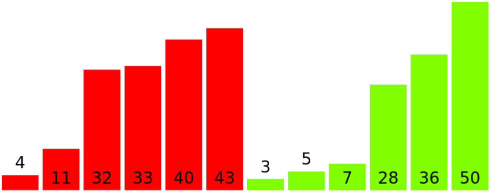
Sorting Algorithms

Comparison based sorting and linear sorting

L.EIC

Algoritmos e Estruturas de Dados

2025/2026



P Ribeiro, AP Tomás

Sorting

- Sorting is a **preprocessing step** for many other algorithms
  - E.g.: finding the median; finding the closest pair for points in a line or in the plane; finding duplicates becomes easier after sorting; ...
- **Different types of sorting** can be suitable for different types of data
  - E.g.: for less general cases, there are linear algorithms for sorting
- It is important to know the sorting functions available in **programming language libraries**
  - E.g.: qsort (C), **STL sort (C++)**, Arrays.sort (Java)

Examples of Applications of Sorting

- Problem: find **frequency** of elements (sort and elements stay together)
- Problem: find **closest pair** of numbers (sort and see differences between consecutive numbers)
- Problem: find **k-th smallest** number (sort and see position k) (there are better algorithms not based on sorting)
- Problem: select **top-k** (sort and see first k)
- Problem: **union** of sets (sort and join - similar to "merge")
- Problem: **intersection** of sets (sort and traverse - similar to "merge")
- ...

Some sorting algorithms

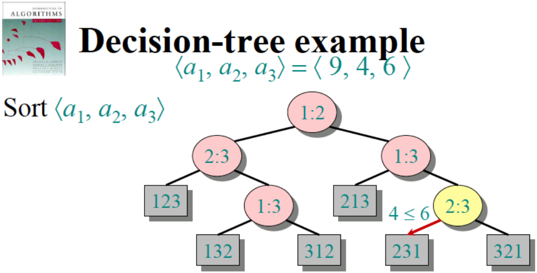
- **Comparative Algorithms**
  - **SelectionSort** (select the max/min)
  - **InsertionSort** (insert in the correct position)
  - **BubbleSort** (swap adjacent elements to push max/min to its position)
  - **MergeSort** (split in two, sort halves and then merge)
  - **QuickSort** "naive" (split in two using a pivot and recursively sort)
  - **Randomized QuickSort** (select the pivot at random)

- **Non Comparative Algorithms**
  - **CountingSort** (count the number of elements of each type)
  - **RadixSort** (sort according to "digits")

You can see some animations in: https://visualgo.net/en/sorting or, e.g., Quick sort with Hungarian, folk dance (https://www.youtube.com/watch?v=3San3uKKHgg)

How fast can we sort?

- What is the lowest possible time complexity for a general sorting algorithm in the worst case? $\Theta(n \log n)$ for the comparative model.
  - **Comparative model (decision trees):** to distinguish elements, we can only use comparisons ($<$, $>$, $\geq$, $\leq$, $=$). How many comparisons do we need?



Each internal node is labeled $i : j$, for $i, j \in \{1, 2, \ldots, n\}$

- ★ The left subtree shows subsequent comparisons if $a_i \leq a_j$
- ★ The right subtree shows subsequent comparisons if $a_i > a_j$

Binary Decision-tree model of computation

A decision tree can model the execution of any comparison sort.

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

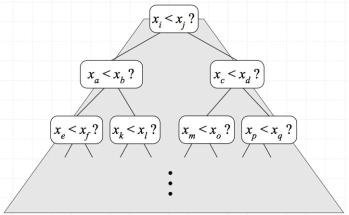


Image from Goodrich and Tamassia textbook

Each possible run of the algorithm on an instance of size n corresponds to a root-to-leaf path in the decision tree. For each permutation of $(1, 2, \ldots, n)$ that is given as input, the path ends up in a distinct leaf.

How fast can we sort?

Theorem

Any comparison-based sorting algorithm performs at least $\boldsymbol{\Omega}(n \log n)$ comparisons in the worst case.

- A sketch of the proof that comparative sorting is $\Omega(n \log n)$ in the worst case:
  - Any deterministic comparison-based sorting algorithm can be represented as a **decision tree with** $n!$ **leaves**.
  - The worst-case running time is at least the depth of the decision tree.
  - All **decision trees with** $n!$ **leaves** have depth $\Omega(n \log(n))$.

    (binary tree with height h has $\leq 2^h$ leaves; $2^h \geq n!$ implies $h \geq \log_2(n!) \in \Omega(n \log n)$, by **Stirling's formula**)
  - So **any comparison-based sorting algorithm** must have **worst-case** running time at least $\Omega(n \log(n))$.

The Sorting Problem

- For the next slides we will assume the following:
  - We want to sort in **ascending order**
  - We are sorting a set of **n** items
  - The items are stored in an array **v[n]** in positions $0..\mathtt{n}-1$
  - Items are **comparable** (through $<$, $>$, $=$,...).
- **ascending order** (or non-decreasing order) means $\mathbf{v[i]} \leq \mathbf{v[i+1]}$, for all **i**, after sorting.

What we will see?

- Time complexity of some **comparative algorithms** with $\mathcal{O}(1)$ time per comparison:
  - **SelectionSort**: $\boldsymbol{\Theta}(\mathbf{n}^2)$
  - **InsertionSort**: $\mathcal{O}(\mathbf{n}^2)$
  - **BubbleSort**: $\mathcal{O}(\mathbf{n}^2)$
  - **MergeSort**: $\mathcal{O}(\mathbf{n} \log \mathbf{n})$
  - **HeapSort**: $\mathcal{O}(\mathbf{n} \log \mathbf{n})$ (to be presented later in AED)
  - **QuickSort** "naive" : $\mathcal{O}(\mathbf{n}^2)$
  - **Randomized QuickSort**: expected time $\mathcal{O}(\mathbf{n} \log \mathbf{n})$

- **HeapSort** and **MergeSort** are asymptotically time optimal in the worst case. **HeapSort** works **in-place**, i.e., it uses $\mathcal{O}(\mathbf{1})$ extra space, whereas **MergeSort** requires an auxiliary vector with $\boldsymbol{\Theta}(\mathbf{n})$ elements and $\mathcal{O}(\log \mathbf{n})$ extra space for handling recursion.

SelectionSort

Idea: look for the minimum and put it into the correct position. Do the same for the remaining elements.

Selection sort (in "pseudo-code")

```
for k from 0 to n-2 do
    pmin = index of the minimum of v[k],v[k+1],...,v[n-1]
    swap v[k] with v[pmin]      (or swap if k is not pmin)
```

- **Loop invariant:** for all $\mathtt{k} \geq 1$, when we are testing the loop condition, the array $\mathtt{v}$ contains the same values it had initially, but possibly in different positions, and $\mathtt{v}[0] \leq \mathtt{v}[1] \ldots \leq \mathtt{v}[\mathtt{k}-1] \leq \min(\mathtt{v}[\mathtt{k}], \ldots, \mathtt{v}[\mathtt{n}-1])$

  For the proof of the loop invariant, it is crucial that $\mathtt{v}[\mathtt{k}-1] \leq \min(\mathtt{v}[\mathtt{k}], \ldots, \mathtt{v}[\mathtt{n}-1])$

- **Correctness:** the loop ends when $\mathtt{k} = \mathtt{n}-1$. From the invariant, we have $\mathtt{v}[0] \leq \mathtt{v}[1] \ldots \leq \mathtt{v}[\mathtt{n}-2] \leq \min(\mathtt{v}[\mathtt{n}-1]) = \mathtt{v}[\mathtt{n}-1]$.

SelectionSort

Example

Sorting $[2, 3, -7, 8, 2, 4, 9, -5]$ by **selection sort**.

2	3	$\boxed{-7}$	8	2	4	9	-5
$\mathbf{-7}$	3	2	8	2	4	9	$\boxed{-5}$
$\mathbf{-7}$	$\mathbf{-5}$	$\boxed{2}$	8	2	4	9	3
$\mathbf{-7}$	$\mathbf{-5}$	**2**	8	$\boxed{2}$	4	9	3
$\mathbf{-7}$	$\mathbf{-5}$	**2**	**2**	8	4	9	$\boxed{3}$
$\mathbf{-7}$	$\mathbf{-5}$	**2**	**2**	**3**	$\boxed{4}$	9	8
$\mathbf{-7}$	$\mathbf{-5}$	**2**	**2**	**3**	**4**	9	$\boxed{8}$
$\mathbf{-7}$	$\mathbf{-5}$	**2**	**2**	**3**	**4**	**8**	9

Answer: $\mathbf{-7}$ $\mathbf{-5}$ **2** **2** **3** **4** **8** **9**

SelectionSort

```
#include <vector>
using namespace std;

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

SelectionSort

Asymptotic time complexity with `vector<int>`

$\Theta(n^2)$ in the best case, i.e., when v is sorted and all elements are different.
Difficult to define worst case but number of `pmin = j` instructions $\leq B_n$ always.

	cost per operation	number of times	Total Time: $\Theta(n^2)$
define parameters `v`	c_0	1	$\Theta(1)$
`unsigned k = 0`	c_1	1	$\Theta(1)$
`test k < v.size()-1`	c_2	n	$\Theta(n)$
`unsigned pmin = k`	c_3	$n-1$	$\Theta(n)$
`unsigned j = k+1`	c_4	$n-1$	$\Theta(n)$
`test j < v.size()`	c_2	A_n	$\Theta(n^2)$
`test v[j] < v[pmin]`	c_5	B_n	$\Theta(n^2)$
`pmin = j`	c_3	$0 \leq t \leq B_n$	Best: $\mathcal{O}(1)$ Worst: $\mathcal{O}(n^2)$
`swap(v[k],v[pmin])`	c_6	$n-1$	$\Theta(n)$
`j++`	c_7	B_n	$\Theta(n^2)$
`k++`	c_7	$n-1$	$\Theta(n)$

with $A_n = \displaystyle\sum_{k=0}^{n-2}(n-k) = \frac{(n+2)(n-1)}{2}$ and $B_n = \displaystyle\sum_{k=0}^{n-2}(n-1-k) = \frac{n(n-1)}{2}$.

InsertionSort

Insertion sort (in "pseudo-code")

```
for k from 1 to n-1 do
  insert v[k] in its correct position wrt v[0],...,v[k-1]
  (by shifting elements to the right, if necessary)
```

- Let $a_0, a_1, \ldots, a_{n-1}$ be the content of array $\mathtt{v}$ initially.
- **Loop invariant:** For all $\mathtt{k} \geq 1$, when we are testing the loop condition, $\mathtt{v}[0], \mathtt{v}[1], \ldots, \mathtt{v}[\mathtt{k}-1]$ contains $a_0, a_1, \ldots, a_{k-1}$ but already sorted, and $v[j] = a_j$, for $\mathtt{k} \leq j < \mathtt{n}$. (provided the insertion step is correct)
- **Correctness:** the loop ends when $\mathtt{k} = \mathtt{n}$. From the invariant, $\mathtt{v}[0], \mathtt{v}[1], \ldots, \mathtt{v}[\mathtt{n}-1]$ contains $a_0, a_1, \ldots, a_{n-1}$ sorted.

InsertionSort

Example

Sorting $[2, 3, -7, 8, 2, 4, 9, -5]$ by **insertion sort**.

Shifts right?

2	$\boxed{3}$	-7	8	2	4	9	-5
2	**3**	$\boxed{-7}$	8	2	4	9	-5
-7	**2**	**3**	$\boxed{8}$	2	4	9	-5
-7	**2**	**3**	**8**	$\boxed{2}$	4	9	-5
-7	**2**	**2**	**3**	**8**	$\boxed{4}$	9	-5
-7	**2**	**2**	**3**	**4**	**8**	$\boxed{9}$	-5
-7	**2**	**2**	**3**	**4**	**8**	**9**	$\boxed{-5}$
-7	**-5**	**2**	**2**	**3**	**4**	**8**	**9**

⇒

2	3	**3**
2	**2**	**3**

⇒

-7	2	3	8	**8**
-7	2	3	**3**	**8**

⇒

-7	2	2	3	8	**8**

⇒

-7	2	2	3	4	8	9	**9**
-7	2	2	3	4	8	**8**	**9**
-7	2	2	3	4	**4**	**8**	**9**
-7	2	2	3	**3**	**4**	**8**	**9**
-7	2	2	**2**	**3**	**4**	**8**	**9**
-7	2	**2**	**2**	**3**	**4**	**8**	**9**

InsertionSort

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

Loop invariant (insertion step): For all $\mathtt{j} \geq \mathtt{k}$, when we are testing the `for(j)`-loop condition (i.e., $\mathtt{j} > 0$ && $\mathtt{tmp} < \mathtt{v}[\mathtt{j}-1]$):

- $\mathtt{v}[0], \mathtt{v}[1], \ldots, \mathtt{v}[\mathtt{j}-1], \mathtt{v}[\mathtt{j}+1], \ldots, \mathtt{v}[\mathtt{k}]$ contains $a_0, a_1, \ldots, a_{k-1}$ sorted;
- $\mathtt{tmp}$ contains a_k and the elements in $\mathtt{v}[\mathtt{j}+1], \ldots, \mathtt{v}[\mathtt{k}]$ are $> a_k$, if $j < k$
- $\mathtt{v}[\mathtt{j}]$ is "free" (can be used for other elements)
- $\mathtt{v}[\mathtt{t}] = a_t$, for all $k+1 \leq t < n$.

InsertionSort

Asymptotic time complexity with `vector<int>`

Best case (v is sorted already): $\Theta(n)$.
Worst case (all elements are distinct and v is sorted in decreasing order): $\Theta(n^2)$
In general: $\mathcal{O}(n^2)$

	Best case	Worst Case	
`define parameters v`	1	1	$\Theta(1)$
`unsigned k = 1`	1	1	$\Theta(1)$
`test k < v.size()`	n	n	$\Theta(n)$
`int tmp = v[k]`	$n-1$	$n-1$	$\Theta(n)$
`unsigned j`	$n-1$	$n-1$	$\Theta(n)$
`j = k`	$n-1$	$n-1$	$\Theta(n)$
`test j > 0 && tmp < v[j-1]`	$n-1$	D_n	$\Theta(n^2)$
`v[j] = v[j-1]`	0	$D_n - (n-1)$	$\Theta(n^2)$
`j--`	0	$D_n - (n-1)$	$\Theta(n^2)$
`v[j] = tmp`	$n-1$	$n-1$	$\Theta(n)$
`k++`	$n-1$	$n-1$	$\Theta(n)$
	$\Theta(n)$	$\Theta(n^2)$	

with $D_n = \sum_{k=1}^{n-1}\sum_{j=0}^{k} 1 = \sum_{k=1}^{n-1}(k+1) = \sum_{k=2}^{n} k = \frac{n(n+1)}{2} - 1 \quad \in \Theta(n^2)$

BubbleSort

Bubble sort (in "pseudo-code")

```
for k from n-1 to 1 with step -1 do
  move max(v[0],v[1],...,v[k]) to position k by swapping
  adjacent elements, v[j] with v[j+1], for 0 <= j < k,
  if necessary.
```

- Let $a_0, a_1, \ldots, a_{n-1}$ be the content of array $\mathtt{v}$ initially.
- **Loop invariant:** When we are testing the loop condition for the t-th time, $\mathtt{k} = \mathtt{n} - t$, the $(t-1)$ largest elements of $a_0, a_1, \ldots, a_{n-1}$ are in $\mathtt{v}[\mathtt{k}+1], \ldots, \mathtt{v}[\mathtt{n}-1]$ and sorted, and the remaining are in $\mathtt{v}[0], \ldots, \mathtt{v}[\mathtt{k}]$.
- **Correctness:** the loop ends when $\mathtt{k} = 0$, i.e, $t = \mathtt{n}$. From the invariant, the $(\mathtt{n}-1)$ largest elements of $a_0, a_1, \ldots, a_{n-1}$ are in $\mathtt{v}[1], \mathtt{v}[2], \ldots, \mathtt{v}[\mathtt{n}-1]$ and sorted, and the remaining one is in $v[0]$. Therefore, $\mathtt{v}$ is sorted.

BubbleSort

Example

Sorting $[2, 3, -5, 7, 2, 8, 9]$ by **bubble sort**.

```
---- iteration 1
2  3  -5  7  2  8  9
(2,3)  no swap    2  3  -5  7  2  8  9
(3,-5) swap       2  -5  3  7  2  8  9
(3,7)  no swap    2  -5  3  7  2  8  9
(7,2)  swap       2  -5  3  2  7  8  9
(7,8)  no swap    2  -5  3  2  7  8  9
(8,9)  no swap    2  -5  3  2  7  8  9
---- iteration 2
2  -5  3  2  7  8  9
(2,-5) swap       -5  2  3  2  7  8  9
(2,3)  no swap    -5  2  3  2  7  8  9
(3,2)  swap       -5  2  2  3  7  8  9
(3,7)  no swap    -5  2  2  3  7  8  9
(7,8)  no swap    -5  2  2  3  7  8  9
---- iteration 3
-5  2  2  3  7  8  9
(-5,2) no swap    -5  2  2  3  7  8  9
(2,2)  no swap    -5  2  2  3  7  8  9
(2,3)  no swap    -5  2  2  3  7  8  9
(3,7)  no swap    -5  2  2  3  7  8  9
Answer: -5 2 2 3 7 8 9
```

We can **stop when there are no swaps** in an iteration (the array is sorted!).

BubbleSort

```
template <class Comparable>
void BubbleSort(vector<Comparable> &v){    // Trivial version
  for (unsigned k = v.size()-1; k > 0; k--)
    for (unsigned j = 0; j < k; j++)
      if (v[j+1] < v[j]) swap(v[j],v[j+1]);
}
```

```
template <class Comparable>
void BubbleSort(vector<Comparable> &v)  {  // Improved version
  bool changes = true;
  for (unsigned k = v.size()-1; changes && k > 0; k--) {
    changes = false;
    for (unsigned j = 0; j < k; j++)
      if (v[j+1] < v[j]) {
        swap(v[j],v[j+1]);
        changes = true;
      }
  }
}
```

BubbleSort

Asymptotic complexity with `vector<int>`

- **Trivial version (without the flag `changes`)**
  $\Theta(n^2)$ time for all instances.

- **Improved version (with the flag `changes`)**
  - Best case (`v` is already sorted): $\Theta(n)$ time.
  - Worst case (`v` is sorted in strictly decreasing order): $\Theta(n^2)$ time
  - So, BubbleSort runs in $\mathcal{O}(n^2)$ time.

- **Space** complexity for both: $\mathcal{O}(1)$ extra space.

- **Further improvement:** the last swap in a given iteration can be used to bound search in the next one. Still, $\mathcal{O}(n^2)$ time.
  (Using flags often leads to "spaghetti code", i.e., difficult-to-maintain and unstructured code. It made some sense here, but can be avoided!)

MergeSort and QuickSort

Based on divide and conquer

- Algorithms that are expressed in a **recursive** way
- Follow the **divide and conquer** strategy:

Divide and Conquer

Divide the problem in a set of subproblems which are smaller instances of the same problem

Conquer the subproblems solving them recursively. If the problem is small enough, solve it directly.

Combine the solutions of the smaller subproblems on a solution for the original problem

MergeSort

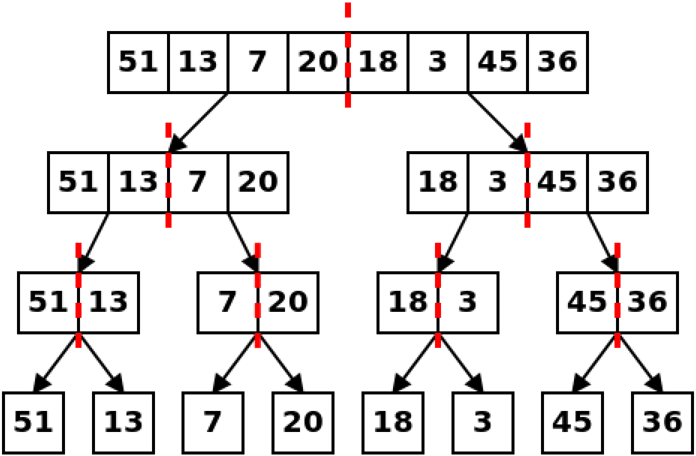
MergeSort

Divide: partition the initial array in two halves

Conquer: recursively sort each half. If we only have one item, it is sorted.

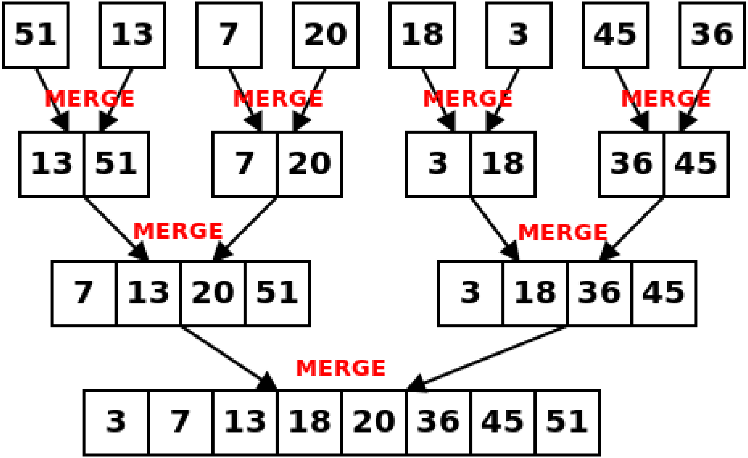
Combine: merge the two sorted halves in a final sorted array

Divide:



51 13 7 20 18 3 45 36
51 13 7 20
18 3 45 36
51 13
7 20
18 3
45 36
51
13
7
20
18
3
45
36

Conquer:



51
13
7
20
18
3
45
36
MERGE
MERGE
MERGE
MERGE
13 51
7 20
3 18
36 45
MERGE
MERGE
7 13 20 51
3 18 36 45
MERGE
3 7 13 18 20 36 45 51

MergeSort

What is the **execution time** of this algorithm?

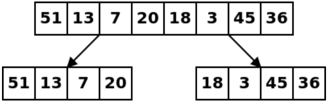
- **D(n)** - Time to partition an array of size n in two halves
- **M(n)** - Time to merge two sorted arrays of size n
- **T(n)** - Time for a MergeSort on an array of size n

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ D(n) + 2T(n/2) + M(n) & \text{if } n > 1 \end{cases}$$

In practice, we are ignoring certain details, but it suffices
(ex: when n is odd, the size of subproblem is not exactly $n/2$)

Divide and Conquer
MergeSort

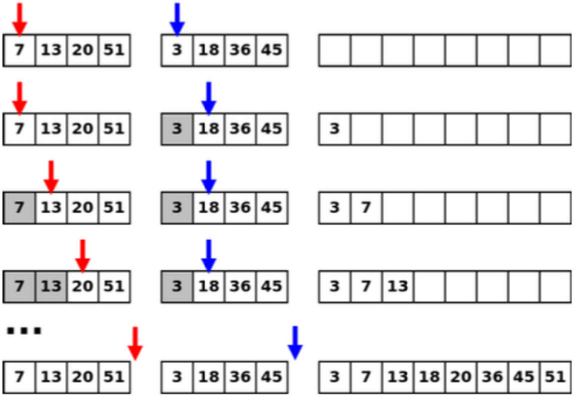
D(n) - Time to partition an array of size n in two halves is **$\Theta(1)$**



- Do not create a copy of the array.
- If we use a function with two arguments `mergesort(a,b)`, to sort from position a to position b:
  - Initially, `mergesort(0, n-1)` (with arrays starting at position 0)
  - Let **middle** $= \lfloor (a+b)/2 \rfloor$ be the middle position
    Calls `mergesort(a,middle)` and `mergesort(middle+1,b)`

(in the implementation, we will have two other arguments: `mergeSort(v,tmpArray,a,b)`)

$\mathbf{M(n)}$ - Time to merge two sorted arrays of size $n/2$ is $\mathcal{O}(\mathbf{n})$



Worst case: $(n-1)$ comparisons $+$ n copies, spending $\mathbf{\Theta(n)}$ (linear time)

MergeSort

The recurrence that defines its runtime

- **D**(**n**) - Time to partition an array of size n in two halves
- **M**(**n**) - Time to merge two sorted arrays of size n
- **T**(**n**) - Time for a MergeSort on an array of size n

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ D(n) + 2T(n/2) + M(n) & \text{if } n > 1 \end{cases}$$

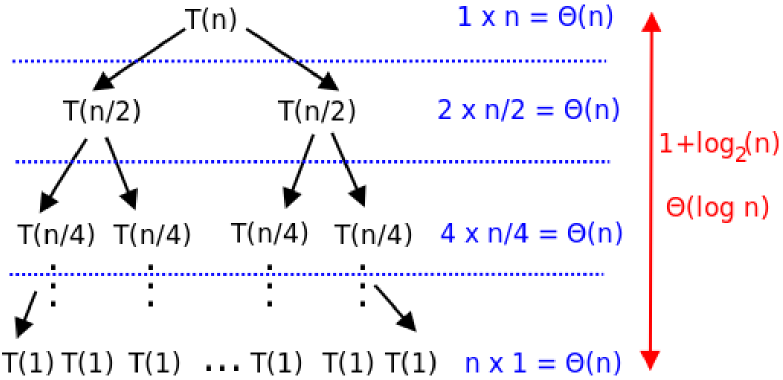
becomes

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

How to solve this recurrence?

(for a cleaner explanation we will assume $n = 2^k$, but the results holds for any n)

Let's draw the **recursion tree**:



Worst case: adding everything we get that **MergeSort** is $\mathbf{\Theta(n \log_2 n)}$

MergeSort

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

// internal method , makes recursive calls
template <class Comparable>
void mergeSort(vector<Comparable> &v,vector<Comparable> &tmp,
                        int left , int right ) {
  if (left < right){
    int middle = left + (right-left)/2;
    mergeSort(v, tmp, left , middle);
    mergeSort(v, tmp, middle + 1, right);
    merge(v, tmp, left, middle + 1, right);
  }
}
```

In addition to the array `tmpArr`, shared by all calls, it uses $\mathcal{O}(\log n)$ **extra space for the recursion**, since the *recursion depth* is $\mathcal{O}(\log n)$ and each call needs $\mathcal{O}(1)$ extra space.

MergeSort – function merge

```
template <class Comparable>
void merge(vector<Comparable> &v, vector<Comparable> &tmp,
                  int leftPos, int rightPos, int rightEnd) {
  int leftEnd = rightPos-1, tmpPos = leftPos;
  int numElements = rightEnd-leftPos+1;

  while (leftPos <= leftEnd && rightPos <= rightEnd)
    if (v[leftPos] <= v[rightPos])
       tmp[tmpPos++] = v[leftPos++];
    else tmp[tmpPos++] = v[rightPos++];

  while(leftPos <= leftEnd)        // if left has not ended
     tmp[tmpPos++] = v[leftPos++];
  while(rightPos <= rightEnd)           // if right has not ended
     tmp[tmpPos++] = v[rightPos++];
  for(int i=0; i < numElements; i++, rightEnd--) // copy to v
     v[rightEnd] = tmp[rightEnd];
}
```

MergeSort – improving merge

Improved version: exploits the fact that, when **"left"** ends before **"right"**, all the remaining items of **"right"** are already in their final position in `v`.

```
template <class Comparable>
void merge(vector<Comparable> &v, vector<Comparable> &tmp,
                  int leftPos, int rightPos, int rightEnd) {
  int leftEnd = rightPos-1, tmpPos = leftPos;
  int auxLeft = leftPos; // needs initial leftPos to copy tmp to v

  while (leftPos <= leftEnd && rightPos <= rightEnd)
    if (v[leftPos] <= v[rightPos])
      tmp[tmpPos++] = v[leftPos++];
    else tmp[tmpPos++] = v[rightPos++];

  while(leftPos <= leftEnd)
    tmp[tmpPos++] = v[leftPos++];
  for(int i = 0; i < tmpPos; i++)  // copy tmp to v
    v[auxLeft++] = tmp[i];
}
```

QuickSort (naive)

Key idea: split according to a pivot and sort recursively

QuickSort (naive)

1. Choose an element (**first**, for example) as the pivot
2. Split the array into two: elements smaller than the pivot and elements larger than (or equal to) the pivot
3. Recursively sort each of the two parts (without the pivot)

- **The choice of the pivot is crucial**.
- If the choice "splits" always well the algorithm takes $O(n \log n)$
- In the worst case, $T(n) = T(n-1) + T(0) + \Theta(n)$, leading to $\mathbf{\Theta(n^2)}$

For an animation:

https://www.youtube.com/watch?v=3San3uKKHgg

QuickSort

```
template <class Comparable>
void quickSort(vector<Comparable> &v, int left, int right){
   if (left < right){
     int pivotPos = partition(v, left, right);
     quickSort(v, left, pivotPos - 1);
     quickSort(v, pivotPos + 1, right);
   }
}
```

Randomized QuickSort

Key idea: split according to a pivot and sort recursively

Randomized QuickSort

1. Choose an element **at random** as the pivot
2. Split the array into two: elements smaller than the pivot and elements larger than (or equal to) the pivot
3. Recursively sort each of the two parts (without the pivot)

- The expected time complexity is $O(n \log n)$
  (check CLRS for a proof, if you are interested to learn more)

- Due to randomization, we cannot find an instance that takes $\mathbf{\Theta(n^2)}$ time in every run.

QuickSort: In-place partition in $\Theta(n)$

Implementing https://www.youtube.com/watch?v=3San3uKKHgg algorithm:

```
template <class Comparable>
int partition(vector<Comparable> &v, int left, int right){
  // int tmp = left + rand() % (right-left+1); // randomized version
  // swap(v[tmp],v[left]);
  int pivotPos = left;
  do {
    while(v[right] >= v[pivotPos] && right > pivotPos)
       right--;
    if (right == pivotPos) return pivotPos;
    swap(v[right],v[pivotPos]);
    pivotPos = right;  left++;
    while(v[left] < v[pivotPos]) left++;
    if (left == pivotPos) return pivotPos;
    swap(v[left],v[pivotPos]);
    right = pivotPos-1;  pivotPos = left;
  }  while (pivotPos < right);
  return pivotPos;
}  // Next slide presents a simpler algorithm
```

QuickSort: In-place partition in $\Theta(n)$

CLRS version

```
template <class Comparable>
int partition(vector<Comparable> &v, int left, int right){

  // ---- randomized version --------------
  // int tmp = left + rand() % (right-left+1);
  // swap(v[tmp],v[left]);
  //-----------------------------

  int pivotPos = left;
  for (int j = left+1; j <= right; j++)
    if (v[j] <= v[left]) {
        pivotPos = pivotPos+1;
        swap(v[pivotPos],v[j]);
    }
  swap(v[left],v[pivotPos]);
  return pivotPos;
}
```

About QuickSort

- Quicksort has been proposed by **C. A. R. Hoare** (in 1961). It is **one of top 10 algorithms of 20th century** in science and engineering.

  https://en.wikipedia.org/wiki/Tony_Hoare, a.k.a, Tony Hoare, Turing Award in 1980

- It has been extensively studied and with many variants (e.g, R. Sedgewick, PhD Thesis, 1975, is about quicksort).
- E.g., the **median of three strategy** is another deterministic heuristic for selecting the pivot:
  - In each recursive call, it looks at the first, middle and last elements of the segment we have to sort, and chooses the **median of those three elements as the pivot**.
  - In that step, it also **sorts these three elements**, by swapping them if needed.
- There are also **hybrid versions**, e.g., combining quicksort with other methods, e.g., with insertion sort when $\texttt{right} - \texttt{left} + 1 \leq 10$.

Non comparison-based sorts

- **CountingSort** (count the number of elements of each type)
  - Assume that $v[j] \in \{1, 2, \ldots, k\}$, for some fixed k
  - **Sorting in linear time**: takes $\mathbf{\Theta(n+k)}$ time to sort n numbers in the range 1 to k.

    (if $k \ll n$ then $\Theta(n+k) = \Theta(n)$)
  - No comparisons between elements.
  - Interesting to have a **stable version**.
  - A **stable sort** keeps the relative order of items that have the same key. Example:

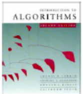
key	5	7	5	3	5	$\rightarrow$	3	5	5	5	7
id	"bi"	"bo"	"ro"	"te"	"ti"		"te"	"bi"	"ro"	"ti"	"bo"

- **RadixSort** (sort according to "digits")

CountingSort

Given an array A with n elements, such that $A[j] \in \{1, 2, \ldots, k\}$, builds B with the elements of A sorted. This version is **stable**.

(can be adapted to deal with other ranges, e.g., $[0, k-1]$ or $[a, b]$)



Analysis

$\Theta(k)$ **for** $i \leftarrow 1$ **to** k
    do $C[i] \leftarrow 0$

$\Theta(n)$ **for** $j \leftarrow 1$ **to** n
    do $C[A[j]] \leftarrow C[A[j]] + 1$

$\Theta(k)$ **for** $i \leftarrow 2$ **to** k
    do $C[i] \leftarrow C[i] + C[i-1]$

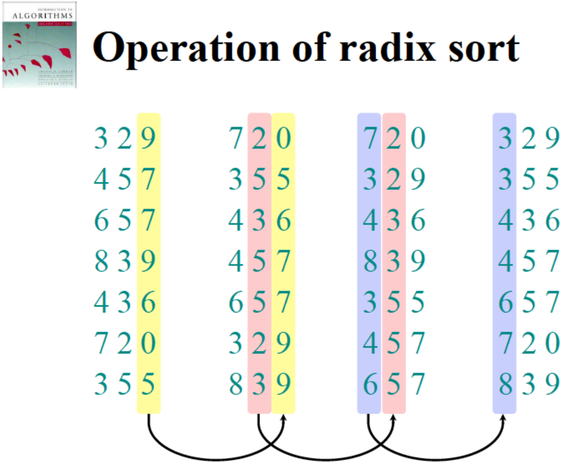
$\Theta(n)$ **for** $j \leftarrow n$ **downto** 1
    do $B[C[A[j]]] \leftarrow A[j]$
        $C[A[j]] \leftarrow C[A[j]] - 1$

$\Theta(n + k)$

Example: Stable CountingSort

```
void countingSort(std::vector<int> &v) {
  auto min_max = std::minmax_element(v.begin(),v.end());
  int min = *min_max.first;
  int max = *min_max.second;
  std::vector<int> count((max-min)+1, 0);  // init count

  for (auto v1:v) ++count[v1-min];

  for (auto it = count.begin()+1; it != count.end(); it++)
    *it += *(it-1);

  std::vector<int> vaux(v.size());
  for (auto it = v.rbegin(); it != v.rend(); it++) {
    vaux[count[(*it)-min]-1] = *it;
    count[(*it)-min]--;
  }

  std::copy(vaux.begin(),vaux.end(),v.begin());
}
```

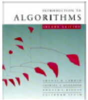
RadixSort



- Digit-by-digit-sort.
- Sort on *least-significant digit first* with auxiliary *stable* sort.

RadixSort

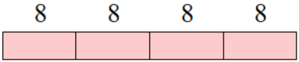
Sort n computer words of b bits each?



Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Example: 32-bit word



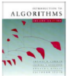
$r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits; or $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

RadixSort

Sort n computer words of b bits each?

(The following result is somehow out of the scope of AED. Refer to CLRS, if you are interested to learn more about RadixSort and variants)



Choosing r

$$T(n,b) = \Theta\left(\frac{b}{r}\left(n + 2^r\right)\right)$$

Minimize $T(n, b)$ by differentiating and setting to 0.

Or, just observe that we don't want $2^r \gg n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.

Choosing $r = \lg n$ implies $T(n, b) = \Theta(bn/\lg n)$.

- For numbers in the range from 0 to $n^d - 1$, we have $b = d \lg n \Rightarrow$ radix sort runs in $\Theta(d\,n)$ time.

Here, $\lg n$ means $\log_2 n$.

Summing up

- There are many (more) sorting algorithms
- The "best" algorithm depends on the case in question
- It is possible to combine several algorithms (**hybrids**)
  - Eg: RadixSort can have another algorithm as an internal step, as long as it is a stable sort (in case of a tie, maintain the initial order). (More about "String sorts": https://algs4.cs.princeton.edu/lectures/)
- In practice, in real implementations, this is what is done (combining): (Note: implementation depends on the compiler and its version)
  - Java: uses Timsort (MergeSort + InsertionSort)
  - C++ STL: uses IntroSort (QuickSort + HeapSort) + InsertionSort

STL algorithms sorting

Checking https://cplusplus.com/reference/algorithm/sort/

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first,
                  RandomAccessIterator last);
```

```
template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first,
                 RandomAccessIterator last,Compare comp);
```

- Sorts the elements in the range [`first`, `last`[into ascending order (i.e., `last` is not included)
- The elements are compared using `operator<` for the first version, and comp for the second. The latter must be a binary predicate that compares two objects, returning true if the first precedes the second (i.e., a strict weak ordering)
- Equivalent elements are not guaranteed to keep their original relative order (but available `std::stable_sort`).

Example

```
#include <vector>
#include<string>
using namespace std;

class Person {
private:
    string id;
    string name;
    int age;

public:
    Person(string i, string n, int a) :
        id(i), name(n), age(a) {}
   string getId() const {return id;}
   string getName() const {return name;}
   int getAge() const {return age;}

   bool operator<(const Person &p2) const  { // ...  next slide
   }
};
```

Example (cont.)

```
bool operator<(const Person &p2) const {
  if (age < p2.age ) return true;
  if (age > p2.age ) return false;
  if (name < p2.name) return true;
  if (name > p2.name) return false;
  return id < p2.id;
}
```

Example (cont.)

```
int main(){
  Person p1("12345", "John", 42);
  Person p2("42424", "Mary", 37);
  Person p3("55555", "Chariloe", 61);

  std::vector<Person> v = {p1, p2, p3};
  sort(v.begin(), v.end());

  for(auto it = v.begin(); it != v.end(); it++)
    std::cout << (*it).getName() << std::endl;
    // std::cout << it -> getName() << std::endl;

  return 0;
}
```

```
$./a.out
Mary
John
Chariloe
```

Example (cont.)

Defining a different order for two Person objects:

```
bool myComp(const Person &p1,const Person &p2)  {
   if (p1.getName() < p2.getName()) return true;
   if (p1.getName() > p2.getName()) return false;
   if (p1.getAge() < p2.getAge()) return true;
   if (p1.getAge()  > p2.getAge()) return false;
  return p1.getId()  < p2.getId();
}
```

In function main:

```
    sort(v.begin(), v.end(), myComp);
```

```
$./a.out
Chariloe
John
Mary
```