Centro de
Investigación en
Computación

Sorting

Ordenamiento

Algoritmos

Complejidad

Course

Operating System (with focus on Security)

Instructor

Acosta Bermejo Raúl et al.

Lecture notes

Instituto
Politécnico
Nacional

Table of contents (outline)

Tabla de contenido

1. Introduction
2. Bubble sort
3. Radix sort
4. Binary tree sort
5. Heapsort
6. Mergesort algorithm

Introduction

Introducción

Some problems become easier once elements are sorted.

- Identify statistical outliers.
- Binary search in a database.
- Remove duplicates in a mailing list.

There are a lot of obvious applications.

Non-obvious applications.

- Convex hull.
- Closest pair of points.
- Interval scheduling / interval partitioning.
- Minimum spanning trees (Kruskal's algorithm).
- ...

Introduction

Introducción

Lists of sorting algorithms (most known)

1. Bubble sort
2. Binary tree sort
3. Bucket sort
4. Heapsort
5. Insertion sort
6. Merge sort
7. Quicksort
8. Radix sort
9. Shell sort
10. Selection sort

En la wikipedia hay 24 (*https://en.wikipedia.org/wiki/Sorting_algorithm*).
https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento.

Introduction

Introducción

Referencias

Algunas genéricas son:

1. Visualgo

   Para ordenamientos

   https://visualgo.net/en/sorting

2. .

Bubble Sort

Ordenamiento de burbuja

Bubble sort

Ordenamiento de burbuja

Sorting problem

Given a list of *n* elements from a totally-ordered universe, rearrange them in ascending order.

The classic (historic) algorithm is:

n=9

```
for(int x=0; x<n; x++){
  for(int y=0; y<n-1; y++){
    if(array[y]>array[y+1]){
      swap( a[y], a[y+1] );
    }
  }
}
```

un registro antes del último
la proxima vez uno menos

0	1	2	3	4	5	6	7	8	9
7	2	1	8	6	3	4	8	11	9

x=0, **y=0**, (0, 1)

0	1	2	3	4	5	6	7	8	9
2	7	1	8	6	3	4	8	11	9

x=0, **y=1**, (1,2)

0	1	2	3	4	5	6	7	8	9
2	1	7	8	6	3	4	8	11	9

x=0, **y=2**, (1,2)

n-1

Bubble sort

Ordenamiento de burbuja

How many comparisons does the algorithm do?

How many swappings?

0	1	2	3	4	5	6	7	8	9
2	1	7	6	3	4	8	8	9	11

Worst-case:

Data is order descendently.

1th cicle: 6 swappings

0	1	2	3	4	5	6	7	8	9
1	2	6	3	4	7	8	8	9	11

2th cicle: 4 swappings

n-1, n-2, n-3, n-4, ... 2, 1

=> (n-1) * (n-2) * (n-3) * ... 1

= (n-1)! equiv (n-1)! x n (si n>0)

$$= \prod_{k=1}^{n} k \quad = O(n^2)$$

How do you debug the program in order to mesure the time?

Bubble sort

Ordenamiento de burbuja

```
bubbleSort( A : list of sortable items )
   n = length(A)
   repeat
      swapped = false;
      for i = 1 to n-1 do
         if A[i-1] > A[i] then
            swap( A[i-1], A[i] )
            swapped = true;     /* remember something changed */
         end if
      end for
   until not swapped.
```

Complexity

Worst-case: $O(n^2)$

Best-case: O(n)

Datos ya ordenados!

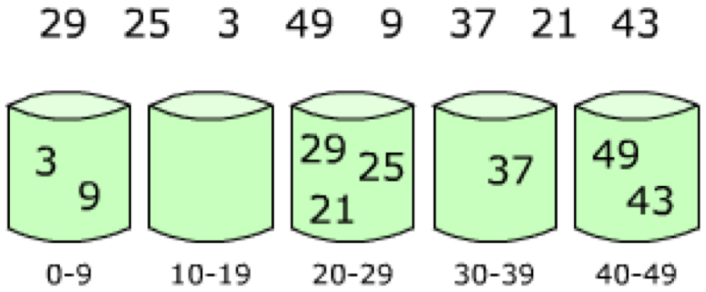
Bucket Sort

Ordenamiento de canastas

Bucket sort

Ordenamiento de canastas (cubo, cubeta)

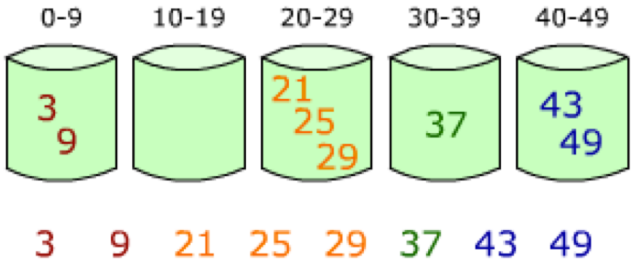
- It works by distributing the elements of an array into a number of buckets.
- Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.



Elements are distributed among bins

Then, elements are sorted within each bin

deposito



Bucket sort

Ordenamiento de canastas

```
bucketSort(array, n)
  buckets ← new array of n empty lists
  for i = 0 to (length(array)-1) do
    insert array[i] into buckets[msbits(array[i], k)]
  for i = 0 to n - 1 do
     nextSort(buckets[i])
  return the concatenation of buckets[0], ...., buckets[n-1]
```

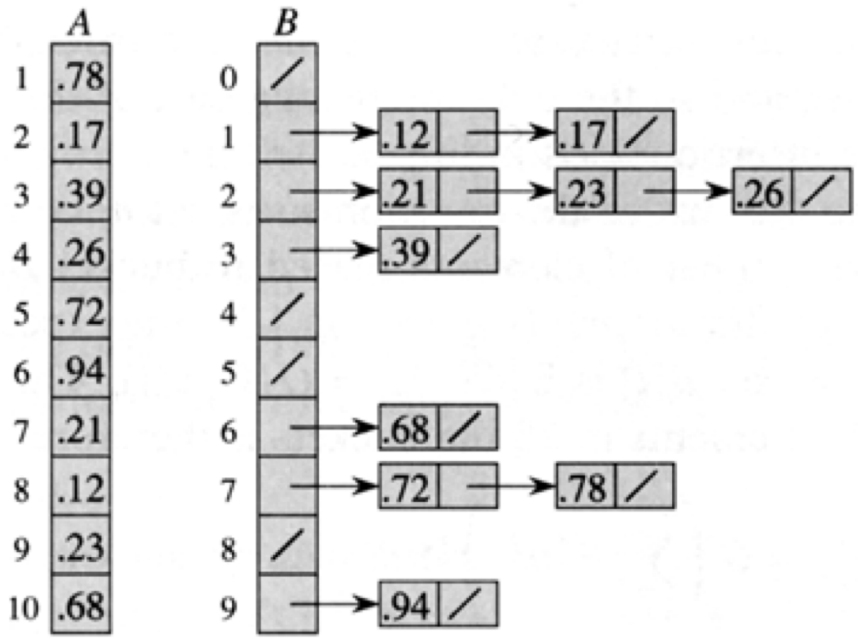
The function

- *msbits*(x,k)
  Returns the k most significant bits of x (floor(x/2^(size(x)-k)));
- nextSort
  It is a sorting function. Usually bucket sort or insert sort.

Bucket sort

Ordenamiento de canastas

Implementation



Some implementations use hash table.

Bucket sort

Ordenamiento de canastas (cubo, cubeta)

- Complexity
  - Worst-case: $O(n^2)$
  - Best-case: $\Omega(n+k)$
  - Average-case: $\Theta(n+k)$

El algoritmo del cartero (variantes)

- El nombre de este algoritmo viene del ejemplo de las oficinas postales. Cuando hay que clasificar una carta para que llegue a su destino primero se clasifica según el país, luego la ciudad, después la calle, etc. (código postal).
- Es una variante del bucket sort utilizada cuando los elementos a ordenar disponen de varias claves y/o subclaves.
- La complejidad computacional es de O(cn), siendo c el número de claves que se utilizan para clasificar.

Bucket sort

Ordenamiento de canastas (cubo, cubeta)

- **Ventajas**
  - Se puede usar cuando los datos no caben en memoria.
  - Para ordenar números reales:
    - https://iq.opengenus.org/time-and-space-complexity-of-bucket-sort/
- **Referencias**
  - Para una descripción detallada de la complejidad

    https://en.wikipedia.org/wiki/Bucket_sort
  - Teoria y Ejemplos
    - https://www.geeksforgeeks.org/bucket-sort-2/
  - Artículos
    - Burnetas, A., Solow, D. & Agarwal, R. An analysis and implementation of an efficient in-place bucket sort. *Acta Informatica* 34, 687–700 (1997). https://doi.org/10.1007/s002360050103.
    - N. Faujdar and S. Saraswat, "The detailed experimental analysis of bucket sort," 2017 7th International Conference on Cloud Computing, Data Science & Engineering - Confluence, 2017, pp. 1-6, doi: 10.1109/CONFLUENCE.2017.7943114.
    - Upper Tail Analysis of Bucket Sort and Random Tries, Ioana O. Bercea, Guy Even. arXiv:2002.10499, 2020.

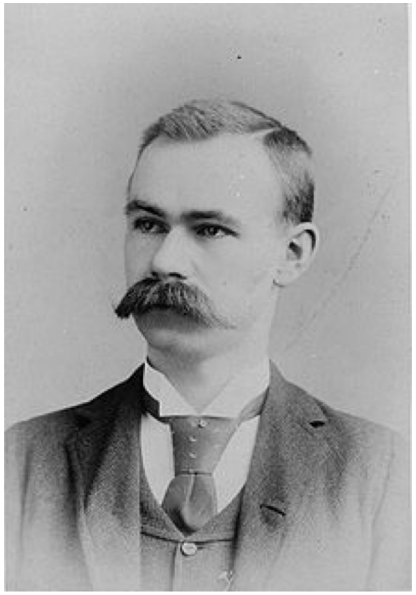
Radix Sort

Ordenamiento

Radix sort

Ordenamiento

- Computers internally represent all of their data as electronic representations of binary numbers.
- Processing the digits of integer representations by groups of binary digit representations is most convenient.
- Two classifications of radix sorts are:
  - Least Significant Digit (LSD) radix sorts. It process the integer representations starting from the least digit and move towards the most significant digit.
  - Most significant Digit (MSD) radix sorts. It work the other way around.
- It dates back as far as 1887 to the work of **Herman Hollerith** on tabulating machines.
- Latter in 1954 at MIT by Harold H. Seward.



Herman Hollerith
1860-1929
Considerado el
primer informático

Radix sort

Ordenamiento

Example

493, 812, 715, 710, 195, 437, 582, 340, 385

we use
1th Digit

unidades

Digit	Sublist
0	340, 710
1	
2	812, 582
3	493
4	
5	715, 195, 385
6	
7	437
8	
9	

Notice

- The numbers were added onto the list in the order that they were found.
- That is why the numbers appear to be unsorted in each of the sublists above.

1000-0001-0010, 0111-0001-0000, 1000-0001-0010, 0011-0100-0000

Radix sort

Ordenamiento

Example

493, 812, 715, 710, 195, 437, 582, 340, 385

Digit	Sublist
0	340, 710
1	
2	812, 582
3	493
4	
5	715, 195, 385
6	
7	437
8	
9	

Then, we gather the sublists (in order from the 0 sublist to the 9 sublist) into the main list again.

340, 710, 812, 582, 493, 715, 195, 385, 437

Radix sort

Ordenamiento

Example

340, 710, 812, 582, 493, 715, 195, 385 437

2nd Digit

decenas

Digit	Sublist
0	
1	710, 812, 715
2	
3	437
4	340
5	
6	
7	
8	582, 385
9	493, 195

710, 812, 715, 437, 340, 582, 385, 493, 195

Radix sort

Ordenamiento

Example

710, 812, 715, 437, 340, 582, 385, 493, 195

3th Digit

centenas

Digit	Sublist
0	
1	195
2	
3	340 385
4	437 493
5	587
6	
7	710 715
8	812
9	

195, 340, 385, 437, 493, 582, 710, 715, 812

Radix sort

Ordenamiento

- It works well for numbers, letters or strings.
- Disadvantages
  - More space than other algo for sublist.
  - If the numbers are not of the same length, then a test is needed to check for additional digits that need sorting.
- **Referencias**

  English versions of Wikipedia are differente (and often better) than spanish:
  - https://en.wikipedia.org/wiki/Radix_sort

    Worst-case : O(wn) n keys which are integers of word size w.
  - https://www.geeksforgeeks.org/radix-sort/

    $O((n+b) * \log_b(k))$ where b is the base, k max possibel values

    $k<=n^C$ (constant)
  - https://www.programiz.com/dsa/radix-sort

    Best, worst, average: O(n+k)

Radix sort

Ordenamiento

- **Tarea optativa**
  - https://iq.opengenus.org/time-and-space-complexity-of-radix-sort/
  - Realizar un análisis de entre 5 y 10 sitios para determinar cual es la complejidad real del algoritmo.
  - Hacer el análisis para los 3 casos: peor, mejor y promedio.
  - Determinar que sitios esta mal la complejidad o no se explica suficientemente en que condiciones se alcanza dicha complejidad, es decir, si simplificando la formula ”mas compleja” se llega a la sencilla.

Mergesort

Ordenamiento por mezcla

Mergesort

Ordenamiento por mezcla

Fue un matemático húngaro-estadounidense. 1903-1957

- Fue desarrollado en 1945 por John Von Neumann.
- Mergesort is a divide and conquer algorithm.
- Mergesort parallelizes well due to use of previous algorithm.
- It has an average and worst-case performance of O(n log n).
- Many implementationes:
  - https://en.wikibooks.org/wiki/Algorithm_Implementation/Sorting/Merge_sort
  - https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort.

Mergesort

Ordenamiento por mezcla

Pseudo-code

```
mergesort(array:m)
   var list left, right, result
Begin
    if length(m) ≤ 1
        return m
    else
        var middle = length(m) / 2
        for each x in m up to middle - 1
            add x to left
        for each x in m at and after middle
            add x to right
        left = mergesort(left)
        right = mergesort(right)
        if last(left) ≤ first(right)
            append right to left
            return left
        result = merge(left, right)
        return result
End
```

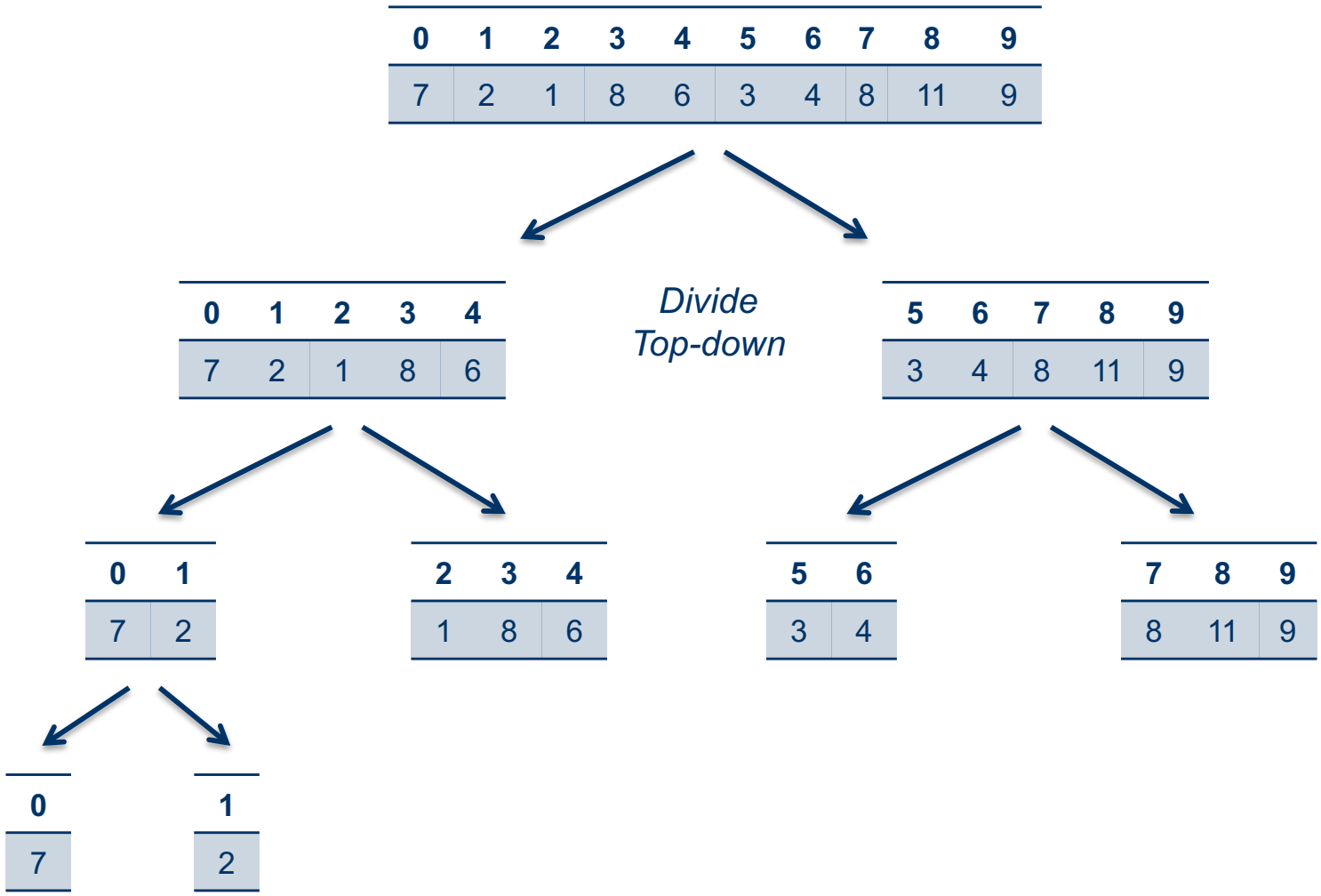
División del problema (partes iguales)

Llamadas recursivas

Unión de soluciones

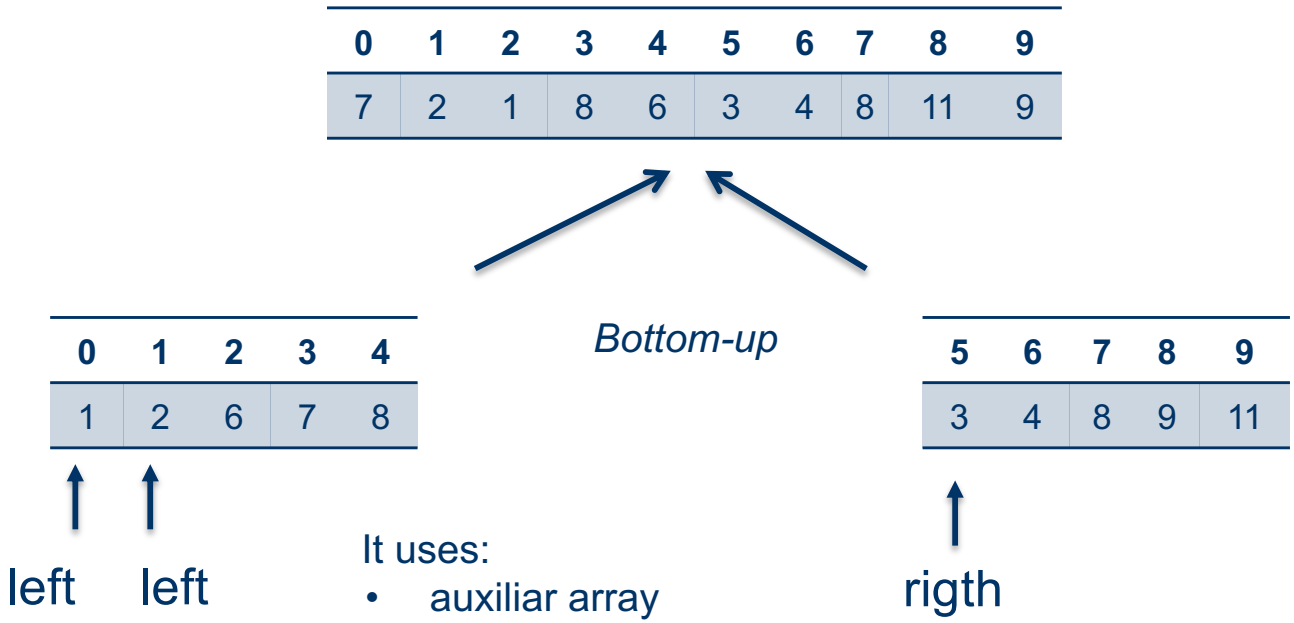
Mergesort

Ordenamiento por mezcla



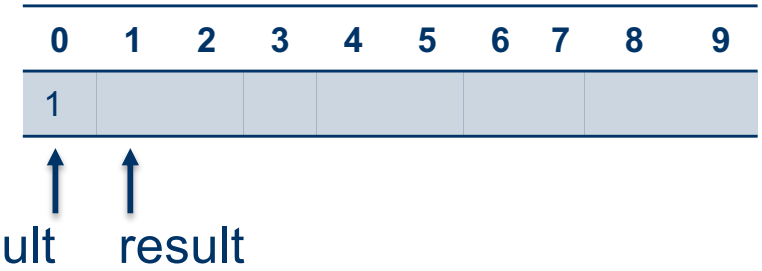
Mergesort

Ordenamiento por mezcla



It uses:

- auxiliar array
  - 3 indices



- Choose the minor data (left & rigth)
- Put it in result
- Increment result & minor

Mergesort

Ordenamiento por mezcla

Definition

T (n) = max number of compares to mergesort a list of size ≤ n.

Note

T (n) is monotone (conserva el orden ≤) nondecreasing.

Mergesort recurrence

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ T(\,n/2\,) + T(\,n/2\,) + n & \text{Otherwise} \end{cases}$$

Solution

$T(n)$ is $O(n \log_2 n)$.

How to Proof that recursive T(n) is Big-O?

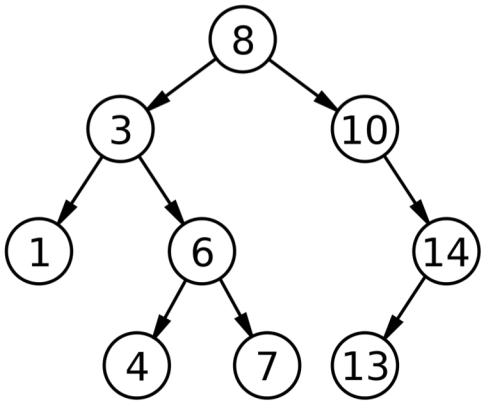
Binary Tree Sort

Ordenamiento de Árbol Binario

Binary tree sort

Ordenamiento de árbol binario

- A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished sub-trees, commonly denoted left and right.
- The tree additionally satisfies the binary search tree property, which states that the key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in the right sub-tree.



Operations
- Searching
- Insertion
- Deletion
- TRaversal
- Verification

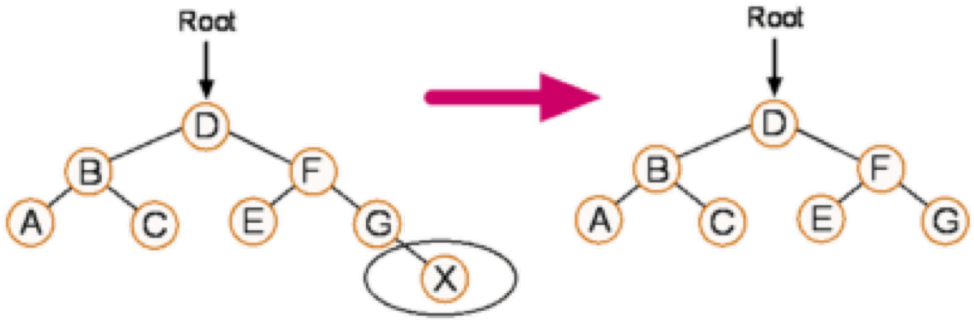
- A **tree sort** is a sort algorithm that builds a **binary search tree** from the elements to be sorted, and then traverses the tree (**in-order**) so that the elements come out in sorted order.

Binary tree sort

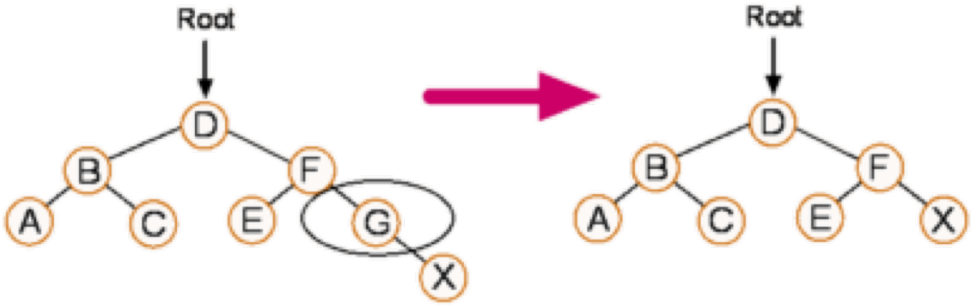
Operaciones

Operation **Deletion**

Leaf deletion



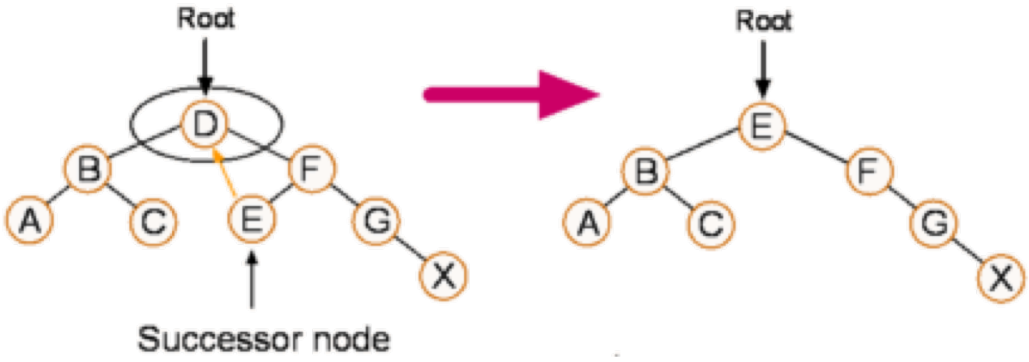
Deleting a node with a single child



Deleting a node with two children:

- Locate the successor node on the right-hand side and replace the deleted node (D) with the succesor Finally remove the successor node.
- Locate the predecessor on the left-hand side and ..

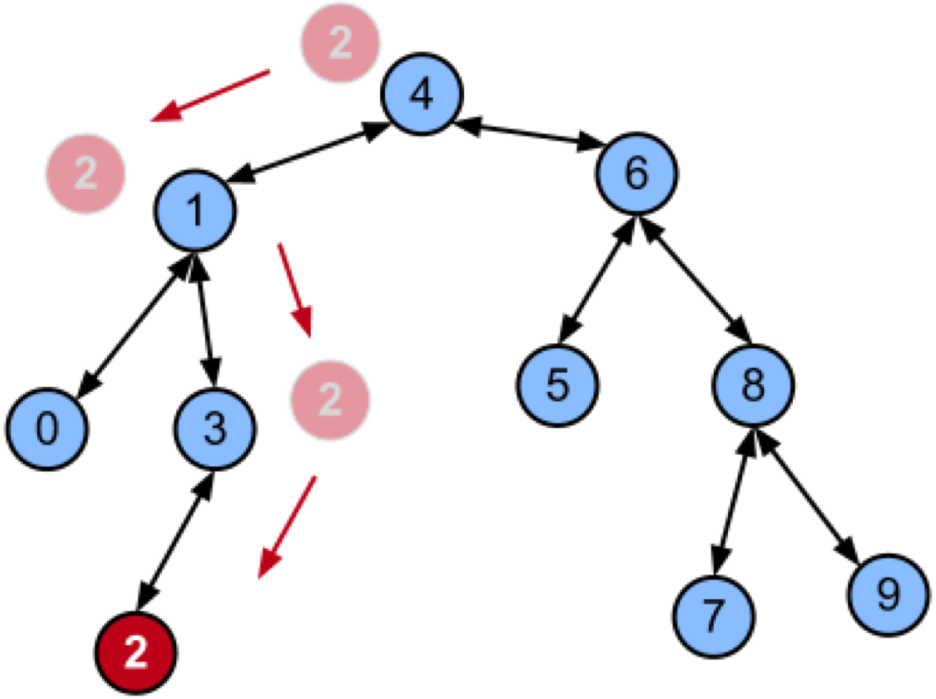
Two children



Binary tree sort

Operaciones

Operation **Insert**



Binary tree sort

Ordenamiento de árbol binario

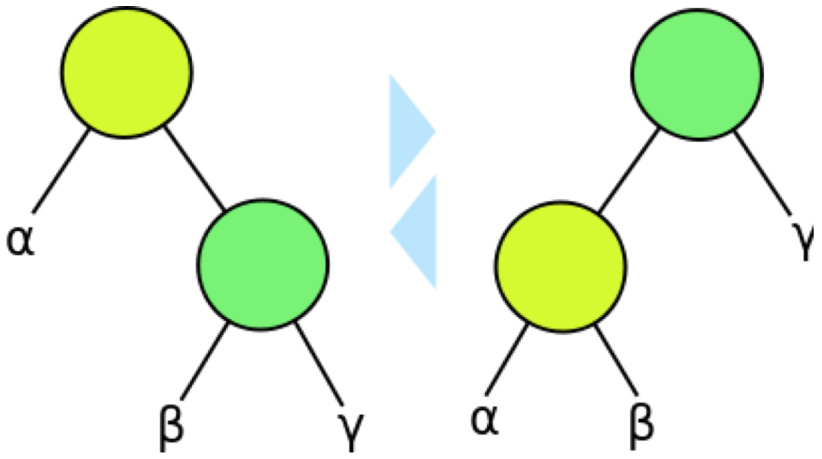
- **Complexity**
  - Adding one item to a binary search tree is on average an $O(\log n)$ process (high), so adding n items is an $O(n \log n)$ process, making tree sort a 'fast sort'.
  - But adding an item to an unbalanced binary tree needs $O(n)$ time in the worst-case, when the tree resembles a linked list (degenerate tree), causing a worst case of $O(n^2)$ for this sorting algorithm.
  - This worst case occurs when the algorithm operates on an already sorted set, or one that is nearly sorted.
- The worst-case behaviour can be improved upon by using a **self-balancing** binary search tree. Using such a tree, the algorithm has an $O(n \log n)$ worst-case performance.

Binary tree sort

Ordenamiento de árbol binario

Optimal binary search trees

- If we do not plan on modifying a search tree, and we know exactly how often each item will be accessed, we can construct an *optimal binary search tree*, which is a search tree where the **average cost** of looking up an item (the *expected search cost*) is minimized.
- Self-balancing



Binary tree sort

Ordenamiento de árbol binario

Splay tree

- It is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again.
  - It defines a Zig-zag rotation.
  - Other balanced trees: AVL, Red-black.
- When using a splay tree as the binary search tree, the resulting algorithm (called **Splaysort**) has the additional property that it is an *adaptive sort*:
  - Its running time is $O(n \log n)$ for inputs that are nearly sorted.

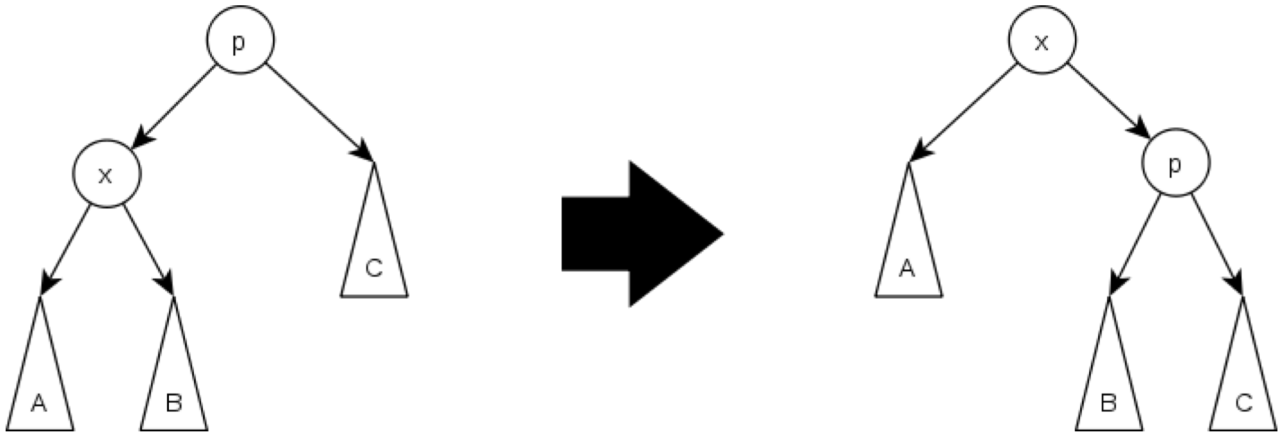
Link

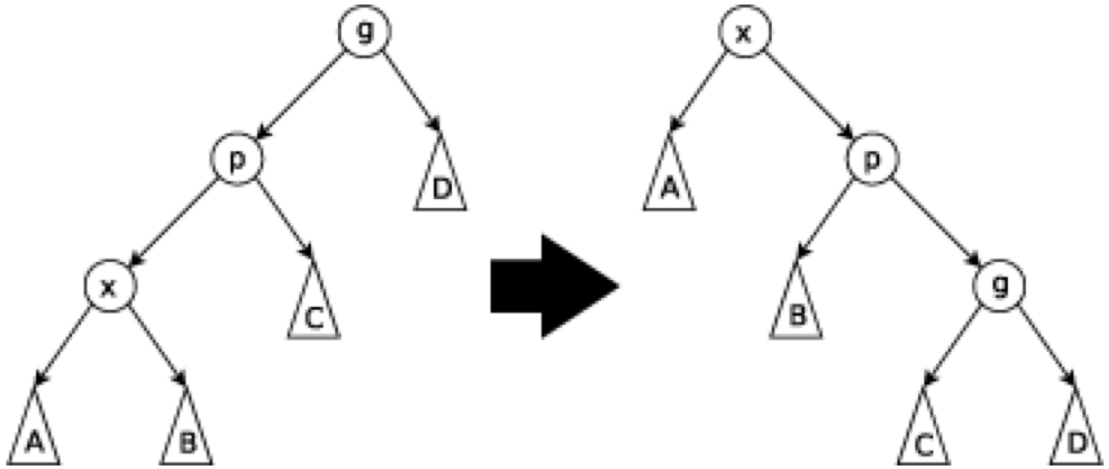
- https://en.wikipedia.org/wiki/Splay_tree
- https://www.geeksforgeeks.org/splay-tree-set-1-insert/

Binary tree sort

Ordenamiento de árbol binario

Splay tree





Heapsort

Ordenamiento de montículos

Heapsort

Ordenamiento de montículos

Introduction

- Heapsort was invented by J. W. J. Williams in 1964.
- This was also the birth of the heap, presented already by Williams as a useful data structure in its own right.
- In the same year, R. W. Floyd published an improved version that could sort an array **in-place**, continuing his earlier research into the tree sort algorithm.
- Complexity
  - Best-case, Average & worst-case: O(n lg n)

Heapsort

Ordenamiento de montículos

What is a “heap”?

- Definitions of heap:
  1. A large area of memory from which the programmer can allocate blocks as needed, and deallocate them (or allow them to be garbage collected) when no longer needed
  2. A balanced, left-justified binary tree in which no node has a value greater than the value in its parent.
- These two definitions have little in common.
- Heapsort uses the second definition.

Heapsort

Ordenamiento de montículos

- Goal:
  - Sort an array using heap representations.
- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains.

Heapsort

Ordenamiento de montículos

Pseudo-code Ver. 1

```
function heapsort(array A[0..n])
    montículo M
    integer i;
    for i = 0..n
      insertar_en_monticulo(M, A[i])
    for i = 0..n
      A[i] = extraer_cima_del_monticulo(M)
    return A
```

Paso 1: construir el heap (árbol)

Paso 2: Convertirlo en una lista ordenada

Heapsort

Ordenamiento de montículos

Pseudo-code Ver. 2

```
function heapSort(a, count)
      heapify(a, count)
      end := count - 1
      while end > 0 do
        swap(a[end], a[0])
        end := end – 1;
        siftDown(a, 0, end);

function heapify(a,count)
   start := (count - 2) / 2
   while start ≥ 0 do
     siftDown(a, start, count-1);
     start := start - 1;
```

```
function siftDown(a, start, end)
   root := start;
   while root * 2 + 1 ≤ end do
     child := root * 2 + 1;
     if child + 1 ≤ end and a[child] < a[child + 1] then
       child := child + 1;
     if a[root] < a[child] then)
       swap(a[root], a[child]);
       root := child;
     else
       return;
```

Heapsort

Ordenamiento de montículos

Example

- http://faculty.simpson.edu/lydia.sinapova/www/cmsc250/LN250_Weiss/L13-HeapSortEx.htm

Heapsort

Ordenamiento de montículos

Time Analysis

- Build Heap Algorithm will run in O(n) time
- There are *n*-1 calls to Heapify each call requires O(log *n*) time
- Heap sort program combine:
  - Build Heap program (*O(n)*), and
  - Heapify (*O(log n)*), therefore.
- Total time complexity: O(n log n).

Insertion Sort

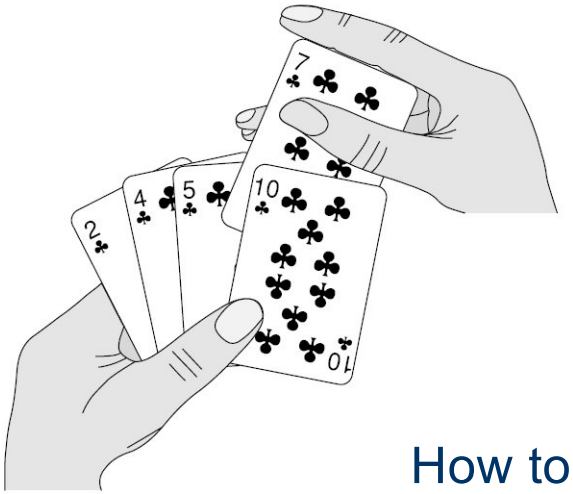
Ordenamiento por inserción

Insertion sort

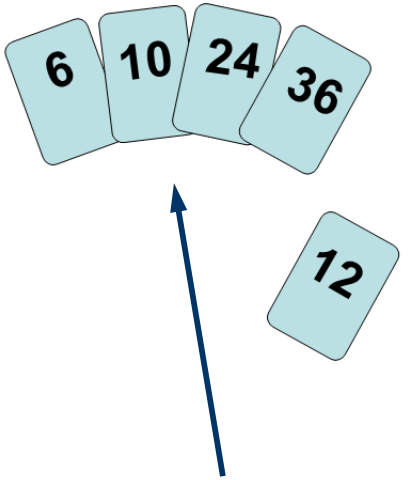
Ordenamiento por inserción

Main idea

The idea is similar to the way we sort playing cards in our hands.
The left side is ordered.



How to make space in arrays efficiently?



To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion sort

Ordenamiento por inserción

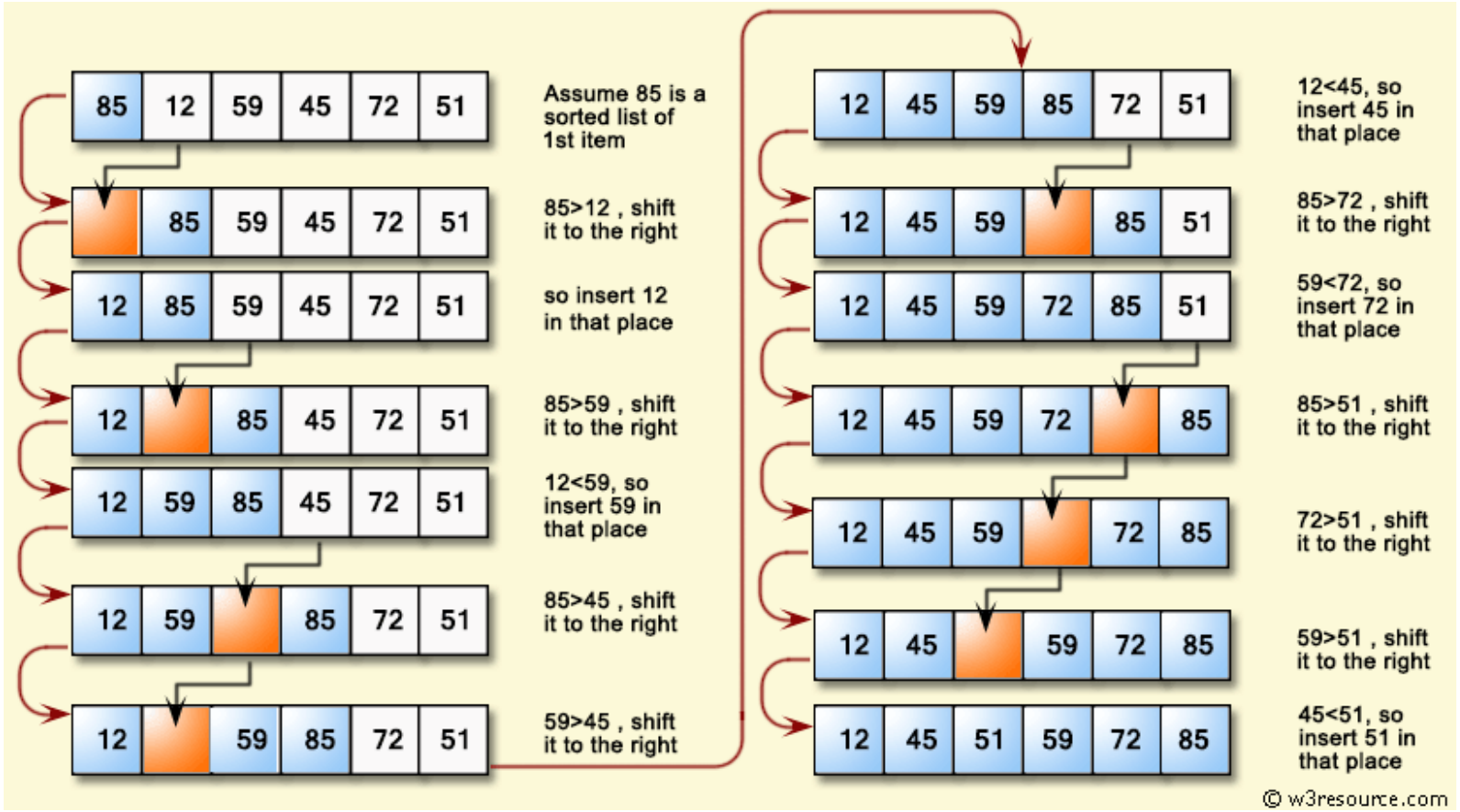
- It is a simple sorting algorithm that builds the final sorted array (or list) one item at a time.
- It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:
  - Simple implementation: Bentley shows a three-line C version, and a five-line optimized version.
  - More efficient in practice (for small data sets) than most other simple quadratic algorithms such as selection sort or bubble sort.
- Complexity
  - Worst-case: $O(n^2)$
  - Best-case: $O(n)$

insertionSort(arr, n)
Loop from i = 1 to n-1
  Pick element arr[i] and
   insert it into sorted sequence arr[0…i-1]

Insertion sort

Ordenamiento por inserción

Example



Insertion sort

Ordenamiento por inserción

Pseudocode

```
Insert(A, size)
for i ← 1 to length(A)-1
    j ← I
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j – 1
    end while
end for
```

The most common variant of insertion sort, which operates on **arrays**, can be described as follows:

- It operates by beginning at the end of the sequence and **shifting** each element one place to the right until a suitable position is found for the new element.
- The function has the side effect of **overwriting** the value stored immediately after the sorted sequence in the array.
- The ordered sequence into which the element is inserted is **stored** at the **beginning** of the array in the set of indices already examined.

Insertion sort

Ordenamiento por inserción

Variants

- **D.L. Shell**
  - He made substantial improvements to the algorithm; the modified version is called **Shell sort**.
  - The sorting algorithm compares elements separated by a distance that decreases on each pass.
  - Shell sort has distinctly improved running times in practical work, with two simple variants requiring $O(n^{3/2})$ and $O(n^{4/3})$ running time.
- In 2006 **Bender, Martin Farach-Colton, and Mosteiro** published a new variant of insertion sort called **library sort** or *gapped insertion sort.*
  - It leaves a small number of unused spaces spread throughout the array.
  - The benefit is that insertions need only shift elements over until a gap is reached.
  - The authors show that this sorting algorithm runs with high probability in $O(n \log n)$ time.

Insertion sort

Ordenamiento por inserción

Referencias

- Ver la implementación de **listas**:
  https://en.wikipedia.org/wiki/Insertion_sort
- ***Knuth, Donald*** *(1998), "5.2.1: Sorting by Insertion", The Art of Computer Programming, 3. Sorting and Searching (second ed.), Addison-Wesley, pp. 80–105, ISBN 0-201-89685-0*.

Quicksort

Ordenamiento rápido

Quicksort

Ordenamiento rápido

Tony Hoare
1934
British computer scientist

Hoare logic. CSP
OCCAM

- Developed by Tony Hoare in 1959, with his work published in 1961.
- It is a **divide and conquer** algorithm.
- It can operate in-place on an array, requiring small additional amounts of memory to perform the sorting.
- It gained widespread adoption, appearing, for example, in Unix as the default library sort function.
- Complexity
  - Worst-case: $O(n^2)$ a rare behavior
  - Average: O(n lg n)
  - Best-case: O(n lg n)

Link
https://en.wikipedia.org/wiki/Quicksort

The algorithm was developed in 1959 by he while in the **Soviet Union**, as a **visiting student** at Moscow State University. At that time, Hoare worked in a project on **machine translation** for the National Physical Laboratory. As a part of the translation process, he needed to sort the words of Russian sentences prior to looking them up in a Russian-English dictionary which was already sorted in alphabetic order on magnetic tape.

Quicksort

Ordenamiento rápido

Description of the Algorithm

- It first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.
- The steps are:
  - Pick an element, called a **pivot**, from the array.
  - **Partitioning**: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
  - **Recursively** apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- The base case of the recursion is arrays of size zero or one, which never need to be sorted.
- The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

Quicksort

Ordenamiento rápido

Partition schemes

- Lomuto (Nico Lomuto/Popularized by Bentley & Cormen in their books)
  - This scheme chooses a pivot which is typically the last element in the array. The algorithm maintains the index to put the pivot in variable i and each time it finds an element less than or equal to pivot, this index is incremented and that element would be placed before the pivot.
  - As this scheme is more compact and easy to understand, it is frequently used in introductory material, although it is less efficient than Hoare's original scheme.
- Hoare
  - It uses two indices that start at the ends of the array being partitioned, then move toward each other, until they detect an inversion: a pair of elements, one greater than the pivot, one smaller, that are in the wrong order relative to each other. The inverted elements are then swapped.
  - When the indices meet, the algorithm stops and returns the final index.
  - There are many variants of this algorithm.

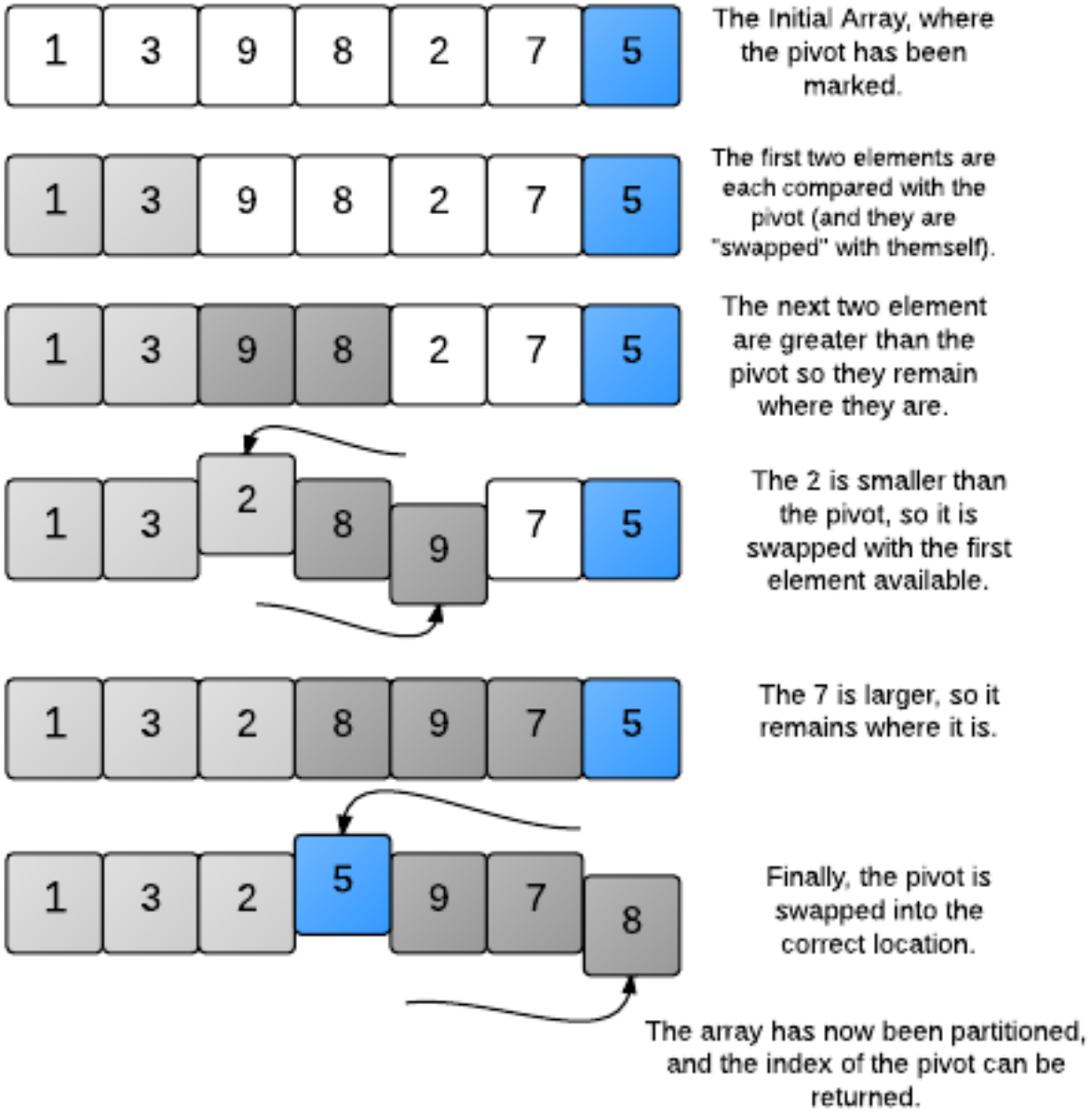
Quicksort

O. rápido

Lomuto partition scheme

There have been various variants proposed to boost performance including:

- Various ways to select pivot.
- Deal with equal elements.
- Use other sorting algorithms such as Insertion sort for small arrays.
- And so on.

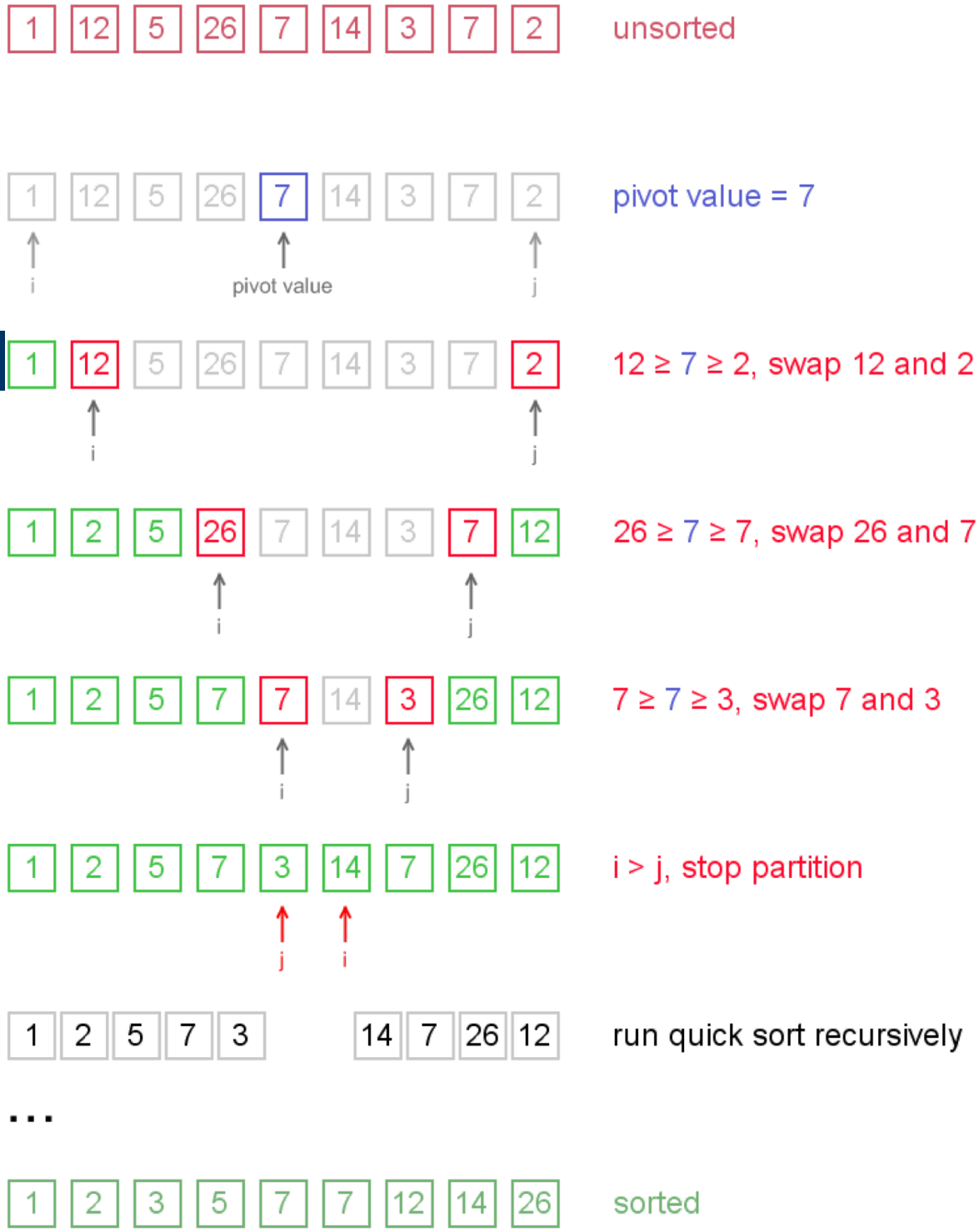


Quicksort

O. rápido

Hoare partition scheme

- Hoare's scheme is more efficient than Lomuto because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal.
- Both schemes causes Quicksort to degrade to $O(n^2)$ when the input array is already sorted
- It also doesn't produce a stable sort.



Quicksort

Ordenamiento rápido

Algorithm

Lomuto partition scheme

```
quicksort(A, lo, hi)
  if lo < hi then
      p := partition(A, lo, hi)
      quicksort(A, lo, p – 1)
      quicksort(A, p + 1, hi)
```

```
partition(A, lo, hi)
    pivot := A[hi]
    i := lo // place for swapping
    for j := lo to hi – 1 do
        if A[j] ≤ pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Sorting the entire array is accomplished by **quicksort**(A, 1, length(A)).

Quicksort

Ordenamiento rápido

Algorithm

Hoare partition scheme

```
quicksort(A, lo, hi)
  if lo < hi then
      p := partition(A, lo, hi)
      quicksort(A, lo, p)
      quicksort(A, p + 1, hi)
```

Diferencia aquí

```
partition(A, lo, hi)
    pivot := A[lo]
    i := lo – 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j – 1
        while A[j] > pivot
        if i >= j then
            return j
        swap A[i] with A[j]
```

Quicksort

Ordenamiento rápido

Space complexity

- The space used by quicksort depends on the version used.
- The in-place version of quicksort has a space complexity of $O(\log n)$, even in the worst case, when it is carefully implemented using the following strategies:
  - In-place partitioning is used. This unstable partition requires $O(1)$ space.
  - After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most $O(\log n)$ space.

Time complexity

- Worst-case analysis
- Best-case analysis
- Average-case analysis

Selection sort

Ordenamiento por selección

Selection sort

Ordenamiento por selección

- Complexity
  - Best-case & worst-case & : $O(n^2)$
- .

Shell sort

Ordenamiento

Shellsort

Ordenamiento

Donald Shell
1924-2015
American computer scientist

- Donald Shell published the first version of this sort in 1959.
- It is an **in-place** comparison sort. It can be seen as either a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort).
- The method starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. Starting with far apart elements, it can move some out-of-place elements into position faster than a simple nearest neighbor exchange.
- The running time of Shellsort is heavily dependent on the gap sequence it uses.
- For many practical variants, determining their time complexity remains an open problem.

Shellsort

Ordenamiento

Description

- Shellsort is a generalization of insertion sort that allows the exchange of items that are far apart.
- The idea is to arrange the list of elements so that, starting anywhere, considering every *h*th element gives a sorted list. Such a list is said to be *h*-sorted.
- Equivalently, it can be thought of as *h* interleaved lists, each individually sorted. Beginning with large values of *h*, this rearrangement allows elements to move long distances in the original list, reducing large amounts of disorder quickly, and leaving less work for smaller*h*-sort steps to do.
- If the file is then *k-sorted* for some smaller integer *k*, then the file remains *h*-sorted. Following this idea for a decreasing sequence of *h* values ending in 1 is guaranteed to leave a sorted list in the end.

Shellsort

Ordenamiento

Pseudocode

```
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]

# Start with the largest gap and work down to a gap of 1
foreach (gap in gaps) {
    # Do a gapped insertion sort for this gap size.
    # The first gap elements a[0..gap-1] are already in gapped order
    # keep adding one more element until the entire array is gap sorted
    for (i = gap; i < n; i += 1) {
        # add a[i] to the elements that have been gap sorted
        # save a[i] in temp and make a hole at position i
        temp = a[i]
        # shift earlier gap-sorted elements up until the correct location for a[i] is found
        for (j = i; j >= gap and a[j - gap] > temp; j -= gap) {
            a[j] = a[j - gap]
        }
        # put temp (the original a[i]) in its correct location
        a[j] = temp
    }
}
```

Shellsort

Ordenamiento

- Shellsort is unstable: it may change the relative order of elements with equal values.
- It is an adaptive sorting algorithm in that it executes faster when the input is partially sorted.
- Links
  - https://en.wikipedia.org/wiki/Shellsort

Samplesort

Ordenamiento

Samplesort

Ordenamiento

- It is a sorting algorithm that is a **divide and conquer** algorithm often used in parallel processing systems.
  - Conventional D&C sorting algorithms partitions the array into sub-intervals or buckets.
  - The buckets are then sorted individually and then concatenated together.
  - However, if the array is non-uniformly distributed, the performance of these sorting algorithms can be significantly throttled.
- It addresses this issue by selecting a sample of size s from the n-element sequence, and determining the range of the buckets by sorting the sample and choosing m -1 elements from the result.
- These elements (called **splitters**) then divide the sample into m equal-sized buckets.
- It is described in the 1970 paper, "Samplesort: A Sampling Approach to Minimal Storage Tree Sorting", by W. D. Frazer and A. C. McKellar.

Samplesort

Ordenamiento

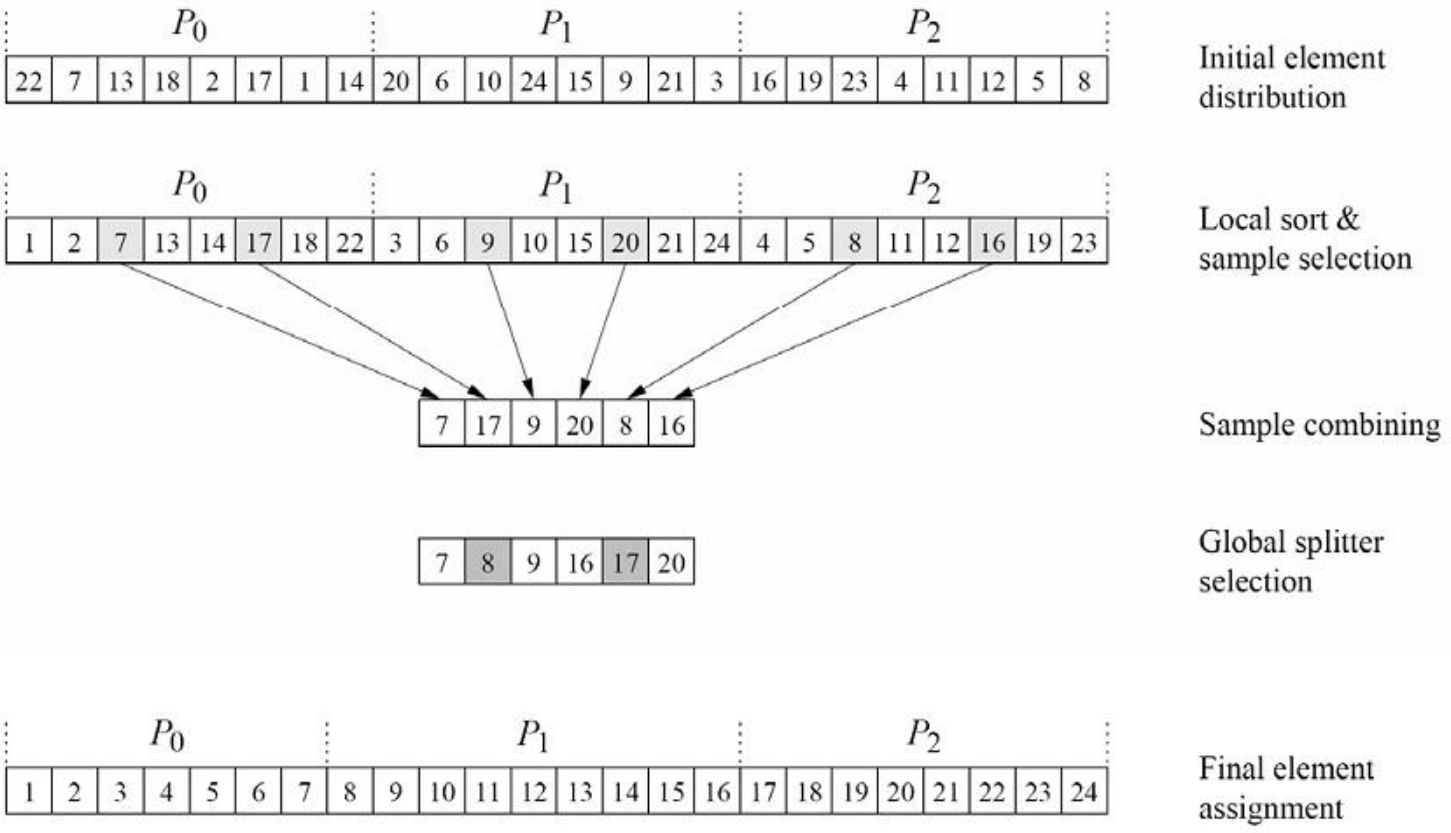
Idea of the Algorithm

- Samplesort can be thought of as a refined quicksort.
- Where quicksort partitions its input into two parts at each step, based on a single value called the **pivot**, samplesort instead takes a larger sample from its input and divides its data into buckets accordingly.
- Like quicksort, it then recursively sorts the buckets.
- To devise a samplesort implementation, one needs to decide on the number of buckets p. When this is done, the actual algorithm operates in **three phases**:
  - Sample $p-1$ elements from the input (the *splitters*). Sort these; each pair of adjacent splitters then defines a *bucket*.
  - Loop over the data, placing each element in the appropriate bucket. (This may mean: send it to a processor, in a multiprocessorsystem.)
  - Sort each of the buckets.
- The full sorted output is the concatenation of the buckets.

Samplesort

Ordenamiento

Graphic representation



Samplesort

Ordenamiento

- A common strategy is to set p equal to the number of processors available.
- The data is then distributed among the processors, which perform the sorting of buckets using some other, sequential, sorting algorithm.

Complexity

- Find the splitters $O(\frac{n}{p} + \log(p))$
- Send to buckets $O(p)$ reading nodes; $O(\log(p))$ broadcasting; $O(\frac{n}{p}\log(p))$ binary search for all keys; $O(\frac{n}{p})$ send keys to bucket
- Sort buckets $O(\frac{c}{p})$ where c is complexity of underlying sequential sorting method

Bibliography

Bibligrafia

Artículos

Bibliography

Bibliografía

Una muestra de artículos:

- Analysis of Pivot Sampling in Dual-Pivot Quicksort: A Holistic Analysis of Yaroslavskiy's Partitioning Scheme
  - 2016 Algorithmica.
  - Nebel, M.E. Wild S., Martinez C.
- Deterministic Sample Sort for GPUs
  - Frank Dehne, Hamidreza Zaboli. 2010.
  - https://arxiv.org/abs/1002.4464

Conclusions

Conclusiones

Complexities

Complejidades

Resume of complexities

Algorithm	Worst-case	Averag-case	Best-case
Bubble sort	$O(n^2)$	$O(n^2)$	O(n) variante
Binary tree sort			
Bucket sort Algoritmo del cartero	$O(n^2)$ O(cn)	$\Theta(n+k)$	$\Omega(n+k)$
Quicksort			
Mergesort			
Heapsort			
Insertion sort			
Radix sort	O(wn)		
Shell sort			
Selection sort			

K number of bits

n keys which are integers of word size w.

Conclusions

Conclusiones

If we compare sorting algorithms, we have:

1. Bubble sort — Iterativo
2. Binary tree sort
3. Bucket sort
4. Quicksort
5. Mergesort — Divide & Conquer
6. Heapsort
7. Insertion sort
8. Radix sort
9. Shell sort
10. Selection sort

The end

Contacto

Raúl Acosta Bermejo

http:www.cic.ipn.mx
http://www.ciseg.cic.ipn.mx/

racostab@ipn.mx
racosta@cic.ipn.mx

57-29-60-00
Ext. 56652