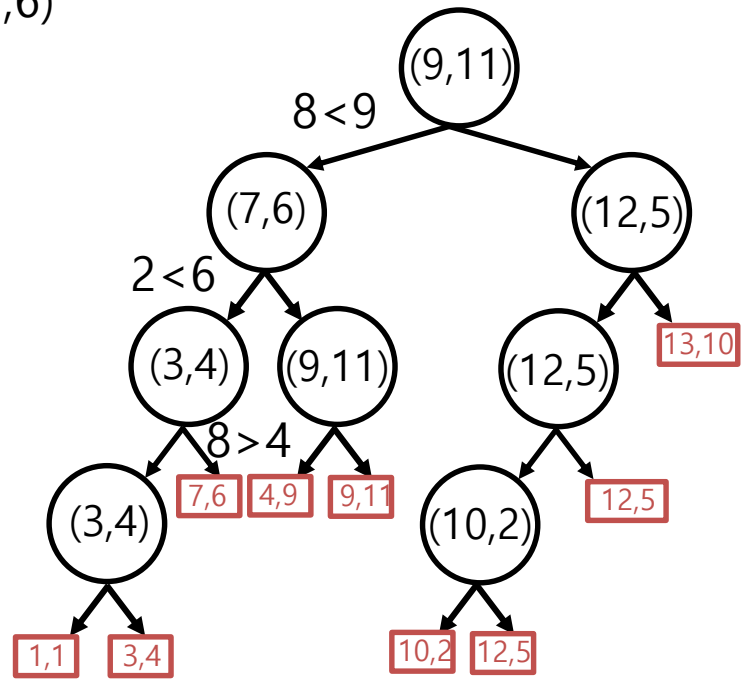


- Go down the tree, the nearest neighbor might be in the in the leaf it belongs to!

NN search

- Query point: (8,2)
  Best_dist = 4.1 / closest_point = (7,6)

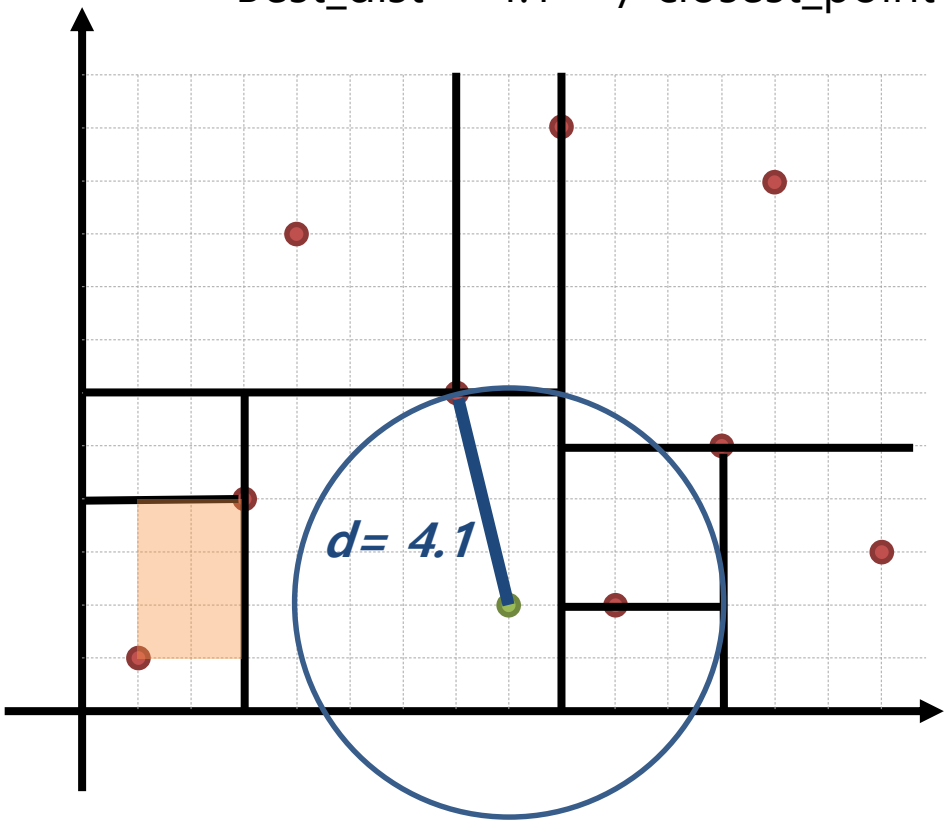


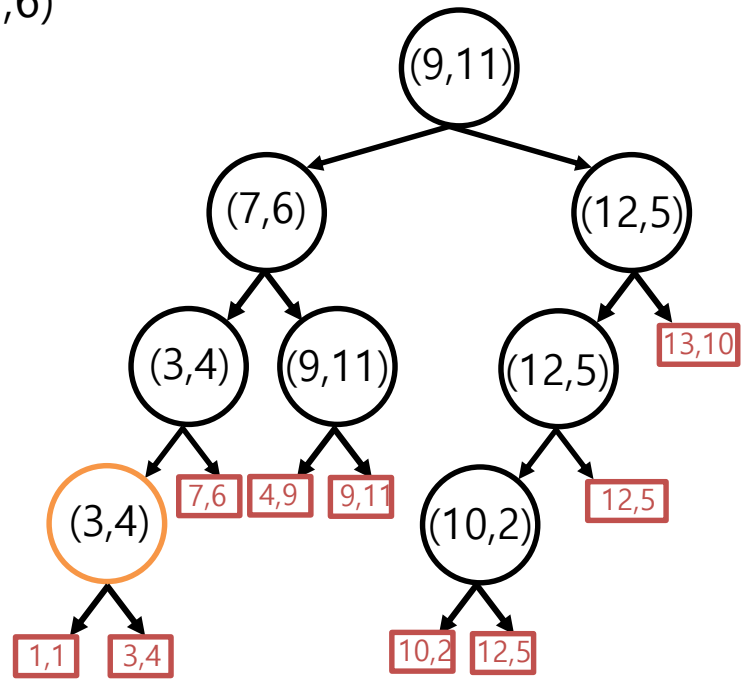


- Compute the distance with the point(s) in this leaf and keep this distance as the best distance so far

NN search

- Query point: (8,2)
  Best_dist = 4.1 / closest_point = (7,6)

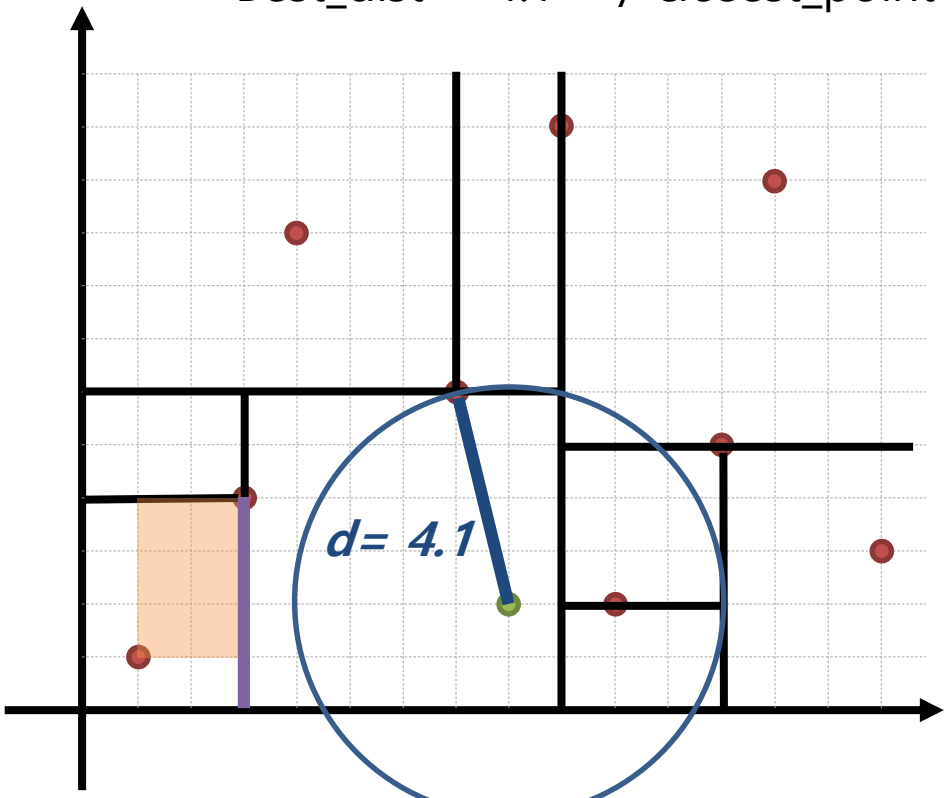


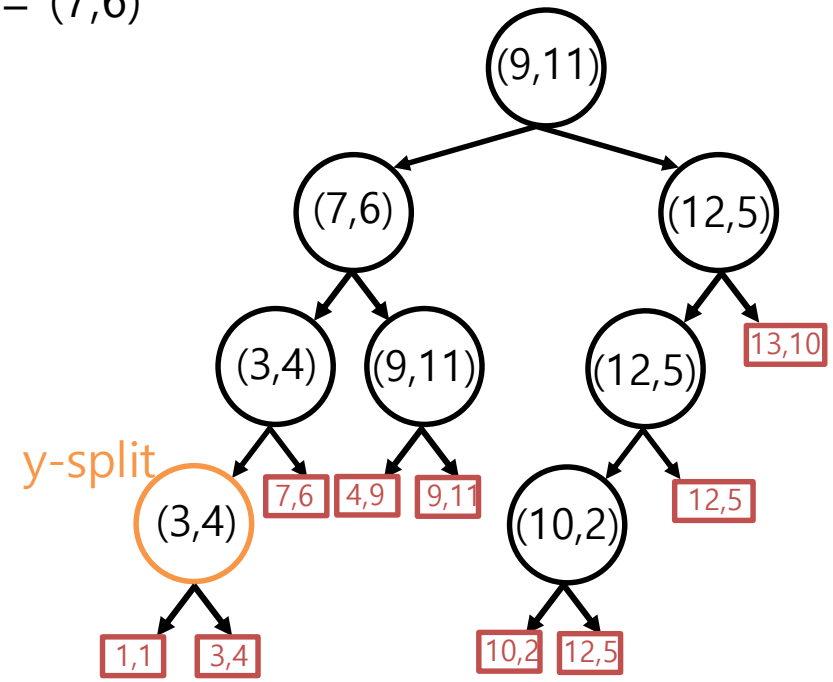


- Check if other nodes are potentially containing a closer point: Use our thigh BB

NN search

- Query point: (8,2)
  Best_dist = 4.1 / closest_point = (7,6)

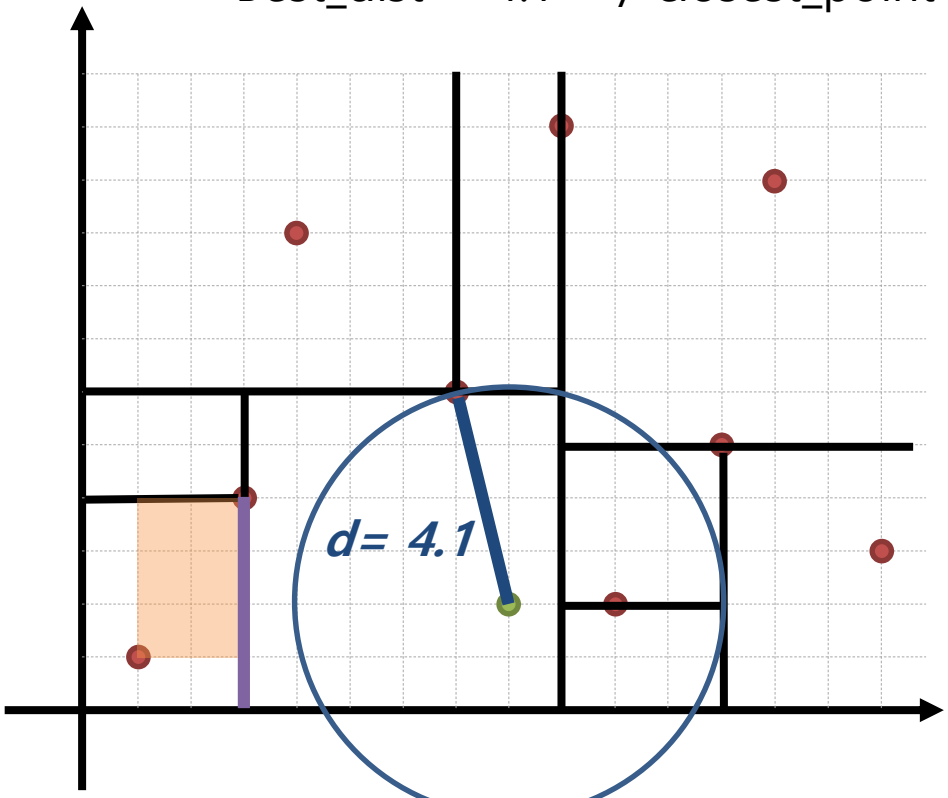


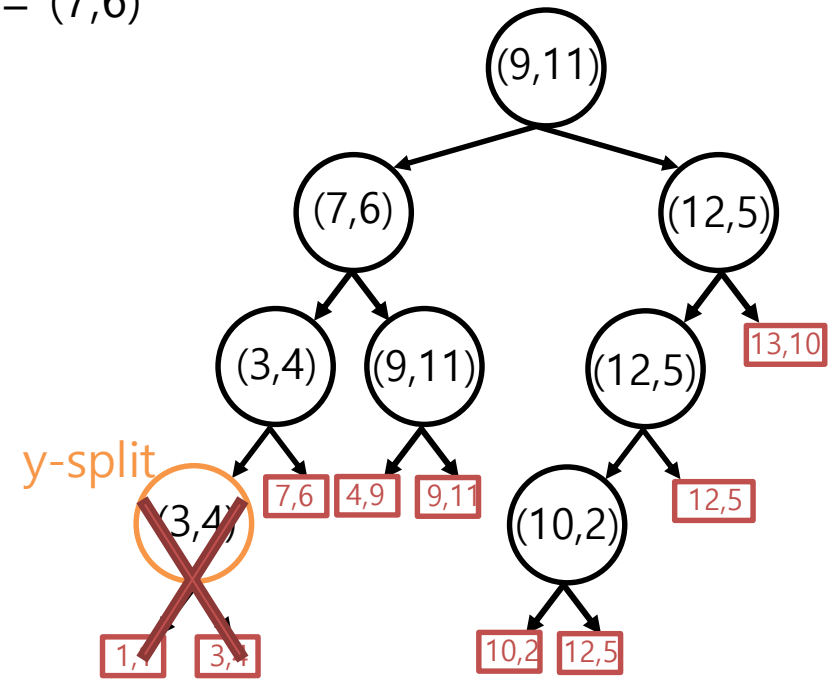


- $if\left(x_{query} - Best_{dist}\right) \leq BB_{ymax}$ then the node might contain a closer point

NN search

- Query point: (8,2)
  Best_dist = 4.1 / closest_point = (7,6)

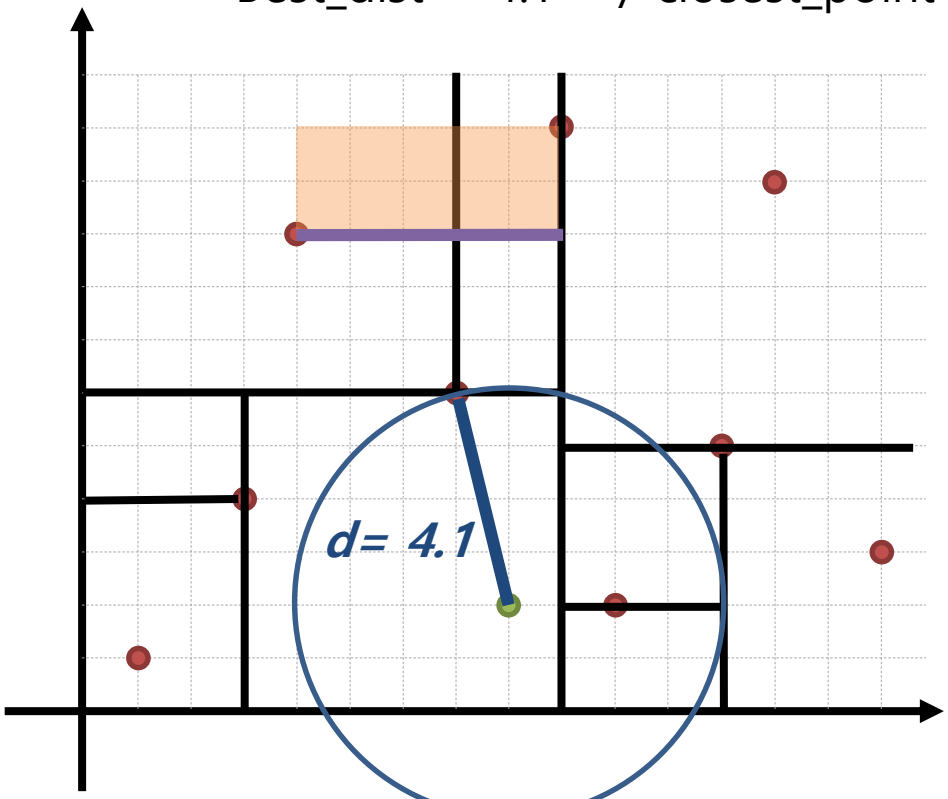


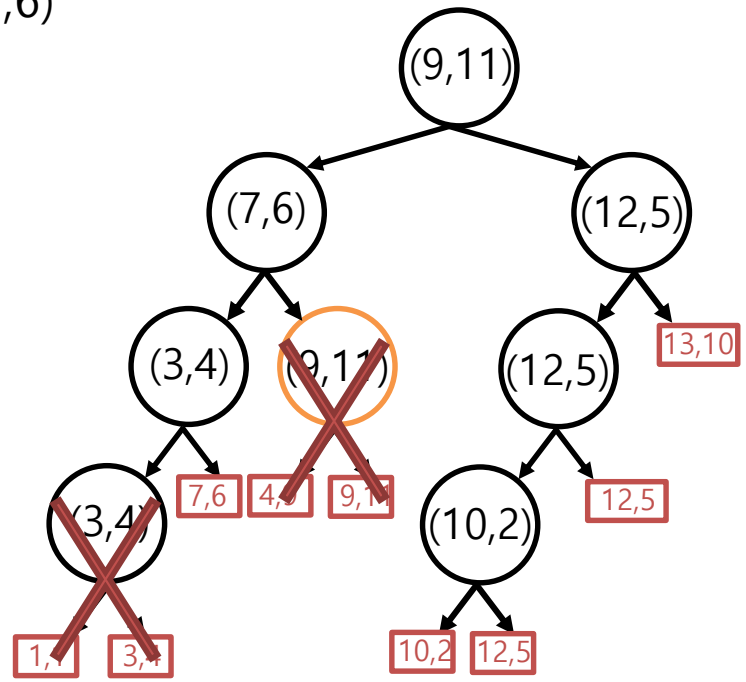


- $if\left(x_{query} - Best_{dist}\right) \leq BB_{xmax} \rightarrow (8 - 4.1) \leq 3$

NN search

- Query point: (8,2)
  Best_dist = 4.1 / closest_point = (7,6)

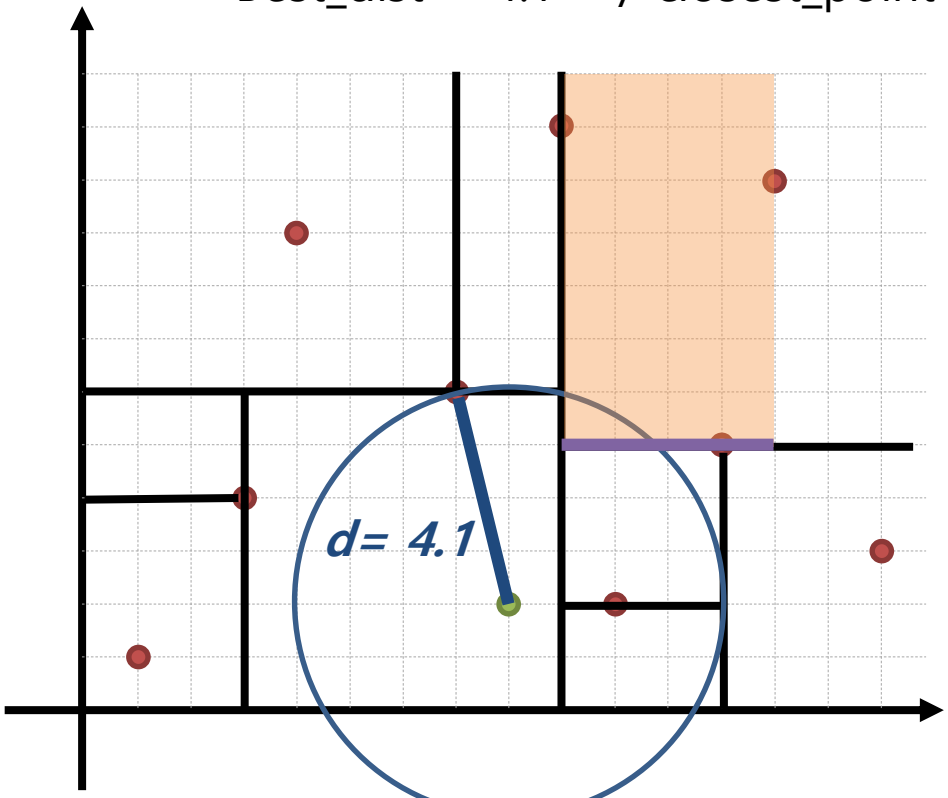


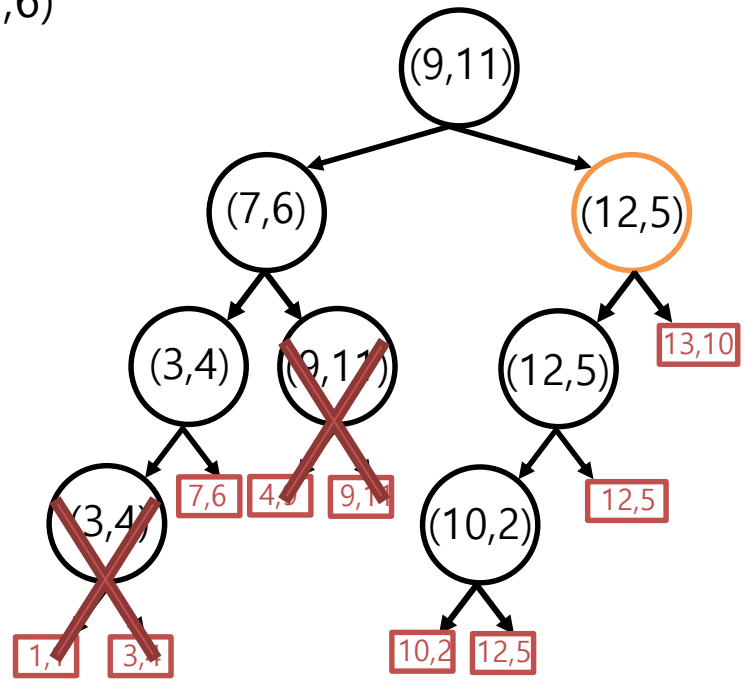


- $if\,(y_{query} + Best_{dist}) \geq BB_{ymin}$ → ~~$(2 + 4.1) \geq 9$~~

NN search

- Query point: (8,2)
  Best_dist = 4.1 / closest_point = (7,6)

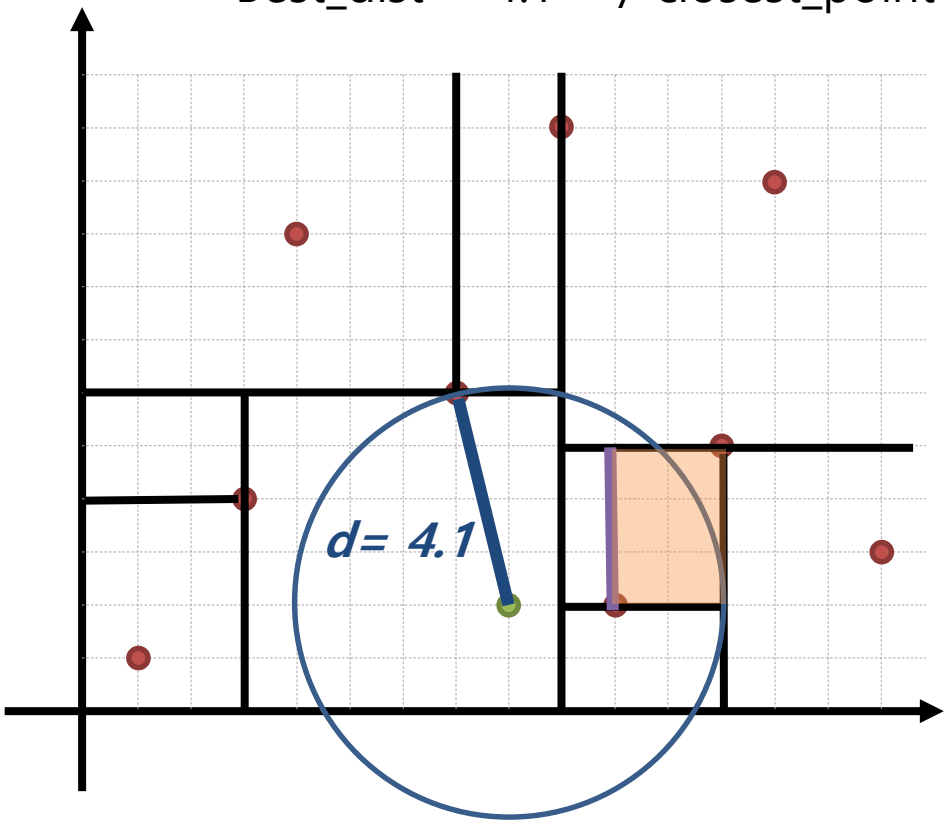


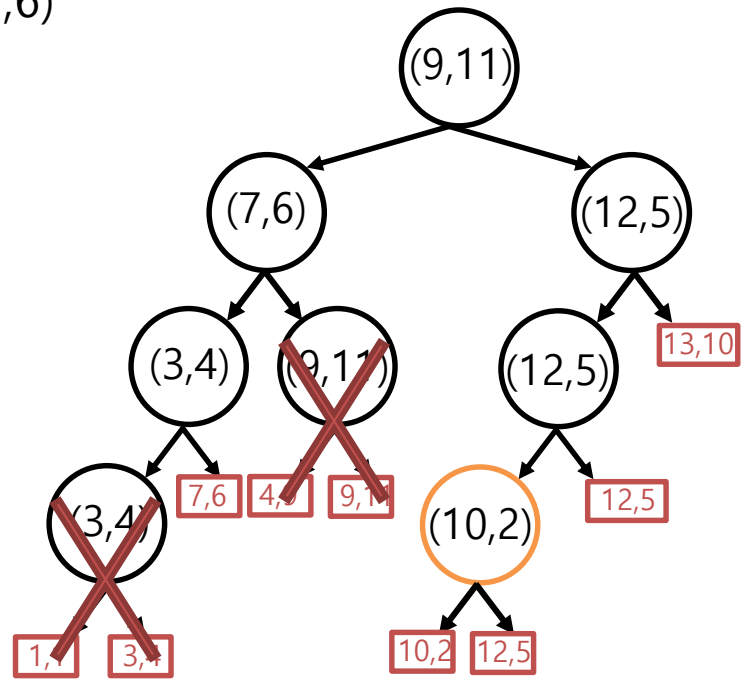


- $if\left(y_{query} + Best_{dist}\right) \geq BB_{ymin}$ → $(2 + 4.1) \geq 5$

NN search

- Query point: (8,2)
  Best_dist = 4.1 / closest_point = (7,6)

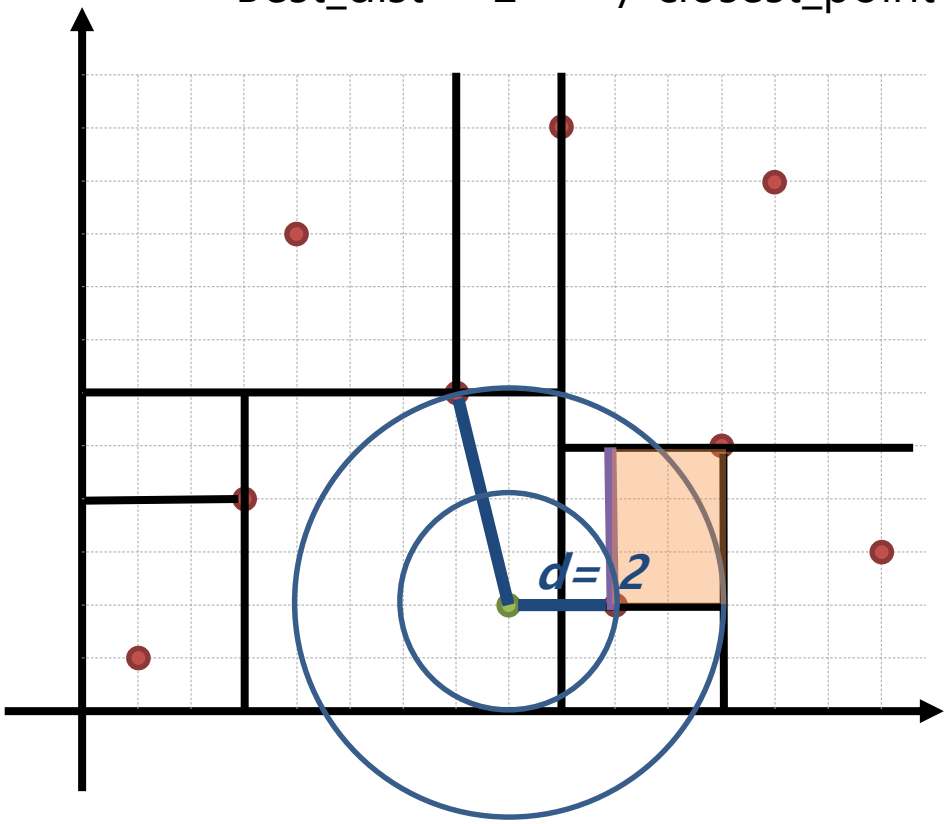


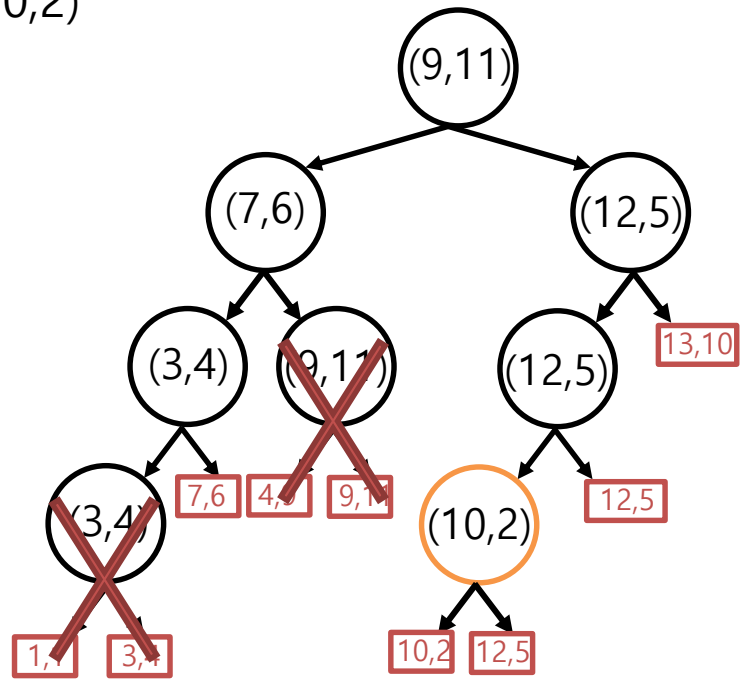


- The leafs of this node might contain a closer point! So we compute the distance to these points

NN search

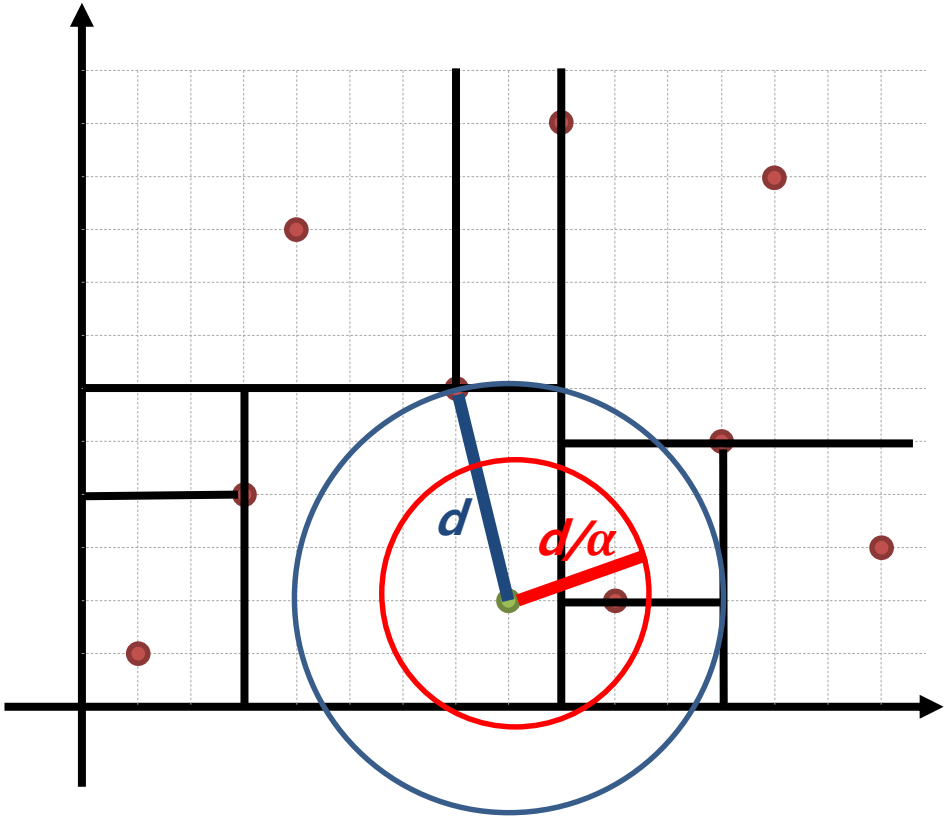
- Query point: (8,2)
  Best_dist = 2 / closest_point = (10,2)





- The leaves of this node might contain a closer point! So we compute the distance to these points

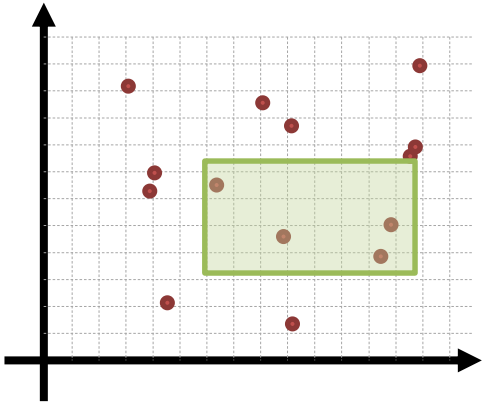
Approximate Nearest neighbor



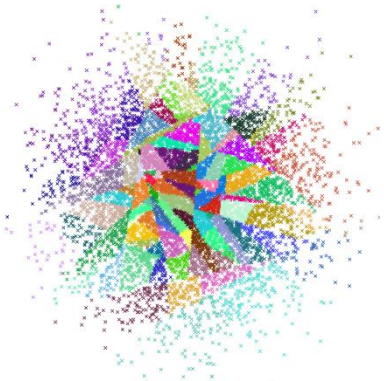
- The NN search can be drastically speed-up by pruning using d/α (where $\alpha > 1$). This approximation will not guarantee an optimal solution but will results in faster search

Conclusion

- Kd-tree is very versatile and can also be used for range search problem!



- Kd-tree is not efficient for large dimensions data:
  - Many alternative exist [1,2,3,4]



- Kd-tree efficiency depends on its configuration:
  - Distance metric
  - Splitting techniques
  - Splitting stop