Selection & Ordering Algorithms

CE-1103 Algorithms and Data Structures

Disclaimer / Descargo de Responsabilidad

Esta presentación corresponde a una guía usada por el profesor durante las clases. La misma ha sido modificada para ser utilizado en el modelo de cursos asistidos por tecnología. No es una versión final, por lo que la misma podría requerir todavía hacer algunos ajustes. Para aspectos de evaluación esta presentación es solo una guía, por lo que el estudiante debe profundizar con el material de lectura asignado y lo discutido en clases para aspectos de evaluación.

This presentation corresponds to a guide material used by the professor during classes. It has been modified to be used in the model of technology-assisted courses. It is not a final version, so it may still require some adjustments. For evaluation aspects, this presentation is only a guide, so the student should delve with the assigned reading material and what has been discussed in class.

Before we begin...

➔ What is an **algorithm**?

➔ How can we represent an algorithm?

➔ How can we measure the **efficiency** of an algorithm?

What is an Algorithm?

- ➔ An algorithm is “a **finite** set of **precise** instructions for performing a computation or for solving a problem

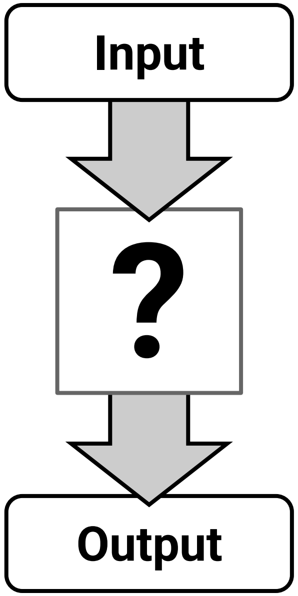
- ➔ An algorithm is “a well-ordered collection of **unambiguous** and **effectively** computable operations that when executed produces a result and halts in a **finite** amount of time”

Characteristics of an Algorithm

➔ **Finiteness**: terminate after a finite number of steps

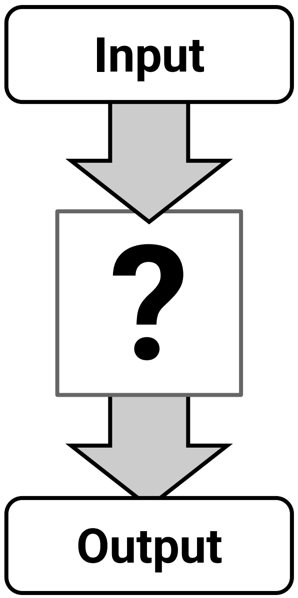
➔ **Definiteness**: each step is precisely defined. There is no ambiguity.

➔ Has a well defined **input**



Characteristics of an Algorithm

- ➔ Has an **output**
- ➔ **Effectiveness**: accomplish its purpose
- ➔ **Deterministic**/Uniqueness
  - ◆ Results of each step are uniquely defined by the input and the result of the preceding steps.



What is algorithm analysis?

→ Algorithm analysis is a method for **estimating the resource consumption** of an algorithm

Why analyze an algorithm?

➔ Discover the characteristics of an algorithm in order to **evaluate its suitability** for various applications.

➔ **Compare** an algorithm with other for the same application/purpose

Why analyze an algorithm?

- ➔ Classify problems and algorithms by difficulty

- ➔ Predict performance, compare algorithms, **tune parameters**

- ➔ Better understand and improve implementations and algorithms

Factors Affecting Run Time

- ➔ Computer System (**CPU**, **Memory**, **Disk**)
- ➔ Compiler
- ➔ Programing Language
- ➔ Operating System

Factors Affecting Run Time

➔ Developer

➔ Size of input

➔ Current applications running on the computer.

Before we continue...

➔ What should we use as a measure of how "good" an algorithm is?

➔ How should we compare two algorithms with each other?

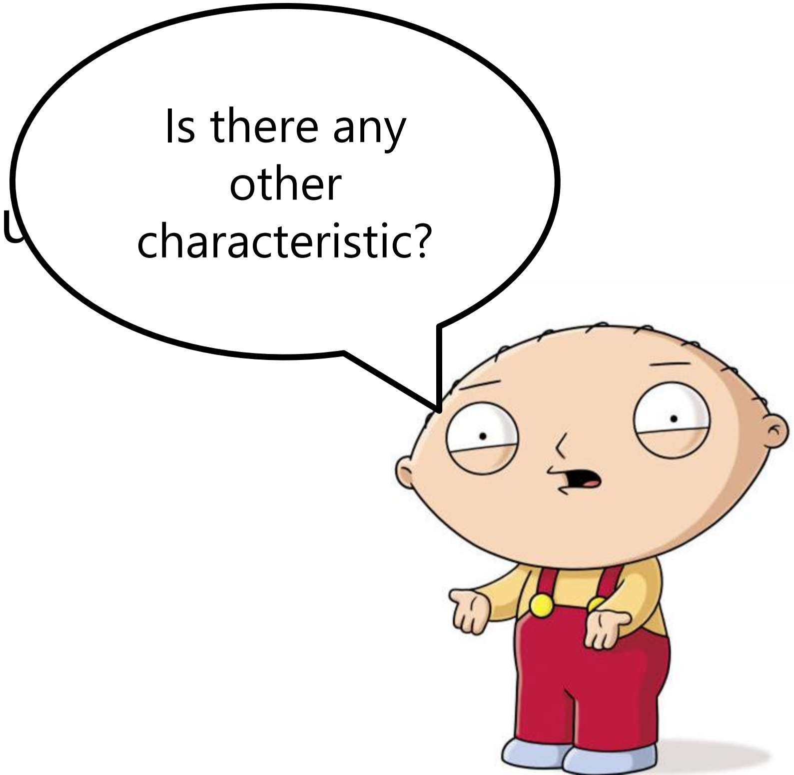
The characteristics of interest

- ➔ Time (CPU)
  - ◆ How long an implementation of a particular algorithm will run on a particular computer
- ➔ Space (Disk/Memory)
  - ◆ How much space it will require

The characteristics of interest

- ➔ Time (CPU)
  - ◆ How long an implementation of a particular algorithm will run on a particular computer

- ➔ Space (Disk/Memory)
  - ◆ How much space it will requ



Other Characteristics

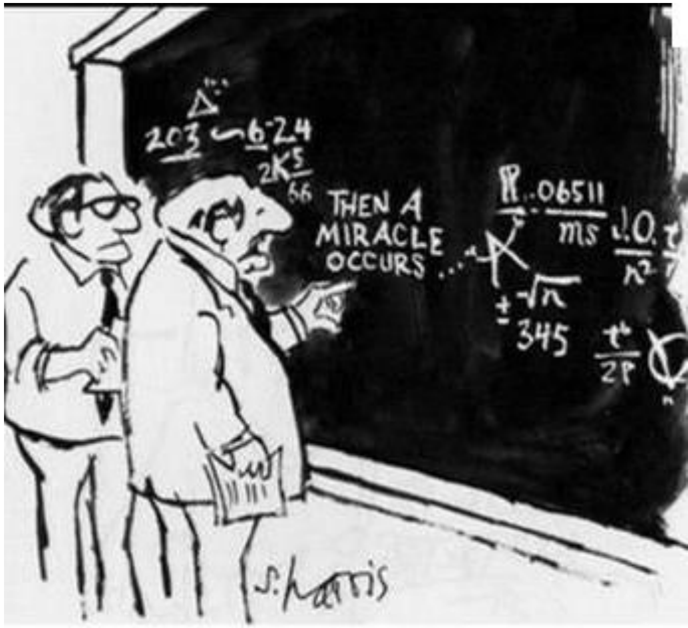
- ➔ Network Bandwidth
- ➔ Disk space
- ➔ Peripheral devices
- ➔ **Any resource** in the computer

Good Measures, bad Measures

➔ Empirical
- Benchmark

➔ Simulational
- Simulated data
- Test cases

➔ Analytical
- Mathematical Model (time & space)



"I think you should be more explicit here in step two."

What is the best **approach?**

Computational Complexity

- ➔ Classifies computational problems according to their difficulty
- ➔ **Time Complexity**
- ➔ Space Complexity

Time Complexity

➔ Classifies the amount of time taken by an algorithm

➔ Represents the algorithm as a function of the size of the input

➔ Focus on dominant operations, the ones which perform more "primitive" operations.

Time Complexity

➔ Time complexity is estimated by **counting elementary instructions**.

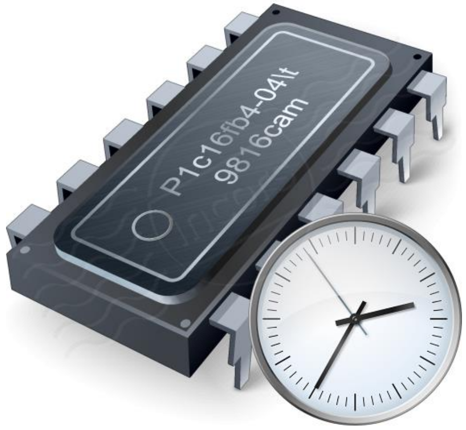
➔ Analyze the growth rate of the function.

What is time?

➔ Wall clock or real time

➔ CPU time

➔ Number of instructions executed



Clock Speed

- ➔ Frequency at which the CPU is running
- ➔ It is measured in hertz or gigahertz
- ➔ Higher frequency is higher clock rate (amount of clock cycles per time unit)
- ➔ How many instructions can I execute per clock cycle?

CPU Performance

- ➔ Cycles per instruction (clock cycles or just clocks)
- ➔ Instructions per cycle (instruction per clock)
- ➔ Instruction per second
- ➔ Elementary Instructions (constant time) vs Composed instructions.

Elementary Instructions

- ➔ Instructions that **always** takes the same time and this time is independent of the input size.
- ➔ We don’t care about how many cycles the instruction needs to be executed
- ➔ **T** is a variable which depends on the architecture and hardware, **T** denotes the time required to execute the instruction.

Constant Time Instructions

➔ Basic Arithmetic Operators (+, -, /, %, ^)

➔ Bitwise operators (<< , >>)

➔ Logical operators (==, !=, and, or, ~ ...)

➔ Jumps (returns values, method calls, ...)

➔ Assignments, access to indexed structures.

Time Complexity Example

```
boolean search(int arr[], int n, int num){
    boolean flag = false;
    for (int i = 0; i < n; i++){
        if (arr[i] == num){
            flag = true;
            break;
        }
    }
    return flag;
}
```

Time Complexity Example

One declaration + One assignment = **2T**

```
boolean search(int arr[], int n, int num){
    boolean flag = false;
    for (int i = 0; i < n; i++){
          if (arr[i] == num){
               flag = true;
               break;
          }
    }
    return flag;
}
```

Time Complexity Example

One declaration + One assignment = **2T**
One comparison = **1T**
One Increment = **1T**

```
boolean search(int arr[], int n, int num){
    boolean flag = false;
    for (int i = 0; i < n; i++){
        if (arr[i] == num){
            flag = true;
            break;
        }
    }
    return flag;
}
```

Time Complexity Example

One declaration + One array access = **2T**

```
boolean sea                                          {
    boolean
    for (int i = 0;      n; i++){
        if (arr[i] == num){
            flag = true;
            break;
        }
    }
    return flag;
}
```

Time Complexity Example

```
boolean sea                                   {
    boolean
    for (int
        if (arr[i]      num){
            flag = true;
            break;
        }
    }
    return flag;
}
```

One assignment = **1T**

Time Complexity Example

```
boolean search(int arr[], int n, int num){
    boole
    for (
        if (a            num){
            flag   true;
            break;
        }
    }
    return flag;
}
```

One jump = **1T**

Time Complexity Example

```
boolean search(int arr[], int n, int num){
    boolean flag = false;
    for (int i = 0; i < n; i++){
        if (arr[i] == num){


        }
    }
    return flag;
}
```

One jump = **1T**

How to calculate complexities

Sequence of statements

```
Statement_1;
  Statement_2;
  …
  …
  …
Statement_N;
```

```
Total time: Time(Stament_1)
          + Time(Stament_2)
      + … + Time(Stament_N)
```

How to calculate complexities

If - Then - Else / Switch

```
if (condition) {
   block of statements 1
} else {
   block of statements 2
}
```

Total time: Time(condition)
+ max(Time(Statement_block_1),
Time(Statement_block_2),
…,
Time((Statement_block_N))

How to calculate complexities

Loops

```
for (int i = 0; i <= n; i++){
        block of statements
}
```

```
                        Total time: time(declaration) +
                                    time(assignment) +
(n+1)(time(comparison_time) + time(increment)) +
                   (n)(time(block of statements)) =
                   2T + (2NT + 2T) + N(T_Block) =
                             3NT + N(T_Block) + 4T
```

Time Complexity Example

```
boolean search(int arr[], int n, int num){
    boolean flag = false;
    for (int i = 0; i < n; i++){
        if (arr[i] == num){
            flag = true;
            break;
        }
    }
    return flag;
}
```

Total time = 7NT + 8T

Best, worst and average

➔ **Best case scenario**:
resource usage at least

➔ **Worst case Scenario**:
resource usage at most

➔ **Average case scenario**:
resource usage on average

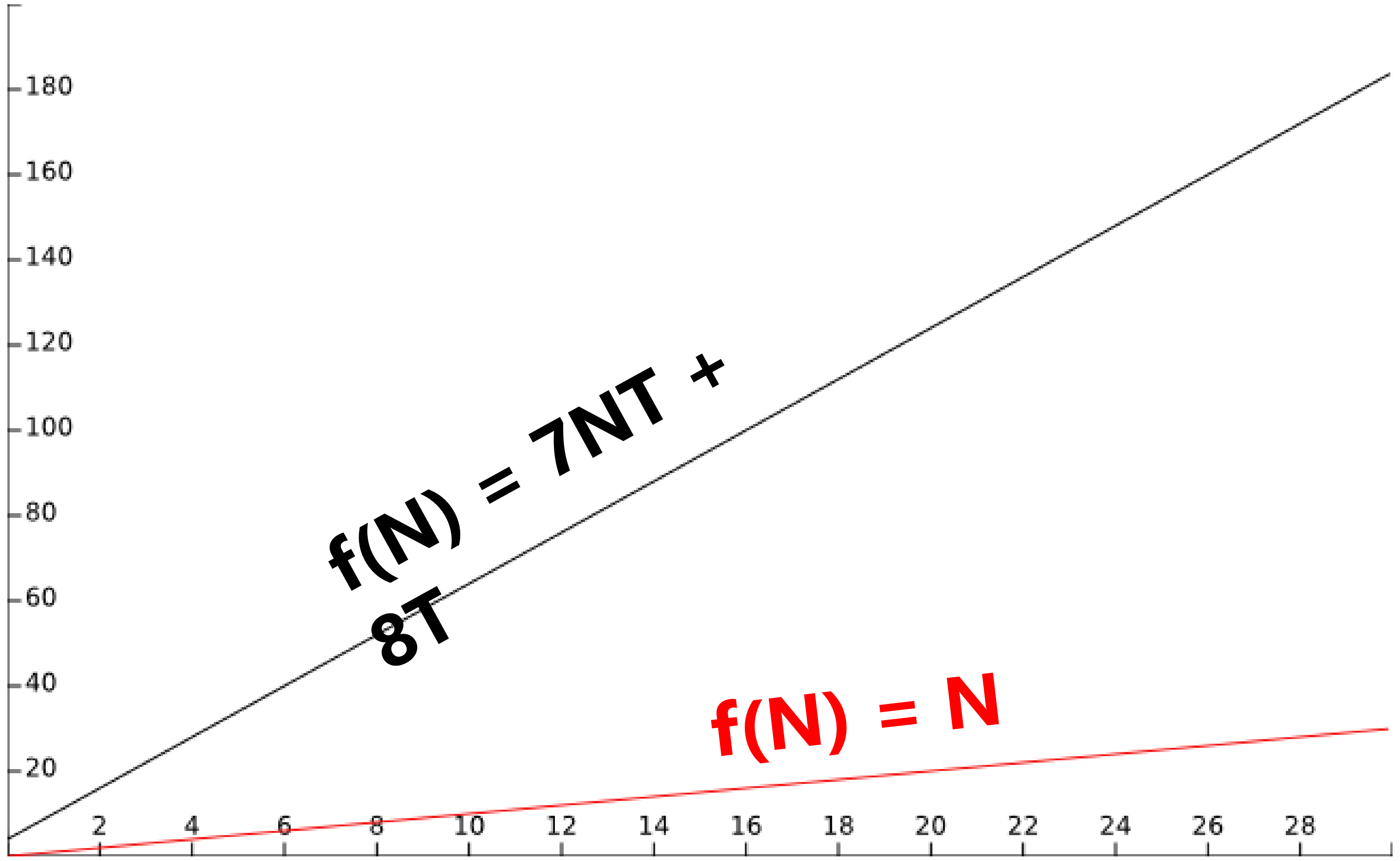
Big-O Notation

➔ Work can be calculated as a function of the size of the input to the algorithm

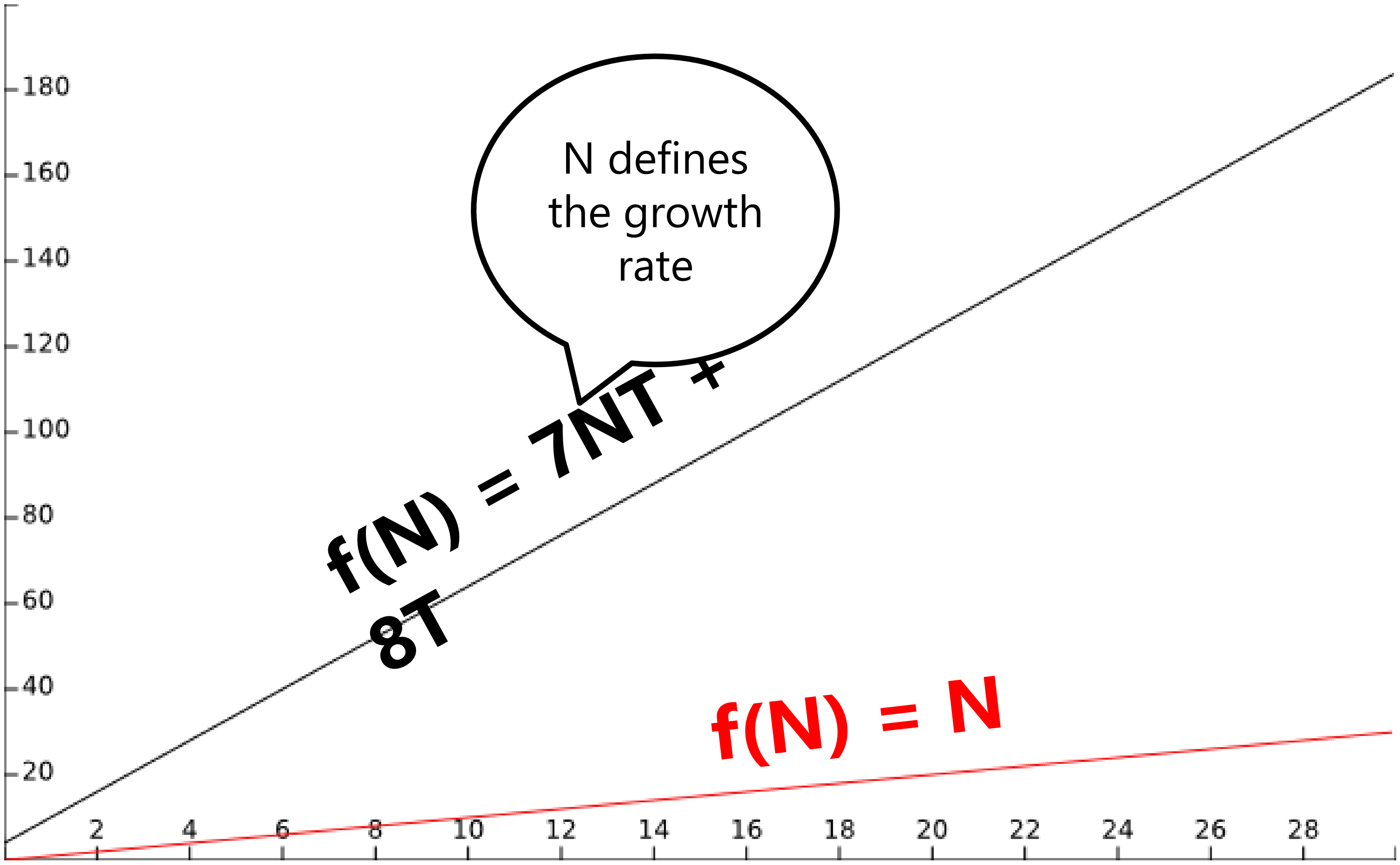
➔ It is possible to express an approximation of this function using a mathematical notation called *order of magnitude or Big-O*

Big-O: A notation that expresses computing time (complexity) as the term in the function that increases most rapidly relative to the size of a problem

Time Complexity Example



Time Complexity Example



Which term defines the function?

➔ $f(n) = 2n^3 + 3n^2 + log_{10}(n) + 333$

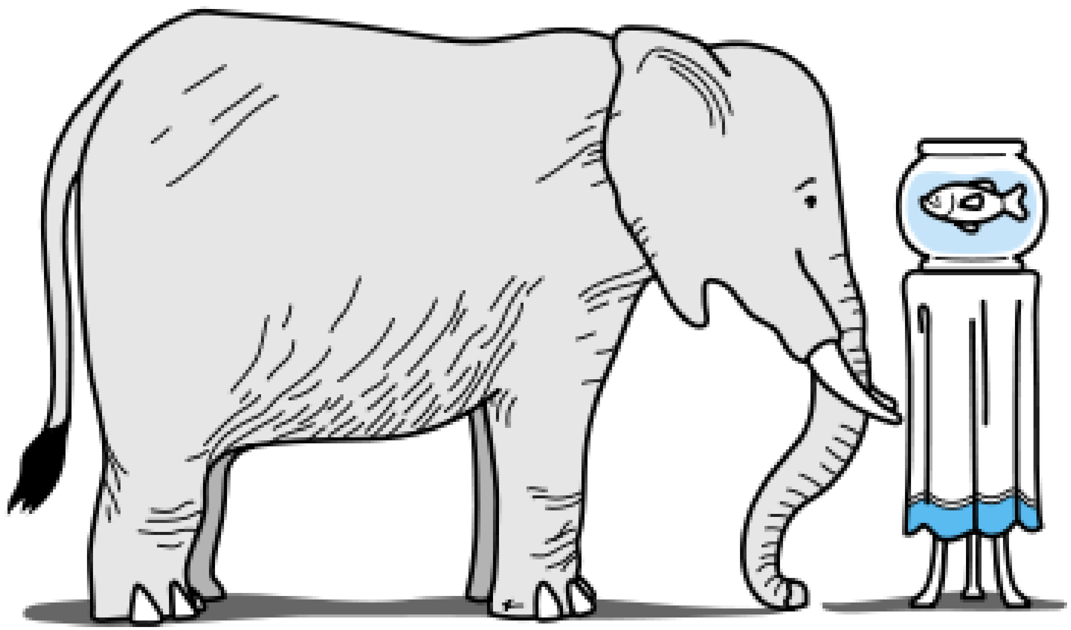
n	**f(n)**	**$2n^3$**	**$3n^2$**	**$log_{10}(n)$**	**333**
1	338	**2**	3	0	333
10	2634	**$2x10^3$**	$3x10^2$	1	333
100	2030335	**$2x10^6$**	$3x10^4$	2	333
1000	2003000336	**$2x10^9$**	$3x10^6$	3	333
10000	2000300000337	**$2x10^{12}$**	$3x10^8$	4	333

Whic… defines the function?

$O(n^3)$

→ $f(n) = 2n^3 + 3n^2 + \log_{10}(n) + 333$

n	**f(n)**	**$2n^3$**	**$3n^2$**	**$\log_{10}(n)$**	**333**
1	338	**2**	3	0	333
10	2634	**$2x10^3$**	$3x10^2$	1	333
100	2030335	**$2x10^6$**	$3x10^4$	2	333
1000	2003000336	**$2x10^9$**	$3x10^6$	3	333
10000	200030000337	**$2x10^{12}$**	$3x10^8$	4	333



Common Orders of Growth

1	Constant	Common Problems
$\log_{10} N$	Logarithmic	
N	Linear	
$N \log_{10} N$	Linear Logarithmic	
N^2	Quadratic	
N^3	Qubic	
2^N	Exponential	Hard Problems
N!	Factorial	

Common Orders of Growth

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296
64	6	384	4,096	262,144	About 1 month's worth of instructions on a supercomputer
128	7	896	16,384	2,097,152	About 10^{12} times greater than the age of the universe in nanoseconds (for a 6-billion-year estimate)
256	8	2,048	65,536	16,777,216	Don't ask!

Sort Algorithms

Sorting Algorithm

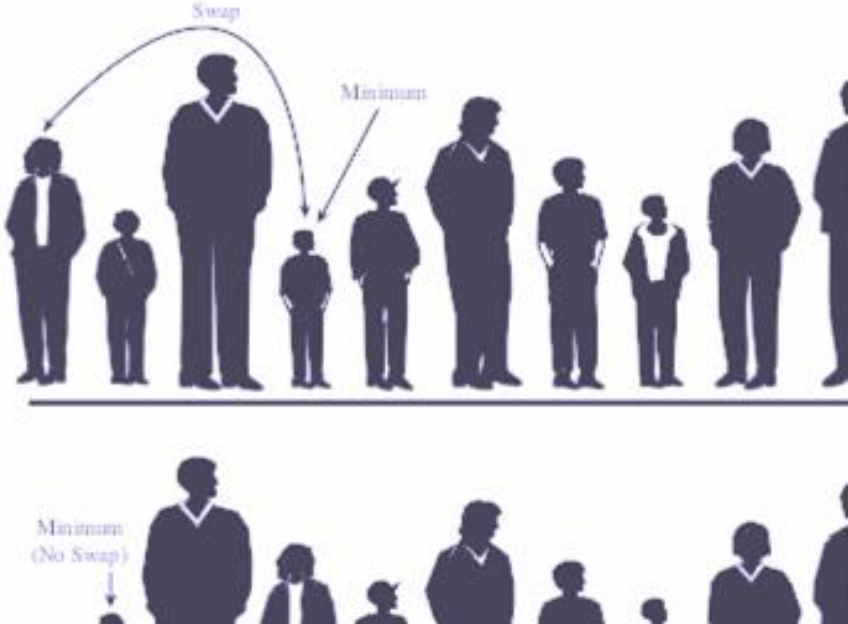
- ➔ Keeping lists of elements in sorted order is important to facilitate searching
- ➔ There are many algorithms to sort lists
- ➔ The main goal is to come up with better, more efficient, sorts

Sorting Algorithm

- ➔ How can we describe efficiency while comparing sorting algorithms?
  - ◆ Pick the central operation for sorting algorithms: the operation that compares two values
  - ◆ Relate the efficiency of each algorithm to the number of elements in the list (N) as a measure

Selection Sort

Sorting algorithm



An example problem

- ➔ We have a list of names and we are asked to put them in alphabetical order. How do you solve it?

Selection sort

The solution

- ➔ Find the name that comes first in alphabetical order and write it on a second sheet of paper
- ➔ Cross the name out on the original list
- ➔ Continue the cycle until all names on the original list have been crossed out and written onto the second list, which is sorted

Selection sort

Space consideration

- ➔ Keeping a second list of elements involves using more memory
- ➔ A slight adjustment involves swapping elements:
  - ◆ As you “cross out” elements of the original list, a free space opens up.
  - ◆ Swap the found element with the “current” position

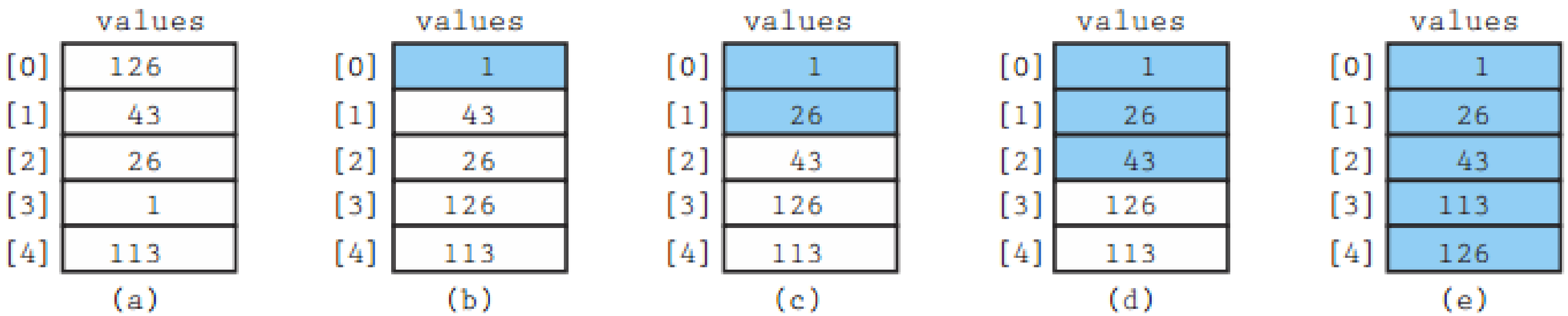
Selection sort

Pseudocode

```
Set current to the index of first item in the array
while more items in unsorted part of array
   Find the index of the smallest unsorted item
   Swap the current item with the smallest unsorted one
   Increment current to shrink unsorted array part
```

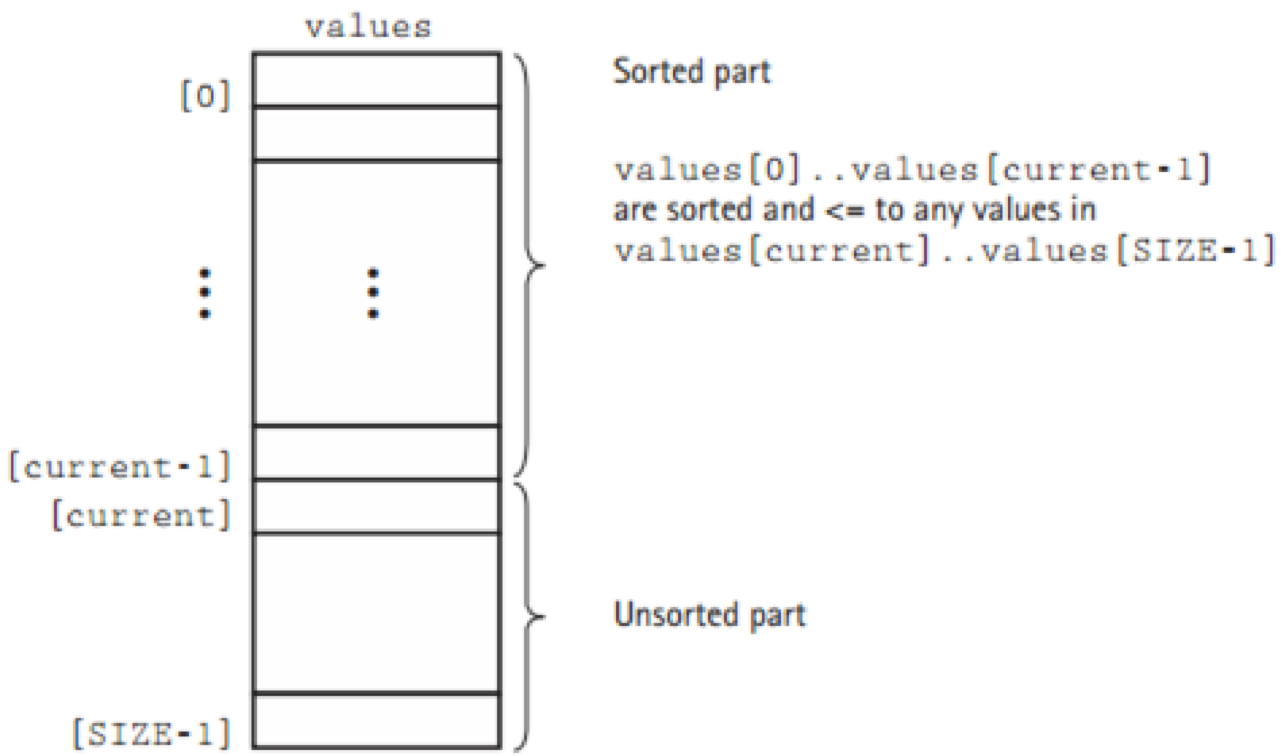
Selection sort

Example



Selection sort

Example



Selection sort

Implementation

```
static int minIndex(int start, int end) {
   int indexOfMin = start;
   for (int index = start + 1; index <= end; index++){
     if (values[index] < values[indexOfMin]){
       indexOfMin = index;
     }
   }
   return indexOfMin;
}
```

Selection sort

Implementation

```
static void selectSort() {
  int endIdex = size - 1;
  for (int current = 0; current < endIndex; current++) {
    swap(current, minIndex(current, endIndex));
  }
}
```

Selection sort

Analyzing the algorithm

```
static int minIndex(int start, int end) {
   int indexOfMin = start;
   for (int index = start + 1; index <= end; index++){
     if (values[index] < values[indexOfMin]){
       indexOfMin = index;
     }
   }
   return indexOfMin;
}
```

➔ In the first call there are N-1 comparisons

➔ Next call is N-2 and so on until only 1 comparison left

Selection sort

Analyzing the algorithm

- ➔ The total number of comparisons is:

$$(N-1) + (N-2) + (N-3) + \ldots + 1 = N(N-1)/2$$

- ➔ In terms of Big-O:

$$1/2N^2 - 1/2N$$

$$\boldsymbol{O(N^2)}$$

Selection sort

Analyzing the algorithm

Number of Items	Number of Comparisons
10	45
20	190
100	4,950
1,000	499,500
10,000	49,995,000

Bubble Sort

Sorting algorithm

Bubble sort

An example problem

➔ Arrange the following line of kids by its height (ascending), but you can only see two adjacent kids at the same time :

The solution

- ➔ Start at the left end of the line and compare the two kids in positions 0 and 1
- ➔ If the one on the left is taller, you swap them
- ➔ If the one on the right is taller, don’t do anything.

Bubble sort

The solution

- ➔ Move over one position and compare the kids in position 1 and 2
- ➔ Continue until you reach the right end
- ➔ At this point you know the tallest kid is on the right end. As the algorithm progresses the biggest items “bubbles up” to the end of the array

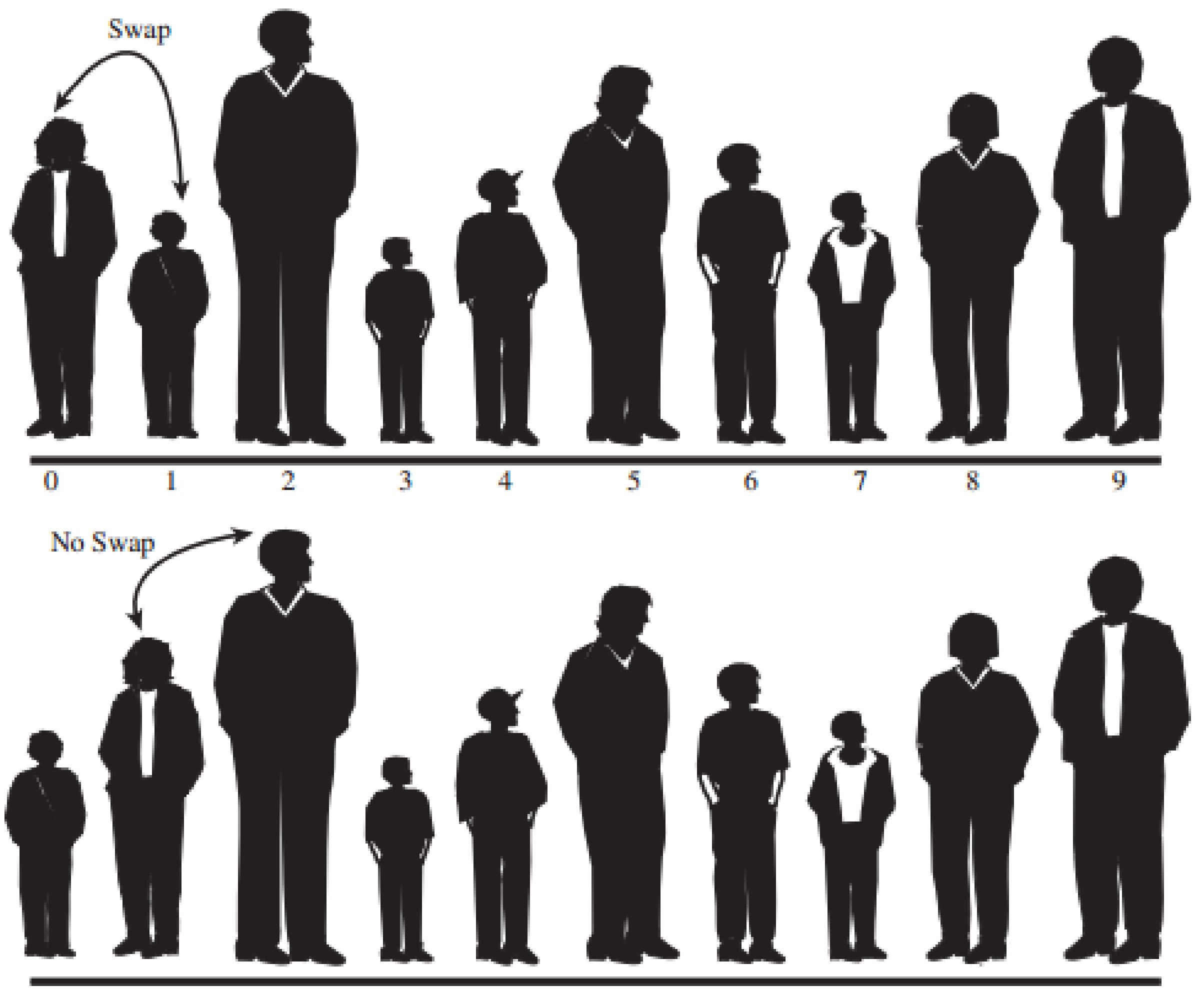
Bubble sort

The solution

- ➔ In the first pass you did N-1 comparisons.
- ➔ Do a second pass but stopping on the N-2th kid, because you know the last kid is the tallest.
- ➔ Continue until all kids are sorted

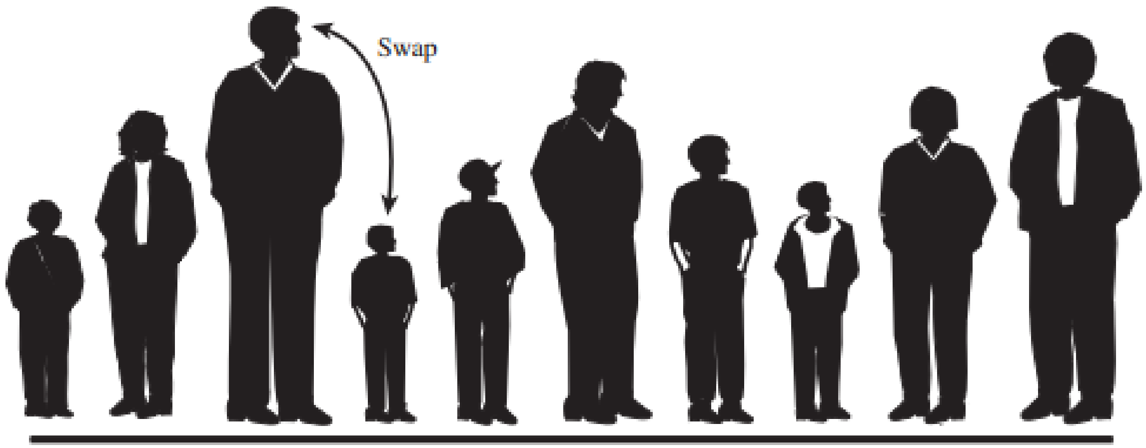
Bubble sort

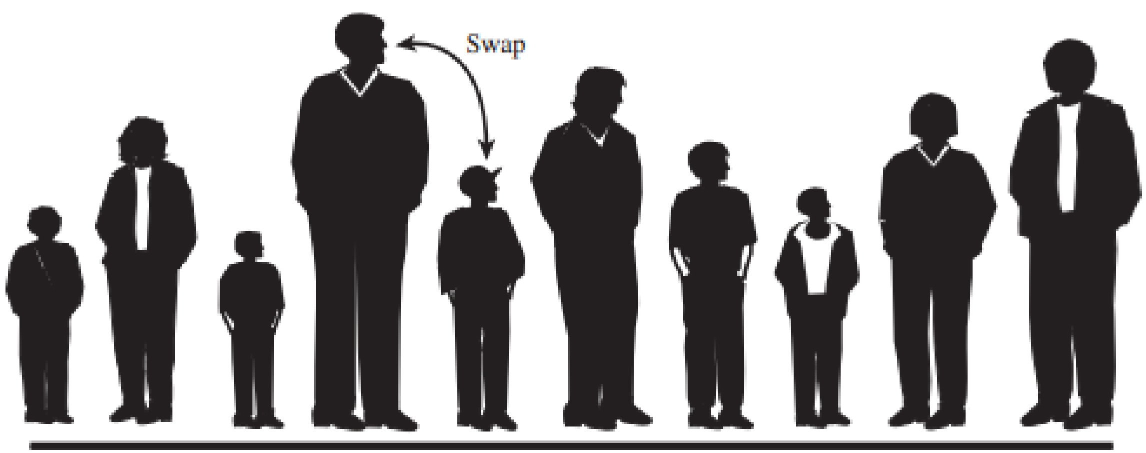
The solution



Bubble sort

The solution





The solution

➔ End of the first pass:

Bubble sort

Implementing the algorithm

```
void bubbleSort() {
  int in;
  int out;

  for (out = nElements - 1; out > 1; out--) {
    for (in = 0; in < out; in++) {
      if (a[in] > a[in + 1]) {
        swap(in, in + 1);
      }
    }
  }
}
```

Bubble sort

Analyzing the algorithm

➔ It is the same as selection sort

➔ $O(N^2)$

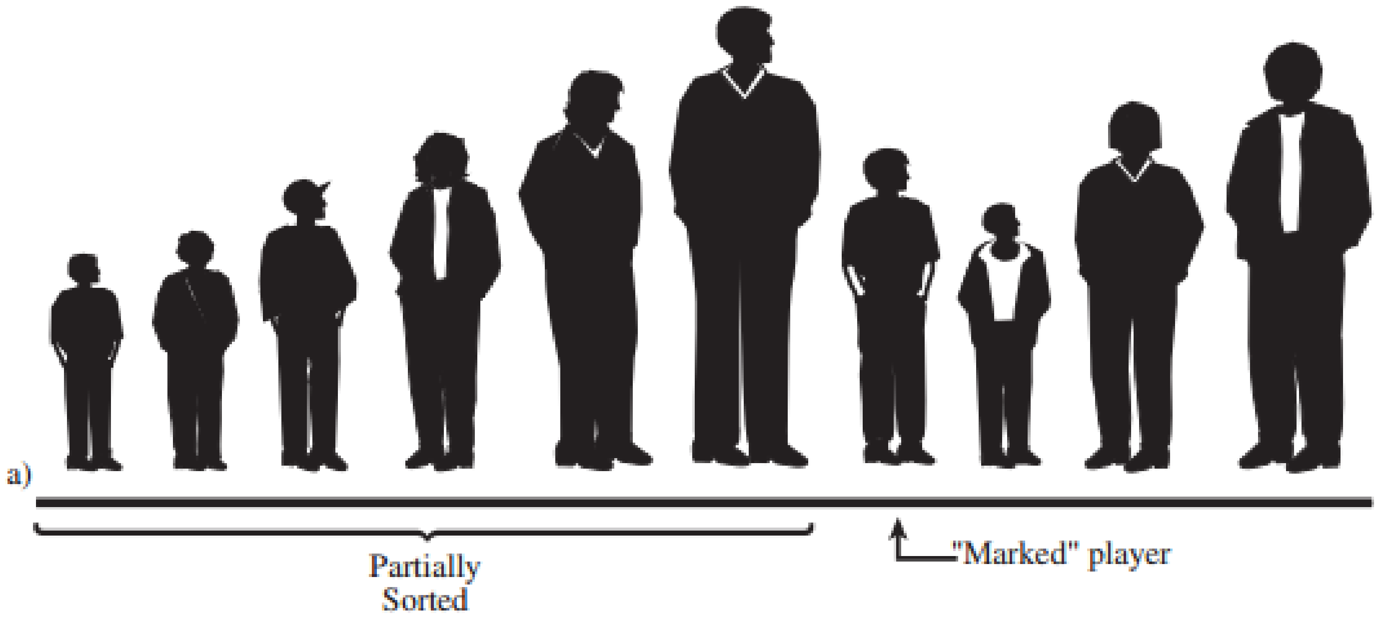
Insertion Sort

Sorting algorithm

Insertion sort

An example problem

➔ Arrange the following line of kids by its height (ascending), but the list is partially sorted :



Insertion sort

An example problem

➔ Arrange the following line of kids by its height (ascending), but the list is partially sorted :

No, it is **not necessary** for the list to be partially sorted from the beginning

Insertion sort

The solution

- ➔ Take the first kid from the unsorted part of the list. (If we are beginning, the sorted part will be comprised by the first element of the list)

- ➔ We need to insert the selected kid into the appropriate place in the sorted group

The solution

- ➔ To do this, we'll need to shift some of the sorted kids to the right to make room
  - ◆ Remove the selected kid
  - ◆ Shift the sorted kids: the tallest kid moved to the removed kid spot, and the next into the tallest kid spot, and so on…

- ➔ The shifting stops when you've shifted the last kid that's taller than the marked kid.

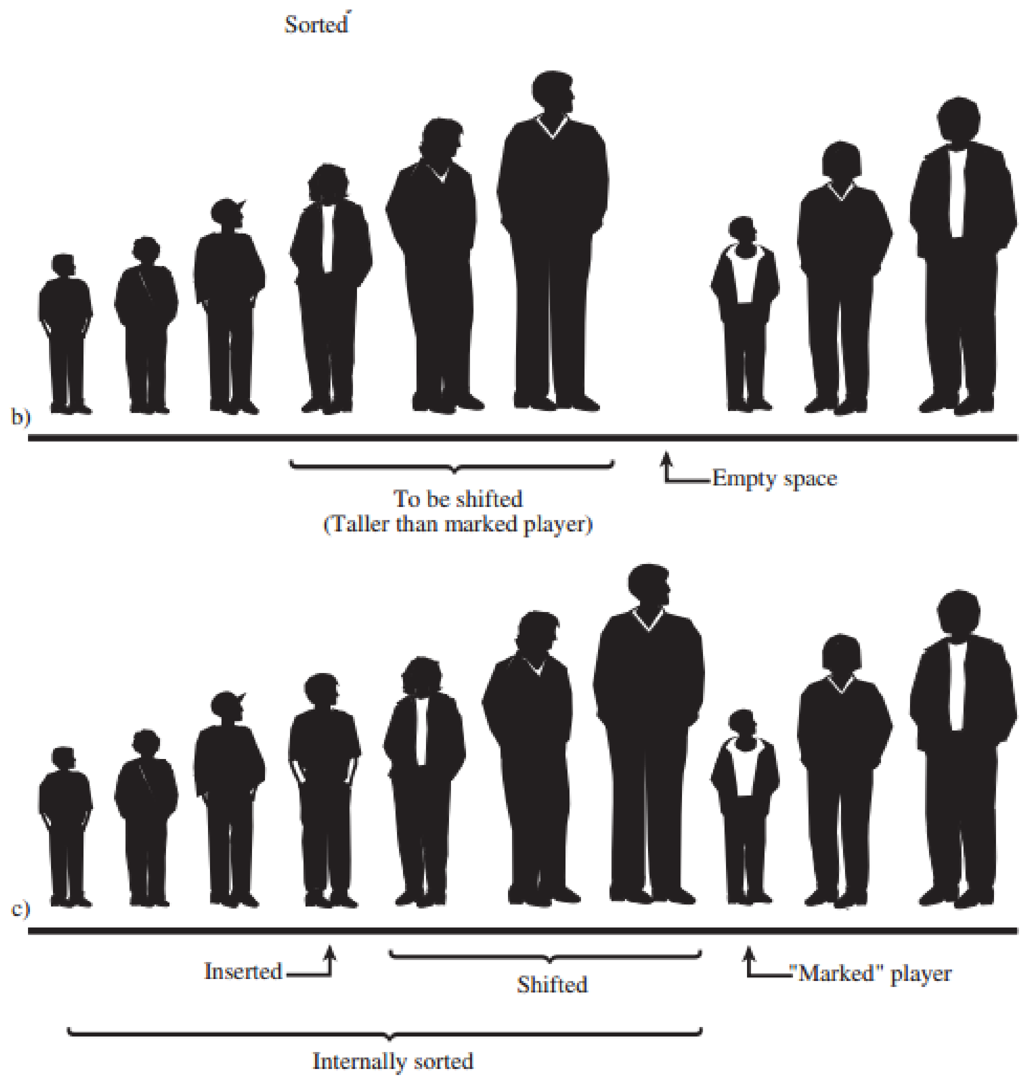
Insertion sort

The solution

➔ The process is repeated until all the unsorted kids have been inserted (*insertion sort*) into the appropriate place in the partially sorted group

Insertion sort

The solution



Implementing the algorithm

```
void insertionSort() {
  int in;
  int out;

  for (out = 1; out < nElems; out++) {
    long temp = a[out];
    in = out;
    while (in > 0 && a[in-1] >= temp) {
      a[in] = a[in-1];
      --in;
    }
    a[in] = temp;
  }
}
```

Insertion sort

Analyzing the algorithm

➔ It is the same as selection sort and bubble sort

➔ $O(N^2)$

Shell Sort

Sorting algorithm

Shell Sort

→ Discovered by Donald L. Shell in 1959

→ Based on insertion sort, but adds a new feature that dramatically improves its performance

→ Good for medium-sized arrays (up to a few thousand items)

Shell Sort

- ➔ Discovered by Donald L. Shell in 1959
- ➔ Based on insertion sort, but adds a new feature that dramatically improves its performance
- ➔ Good for medium-sized arrays (up to a few thousand items)

Shell Sort

- ➔ Breaks the original list in sublists, each of which is sorted using insertion sort

- ➔ Instead of breaking the list into sublists of contiguous elements, the shell uses an increment *h*, called the gap, creating a list that is *h* elements apart.

Shell Sort

Example problem

➔ Sort the following array:

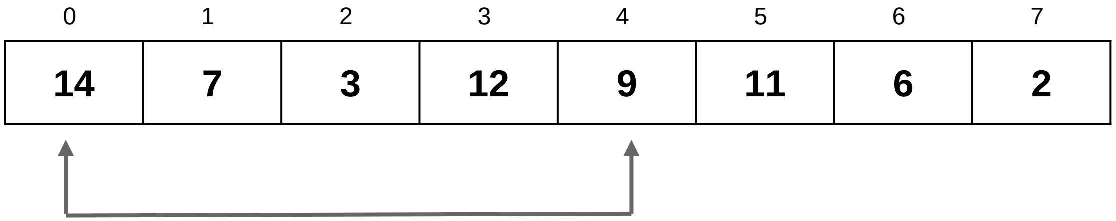
14	7	3	12	9	11	6	2

Define an h...

```
h = 1;
while(h <= nElems/3)
h = h*3 + 1;
```

Shell Sort

The solution



Shell Sort

The solution

0	1	2	3	4	5	6	7
9	**7**	**3**	**12**	**14**	**11**	**6**	**2**

Shell Sort

The solution

0	1	2	3	4	5	6	7
9	**7**	**3**	**12**	**14**	**11**	**6**	**2**

Shell Sort

The solution

0	1	2	3	4	5	6	7
9	**7**	**3**	**12**	**14**	**11**	**6**	**2**

Shell Sort

The solution

0	1	2	3	4	5	6	7
9	**7**	**3**	**12**	**14**	**11**	**6**	**2**

Shell Sort
The solution

0	1	2	3	4	5	6	7
9	**7**	**3**	**2**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
9	**7**	**3**	**2**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
7	**9**	**3**	**2**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
7	**9**	**3**	**2**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
7	**3**	**9**	**2**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
7	**3**	**9**	**2**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
3	**7**	**9**	**2**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
3	**7**	**9**	**2**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
3	**7**	**2**	**9**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
3	**7**	**2**	**9**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
3	**2**	**7**	**9**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
3	**2**	**7**	**9**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**14**	**11**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**11**	**14**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**11**	**14**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**11**	**14**	**6**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**11**	**6**	**14**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**11**	**6**	**14**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**6**	**11**	**14**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**9**	**6**	**11**	**14**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**6**	**9**	**11**	**14**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**7**	**6**	**9**	**11**	**14**	**12**

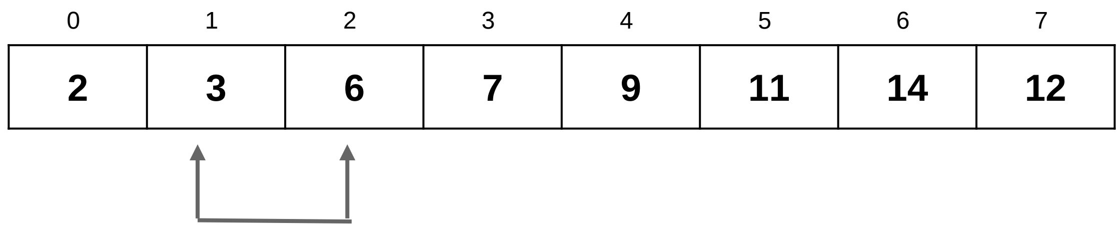
Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**6**	**7**	**9**	**11**	**14**	**12**

Shell Sort

The solution



Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**6**	**7**	**9**	**11**	**14**	**12**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**6**	**7**	**9**	**11**	**12**	**14**

Shell Sort

The solution

0	1	2	3	4	5	6	7
2	**3**	**6**	**7**	**9**	**11**	**12**	**14**

Shell Sort

Implementing the algorithm

```
public static void sort(Comparable[] a) {
  int N = a.length;

  // 3x+1 increment sequence:  1, 4, 13, 40, 121, 364, 1093,
  int h = 1;
  while (h < N/3) h = 3*h + 1;

  while (h >= 1) {
    // h-sort the array
    for (int i = h; i < N; i++) {
      for (int j = i; j >= h && a[j] < a[j-h]; j -= h) {
          swap(a, j, j-h);
      }
    }
    h /= 3;
 }
}
```

Shell Sort

Analyzing the algorithm

- Picking the right gap is not standard
  - Different sequences can do the job

- No one so far has been able to analyze the efficiency theoretically

- Based on experiments, there are various estimates

Shell Sort

Analyzing the algorithm

- ➔ Picking the right gap is not standard
  - ◆ Different sequences can do the job

- ➔ No one so far has been able to analyze the efficiency theoretically

- ➔ Based on experiments, there are various estimates

Shell Sort

Analyzing the algorithm

O() Value	Type of Sort	10 Items	100 Items	1,000 Items	10,000 Items
N^2	Insertion, etc.	100	10,000	1,000,000	100,000,000
$N^{3/2}$	Shellsort	32	1,000	32,000	1,000,000
$N*(\log N)^2$	Shellsort	10	400	9,000	160,000
$N^{5/4}$	Shellsort	18	316	5,600	100,000
$N^{7/6}$	Shellsort	14	215	3,200	46,000
$N*\log N$	Quicksort, etc.	10	200	3,000	40,000

Merge Sort

Sorting algorithm

Merge Sort

Example problem

➔ Sort the following array:

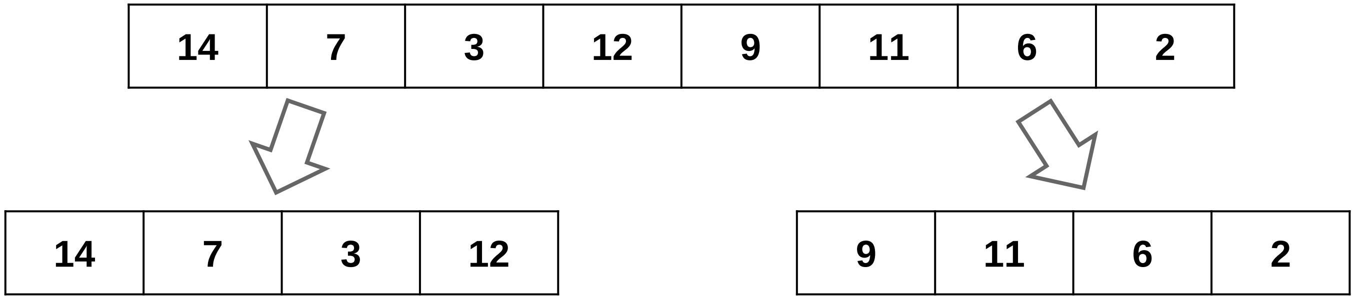
14	7	3	12	9	11	6	2

The solution

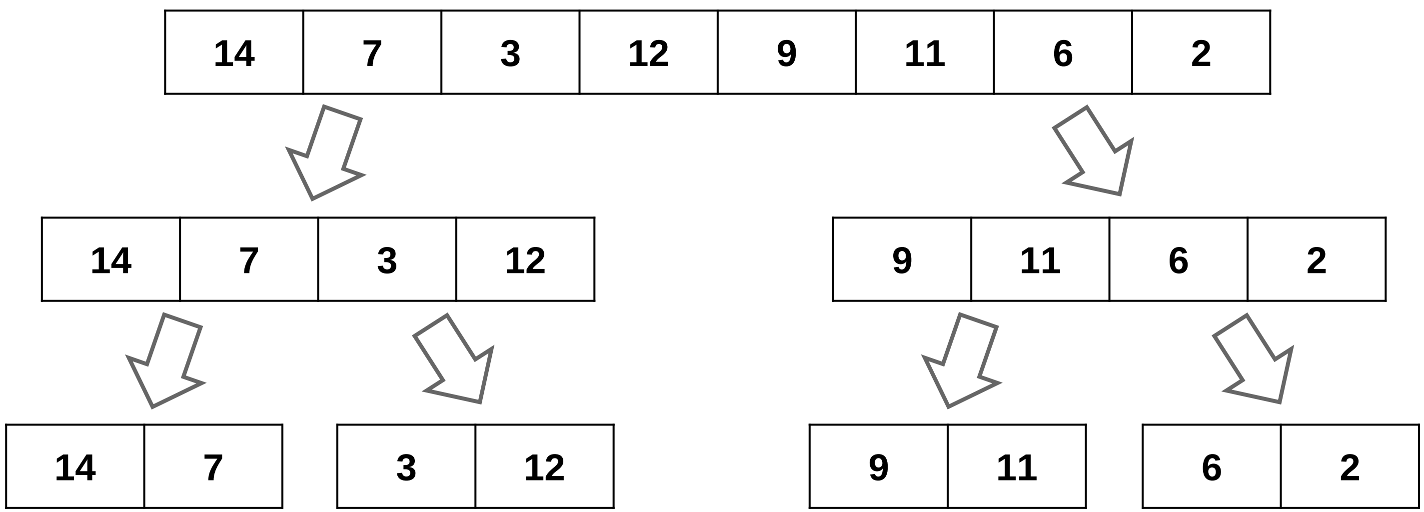
- ➔ Cut the array in half
- ➔ MergeSort the left half
- ➔ MergeSort the right half
- ➔ Merge the sorted halves into one sorted array

Merge Sort

The solution

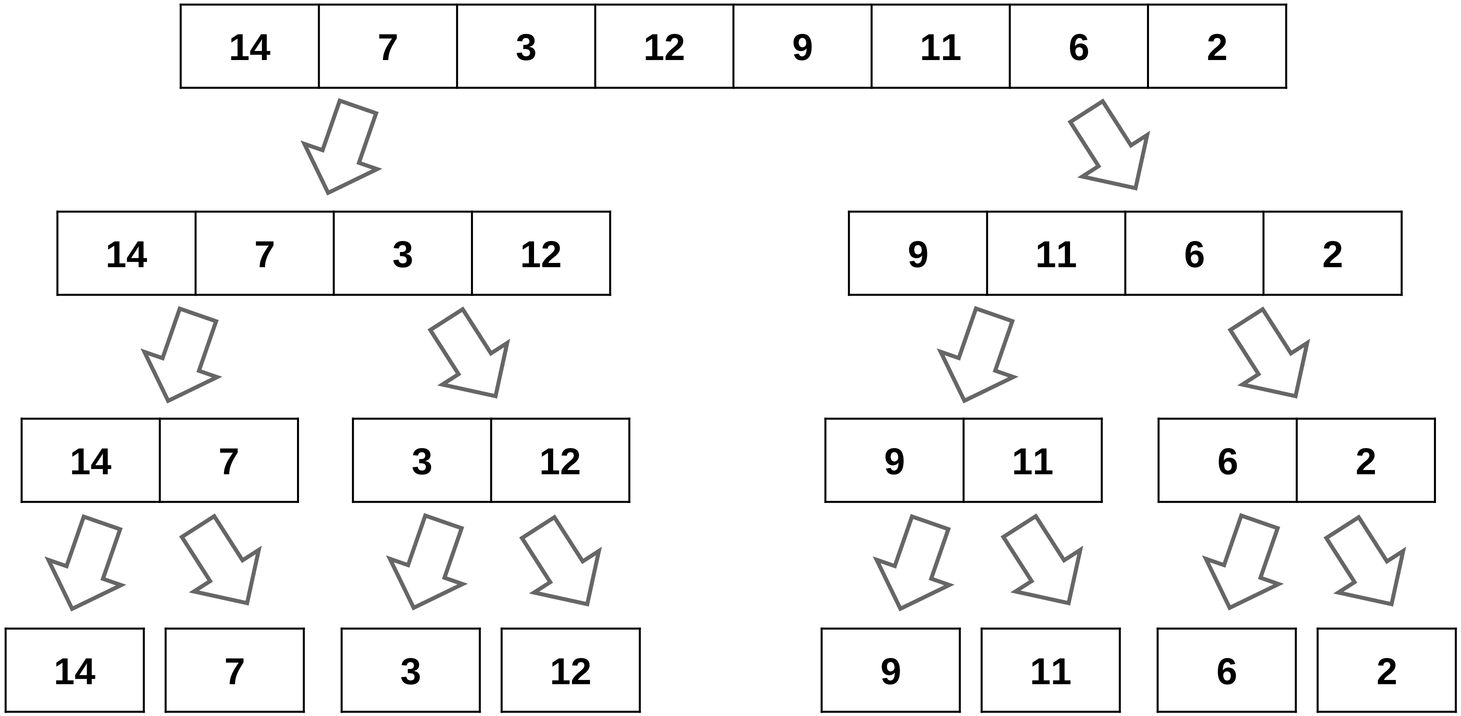


Merge Sort
The solution



Merge Sort

The solution



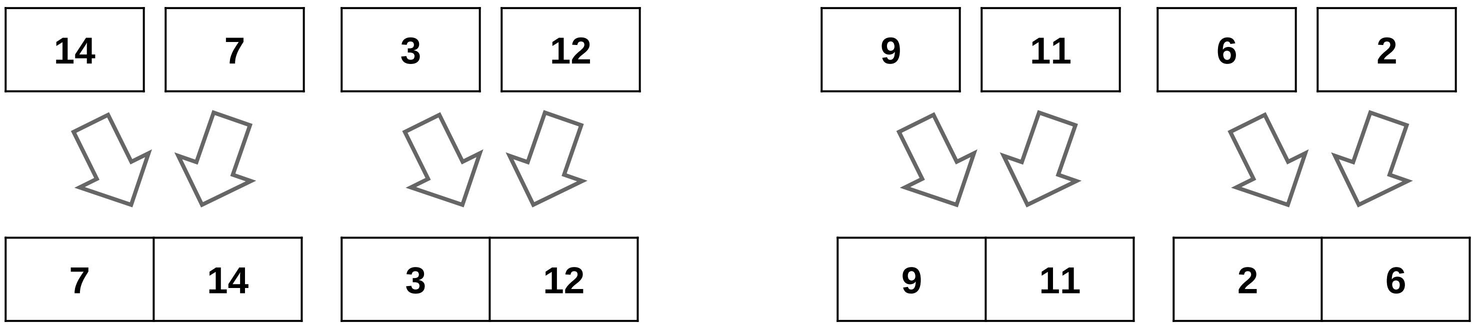
Merge Sort

The solution

14	7	3	12	9	11	6	2

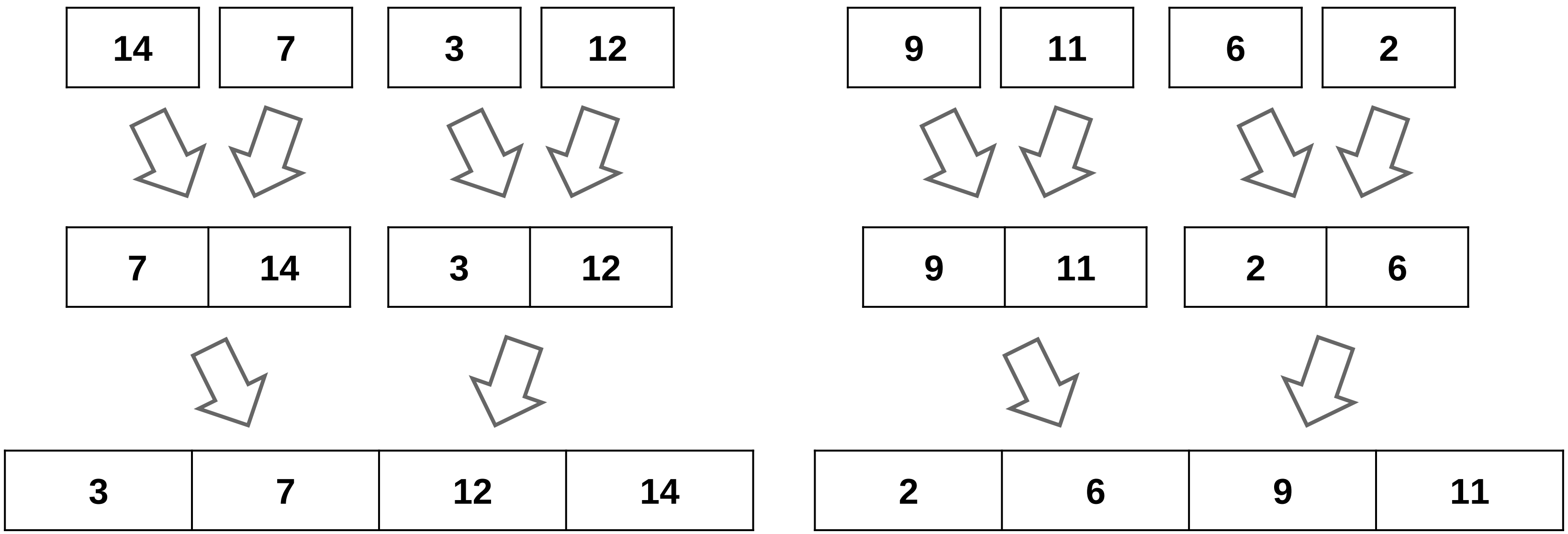
Merge Sort

The solution



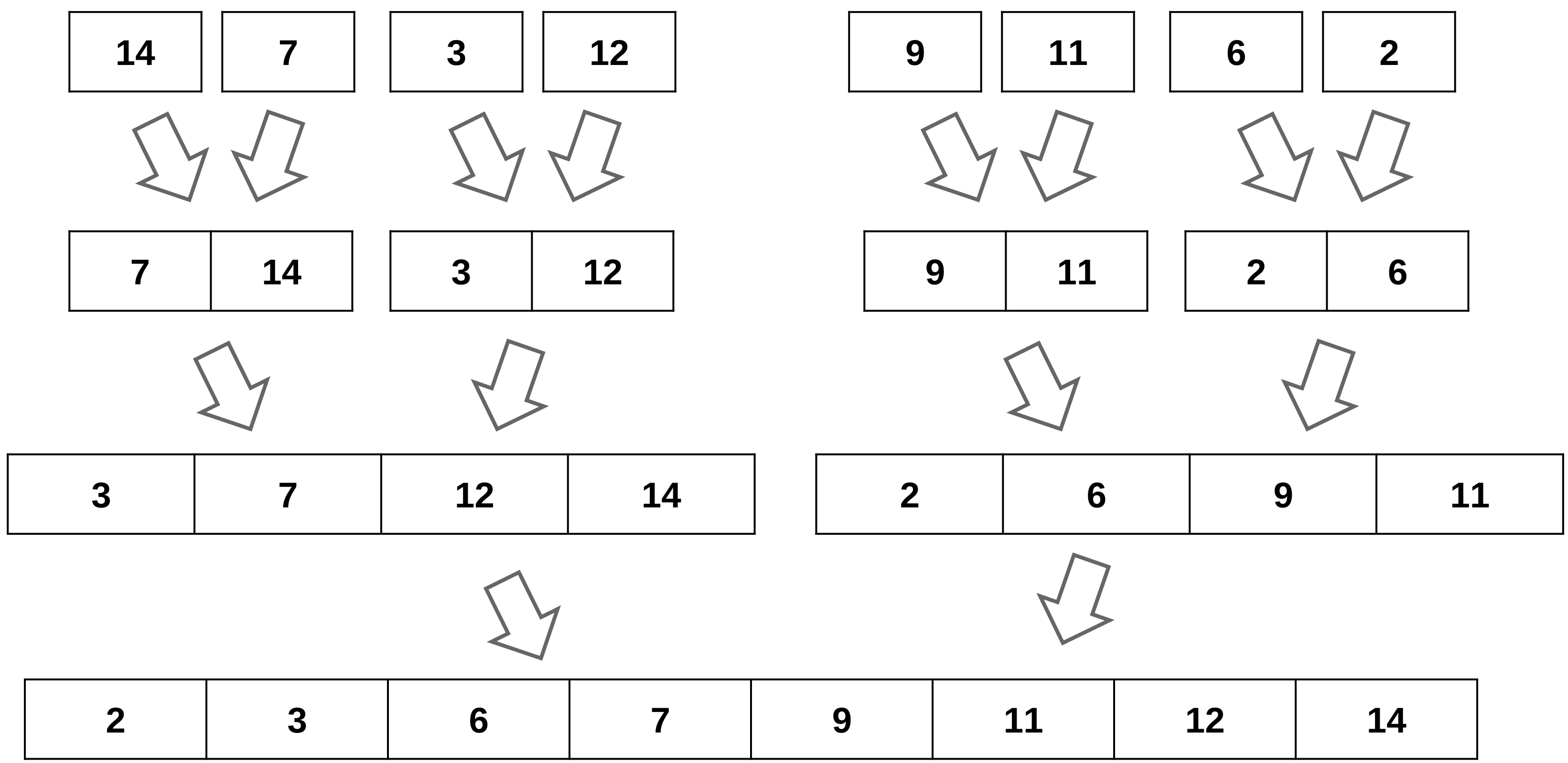
Merge Sort

The solution



Merge Sort

The solution



Merge Sort

Implementing the algorithm

```
void mergeSort(int[] a, int low, int high) {
    int mid;
    if (low < high) {
        mid = (low + high) / 2;
        mergesort(a, low, mid);
        mergesort(a, mid + 1, high);
        merge(a, low, high, mid);
    }
    return;
}
```

Merge Sort

Implementing the algorithm

```
void merge(int[] a, int low, int high, int mid) {
   int i, j, k, c[50];
   i = low;
   k = low;
   j = mid + 1;
   while (i <= mid && j <= high) {
       if (a[i] < a[j]) {
           c[k] = a[i];
           k++;
           i++;
       } else {
           c[k] = a[j];
           k++;
           j++;
       }
   }
```

Merge Sort

Implementing the algorithm

```
    while (i <= mid) {
        c[k] = a[i];
        k++;
        i++;
    }
    while (j <= high) {
        c[k] = a[j];
        k++;
        j++;
    }
    for (i = low; i < k; i++) {
        a[i] = c[i];
    }
}
```

Merge Sort

Analyzing the algorithm

➔ Splits the original array into two halves

➔ Sorts the first half of the array using the divide and conquer approach

➔ Sort the second half using the same approach

➔ Merges the two halves

Merge Sort

Analyzing the algorithm

- ➔ Merge sort continuously divides the original array in two until it has created N one element subarrays
- ➔ To divide the array O(N) is required and merging each level is also O(N)
- ➔ How many levels are generated? $\log_2 N$

Merge Sort

Analyzing the algorithm

➔ So merge sort is $O(N\log_2 N)$

N	$\log_2 N$	N^2	$N\log_2 N$
32	5	1,024	160
64	6	4.096	384
128	7	16,384	896
256	8	65,536	2,048
512	9	262,144	4,608
1024	10	1,048,576	10,240
2048	11	4,194,304	22,528
4096	12	16,777,216	49,152

Quick Sort

Sorting algorithm

Quicksort

- ➔ Is the most popular sorting algorithm
  - ◆ In the majority of situations is the fastest (for in memory sorting)

- ➔ Works by partitioning an array into two subarrays and then calling itself recursively to quick-sort each of these subarrays

Example problem

➔ Sort the following array:

14	7	3	12	9	11	6	2

QuickSort

The solution

0	1	2	3	4	5	6	7
14	**7**	**3**	**12**	**9**	**11**	**6**	**2**

QuickSort

The solution

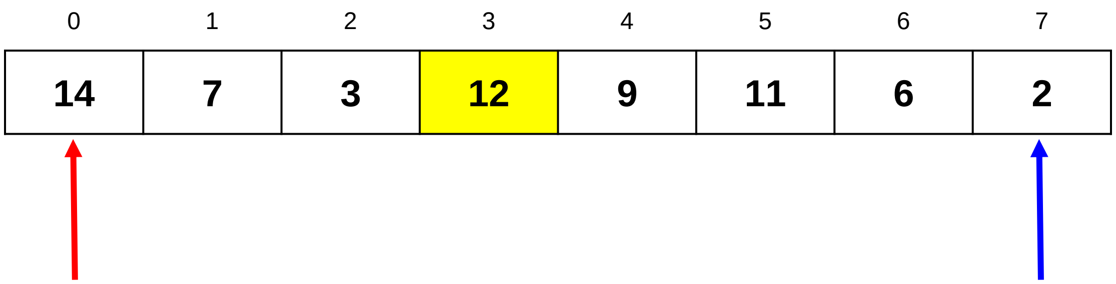
0	1	2	3	4	5	6	7
14	**7**	**3**	**12**	**9**	**11**	**6**	**2**

Let's calculate the pivot

0 + ((7- 0) / 2) = 3

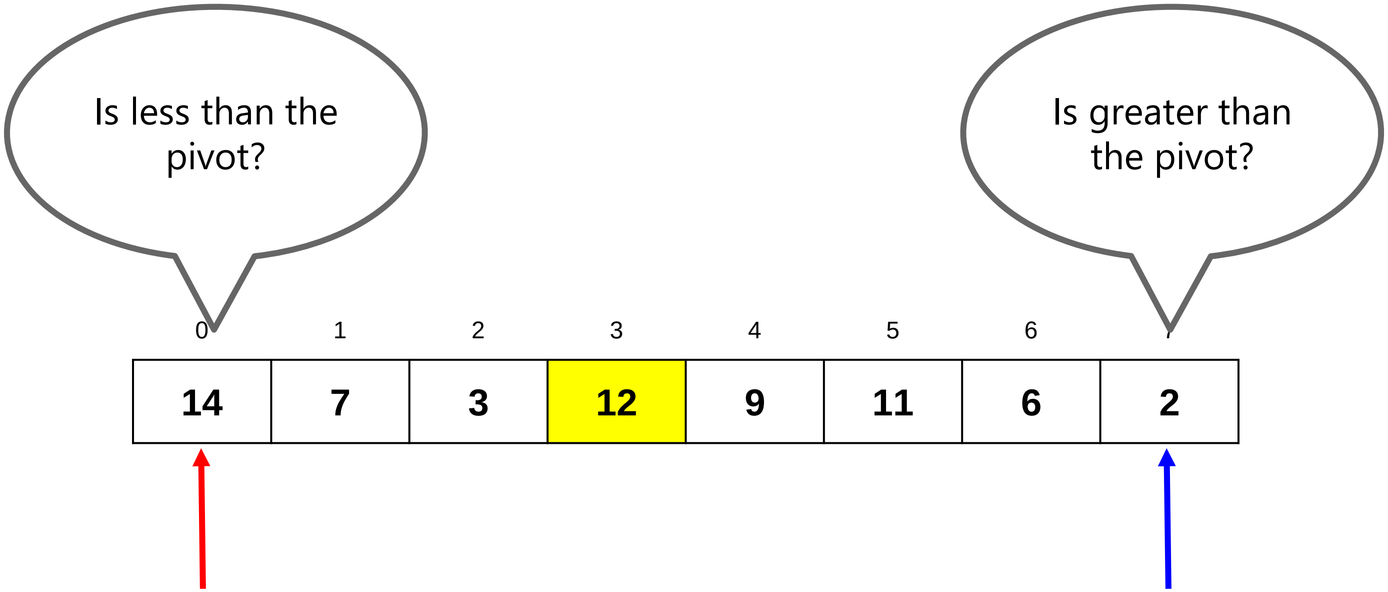
QuickSort

The solution

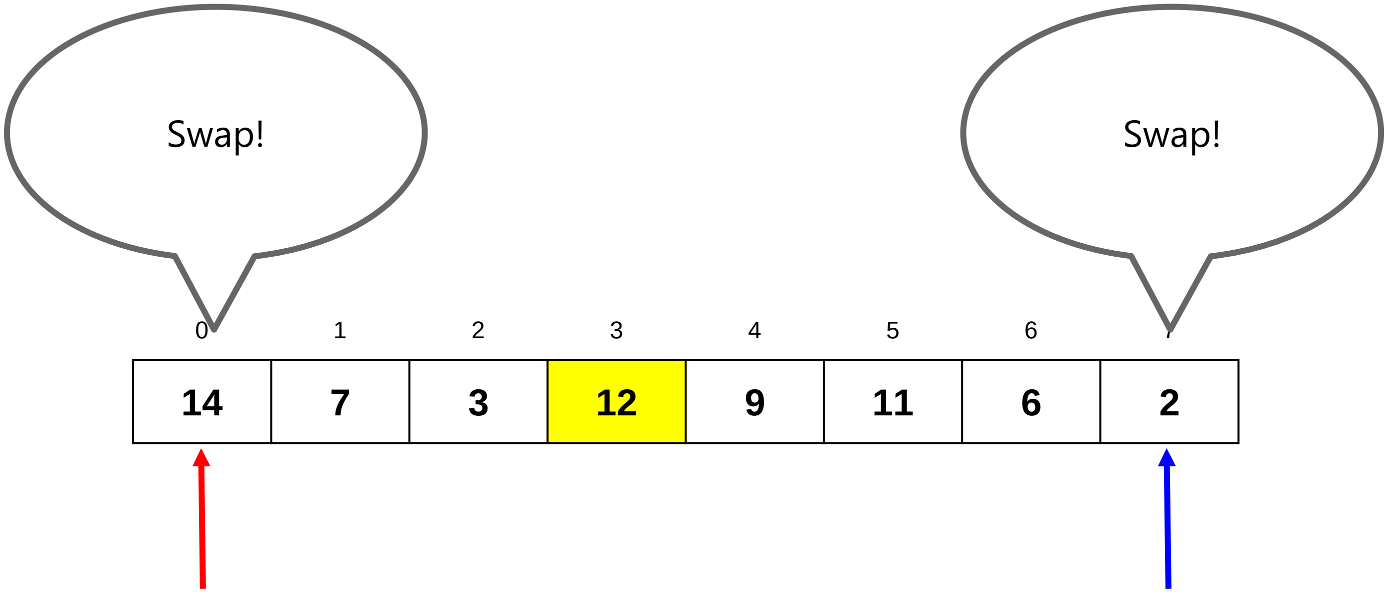


QuickSort

The solution

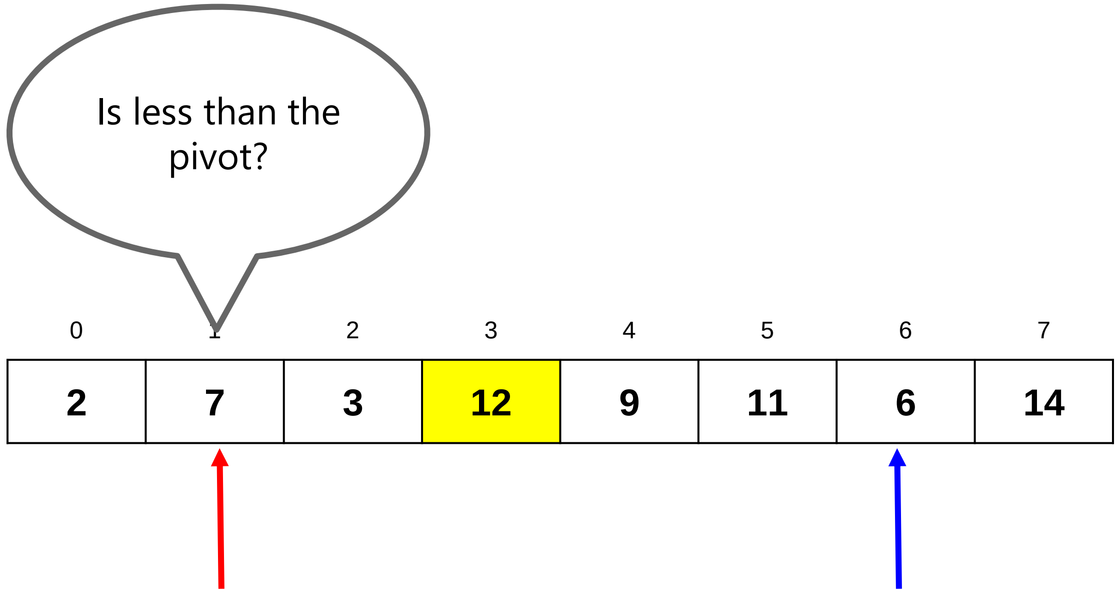


QuickSort
The solution

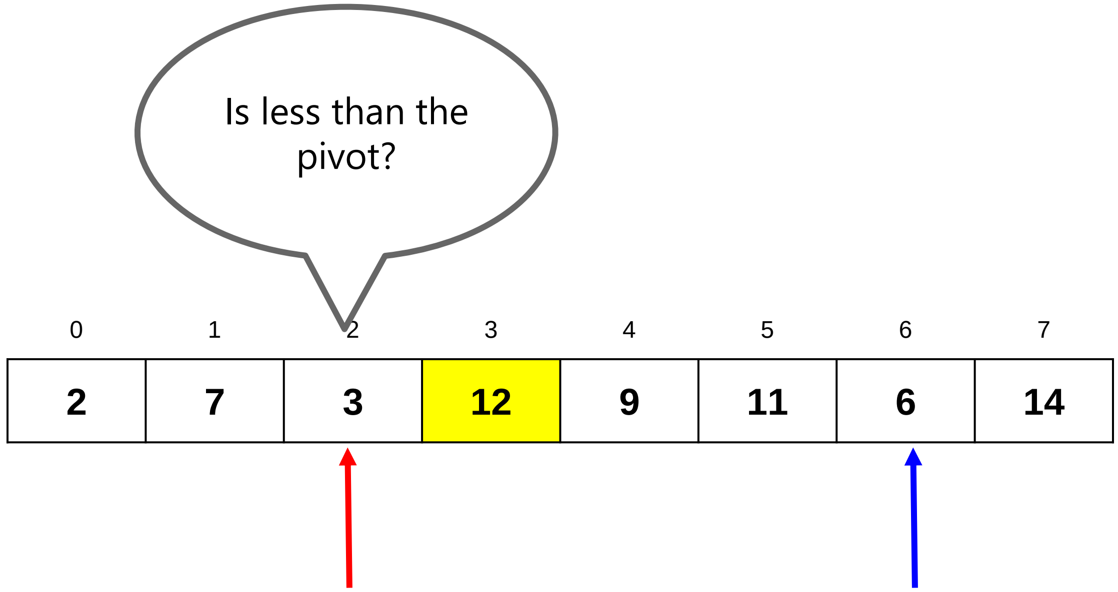


QuickSort

The solution

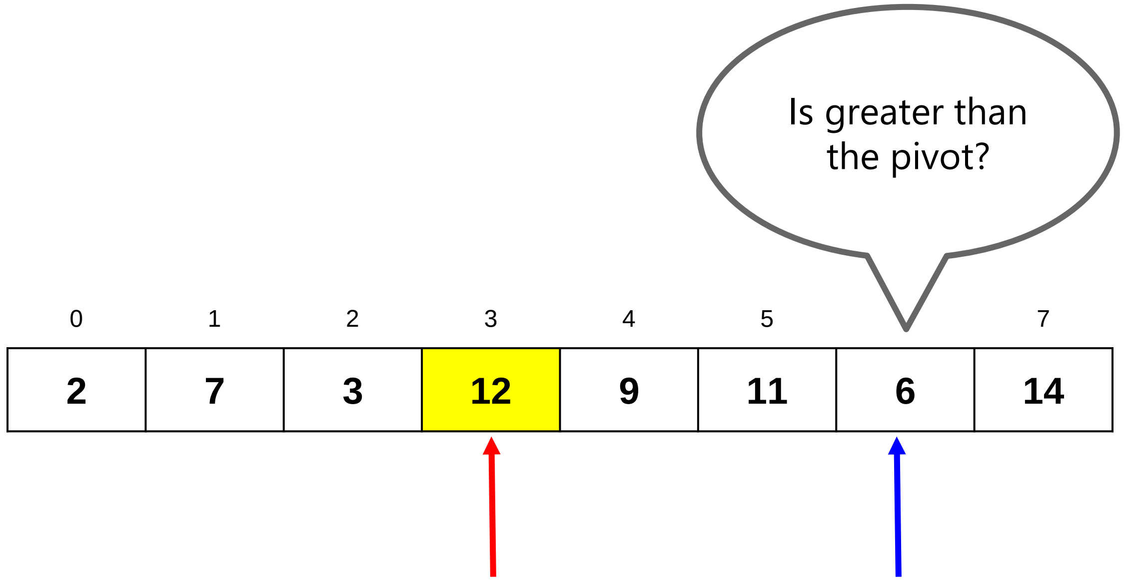


QuickSort
The solution

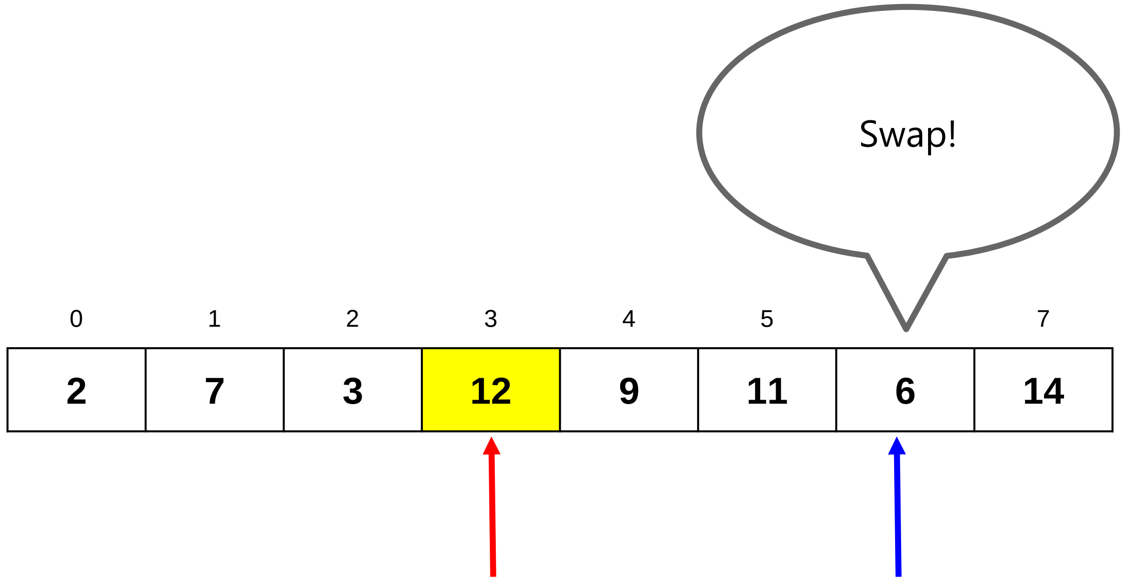


QuickSort

The solution

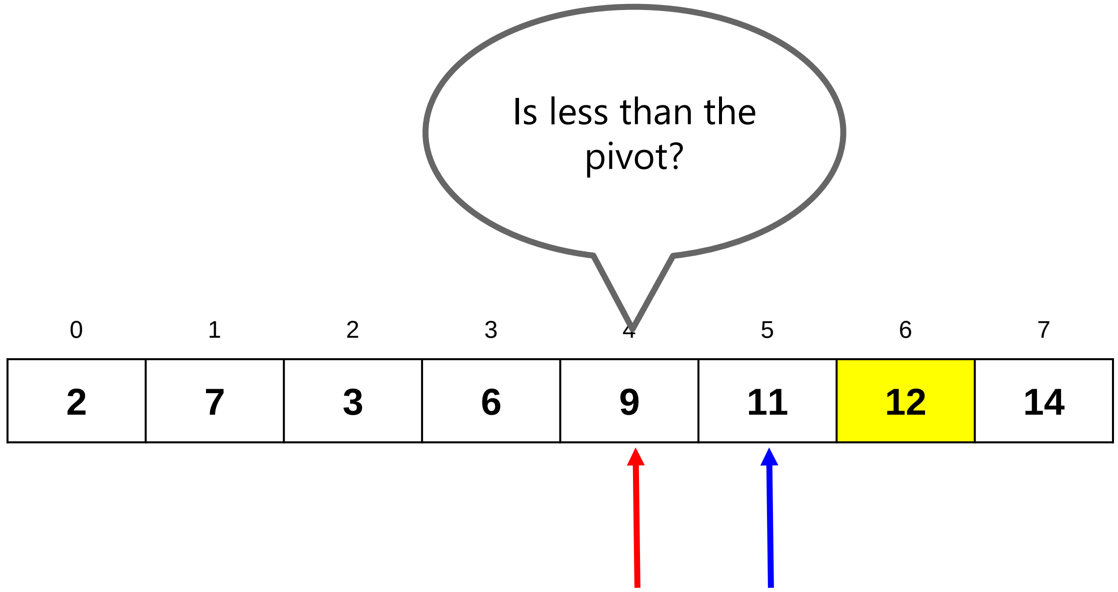


QuickSort
The solution



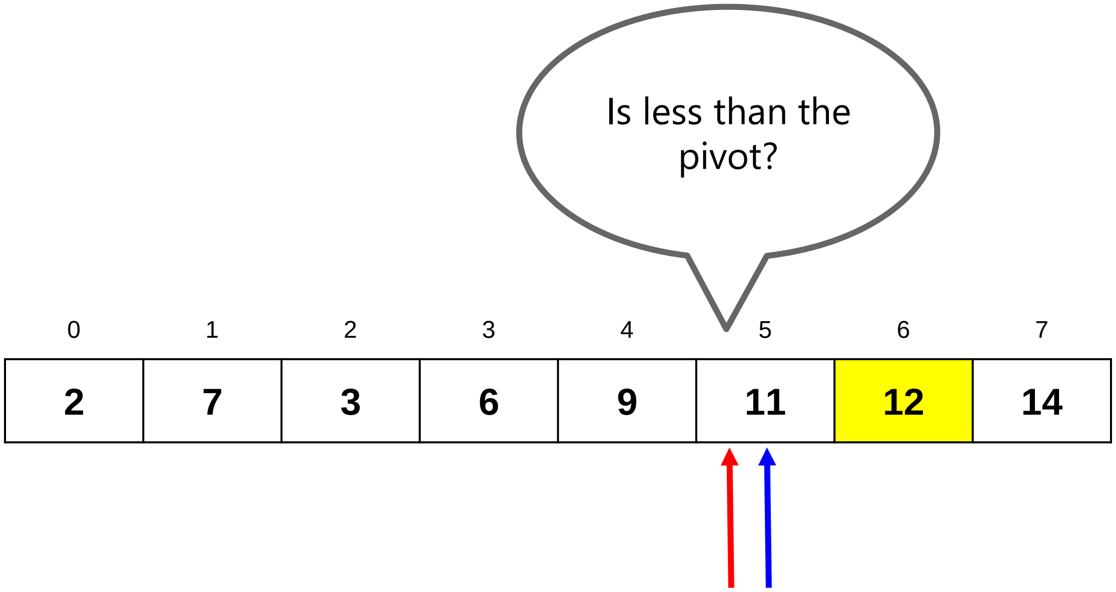
QuickSort

The solution



QuickSort

The solution



QuickSort

The solution

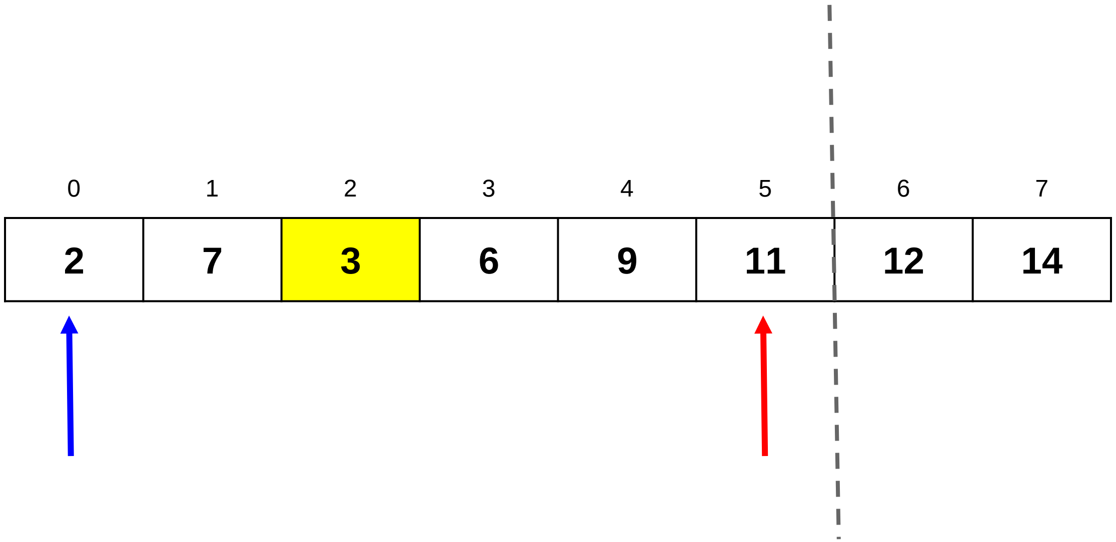
0	1	2	3	4	5	6	7
2	**7**	**3**	**6**	**9**	**11**	**12**	**14**

QuickSort
The solution

Recurse!

0	1	2	3	4	5	6	7
2	**7**	**3**	**6**	**9**	**11**	**12**	**14**

QuickSort
The solution

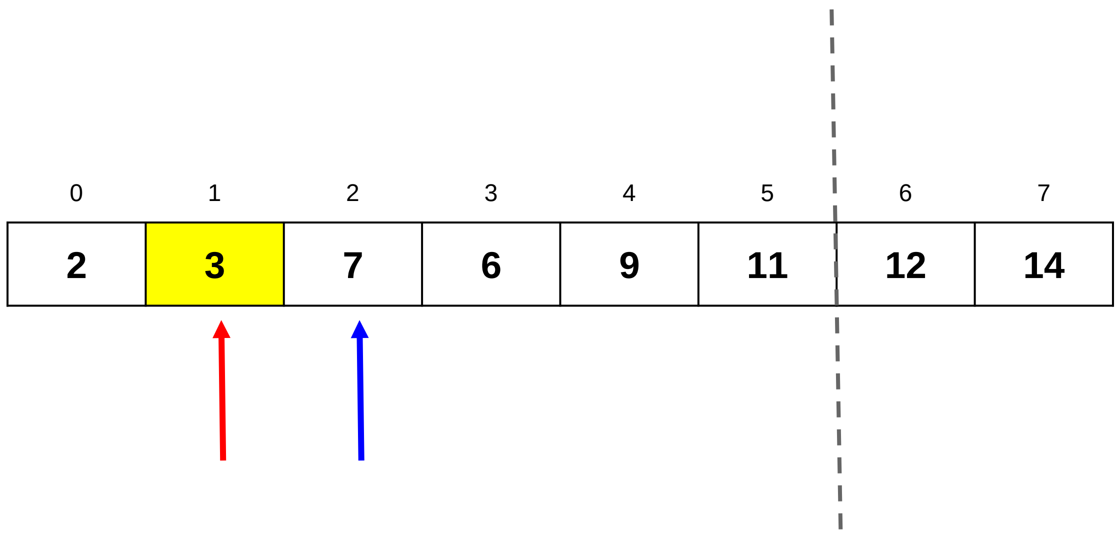


QuickSort

The solution

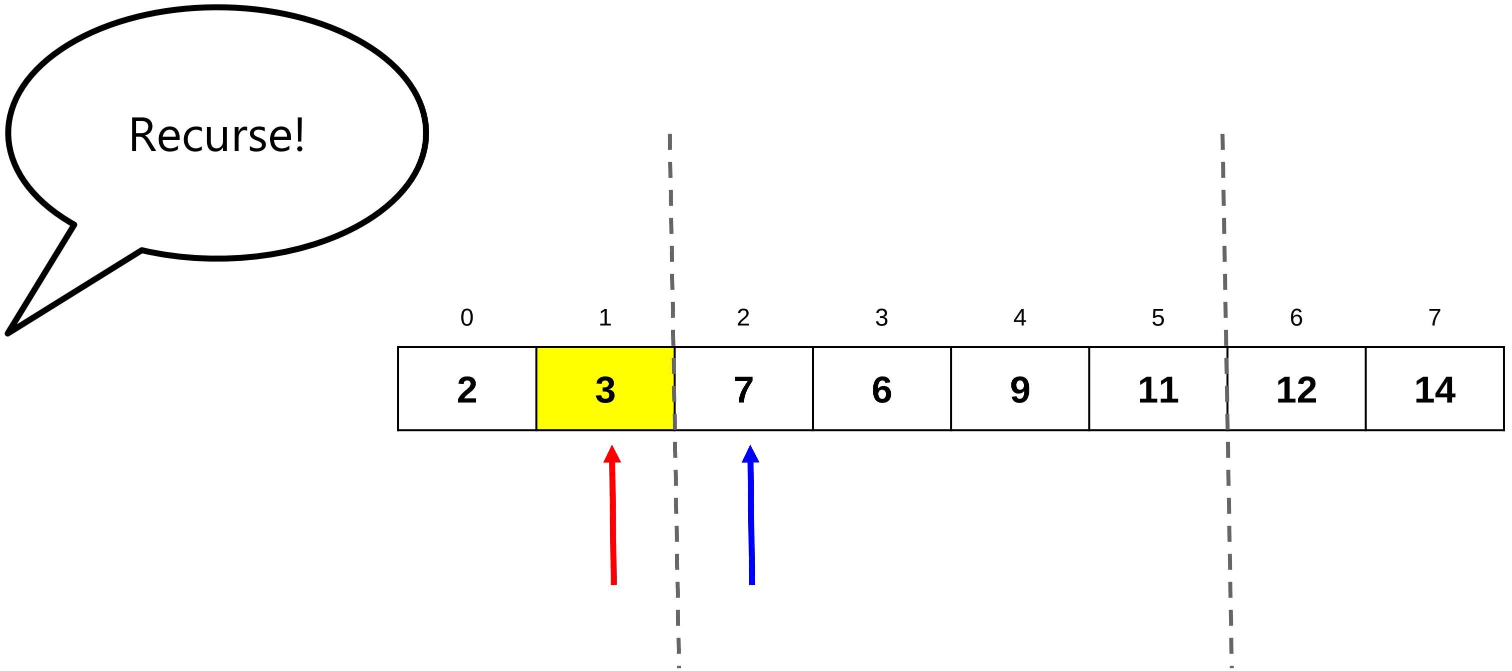
0	1	2	3	4	5	6	7
2	**7**	**3**	**6**	**9**	**11**	**12**	**14**

QuickSort
The solution



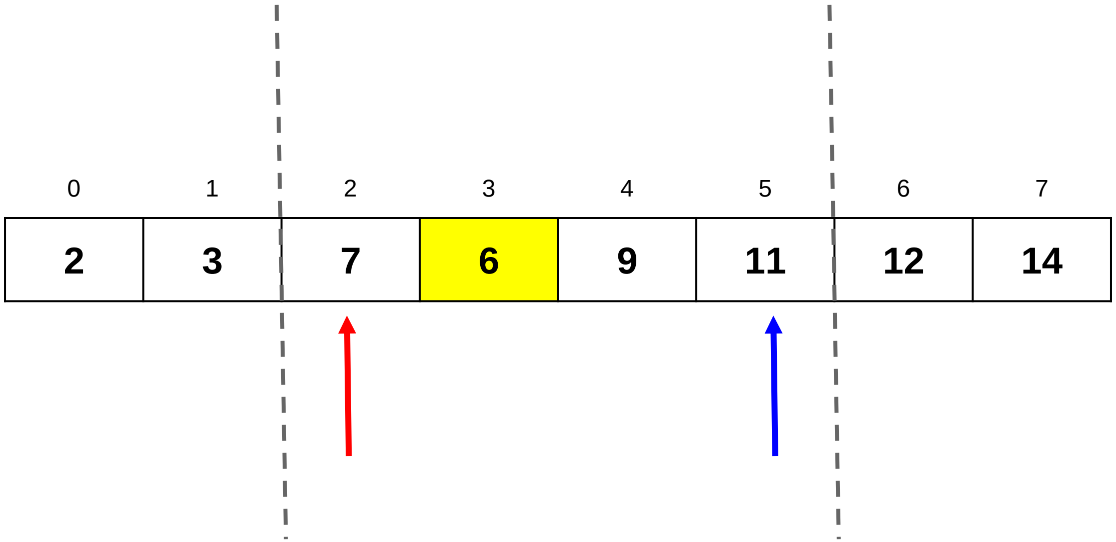
QuickSort

The solution



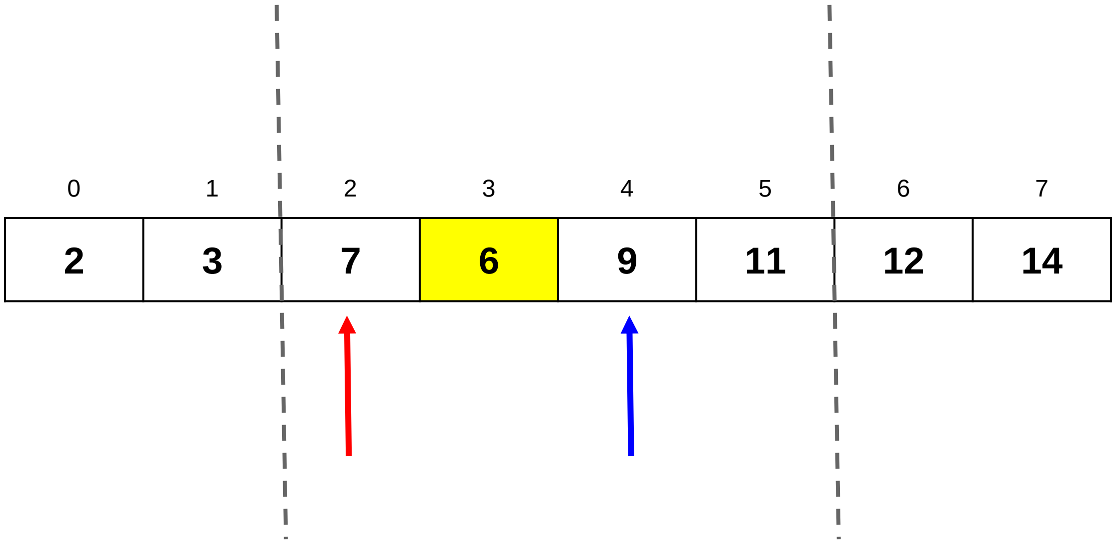
QuickSort

The solution



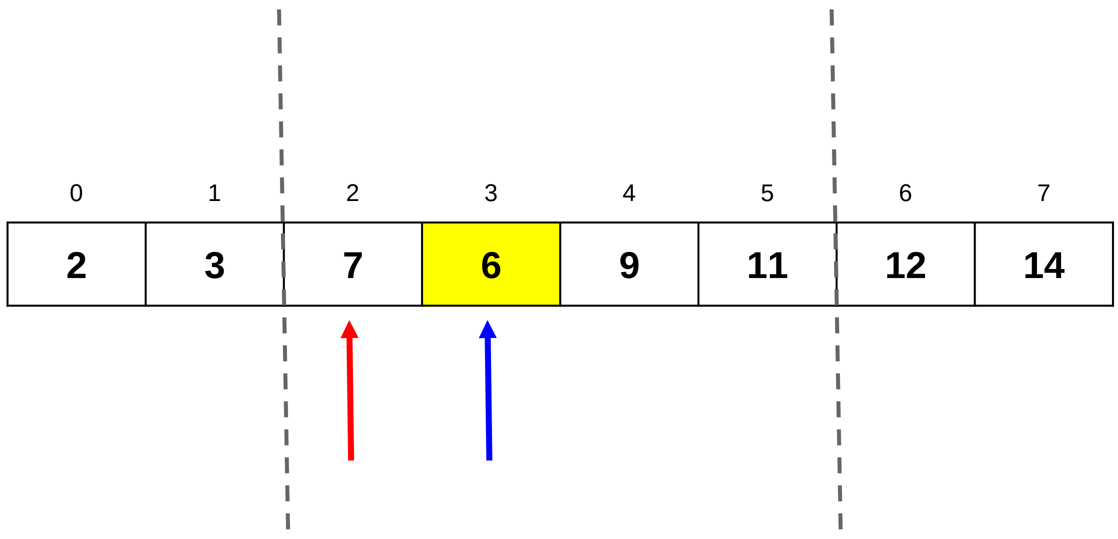
QuickSort

The solution



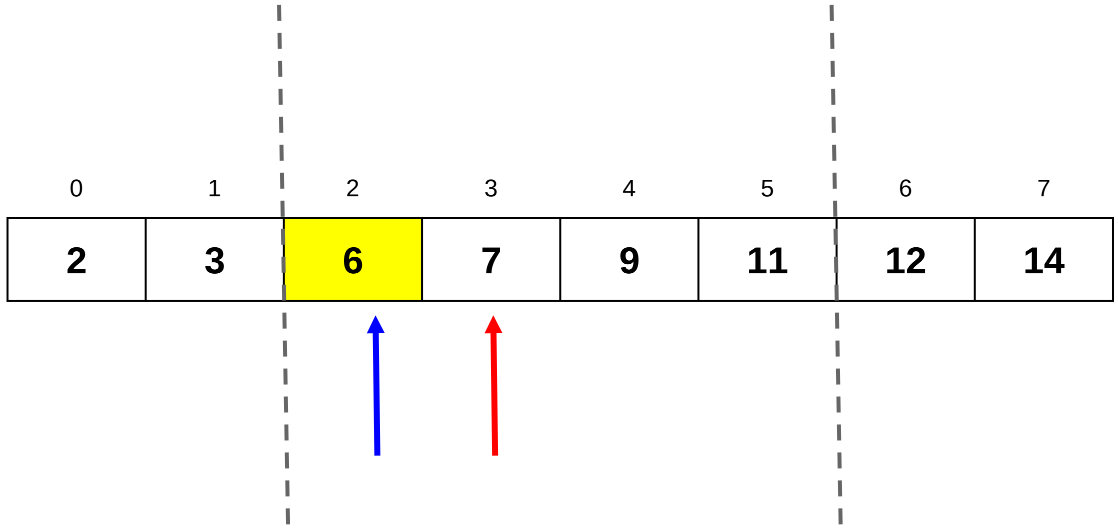
QuickSort

The solution



QuickSort

The solution



Quicksort

Implementing the algorithm

```java
public class Quicksort  {
  private int[] numbers;
  private int number;

  public void sort(int[] values) {
    // check for empty or null array
    if (values ==null || values.length==0){
      return;
    }
    this.numbers = values;
    number = values.length;
    quicksort(0, number - 1);
  }
```

Quicksort

Implementing the algorithm

```
private void quicksort(int low, int high) {
  int i = low, j = high;
  int pivot = numbers[low + (high - low) / 2];

  while (i <= j) {
    while (numbers[i] < pivot) {
      i++;
    }
    while (numbers[j] > pivot) {
      j--;
    }
    if (i <= j) {
      exchange(i, j);
      i++;
      j--;
    }
  }
```

Quicksort

Implementing the algorithm

```
    if (low < j)
      quicksort(low, j);
    if (i < high)
      quicksort(i, high);
  }
```

http://me.dt.in.th/page/Quicksort/

Quicksort

Analyzing the algorithm

➔ Same as MergeSort

➔ $O(N\log_2 N)$

Radix Sort

Sorting algorithm

Radix Sort

➔ Uses a different approach to the rest of algorithms

➔ Other algorithms see the key of each element as an atomic unit

➔ Radix sort disassembles the key into digits and arranges the data items according to the value of the digits

Radix Sort

- ➔ *Radix* means the base of a system of numbers
  - ◆ Ten is the radix of the decimal system (base-10)
  - ◆ Two is the radix of the binary system (base-2)

Radix Sort

Example problem

➔ Sort the following array:

170	45	75	90	802	24	2	66

Radix Sort

Example problem

170	45	75	90	802	24	2	66

Take the least significant digit (1s)

Radix Sort

The solution

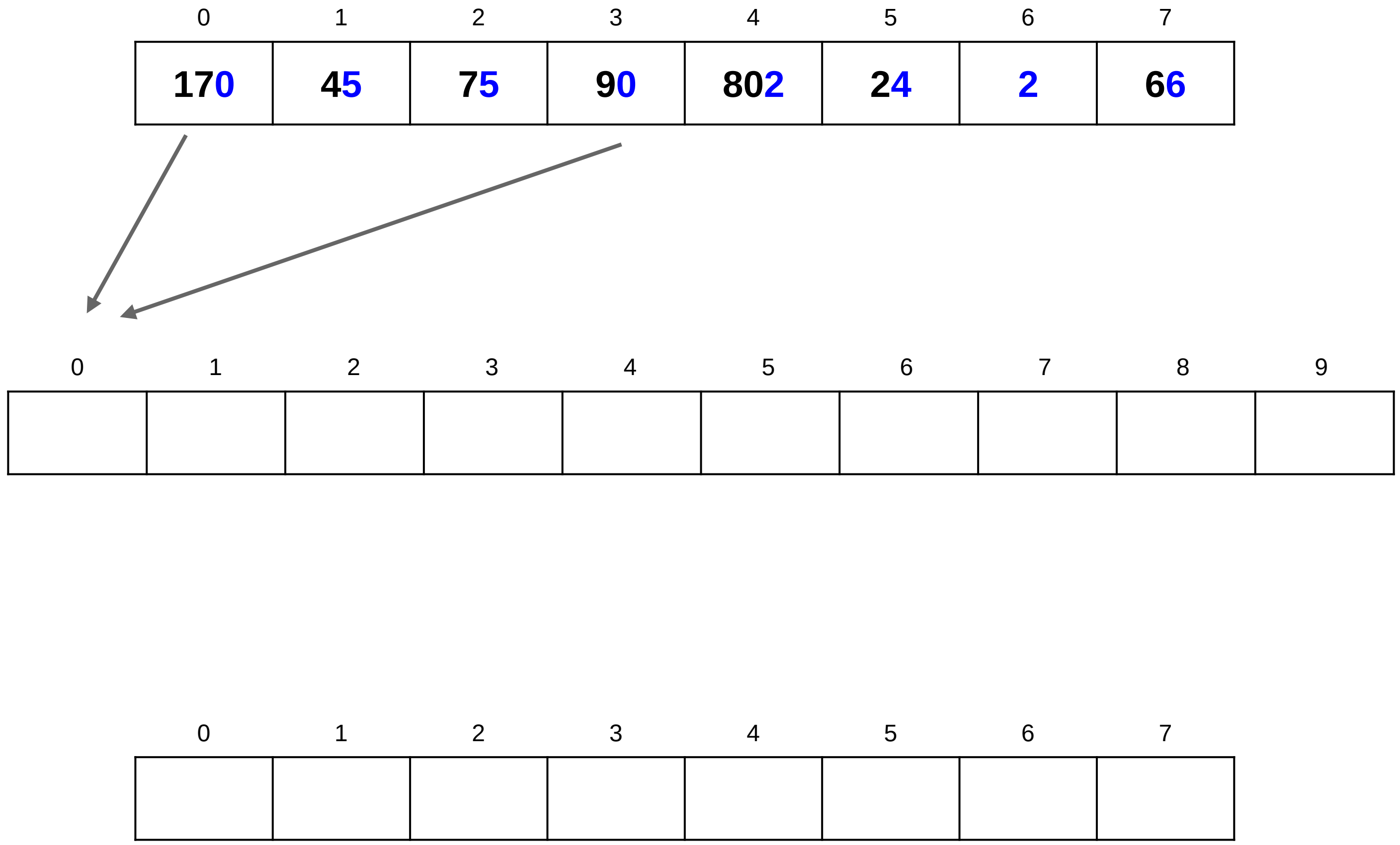
0	1	2	3	4	5	6	7
170	45	75	90	802	24	2	66

0	1	2	3	4	5	6	7	8	9

0	1	2	3	4	5	6	7

Radix Sort

The solution



Radix Sort

The solution

0	1	2	3	4	5	6	7
170	**45**	**75**	**90**	**802**	**24**	**2**	**66**

0	1	2	3	4	5	6	7	8	9
2	0	2	0	1	2	1	0	0	0

0	1	2	3	4	5	6	7

Radix Sort

The solution

0	1	2	3	4	5	6	7
170	**45**	**75**	**90**	**802**	**24**	**2**	**66**

0	1	2	3	4	5	6	7	8	9
2	2	4	4	5	7	8	8	8	8

Adjust count
for the
output

0	1	2	3	4	5	6

Radix Sort

The solution

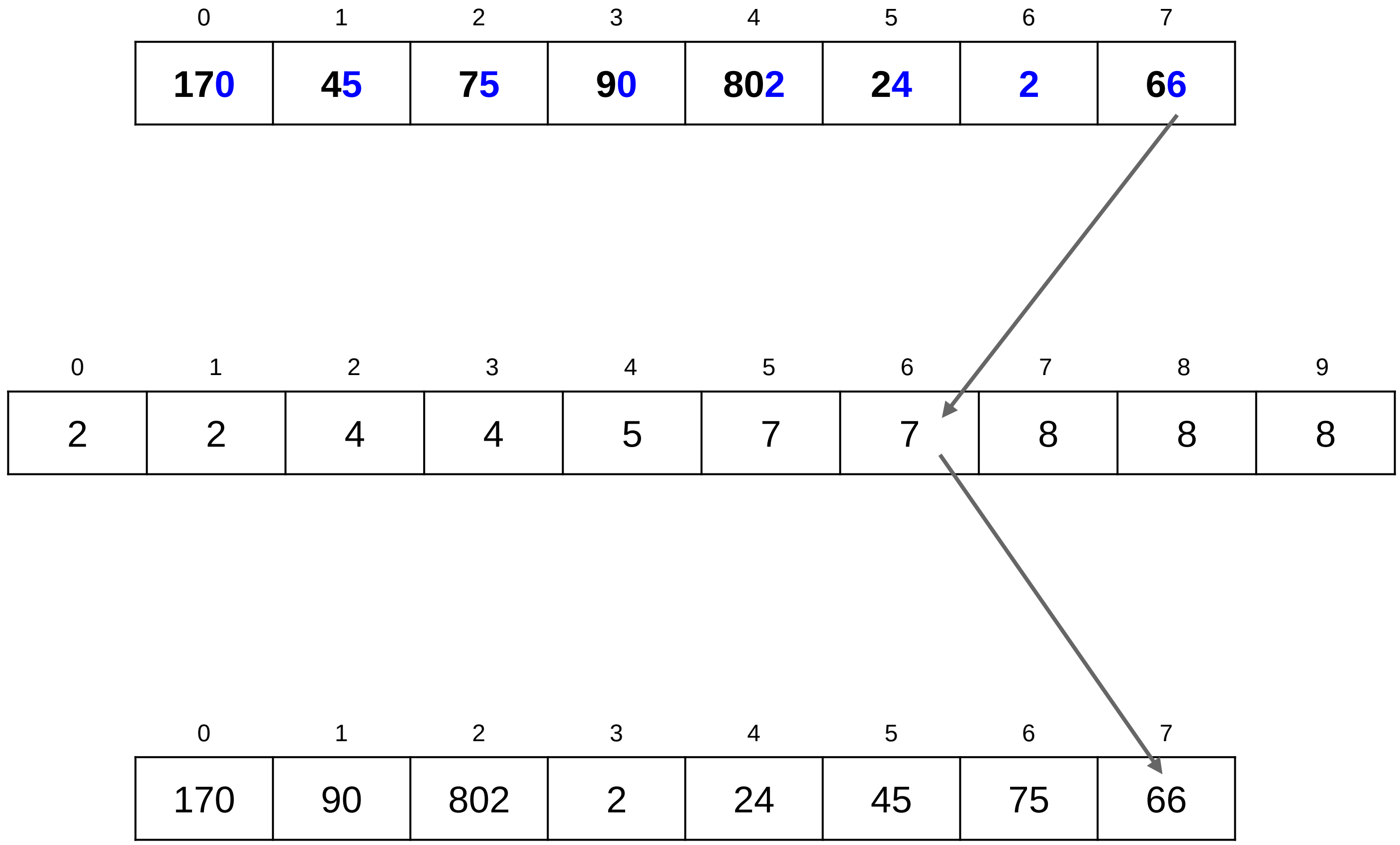
0	1	2	3	4	5	6	7
170	45	75	90	802	24	2	66

0	1	2	3	4	5	6	7	8	9
2	2	4	4	5	7	8	8	8	8

0	1	2	3	4	5	6	7

Radix Sort

The solution



Radix Sort

The solution

0	1	2	3	4	5	6	7
17**0**	4**5**	7**5**	9**0**	80**2**	2**4**	**2**	6**6**

0	1	2	3	4	5	6	7	8	9
1	2	4	4	5	7	7	8	8	8

0	1	2	3	4	5	6	7
1**7**0	**9**0	8**0**2	**0**2	**2**4	**4**5	**7**5	**6**6

0	1	2	3	4	5	6	7	8	9
2	2	3	3	4	4	5	7	7	8

0	1	2	3	4	5	6	7
802	**0**02	**0**24	**0**45	**0**66	**1**70	**0**75	**0**90

0	1	2	3	4	5	6	7	8	9
6	7	7	7	7	7	7	7	8	8

0	1	2	3	4	5	6	7
002	024	045	066	075	090	170	802

Radix Sort

Implementing the algorithm

```
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit.
    // Note that instead of passing digit number,
    // exp is passed. exp is 10^i where i is current
    // digit number
    for (int exp = 1; m/exp > 0; exp *= 10) {
        countSort(arr, n, exp);
    }
}
```

Radix Sort

Implementing the algorithm

```
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit.
    // Note that instead of passing digit number,
    // exp is passed. exp is 10^i where i is current
    // digit number
    for (int exp = 1; m/exp > 0; exp *= 10) {
        countSort(arr, n, exp);
    }
}
```

Radix Sort

Implementing the algorithm

```
void countSort(int arr[], int n, int exp) {
    int output[n]; // output array
    int i, count[10] = {0};

    for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;

    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    for (i = n - 1; i >= 0; i--) {
        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

Radix Sort

Analyzing the algorithm

→ Same as QuickSort : $O(NLog_2N)$

→ But uses twice as much memory as quickSort

Binary Search

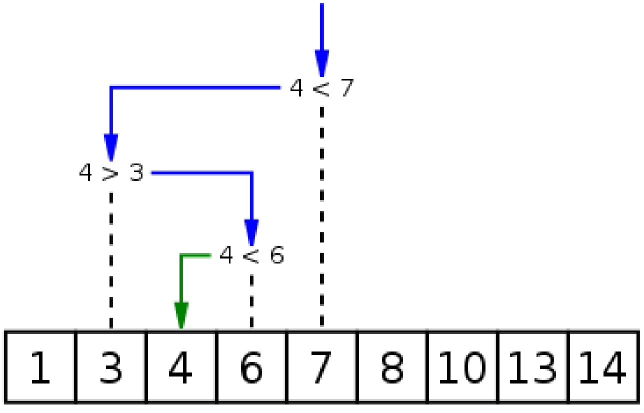
Search algorithm

Binary Search

➔ Finds a number in a **sorted** array

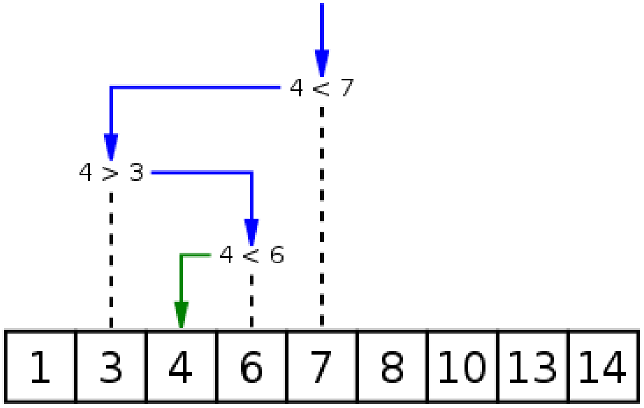
➔ Compares the input element with the **middle** of the array.

➔ If the input element matches, return the index to that element



Binary Search

- ➔ Otherwise, runs the algorithm with the right or left **subarray**, depending if the input key is greater or less than the middle element.
- ➔ Array must be sorted
- ➔ O(log (N))



Binary Search

Implementing the algorithm

```
int binarySearch(int pArray[], int pKey,
                 int pIndexMin, int pIndexMax) {
   while (pIndexMax >= pIndexMin) {
     int middle = (int)((pIndexMax + pIndexMin) / 2);

     if (pArray[middle] < pkey)
        pIndexMin = middle + 1;
     else if (pArray[middle] > pKey)
        pIndexMax = middle - 1;
     else
        return middle;
   }
   return -1;
}
```

Interpolation Search

- ➔ Modification of Binary Search
- ➔ In each step tries to calculate where the number might be.
- ➔ Based on the idea of looking for a person in the phonebook. If you're looking for Bob, you know it should be at the beginning…
- ➔ O(n)

Interpolation Search

Implementing the algorithm

```
middle = low + ((number - array[low]) * (high - low))
       / (array[high] - array[low]);
```

Interpolation Search

Implementing the algo

Rest of the code is equal to binary search

```
middle = low + ((number - array[low]) * (high - low))
        / (array[high] - array[low]);
```

Interpolation Search

Implementing the algorithm

```
middle = low + ((number - array[low]) * (high - low))
        / (array[high] - array[low]);
```

5	6	9	11	15	18	20	25	28	39

Let's search for 28

Interpolation Search

Implementing the algorithm

```
middle = low + ((number - array[low]) * (high - low))
        / (array[high] - array[low]);
```

5	6	9	11	15	18	20	25	28	39

The middle will be: 6

Interpolation Search

Implementing the algorithm

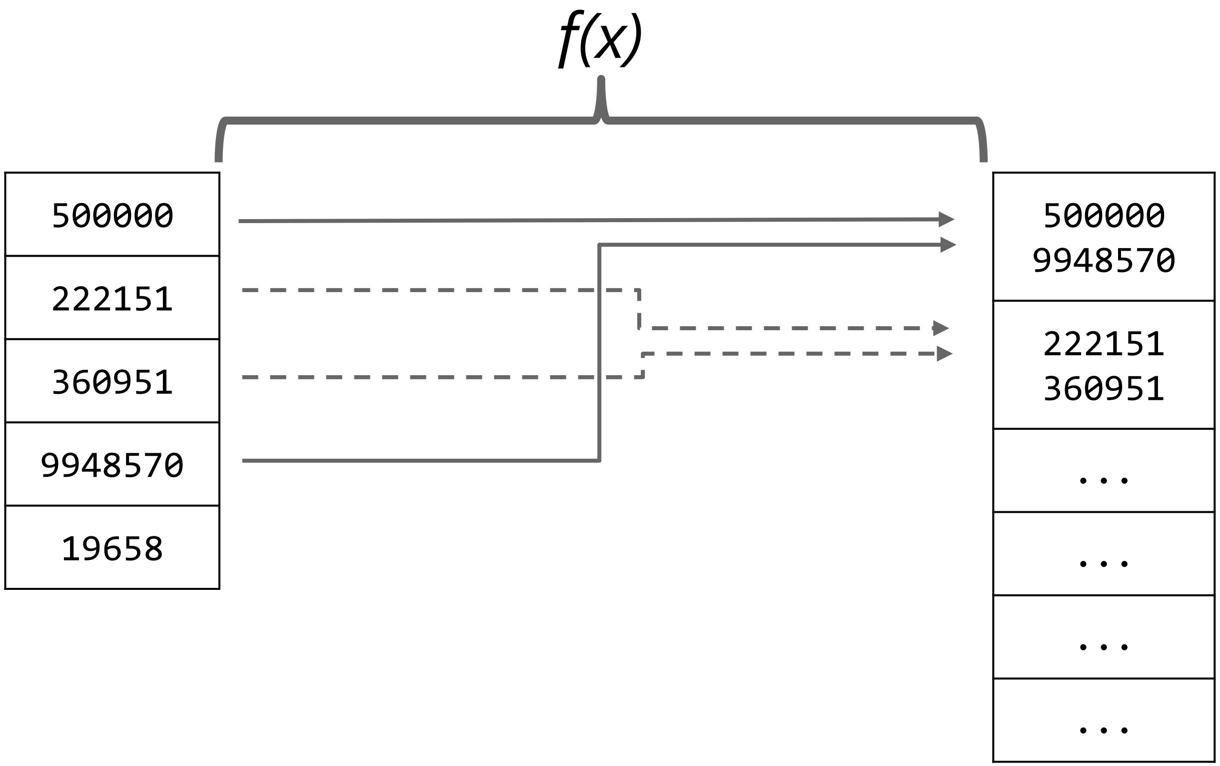
```
middle = low + ((number - array[low]) * (high - low))
          / (array[high] - array[low]);
```

5	6	9	11	15	18	20	25	28	39

Hash Search

Search algorithm

Hashing



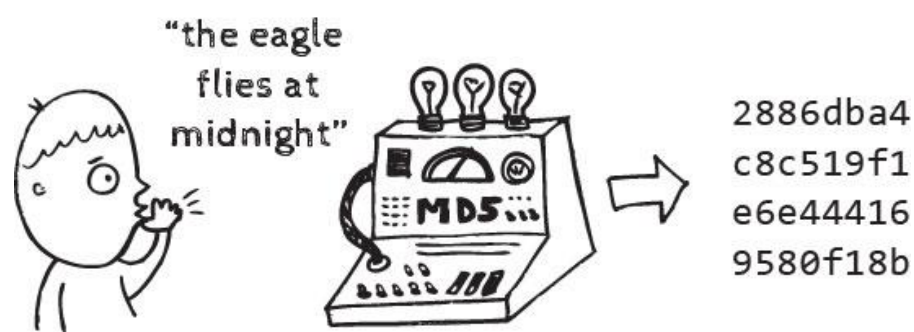
f(x)
500000
222151
360951
9948570
19658
500000
9948570
222151
360951
...
...
...
...

Hashing

➔ Maps large sets of data to small sets.

➔ It's a fast search method

➔ Hash Function allows find and assign an index to a key value.

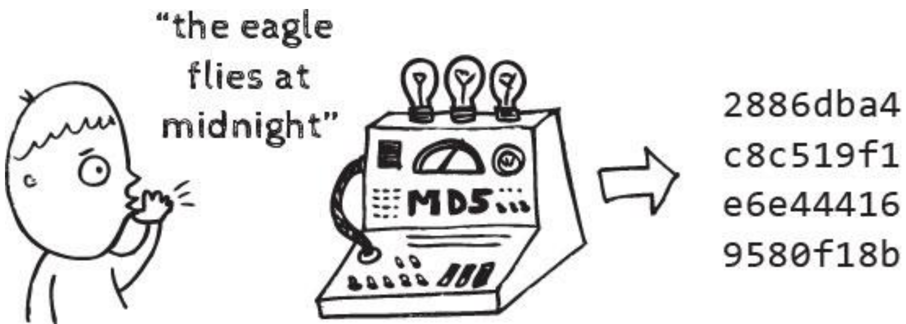


Hashing

➔ It can map several keys to the same index

➔ Each slot in the hash table has assigned a set of data

➔ Each slot is called bucket

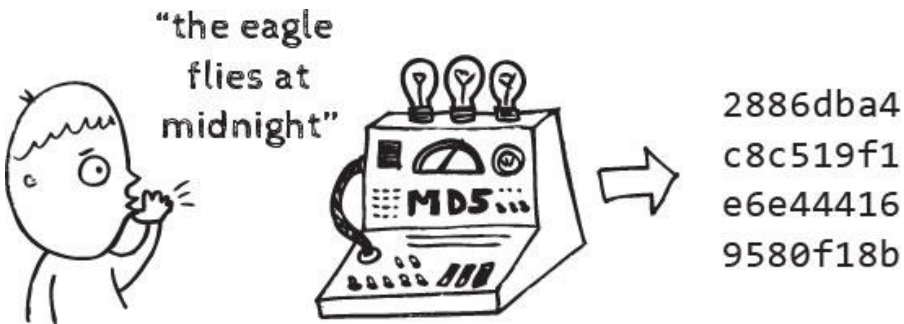


Hashing

- ➔ Transforms keys to indexes

- ➔ The basic hash function for numbers is the **identity function**. It is not used.

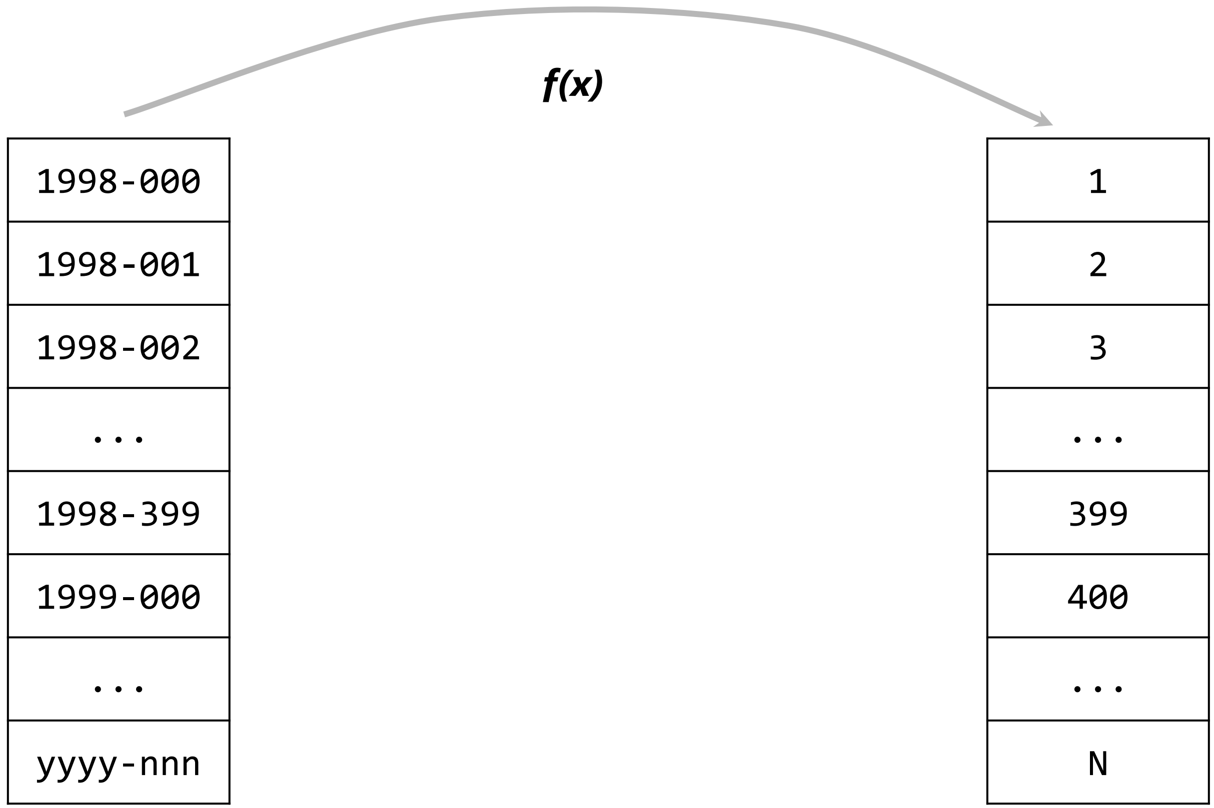
- ➔ Ideal function is a bijective function or injective function. These kind of functions are not used.



Hash Functions

Successive substractions

➔ What is the function?



Hash Functions

Successive substractions

➔ What is the function?

f(x)

1998-000	1998**000** - 1998**000**	0
1998-001	1998**001** - 1998**000**	1
1998-002	1998**002** - 1998**000**	2
...	...	...
1998-399	1998**399** - 1998**000**	399
1999-000	(1999**000** - 1998**000**) + **399+1**	400
...	...	...
yyyy-nnn	yyyy**nnn** - 1998**000** + (**400** * (**yyyy-1998**))	N

Hash Functions

Modular Arithmetic

- ➔ Use a prime Number
- ➔ Index is the residue of divide the key between a number (module).
- ➔ The number defines the amount of buckets of the hash table.

Hash Functions

Modular Arithmetic

f(x)

13000000	13000000 **mod 13**	0
12345678	12345678 mod 13	7
13602499	13602499 mod 13	1
71140205	71140205 mod 13	6
73062138	73062138 mod 13	6

Mid-Square Method

➔ Squares the key value

➔ Takes the middle *r* digits of the result

➔ It gives a value between 0 and (2^*r*) -1

Hash Functions

Mid-Square Method

f(x)

123	123 * 123 = 1**51**29	**51**
136	136 * 136 = 1**84**96	**84**
730	730 * 730 = 53**29**00	**29**
301	301 * 301 = 9**06**01	**06**
625	625 * 625 = 39**06**25	**06**

Hash Functions

Truncation method

- ➔ Ignore part of the key and use the rest as the array index.

- ➔ You don't need to get successive numbers

	f(x)	
13000000	**1**3**0**0000**0**	100
12345678	**1**2**3**4567**8**	138
13602499	**1**3**6**0249**9**	169
71140205	**7**1**1**4020**5**	715
73162135	**7**3**1**6213**5**	715

Folding method

➔ Divide the key in parts

➔ Combine this parts (might be using operator / * + -)

➔ For example, divide a number of 8 digits in groups of 3 digits and sum this groups

Hash Functions

Folding method

f(x)

13000000	130 + 000 + 00 = 0**130**	130
12345678	123 + 456 + 78 = 0**657**	657
13602499	711 + 402 + 05 = 1**118**	118
71140205	136 + 024 + 99 = 0**259**	259
73162135	250 + 000 + 09 = 0**259**	259

What's the problem of hash tables?

Hash Functions

Collisions

- ➔ Collisions are practically unavoidable
- ➔ Two or more keys with the same index.
- ➔ Wrong choice of hash function

Collisions

➔ Almost all the hash slots remaining are empty while a few are full and present a lot of collisions.

➔ Small hash table and too much keys to be sorted.

➔ Collisions Treatment in some cases is very expensive.

Selection & Ordering Algorithms

CE-1103 Algorithms and Data Structures