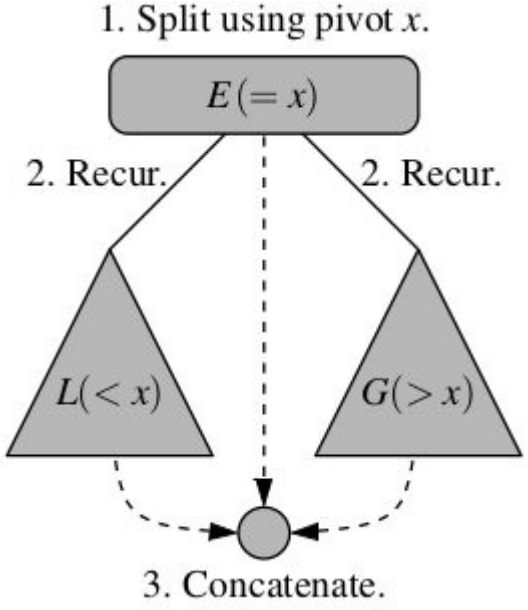
Renteria Magaña Rayni Damian
Algoritmos de ordenación internos
High-Level Description of Quick-Sort

The quick-sort algorithm sorts a sequence S using a simple recursive approach. The main idea is to apply the divide-and-conquer technique, whereby we divide S into subsequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation. In particular, the quick-sort algorithm consists of the following three steps (see Figure 12.8):

1. Divide: If S has at least two elements (nothing needs to be done if S has zero or one element), select a specific element x from S, which is called the pivot. As is common practice, choose the pivot x to be the last element in S. Remove all the elements from S and put them into three sequences:
• L, storing the elements in S less than x
• E, storing the elements in S equal to x
• G, storing the elements in S greater than x

Of course, if the elements of S are distinct, then E holds just one element—the pivot itself.

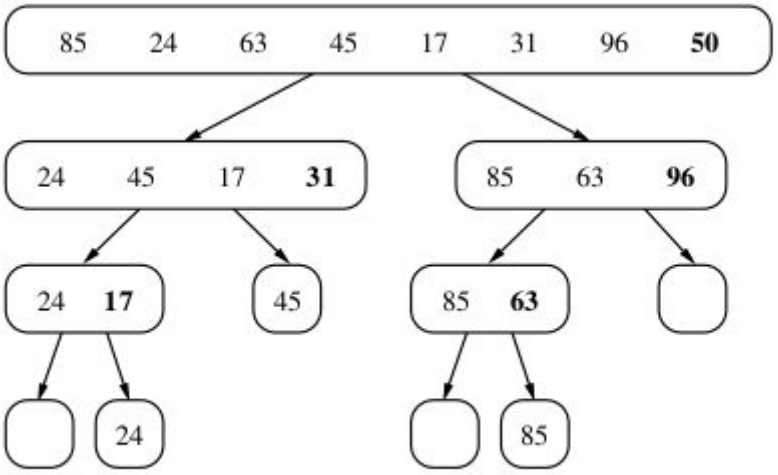
2. Conquer: Recursively sort sequences L and G.
3. Combine: Put back the elements into S in order by first inserting the elements of L, then those of E, and finally those of G.



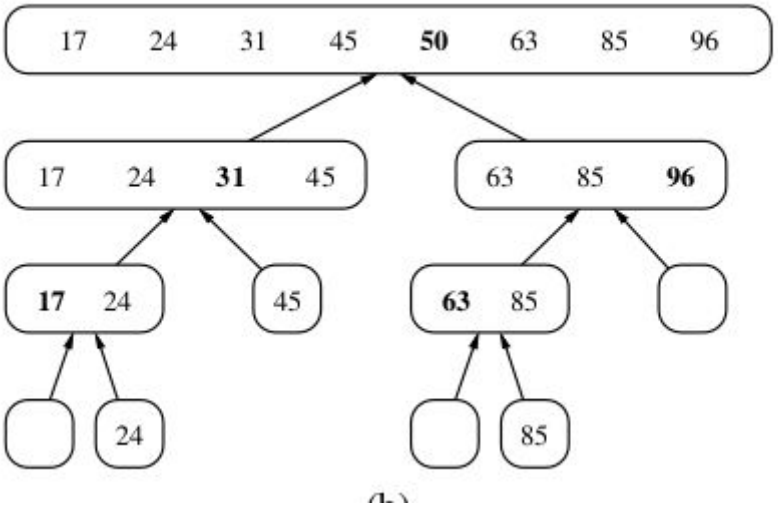
ike merge-sort, the execution of quick-sort can be visualized by means of a binary recursion tree, called the quick-sort tree. Figure 12.9 summarizes an execution of the quick-sort algorithm by showing the input and output sequences processed at each node of the quick-sort tree. The step-by-step evolution of the quick-sort tree is shown in Figures 12.10, 12.11, and 12.12.

Unlike merge-sort, however, the height of the quick-sort tree associated with an execution of quick-sort is linear in the worst case. This happens, for example, if the sequence consists of n distinct elements and is already sorted. Indeed, in this case, the standard choice of the last element as pivot yields a subsequence L of size n − 1, while subsequence E has size 1 and subsequence G has size 0. At each invocation

of quick-sort on subsequence L, the size decreases by 1. Hence, the height of the quick-sort tree is n − 1.



(a)



```
def quick_sort(S):
  """Sort the elements of queue S using the quick-sort algorithm."""
  n = len(S)
  if n < 2:
    return                          # list is already sorted
  # divide
  p = S.first( )                    # using first as arbitrary pivot
  L = LinkedQueue( )
  E = LinkedQueue( )
  G = LinkedQueue( )
  while not S.is_empty( ):          # divide S into L, E, and G
    if S.first( ) < p:
      L.enqueue(S.dequeue( ))
    elif p < S.first( ):
      G.enqueue(S.dequeue( ))
    else:                           # S.first() must equal pivot
      E.enqueue(S.dequeue( ))
  # conquer (with recursion)
  quick_sort(L)                     # sort elements less than p
  quick_sort(G)                     # sort elements greater than p
  # concatenate results
  while not L.is_empty( ):
    S.enqueue(L.dequeue( ))
  while not E.is_empty( ):
    S.enqueue(E.dequeue( ))
  while not G.is_empty( ):
    S.enqueue(G.dequeue( ))
```

The radix-sort algorithm sorts a sequence S of entries with keys that are pairs, by applying a stable bucket-sort on the sequence twice; first using one component of the pair as the key when ordering and then using the second component. But which order is correct? Should we first sort on the k's (the first component) and then on the l's (the second component), or should it be the other way around?

Example 12.5: *Consider the following sequence S (we show only the keys):*

$$S = ((3,3),(1,5),(2,5),(1,2),(2,3),(1,7),(3,2),(2,2)).$$

If we sort S stably on the first component, then we get the sequence

$$S_1 = ((1,5),(1,2),(1,7),(2,5),(2,3),(2,2),(3,3),(3,2)).$$

If we then stably sort this sequence S_1 using the second component, we get the sequence

$$S_{1,2} = ((1,2),(2,2),(3,2),(2,3),(3,3),(1,5),(2,5),(1,7)),$$

which is unfortunately not a sorted sequence. On the other hand, if we first stably sort S using the second component, then we get the sequence

$$S_2 = ((1,2),(3,2),(2,2),(3,3),(2,3),(1,5),(2,5),(1,7)).$$

If we then stably sort sequence S_2 using the first component, we get the sequence

$$S_{2,1} = ((1,2),(1,5),(1,7),(2,2),(2,3),(2,5),(3,2),(3,3)),$$

which is indeed sequence S lexicographically ordered.

So, from this example, we are led to believe that we should first sort using the second component and then again using the first component. This intuition is exactly right. By first stably sorting by the second component and then again by the first component, we guarantee that if two entries are equal in the second sort (by the first component), then their relative order in the starting sequence (which is sorted by the second component) is preserved. Thus, the resulting sequence is guaranteed to be sorted lexicographically every time. We leave to a simple exercise (R-12.18) the determination of how this approach can be extended to triples and other d-tuples of numbers. We can summarize this section as follows:

Proposition 12.6: Let S be a sequence of n key-value pairs, each of which has a key (k 1 , k 2 , . . . , k d) , where k i is an integer in the range [0, N − 1] for some integer N ≥ 2. We can sort S lexicographically in time O(d(n + N)) using radix-sort.

Radix sort can be applied to any key that can be viewed as a composite of smaller pieces that are to be sorted lexicographically. For example, we can apply it to sort character strings of moderate length, as each individual character can be

represented as an integer value. (Some care is needed to properly handle strings with varying lengths.)

C-7.38 There is a simple, but inefficient, algorithm, called ***bubble-sort***, for sorting a list L of n comparable elements. This algorithm scans the list $n-1$ times, where, in each scan, the algorithm compares the current element with the next one and swaps them if they are out of order. Implement a bubble_sort function that takes a positional list L as a parameter. What is the running time of this algorithm, assuming the positional list is implemented with a doubly linked list?

The bubble sort algorithm moves sequentially through a list of data structures dividing it into a sorted part, and an unsorted part. A bubble sort starts at end of the list, compares adjacent elements, and swaps them if the right-hand element (the element later in the list) is less than the left-hand element (the element with earlier in the list). With successive passes through the list, each member *percolates* or *bubbles* to its ordered position at the beginning of the list. The pseudocode looks like this:

```
numElements = number of structures to be sorted
for ( inx = 0 ; inx < numElements - 1 ; ++inx )
    for ( jnx = numElements - 1 ; jnx != inx ; --jnx )
        if ( element( jnx ) < element( jnx - 1 ) )
            swap( element( jnx ), element( jnx - 1 ) )
```

Shellsort: Java implementation

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

      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ...

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

         h = h/3;
      }
   }

   private static boolean less(Comparable v, Comparable w)
   { /* as before */ }
   private static boolean exch(Comparable[] a, int i, int j)
   { /* as before */ }
}
```

3x+1 increment sequence

insertion sort

move to next increment

39

La idea básica del algoritmo Radix sort es considerar que las claves están formadas por dígitos. Así, para ordenar las claves, el método las ordena por cada uno de sus dígitos, de menos peso a más peso. A cada ordenación de las claves por un dígito se le llamará iteración. La versión estable de Radix sort necesita un vector auxiliar del mismo tamaño del vector a ordenar.

En cada iteración, Radix sort puede utilizar cualquier método de ordenación para ordenar un dígito de las claves. En este documento, la implementación utilizada ordena cada dígito con el algoritmo de conteo que se puede ver en el Algoritmo 1. El

algoritmo de conteo realiza 4 pasos para la ordenación de cada dígito. A continuación se explica dicho algoritmo con la ayuda del ejemplo de la Figura 2.1, que muestra el procedimiento para claves de 2 dígitos decimales. Por claridad, en la figura no se muestra el puntero asociado a cada clave ya que no aporta nada a la comprensión del algoritmo.

Los pasos para el algoritmo de conteo, suponiendo que lo aplicamos al dígito de menos peso (primera iteración) del ejemplo de la Figura 2.1 son los siguientes:

1. **Paso (1) - Inicialización de Contadores:** Inicializa a cero los contadores que el algoritmo va a utilizar. En el caso del ejemplo, inicializaríamos un vector de 10 contadores, uno para cada posible valor (de 0 a 9) de un dígito decimal. Éste es el vector C de la Figura 2.1. Este paso no se muestra en la Figura 2.1.

2. **Paso (2) - Conteo:** El paso de conteo calcula un histograma de los valores del dígito para las claves almacenadas en el vector fuente S. En el vector C de la Figura 2.1 se muestra, por ejemplo, que el dígito de menos peso de la clave tiene 6 y 2 ocurrencias de los valores 1 y 7 respectivamente. Esto se se realiza en las líneas 4 a 7 del Algoritmo 1.

3. **Paso (3) - Suma Parcial:** En este paso, líneas 9 a 15 del Algoritmo 1, se calcula la suma parcial de los contadores. Esta suma se muestra en una instancia diferente del vector C en la Figura 2.1.

4. **Paso (4) - Movimiento:** En el paso de movimiento se leen cada una de las claves y punteros del vector S para escribirlas en el vector D (líneas 17 a 21 del Algoritmo 1). La posición del vector D en la que se escribe una clave y su puntero se obtiene del contador del vector de contadores C, indexado por el valor del dígito de la clave. Después de copiar la clave y el puntero del vector S al vector D, se incrementa el contador correspondiente en el vector C. Por ejemplo, cuando el algoritmo lee la primera clave del vector S (valor 10 de la posición 0), indexa el vector C con el valor del dígito de menos peso de la clave, que es 0, y utiliza el valor del contador como índice para almacenar la clave en el vector D. Después, se incrementa la posición 0 del vector C. En el ejemplo de la Figura 2.1, también se muestra el estado del vector C después de mover las primeras 9 claves (y punteros, aunque no se muestren) del vector S al vector D.

Durante la segunda iteración de Radix sort se aplica el mismo procedimiento al dígito de más peso de la clave. En este caso, los vectores S y D cambian sus papeles. En la Figura 2.1 también se muestra el vector final ordenado.

References

Goodrich, M. T., Goldwasser, M. H., & Tamassia, R. (2013). *Data structures and algorithms in Python*. Wiley.

Algorithms ROBERT S EDGEWICK |KEVIN WAY N E

N.Wirth. Algorithms and Data Structures. Oberon version