Searching

L.EIC

Algoritmos e Estruturas de Dados

2023/2024

AP Rocha, P Diniz

A Costa, B Leite, F Ramos, J Pires, PH Diniz, V Silva

The Search problem

Problem: given an array ***v*** storing ***n*** elements, and a target element ***el***, locate the position in ***v*** (if it exists) where $v[i] = el$

– variants for the case of arrays with repeated values:
  a) indicate the position of the first occurrence
  b) indicate the position of the last occurrence
  c) indicate the position of any occurrence

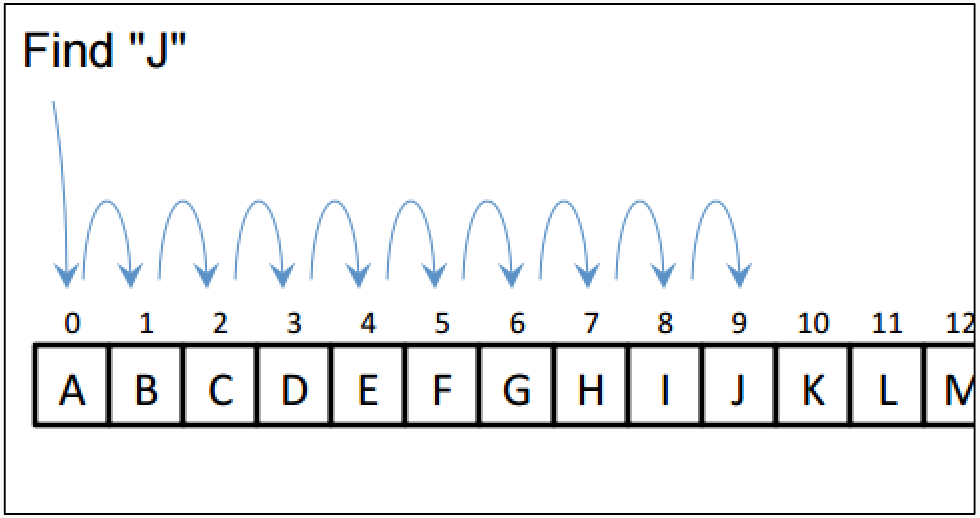
– when the target ***el*** does not exist, return an undefined position, such as *-1*

Sequential Search

- <u>Algorithm</u> *(sequential search)*

  sequentially checks each element of the array, from the first to the last [a] or from the last to the first [b], until a match is found or the end of the array is reached

  - [a] if you want to know the position of the first occurrence
  - [b] if you want to know the position of the last occurrence



suitable for unordered or small arrays

Sequential Search

variant a)

```
/* Search for an element el in a vector v of comparable elements
with the comparison operators. Returns the index of the first
occurrence of el in v, if found; otherwise, returns -1 */

template <class T>
int SequentialSearch(const vector<T> &v, T el)
{
   for (unsigned i = 0; i < v.size(); i++)
      if (v[i] == el)
         return i; // found

   return -1; // not found
}
```

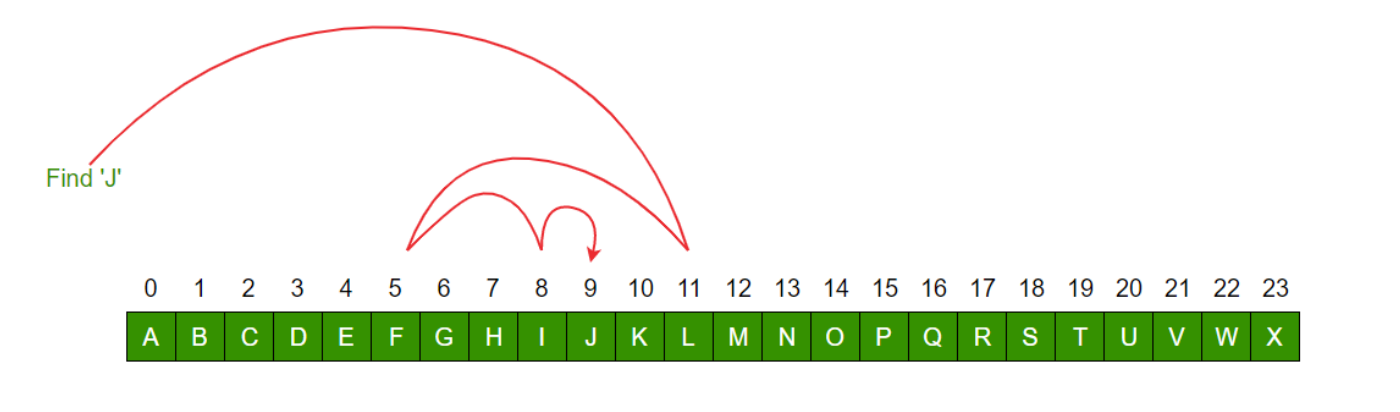
Sequential Search complexity

- Sequential Search time complexity
  - the operation performed most often is the test "`if (v[i] == el)`", at most **n+1** times (in case it doesn't find the target element).
  - if the target element exists in the vector, the test is performed approximately **n/2** times on average (1 time in the best case)
  - ***T(n) = O(n)*** in worst case and average case

- Sequential Search space complexity
  - space on local variables (including arguments)
  - since vectors are passed "by reference", the space taken up by the local variables is constant and independent of the vector size.
  - ***S(n) = O(1)***

Searching in sorted arrays

- Suppose the array is ordered (arranged in increasing or non-decreasing order)
  - Sequential search on a sorted array still yields the same analysis $T(n) = O(n)$
  - Can exploit sorted structure by performing **binary search**
  - *Strategy:* inspect middle of the structure so that half of the structure is discarded at every step



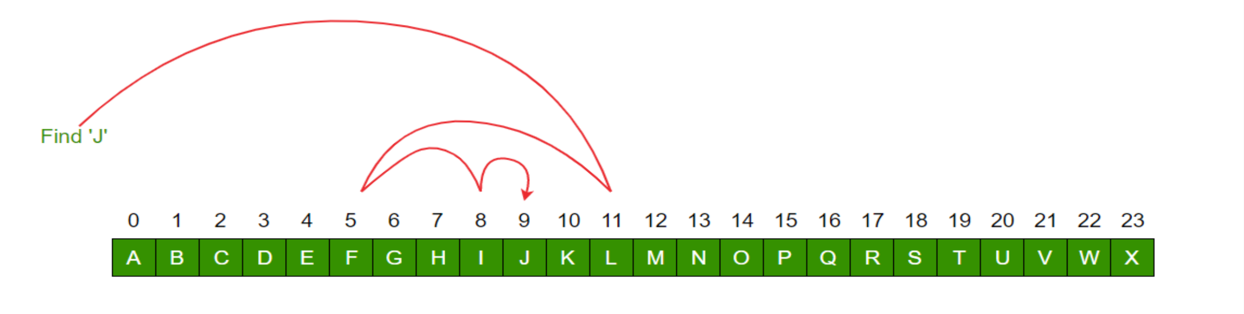
Binary Search

- <u>Algorithm</u> *(binary search)*

compares the element in the middle of the array with the target element:

- is equal to the target element → found
- is greater than the target element → continue searching (in the same way) in the sub-array to the left of the inspected position
- is less than the target element → continue searching (in the same way) in the sub-array to the right of the inspected position

if the sub-array to be inspected reduces to an empty vector, it is concluded that the target element does not exist

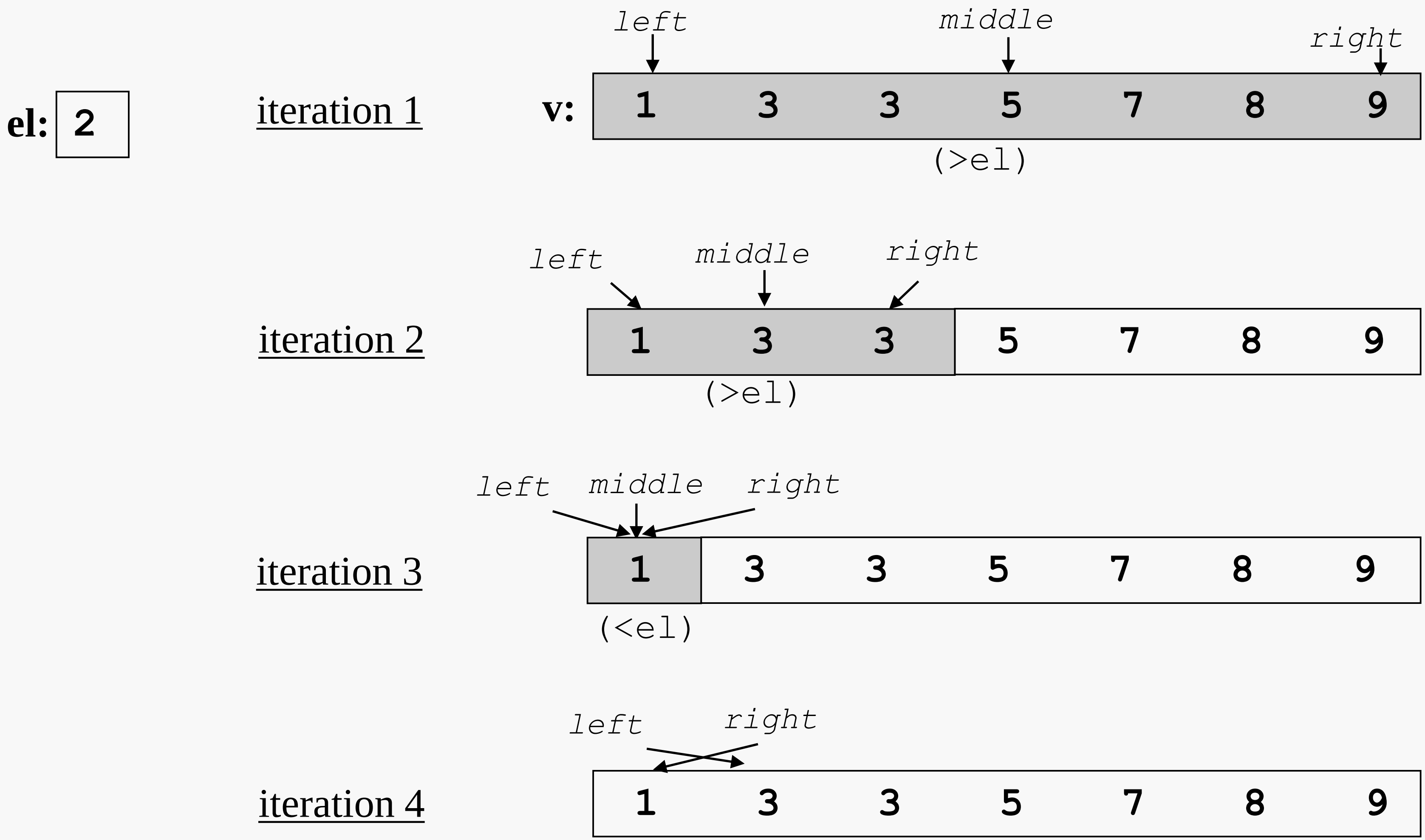


Binary Search

```
/* Search for an element el in an ordered vector v of comparable elements
with the comparison operators. Returns the index of one occurrence of el
in v, if found; otherwise returns -1. */

template <class T>
int BinarySearch(const vector<T> &v, T el)
{
    int left = 0, right = v.size() - 1;
    while (left <= right)
    {
        int middle = (left + right) / 2;
        if (v[middle] < el)
            left = middle + 1;
        else if (el < v[middle])
            right = middle - 1;
        else
            return middle; // found
    }
    return -1; // not found
}
```

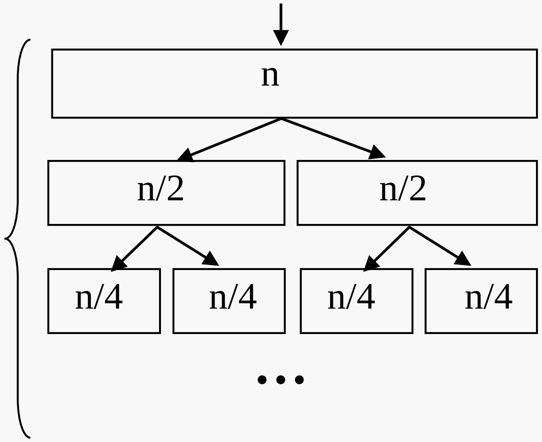
Binary Search



sub-array is empty → element 2 does not exist!

Binary Search complexity

- Binary Search time complexity
  - in each iteration, the size of the sub-vector to be analyzed is divided by ≈ 2
  - after **k** iterations, the size of the sub-vector to analyze is approximately $\mathbf{n/2^k}$
  - if the target element does not exist in the vector, the cycle only ends when



$$n/2^k \approx 1 \;\rightarrow\; n \approx 2^k \;\rightarrow\; k \approx \log_2 n$$

  - so, in the worst case, the number of iterations is $k \approx \log_2 n$, ***T(n) = O(log n)***

- Binary Search space complexity

S(n) = O(1)

The Painter's Partition Problem: using binary search

Problem

- there are paint ***n*** boards of length $\{l_1, l_2 \ldots l_n\}$ and there are **k** painters available
- each painter takes 1 unit of time to paint 1 unit of the board
- any painter will only paint continuous sections of boards
- the problem is to find the minimum time to get this job done

example: ***k*** = 2, ***board*** = [10, 20, 30, 40]

algorithm:

- apply binary search on the search space and
- according to the problem reduce the search space which will finally give the final result

The Painter's Partition Problem: using binary search

example: ***k*** = 2, ***board*** = [10, 20, 30, 40]

- Search space is the maximum range where the answer contains:
  - the maximum time will be (10+20+30+40) = 100
  - the minimum time will be 40
- the search space will be [40 – 100]

40	41	42	. . . .	98	99	100

- divide the search space, *middle* = 40 + (100 - 40) / 2 = 70

40	41	42	...	55	...	68	69	70

  - assume that no painter will paint more than 70 units of the board
  - how many painters will be required? ***k****=2* is enough? yes, so the search space will be reduced and will change to [40, 70]
- …

The Painter's Partition Problem: using binary search

```
int partition(vector<int> &board, int k)
{
    int n = board.size(), s = 0, m = 0;
    for(int i = 0; i < n; i++)
    {
        m = max(m, board[i]);
        s += board[i];
    }

    int low = m, high = s;
    while (low < high)
    {
        int mid = low + (high - low) / 2;
        int painters = findkp(board, mid);

        if (painters <= k) high = mid;
        else low = mid + 1;
    }
    return low;
}
```

The Painter's Partition Problem: using binary search

```
int findkp(vector<int> &board, int atmost)
{
    int n = board.size();
    int s = 0, painters = 1;

    for (int i = 0; i < n; i++)
    {
        s += board[i];
        if (s > atmost)
        {
            s = board[i];
            painters++;
        }
    }
    return painters;
}
```

STL algorithms

- Sequential Search in vectors

  iterator find(iterator start, iterator end, const T& val);

  – looks for first occurrence of an element identical to *val* in *[start, end[* (comparison performed by operator ==)
    - success, returns iterator for the found element
    - no success, returns iterator to “the end” of the vector (*v.end()*)

  iterator find_if(iterator start, iterator end, Predicate pred);

  – looks for first occurence for which unary predicate *pred* is true in *[start, end[*

STL algorithms

- Binary Search in vectors

  bool binary_search(iterator start, iterator end, const T& val);

  - looks for one occurrence of an element identical to *val* in *[start, end[*
  - uses operator <

  bool binary_search(iterator start, iterator end, const T& val, Compare comp);

  - looks for one occurrence of an element identical to *val* in *[start, end[*
  - uses predicate *comp* (*comp* compares two elements)

Example

```
class Person {
   string cc;
   string name;
   int age;
public:
   Person (string c, string nm="", int a=0);
   bool operator < (const Person & p2) const;
   bool operator == (const Person & p2) const;
   // …
};
```

```
bool Person::operator < (const Person & p2) const {
   return name < p2.name;
}

bool Person::operator == (const Person & p2) const {
   return cc == p2.cc;
}
```

Example

```
bool isTeenager(const Person &p1)  {
   return p1.getAge() <= 20;
}

bool younger(const Person &p1, const Person &p2)  {
   if (p1.getAge() < p2.getAge()) return true;
   else return false;
}


template <class T> void writeVector(vector<T> &v)  {
   for (val:v)
      cout << val << endl;
   cout << endl;
}
```

Example

```
int main()
{
  vector<Person> vp;
  vp.push_back(Person("6666666","Rui Silva",34));
  vp.push_back(Person("7777777","Antonio Matos",24));
  vp.push_back(Person("1234567","Maria Barros",20));
  vp.push_back(Person ("7654321","Carlos Sousa",18));
  vp.push_back(Person("3333333","Fernando Cardoso",33));

  cout << “initial vector:" << endl;
  writeVector(vp);
```

```
initial vector:
cc: 6666666, name: Rui Silva, age: 34
cc: 7777777, name: Antonio Matos, age: 24
cc: 1234567, name: Maria Barros, age: 20
cc: 7654321, name: Carlos Sousa, age: 18
cc: 3333333, name: Fernando Cardoso, age: 33
```

Example

```
Pessoa px("7654321");
vector<Person>::iterator it = find(vp.begin(),vp.end(),px);
if (it == vp.end())
   cout << px << " does not exist in vector" << endl;
else
   cout<< px << " exists in vector as:" << *it <<endl;
```

cc: 7654321, name: , age: 0 exists in vector as
cc: 7654321, name: Carlos Sousa, age: 18

```
it = find_if(vp.begin(),vp.end(),isTeeenager);
if (it == vp.end())
  cout << "there is no teenager in the vector" << endl;
else
  cout << "teenager found " << *it << endl;
```

teenager found cc: 1234567, name: Maria Barros, age: 20

Example

```
// note that vector vp2 is sorted by age
vector<Person> vp2;
vp2.push_back(Person("7654321","Carlos Sousa",18));
vp2.push_back(Person("1234567","Maria Barros",20));
vp2.push_back(Person("7777777","Antonio Matos",24));
vp2.push_back(Person("3333333","Fernando Cardoso",33));
vp2.push_back(Person("6666666","Rui Silva",34));

Person py("xx","xx",24);
bool exist = binary_search(vp2.begin(),vp2.end(),py,younger);
if (exist == true)
   cout << "there is a person aged " << py.getIdade() << endl;
else
   cout << "there is no person aged " << py.getIdade()<<  endl;
```

there is a person aged 24

Sorting

L.EIC

Algoritmos e Estruturas de Dados

2023/2024

AP Rocha, P Diniz

A Costa, B Leite, F Ramos, J Pires, PH Diniz, V Silva

The Sorting problem

Problem: given an array $\boldsymbol{v}$ storing $\boldsymbol{n}$ elements, rearrange these elements in ascending order (or rather, in non-descending order, because there may be repeated elements)

- there are several sorting algorithms with complexity $\boldsymbol{O}(\boldsymbol{n}^2)$ that are very simple (Insertion Sorting, BubbleSort, …)

- there are sorting algorithms that are more difficult to code with complexity $\boldsymbol{O}(\boldsymbol{n} \times \boldsymbol{log}\, \boldsymbol{n})$

Sorting

(some) Algorithms

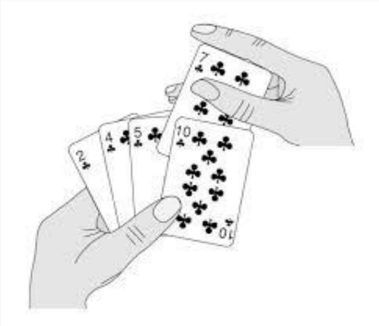
Comparative algorithms

- InsertionSort - $O(n^2)$
- SelectionSort - $O(n^2)$
- BubbleSort - $O(n^2)$
- MergeSort - $O(n \times log\, n)$
- QuickSort - $O(n \times log\, n)$
- HeapSort - $O(n \times log\, n)$, to study later

Non-comparative algorithms

- CountingSort
- RadixSort

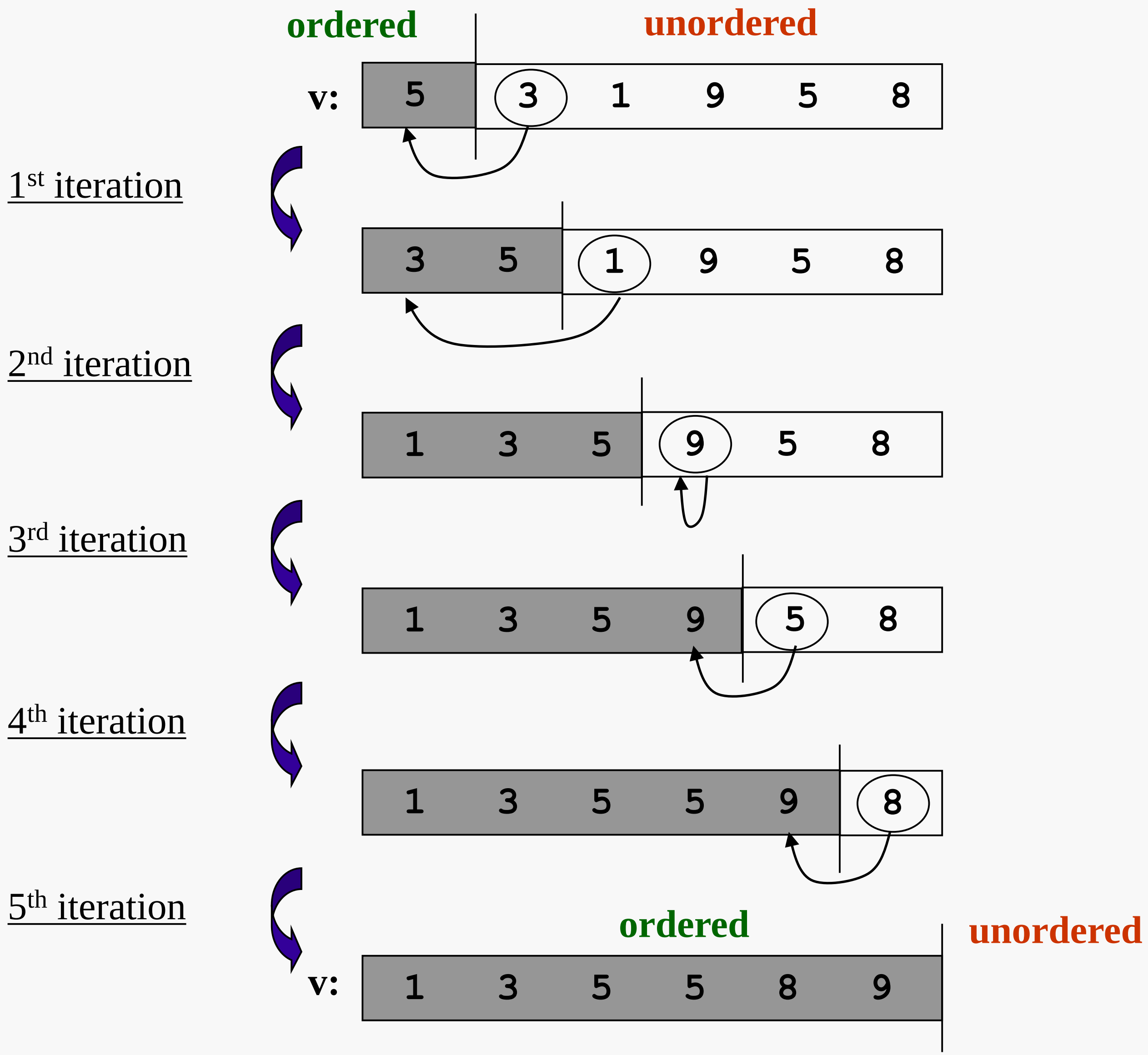
InsertionSort



Idea: insert each element in the correct position

- Algorithm (sort by insertion):
  - considers the vector divided into two sub-vectors (left and right)
    - the left sub-vector is ordered and
    - the right sub-vector is unordered
  - starts with just an element in the left sub-vector
  - moves one element at a time from the right sub-vector to the left sub-vector, placing it in the correct position to keep the left sub-vector sorted
  - ends when the right sub-vector is empty

InsertionSort



InsertionSort

```
// Sorts vector v
// Comparable: must have copy constructor,
//      assignment operator (=) and comparison operator (<)

template <class Comparable>
void insertionSort(vector<Comparable> &v) {
    for (unsigned p = 1; p < v.size(); p++) {
        Comparable tmp = v[p];
        unsigned j;
        for (j = p; j > 0 && tmp < v[j-1]; j--)
            v[j] = v[j-1];
        v[j] = tmp;
    }
}
```

see animation in [VisuAlgo]

InsertionSort complexity

- InsertionSort time complexity
  - the number of iterations of the for interior cycle is:
    - *1*, best case
    - *n*, worst case
    - *n/2*, average case
  - the total number of iterations of the for exterior cycle is:
    - best case, $1 + 1 + \ldots + 1 = n - 1 \approx n$
    - worst case, $1 + 2 + \cdots + n - 1 = (n-1) \times (1 + n - 1)/2 = n \times (n-1)/2 \approx n^2/2$
    - average case, $\approx n^2/4$

$$T(n) = O(n^2)$$

- InsertionSort space complexity
  - space taken up by the local variables is constant and independent of the vector size

$$S(n) = O(1)$$

SelectionSort

Idea: select minimum

look for the minimum and put in his position

- Algorithm (probably the most intuitive)
  - find the minimum of the vector
  - swap with the first element
  - continue for the rest of the vector (excluding the first one)

SelectionSort

```
template <class Comparable>
void selectionSort(vector<Comparable> &v) {
    for (unsigned i = 0; i < v.size()-1; i++) {
        unsigned imin = i;
        for (j = i+1; j < v.size(); j++)
            if (v[j] < v[imin])
                imin = j;
        swap(v[i], v[imin]);
    }
}
```

- SelectionSort time complexity
  - two nested cycles, each can have *n* iterations $T(n) = O(n^2)$

BubbleSort

Idea: swap adjacent elements that are out of order

- Algorithm
  - compare adjacent elements; if the second is smaller than the first, swap them
  - repeat for all elements except the last one (which is already correct)
  - repeat the two previous steps, using one less pair in each iteration until there are no more pairs (or no swaps)

BubbleSort

```
template <class Comparable>
void bubbleSort(vector<Comparable> &v) {
    for(unsigned int j = v.size()-1; j > 0; j--) {
        bool troca=false;
        for(unsigned int i = 0; i < j; i++)
            if(v[i+1] < v[i]) {
                swap(v[i],v[i+1]);
                troca = true;
            }
        if (!troca) return;
    }
}
```

- BubbleSort time complexity
  - two nested cycles, each can have n iterations $T(n) = O(n^2)$

MergeSort

Idea: divide in two (easy to order), order and merge

- Algorithm (recursive approach)
  - divide the vector in half
  - sort each half, using MergeSort recursively
  - merge the two halves already ordered

- MergeSort time complexity
  - 2 recursive calls of size $n/2$
  - vector merge operation: $O(n)$

$$T(n) = O(n \times \log n)$$

- MergeSort space complexity
  - vector merge requires extra space $S(n) = O(n)$

QuickSort

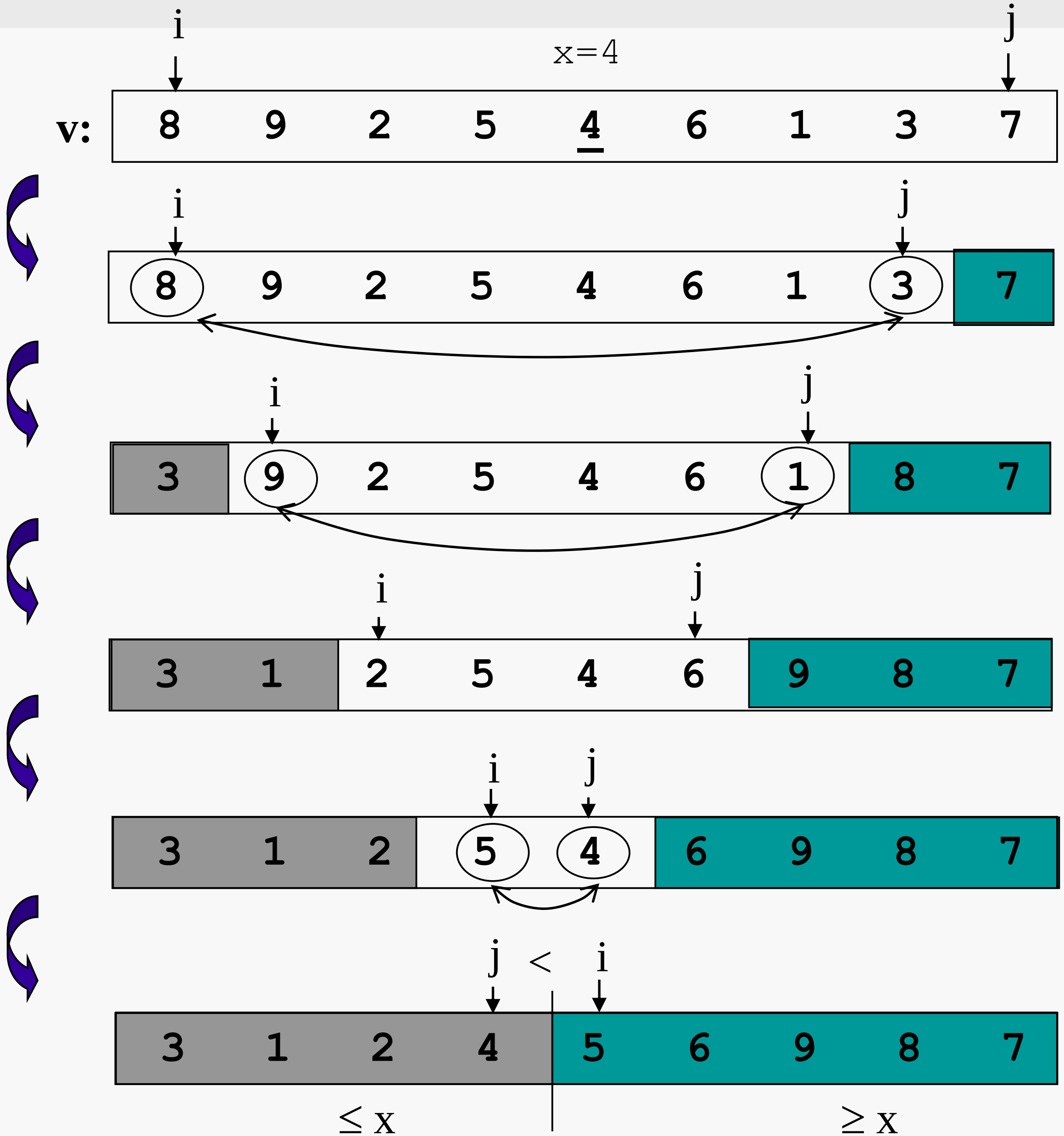
Idea: partition according to a pivot

invented by Tony) Hoare in 1959

- Algorithm (recursive approach)
  - choose an "arbitrary" element (x) of the vector (called pivot)
  - break the initial vector into two sub-vectors (left and right), with
    - $\leq$ x values in the left sub-vector and
    - $\geq$ x values in the right sub-vector
  - sort the left and right sub-vectors using the same method recursively

if the number (n) of elements to sort is too small, use another algorithm *(insertionSort)*

QuickSort



x=4
v: 8 9 2 5 4 6 1 3 7
8 9 2 5 4 6 1 3 7
3 9 2 5 4 6 1 8 7
3 1 2 5 4 6 9 8 7
3 1 2 5 4 6 9 8 7
j < i
3 1 2 4 5 6 9 8 7
≤ x
≥ x

QuickSort

```
template <class Comparable>
void quickSort(vector<Comparable> &v, int left, int right){
    if (right-left <= 10)      // if small vector
        insertionSort(v, left, right);
    else {
        Comparable x = median3(v, left, right);    // x is the pivot
        int i = left; int j = right-1;
        for(; ; ) {
            while (v[++i] < x) ;
            while (x < v[--j]) ;
            if (i < j)
              swap(v[i], v[j]);
            else break;
        }
       swap(v[i], v[right-1]);    //reset pivot
        quickSort(v, left, i-1);
        quickSort(v, i+1, right);
   }
}
```

QuickSort

```
template <class Comparable>
void quickSort(vector<Comparable> &v){
   quickSort(v, 0, v.size()-1);
}
```

```
template <class Comparable>
const Comparable &median3(vector<Comparable> &v, int left, int right){
   int center = (left+right) /2;
   if (v[center] < v[left])
      swap(v[left], v[center]);  //swap elements if order incorrect
   if (v[right] < v[left])
      swap(v[left], v[right]);   //swap elements if order incorrect
   if (v[right] < v[center])
      swap(v[center], v[right]);  //swap elements if order incorrect

   swap(v[center], v[right-1]);  //puts pivot in position right-1
   return v[right-1];
}
```

QuickSort

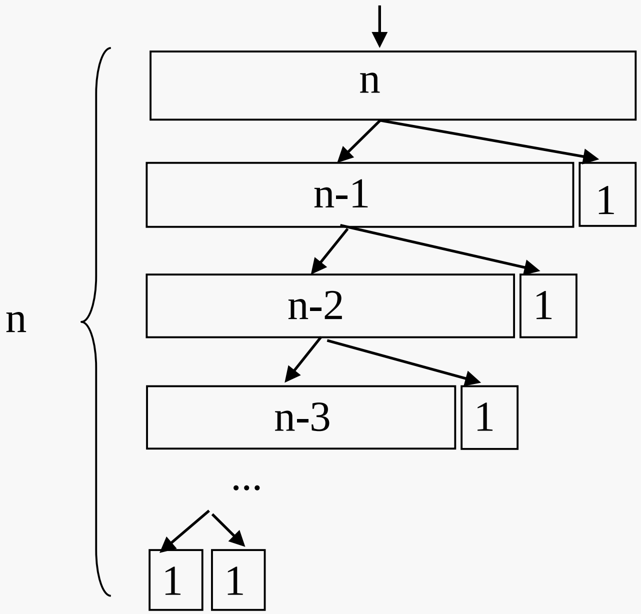
- Choice of pivot determines efficiency
  - *worst case*: pivot is the smallest element
    - $T(n) = O(n^2)$
  - *best case*: pivot is the middle element
    - $T(n) = O(n \times \log n)$
  - *average case*: pivot cuts vector arbitrarily
    - $T(n) = O(n \times \log n)$

- Choice of pivot
  - one of the extreme elements of the vector
    - bad choice: $O(n^2)$ if ordered vector
  - random element
    - one more heavy function
  - median of three elements (extremes and middle of vector)
    - recommended

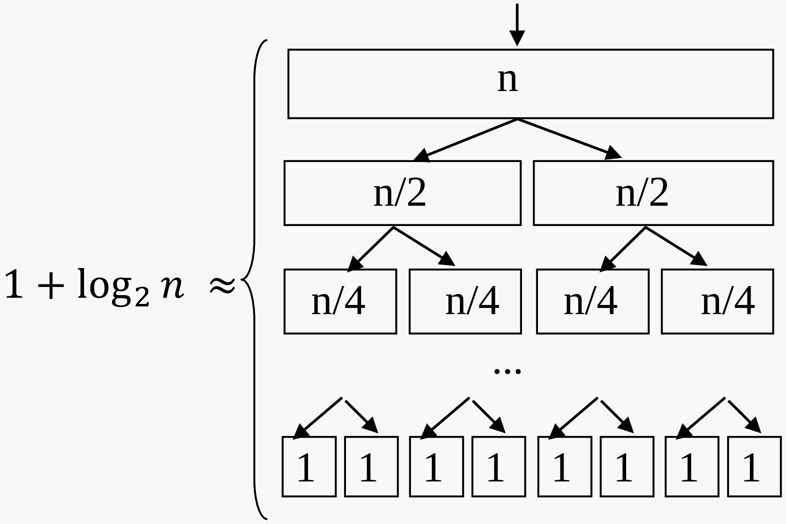
QuickSort

- QuickSort time complexity

worst case



best case



- recursion depth: n
- execution time

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

- recursion depth: $\approx 1 + \log_2 n$
- execution time (each line is n)

$$T(n) = (1 + \log_2 n) \times n$$
$$= O(n \times \log n)$$

QuickSort

- QuickSort space complexity
  - the memory space required by each QuickSort call, not counting recursive calls, is independent of the size (n) of the vector.
  - the total memory space required by the QuickSort call, including recursive calls, is therefore proportional to the recursion depth
  - QuickSort space complexity is:
    - *worst case*: $S(n) = O(n)$
    - *best case, average case*: $S(n) = O(\log n)$

Comparative sorting algorithms (some comparison)

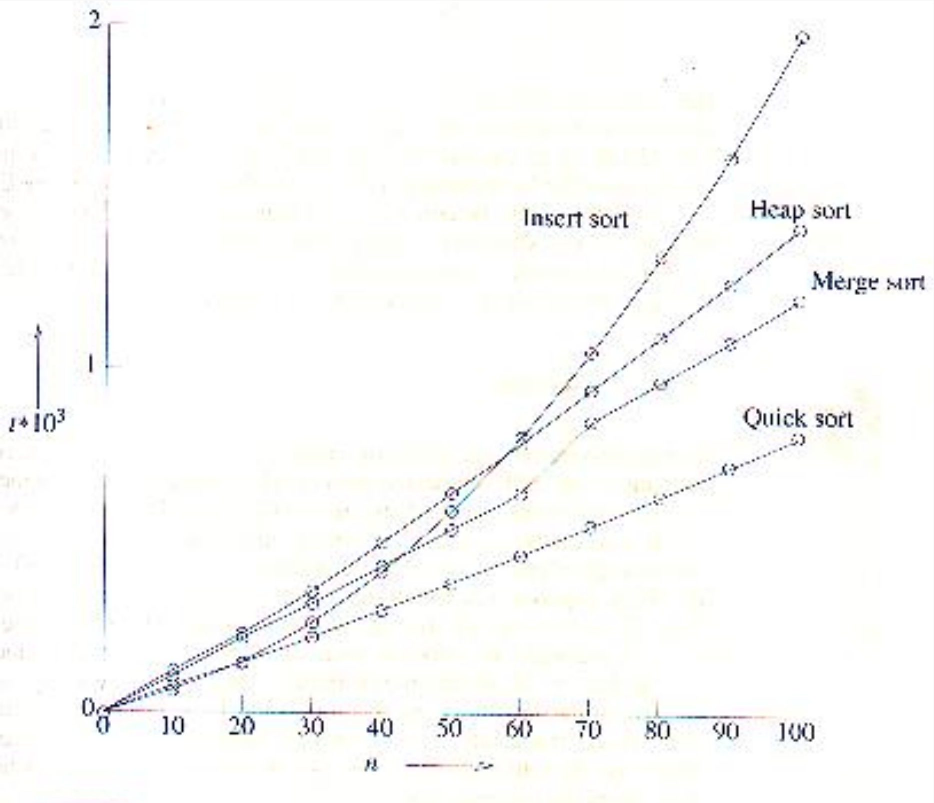


Figure 14.12 Plot of average times

Sahni, "Data Structures, Algorithms and Applications in C++"

– QuickSort is in practice the most efficient
– except for small vectors where the insertion sort method (insertionSort) is better

Sorting (comparative/non-comparative)

- Comparative based sorting algorithms:
  - Several sorting algorithms have been discussed and the best ones, so far:

$$T(n) = O(n \times \log n)$$

  - It can be proven that any deterministic comparison based sorting algorithm will need to carry out at least $O(n \times \log n)$ steps

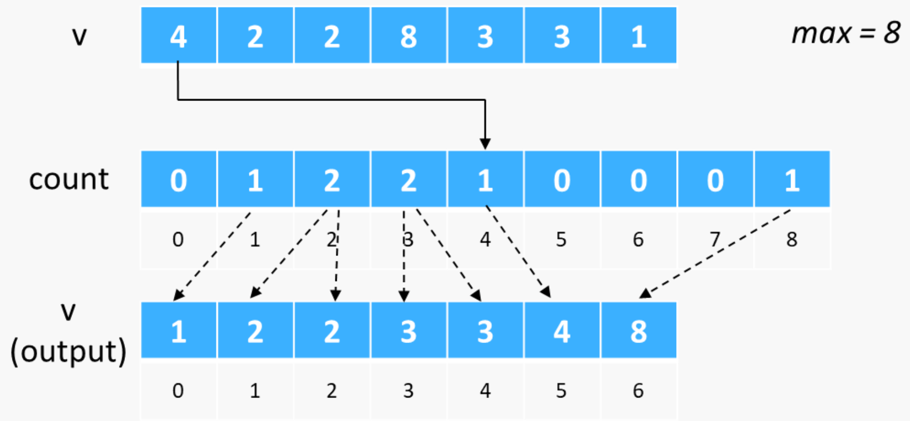
- Non-comparative based sorting algorithms:
  - we can avoid comparing elements and still sort
  - this is fast if the range of data is small

  - for simplicity we will assume that elements are numbers, but
  - idea can be generalized to other types of data

CountingSort

Idea: count the number of occurrences of an element

- Algorithm:
  - count the number of occurrences of each element
  - count is stored in an auxiliary array
  - mapping the count as an index of the auxiliary array



$T(n) = O(n+k)$, k is the maximum element

Counting Sort

```
void countingSort(vector<int> &v){
    if (v.empty())
        return;
    auto min_max = std::minmax_element(v.begin(), v.end());
    int min = *min_max.first;
    int max = *min_max.second;
    std::vector<unsigned> count((max - min) + 1, 0);
    for (auto v1:v)
        ++count[v1-min];
    unsigned i=0;
    for (auto c = 0; c < count.size(); c++)
        for (auto j = 0; j < count[c]; j++) {
            v[i]= c+min;
            i++;
        }
}
```

see animation in [VisuAlgo]

RadixSort

Idea: repeatedly sort by digit

- Algorithm:

  we have to go through all the significant places of all elements

  – go through each significant place one by one

  – use any stable* sorting technique to sort the digits at each significant place (counting sort)

```
for each digit i=0 to d-1     (0 is the least significant digit)
   countingSort by digit i     (or other stable sorting algorithm)
```

$T(n) = O(d * (n + k))$, d is the maximum number of digits of an element

* a sorting algorithm is stable if it always leaves elements with equal keys in their original order

Radix Sort

v

12	58	47	64	52	9	37	20

... *(countingsort)*

sort by digits in units places

20	12	52	64	47	37	58	09

... *(countingsort)*

sort by digits in tens places

9	12	20	37	47	52	58	64

see animation in [VisuAlgo]

Sorting

- There are many sorting algorithms (many other exist)

- The "best" algorithm depends on the specific case

- It is possible to combine several sorting algorithms. In practice, in real implementations, this is what is done.
  - when size or portion of unordered array is small use insertion sort
  - C++ STL uses *IntroSort*: (QuickSort + HeapSort) + InsertionSort

STL algorithms

- Sorting

LegacyRandomAccessIterator !

void sort(iterator start, iterator end);

- sorts the elements between *[start, end[* in ascending order, using the operator <

void sort(iterator start, iterator end, StrictWeakOrdering cmp);

- sorts the elements between *[start, end[* in ascending order, using the *StrictWeakOrdering* function

Algoritmo de ordenação implementado em *sort()* é o algoritmo *IntroSort*:

– (QuickSort + HeapSort) + InsertionSort

Example

```
class Person {
   string cc, name;
   int age;
public:
   Person (string c, string nm="", int a=0);
   bool operator < (const Person & p2) const;
};
```

```
bool Person::operator < (const Person & p2) const {
   return name < p2.name;
}

bool compPerson(const Person &p1, const Person &p2) {
    return p1.getAge() < p2.getAge();
}
```

Example

```
int main()
{
  vector<Person> vp;
  vp.push_back(Person("6666666","Rui Silva",34));
  vp.push_back(Person("7777777","Antonio Matos",24));
  vp.push_back(Person("1234567","Maria Barros",20));
  vp.push_back(Person("7654321","Carlos Sousa",18));
  vp.push_back(Person("3333333","Fernando Cardoso",33));
  vector<Person> vp1=vp;
  vector<Person> vp2=vp;

  cout << "initial vector:" << endl;
  write_vector(vp);
```

```
initial vector:
v[0] = (cc: 6666666, name: Rui Silva, age: 34)
v[1] = (cc: 7777777, name: Antonio Matos, age: 24)
v[2] = (cc: 1234567, name: Maria Barros, age: 20)
v[3] = (cc: 7654321, name: Carlos Sousa, age: 18)
v[4] = (cc: 3333333, name: Fernando Cardoso, age: 33)
```

Example

```
  sort(vp1.begin(),vp1.end());
  cout << "sort using 'operator <':" << endl;
  write_vector(vp1);

  sort(vp2.begin(),vp2.end(),compPerson);
  cout << "sort using comparison function:" << endl;
  write_vector(vp2);
}
```

```
sort using 'operator <':
v[0] = (cc: 7777777, name: Antonio Matos, age: 24)
v[1] = (cc: 7654321, name: Carlos Sousa, age : 18)
v[2] = (cc: 3333333, name: Fernando Cardoso, age : 33)
v[3] = (cc: 1234567, name: Maria Barros, age : 20)
v[4] = (cc: 6666666, name: Rui Silva, age : 34)
```

```
sort using comparison function:
v[0] = (cc: 7654321, name: Carlos Sousa, age : 18)
v[1] = (cc: 1234567, name: Maria Barros, age : 20)
v[2] = (cc: 7777777, name: Antonio Matos, age : 24)
v[3] = (cc: 3333333, name: Fernando Cardoso, age : 33)
v[4] = (cc: 6666666, name: Rui Silva, age : 34)
```