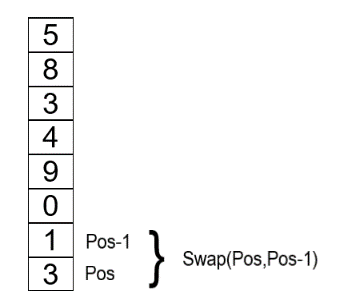
Laboratori 1

Sorting Benchmark

Autors: Èric Bitriá i Adrià Fernández

# bubbleSort

L’algoritme d’ordenació bubbleSort pren aquest nom per la forma similar a les bombolles d’aire al pujar a la superfície. Es basa en iterar entre posicions consecutives de forma ascendent agafant sempre l’element més petit i situant-lo a la posició més elevada possible.



[2] First Iteration



[1] Unsorted List

Com es pot observar la figura [2], l’algoritme compara l’element Pos i Pos-1, en cas que l’element a Pos sigui més petit que Pos-1 es realitzarà un swap, elevant l’element més petit en una posició.

A close-up of a number

Description automatically generated

[3] First element sorted

Com es veu en la figura [3], al iterar sobre tota la llista hem agafat l’element 0 i aquest s’ha situat a la zona més elevada. Per cada iteració per tota la llista que realitzem tenim garantit que almenys una posició estarà ordenada reduint la longitud de les comparacions que realitzem en 1. D’aquesta manera podem implementar la part fonamental del codi:

for (int j = array.length - 1; j > i; j--) {

// If the number in j is smaller than j-1 we swap them

if (array[j] < array[j - 1]) {

swap(j, j - 1);

}

}

Si ens fixem en la declaració del for, sempre comencem des de el final. I iterem fins arribar a “i” que es el nombre de voltes ja realitzades i per tant nombre mínim d’elements ordenats. Seguidament tenim el comparador if dels elements seguit del swap.

En el codi també disposem d’optimitzacions en casos simples com tenir un element en total, o no haver-se realitzat cap swap el que significa que ja esta ordenada la llista.

Analitzant el codi en notació Big(O), obtenim un cost computacional d’ordre n2 al disposar de dos bucles for que recorren n posicions, encara que el segon bucle disminueix les voltes de forma lineal.

# selectionSort

Aquest algoritme d’ordenació per selecció és bastant senzill. Consisteix en una llista d’elements que nosaltres dividim en dues parts. Tenim la part ordenada que començarà buida i la part desordenada que tindrà tota la llista. L’algoritme busca l’element més petit a la part desordenada i l’afegeix a la primera posició de la part ordenada. Per aconseguir això, hem d’anar recorrent la llista d’un en un i emmagatzemar el nombre més petit. Un cop l’acabem de recórrer tindrem seleccionat el nombre més petit i l’afegirem a la posició següent del nombre anterior de la part ordenada. Bàsicament, realitzarem aquest pas tants cops com elements tingui la nostra llista.

Per crear el codi iniciarem la variable ‘smallest’ amb valor 0, així farem un seguiment de la posició de l’element més petit de la part no ordenada. Després ens haurem d’assegurar que la llista tingui més d’un element, perquè si no és així ja tindrem la nostra llista ordenada, ja que sol hi haurà un element.

A number grid with numbers and arrows

Description automatically generated with medium confidence

[4] Representació del selection sort

El bucle recorre la llista sense ordenar fins l’últim element. Després del bucle la variable ‘smallest’ la iniciarem amb la posició actual ‘i’. El bucle interior començarà des de la posició ‘i’ fins a l’últim element, buscant l’element més petit de la part no ordenada. Si trobéssim un element més petit que l’actual (‘smallest’), s’actualitzaria aquest valor amb el nou valor ‘j’. Després de sortir del bucle interior cridem a la funció ‘swap’ i aquesta ens intercanviarà el valor de la posició ‘i’ amb l’element més petit ‘smallest’. L’algoritme acaba quan el bucle exterior recorre tota la llista. Així ja tindrem ordenats tots els elements.

for (int j = i; j < array.length; j++) {

if (array[smallest] > array[j]) {

smallest = j;

}

}

swap(i, smallest);

Analitzant el codi en notació Big(O), obtenim un cost computacional d’ordre n2. I com ens succeïa al mètode anterior el segon bucle va reduint el nombre de voltes que ha de realitzar.

# partitionIterative

El mètode partitionIterative és el codi que permet a quickSort ordenar la llista en base a un element “pivot” que anirà variant a mesura que dividim la llista de 2 en 2. Utilitza tres variables i una sèrie de comparacions: El pivot, element per comparar, inf i sup que son els limitadors superiors i inferiors de la part de la llista que volem ordenar.



[7] Inf++, Sup--



A white rectangular with black numbers

Description automatically generated

[6] Swap





[5] First Iteration



[8] Inf++



[9] Inf == Sup, Return inf



El nostre codi comença amb un bucle ‘while’, on tenim dos índexs ‘inf’ i ‘sup’. El bucle funcionarà sempre que ‘inf’ sigui més petit que ‘sup’. La idea és controlar la llista en dues parts, l’índex ‘inf’ es desplaçarà cap a la dreta i l’índex ‘sup’ es mourà cap a l’esquerra, figura [8]. El pivot tindrà un valor, si aquest és més gran o igual al valor de ‘inf’, aquest índex es desplaçarà. Amb l’índex ‘sup’ serà de la forma contrària, quan aquest sigui més gran que el pivot, el desplaçarem. Si cap d’aquestes condicions es compleixen, llavors utilitzarem el ‘swap’ figura [6], intercanviarem les posicions de ‘inf’ i ‘sup’. Aquests passos s’aniran repetint fins que els índexs es creuin o s’igualin, llavors el bucle acabarà, de forma que els elements més a la esquerra seran més petits que el pivot i els elements a la dreta més grans que el pivot. Per últim, es retorna la posició de ‘inf’.

while(inf<sup){

if(array[inf] <= pivotValue){

inf++;

} else if (array[sup - 1] > pivotValue){

sup--;

} else {

swap(inf, sup - 1);

inf++;

sup--;

}

}

return inf;

Encara que el quickSort és l’algoritme que menys temps ha tardat en executar-se en els benchmarks, segueix tenint una notació en els pitjor dels casos d’ordre n2 , que en realitat de mitja aconsegueix resultats de O(n\*log(n))