Universitat de Lleida

Escola Politècnica Superior

Grau en Enginyeria Informàtica

Estructures de dades

Pràctica 1 (Sorting Algorithms)

Daniel Hernández Calvo
Pau Esquerda Masip

Data d'entrega: 14 d'octubre de 2025

Index

Bubble Sort

Fonaments de l'algorisme

L'*array* es divideix en dues zones: la zona ordenada i la zona desordenada. L'algorisme *Bubble Sort* es basa en comparar parelles adjacents d'elements de la zona desordenada, començant des del final de l'*array*. Si l'ordre de la parella és incorrecte (és a dir, que l'element de baix és més petit que el de dalt), s'intercanvien. D'aquesta forma, l'element més petit de la part desordenada va anant cap amunt fins a la primera posició de la zona desordenada. Quan hi arriba, ja es considera com un element de la zona ordenada i llavors es repeteix el mateix procés per a la nova zona desordenada.

Això es va repetint fins que només queda un element a la zona desordenada, cosa que vol dir que és l'últim element de la zona ordenada.

La nostra implementació

Variables i funcions clau

- *array*: Array a ordenar.
- *ini*: Índex que marca el final de la zona ordenada, començant des de l'inici de tot l'array.
- *swapped*: Boolean que indica si s'ha realitzat l'intercanvi en la iteració actual.
- *i*: Índex que recorre la zona desordenada des del final fins al principi.
- *swap(a, b)*: Intercanvia array[a] per array[b].

Primer bucle

Primer de tot, *i* va des de 0 fins a *array.length - 1*, fent que es recorri tot l'*array* i s'hi vagi afegint un nou element a la zona ordenada a cada iteració, i al mateix temps evitant que es faci la comparació amb l'últim element, cosa innecessària perquè ja estarà ordenat.

Al mateix temps, el booleà *swapped* comprova si s'ha fet alguna iteració al segon bucle, i si no s'ha fet, trenca aquest primer bucle perquè vol dir que ja estan tots els elements ordenats. Aquesta implementació de *swapped* la vam afegir després d'haver plantejat els dos bucles. Encara que, a nivell de rendiment, per a *arrays* grans no sigui gaire important —ja que els temps són gairebé els mateixos—, vam voler fer la implementació igualment.

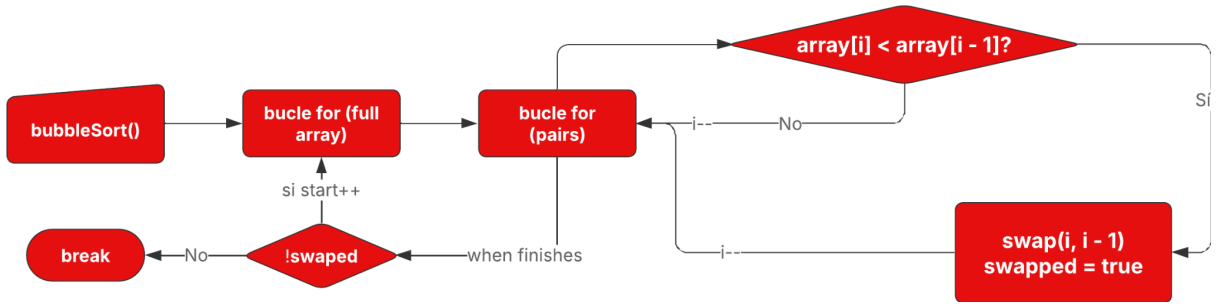
Segon bucle

Comença amb *i* i fa el recorregut de la zona desordenada des del final fins al principi. Mitjançant un *if*, es fa la comparació d'elements, i si estan ordenats de forma incorrecta, mitjançant la funció *swap*, es fa l'intercanvi de posicions. Si s'ha fet l'intercanvi, es posa *swapped* com a *true* per avisar a l'altre bucle que s'ha fet, com a mínim, un intercanvi, i que, per tant, ha de seguir ordenant. Finalment, el bucle s'atura quan *i* arriba a *ini + 1*, evitant treballar en la zona que ja està ordenada.

Complexitat

Aquest algorisme té una complexitat de $O(n^2)$ en la majoria de casos i en el cas pitjor, ja que per a cada *ini* ($n-1$ iteracions) es recorre la zona desordenada una mitjana de n/2 elements. Amb la nostra implementació de *swapped*, encara que la probabilitat sigui molt baixa, si es dona el cas millor, la complexitat seria $O(n)$, perquè amb una sola passada es detectaria que l'*array* ja està ordenat.

Diagrama de flux de la implementació



Selection Sort

Fonaments de l'algorisme

L'*array* es divideix en dues zones: la zona ordenada i la zona desordenada. L'algorisme en si és simple i intuïtiu: es selecciona l'element més petit de la zona desordenada i es col·loca a la primera posició de la zona ordenada. Tot seguit, es torna a buscar el següent element més petit de la zona desordenada (ara reduïda en comparació amb l'anterior iteració) i es repeteix el procés fins que la zona desordenada desapareix.

La nostra implementació

Variables i funcions clau

- *array*: Array a ordenar.
- *ini*: Índex de l'inici de la zona no ordenada.
- *returnMin(ini, array.length)*: Retorna la posició de l'element mínim dins d'un rang d'un array.
- *swap(a, b)*: Intercanvia array[a] per array[b].

Bucle principal

Primer de tot, *ini* va des de 0 fins a *array.length − 1*, i en cada iteració es calcula la posició del valor mínim mitjançant *returnMin()*. Llavors, si la posició del valor mínim no és *ini*, es fa *swap(ini, posMin)* per col·locar el mínim a la posició *ini*. En canvi, si la posició del valor mínim ja és *ini*, no es fa res i es continua amb les següents iteracions.

ReturnMin()

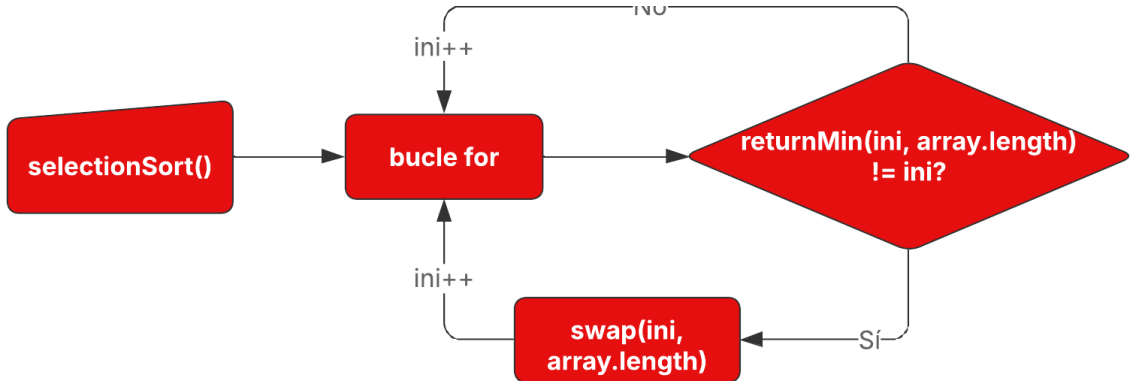
Aquest mètode, primer de tot, suposa que la posició mínima és la inicial; per tant, inicialitza *minPos* a *ini*. Després, recorre l'*array* fins a *end − 1*. Mitjançant aquest recorregut, si troba

un valor més petit que l'especificat en *minPos*, canvia el valor que tenia pel de l'actual. Finalment, acaba retornant *minPos*, que és l'element més petit de la zona desordenada.

Complexitat

Aquest algorisme té una complexitat de $O(n^2)$ en tots els casos, ja que per a cada *ini* (n iteracions) es busca el mínim recorrent una mitjana de n/2 elements.

Diagrama de flux de la implementació



Quick Sort

Fonaments de l'algorisme

Es determina un *pivot* que dividirà l'*array* en dos sub-*arrays*: el dels elements majors i el dels menors al valor del *pivot*. S'aplica el mateix sistema als sub-*arrays* generats de manera recursiva i, finalment, es combinen les solucions per obtenir tot l'*array* ordenat. Aplica el principi de *"divide y vencerás"*.

La nostra implementació

Variables i funcions clau

- *array*: Array a ordenar.
- *left* i *right*: Defineixen el rang actual del subarray a ordenar.
- *pivotPos*: Posició inicial del pivot.
- *pivotValue*: Valor de l'element pivot utilitzat per a les comparacions.
- *pos*: Posició retornada pel mètode *partitionIterative* que indica on dividir l'array.
- *inf* i *sup*: Límits del subarray al qual es farà la partició.
- *partitionIterative(pivotValue, inf, sup)*: Funció que reorganitza els elements al voltant del pivot. Aquesta ha sigut la que em implementat nosaltres.
- *swap(a, b)*: Intercanvia array[a] per array[b].

Mètode principal

El mètode *quickSort* ordena, mitjançant recursivitat, un *array* definit pels índexs *left* i *right*. Si l'*array* té més d'un element, selecciona un *pivot* situat al mig i el mou al principi amb el mètode *swap*. Després, crida a *partitionIterative* per fer la partició de la resta de l'*array* al voltant del valor del *pivot*, mètode que retorna la posició *pos* on es divideix l'*array*.

Seguidament, retorna el *pivot* a la seva posició correcta intercanviant-lo amb l'element a *pos − 1*. Finalment, fa la crida recursiva a *quickSort* per a la part esquerra, que són els elements menors o iguals al *pivot*, i per a la part dreta, que són els elements majors al *pivot*.

S'ha realitzat l'explicació d'aquest mètode perquè, encara que no haguem fet la implementació, ens anirà millor per entendre el codi en un futur si necessitem revisar aquesta pràctica.

PartitionIterative

El mètode *partitionIterative* parteix el sub-*array* definit entre *inf* i *sup* al voltant del valor *pivot pivotValue*. Inicialitza dos punters: *left* a *inf* i *right* a *sup*. Mentrestant, mentre fa el bucle, si l'element a *left* és menor o igual al *pivot*, s'incrementa *left* perquè ja està ben posicionat. Si l'element a *right − 1* és major que el *pivot*, es fa el decrement de *right* perquè ja està ben col·locat. En canvi, si no es compleixen cap d'aquestes condicions, s'intercanvien els elements a *left* i *right − 1* i es mouen els dos punters. Tot aquest funcionament es fa mitjançant *if*, *else if* i *else*.

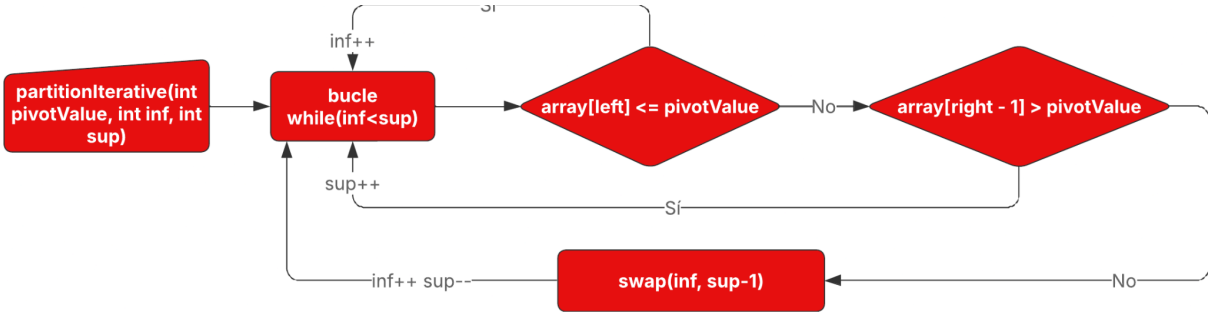
El bucle continua fins que *left* i *right* es troben, que és quan retorna *left*, que és la posició que indica on l'*array* queda dividit.

Complexitat

En la majoria de casos i en el cas millor, la complexitat del *Quick Sort* és de $O(n \cdot log(n))$. La part iterativa realitza $O(n)$ operacions en cada nivell de recursió, mentre que la pròpia recursió té una complexitat de $O(log(n))$ operacions. Per això mateix, per calcular la complexitat de tot l'algorisme es multipliquen aquests dos valors.

Per calcular la complexitat de la part recursiva, ens ha servit de gran ajuda la part de *Recursion and Recurrences* del tema 1 de teoria, ja que ens ensenya a expressar el cost com una equació de recurrència i a resoldre la pròpia recurrència.

Diagrama de flux de la implementació



Insertion Sort

Com aquest algorisme ja venía implementat i amb codi explicat per part del professorat, només explicarem els fonaments del propi algorisme i la seva complexitat.

Fonaments de l'algorisme

L'algorisme d'ordenació per inserció fa que, a cada iteració, s'agafi el primer element de la zona desordenada i s'insereixi a la posició correcta de la zona ordenada, mantenint l'ordenació d'aquesta. Inicialment, la zona ordenada comença amb un element, el primer de

l'*array*, ja que un *array* d'un sol element està sempre ordenat, i, a mesura que avança el bucle, la zona ordenada va creixent fins a cobrir tot l'*array*.

Complexitat

Aquest algorisme té una complexitat de $O(n^2)$ en la majoria de casos i en el cas pitjor, ja que per a cada $n-1$ iteracions pot necessitar recórrer i desplaçar fins a *end* elements cap enrere per trobar la posició correcta. Encara així, en el millor cas, la complexitat seria de $O(n)$ gràcies a la implementació del *break*, que atura el bucle intern quan detecta que l'element ja està en la posició correcta.

Conclusions

Gràcies a la utilització de la classe *Benchmark*, que ja se'ns va proporcionar, i mitjançant els diferents càlculs de les complexitats dels quatre algoritmes, hem arribat a les següents conclusions.

L'algorisme més ràpid, i com ens indica el seu nom, és el *Quick Sort*, que gràcies a la seva part recursiva aconsegueix reduir molt el temps d'execució mig per als diferents *arrays* amb què es prova a la classe *Benchmark*. Cal destacar que aquest algorisme es pot utilitzar (i de fet, s'utilitza) per a gairebé qualsevol exercici d'ordenació de dades en un *array*, gràcies al que ja s'ha comentat.

El segon algorisme més òptim és el *Insertion Sort*, gràcies al *break* implementat, que impedeix que es facin iteracions innecessàries a l'hora de fer l'ordenació d'un *array*. Aquest algorisme funciona molt correctament per a *arrays* de no gaire gran tamany, arribant a ser fins i tot més ràpid que el *Quick Sort*, però, a mesura que anem augmentant el tamany d'aquests, es va quedant enrere com els altres

En tercer lloc, i no amb gaire diferència, tenim el *Selection Sort*, que és una mica més òptim que l'últim, però no amb molta diferència. Per a ús acadèmic i per aprendre d'aquests algoritmes, pot arribar a ser molt útil, però, per a la seva pròpia funció, que és la d'ordenar un *array*, si la velocitat ens importa, tenim millors opcions.

Per últim, tenim el *Bubble Sort*, que és el que ha donat els pitjors resultats a tots els *arrays* de qualsevol mida. Ha passat els tests correctament, i, per tant, vol dir que la seva funció d'ordenar la compleix, però, a la vida real, i com ja hem comentat en el cas anterior, és molt important la velocitat d'aquests algorismes, cosa que, en aquest, és la pitjor característica.

A part de les diferents conclusions sobre els algoritmes, també podem dir que aquesta pràctica ha servit sobretot per acabar d'entendre i de poder implementar cadascun d'aquests, ja que, encara que ja els havíem vist en forma d'esquema a les classes de teoria i pràctica, fer la seva implementació en Java ha estat de gran ajuda.

Ús de la IA

Com s'especifica a l'enunciat del treball, en aquest apartat s'especifica l'ús de la IA en la realització de la nostra pràctica. En aquesta, només hem fet servir aquesta eina generativa per poder corregir les faltes d'ortografia i de format en aquest document, fent així un ús correcte d'aquesta i seguint les directrius generals de la IA en l'assignatura.