**SORTING TYPES**

**SELECTION SORT**

Select the minimum, swap with first element. Repeat for array starting from index 1, index 2, etc...

We have to do this n-1 times. When we put second largest in its place, no need to look at last element, it is already the largest in the array.

All items to be sorted must be Comparable objects. You can use comparator as well.

Text

Description automatically generated

Outer loop iterates n-fill times.

There are n-1 exchanges

Running time is

Line 4 (comparison) is performed (n-1-fill) times for each value of fill and can be represented by the following series:

(n-1) + (n-2) + … + 3 + 2 + 1 =

number of comparisons = =

number of exchanges = n-1 =

Comparison usually takes more time than exchange bc we don’t know type, it may have int or String. compareTo method is different for them.

In exchange, we just do reference assignment.

In C, exchange requires copying of objects (some structure). So in that case, exchange is slower.

Text, letter

Description automatically generated

After first iteration, we are sure first value is sorted. When we get fill, elements until fill are sorted.

Invariant says that posminth element is the smallest element in array from fill to next-1.

Invariant says statement is correct each time we come to that point.

assert says that we are sure that posminth element is the smallest element in array from fill to n-1. So invariant is satisfied.

Invariants and assertions help us to control what kind of things have to be checked when we come to that point. You can write a small code checking whether invariant/assertion is correct or not.

An invariant is a condition that can be relied upon to be true during execution of a program. For example, a loop invariant is a condition that is true at the beginning and end of every execution of a loop.

An assertion is a predicate (a true–false statement) placed in a program to indicate that the developer thinks that the predicate is always true at that place. Programmers often use assertions in their code to make invariants explicit.

When we come to invariant, statement has to be true.  
When we come to assert, we assert the statement.

You can prove your algorithm is correct by using assertions and invariants.

To be able to prove invariant is correct, you may need induction.

**BUBBLE SORT**

Always compares 2 elements next to each other. At each iteration, we look through 1 less size (0 to n, 0 to n-1, 0 to n-2, …)

n-1 comparisons are performed (at most). After comparisons, we are sure that largest element is in the last position.

Bubble sort is also quadratic.

Text

Description automatically generated

At the end of first pass, largest element will be at the end.  
At the end of 2nd pass, 2nd largest element will be second from last  
…

Sometimes an array will be sorted before n-1 passes. This can be detected if there are no exchanges made during a pass through the array.

If there is no exchange in one pass, array is sorted. We can modify algorithm to detect exchanges:

Text

Description automatically generated

Tb(n) = --> array is already sorted, no exchanges are required, inner loop is performed just once, it performs n-1 comparisons ( comparisons, exchanges)

Tw(n) = --> if array is inversely sorted or smallest element is at the end, we need n-1 iteration for the outer loop and we require computation time

The number of comparisons and exchanges is represented by:

(n-1) + (n-2) + … + 3 + 2 + 1

Number of comparisons:

* Best case : n-1
* Worst case : Outer loop iterates n-1 times, inner loop n-1, n-2, …, 1 🡪

Number of exchanges:

* Best case : 0 --> array is already sorted
* Worst case :

Compared to selection sort with its comparisons and exchanges, bubble sort is usually performs worse

If the array is sorted early, the later comparisons and exchanges are not performed, and performance is improved

For general running time, bubble sort (O()) is better than selection sort.  
For # of comparisons, bubble sort is better as well.  
For # of exchanges, selection sort is better.

Comparison and exchange don’t have the same code. One is more expensive than the other in some cases in some environments (computer, OS, programming languages). So you should choose your sorting algorithm wisely.

Text, letter

Description automatically generated

**INSERTION SORT**

Tek tek elemanları uygun yere ekleme.

Kart destesi elindeyken yeni kart gelince uygun yere eklemek gibi.

In the beginning, element at index 0 (1 element array) is already sorted.

A screenshot of a computer

Description automatically generated with medium confidence

Text

Description automatically generated with low confidence

for loop is always executed n-1 times.

Lines 2-3-5-6 are constant time.

while loop is executed:

* 0 times for best case (nextVal is the greater than all the elements comes before it)
* i times for worst case where i is the position of the inserted element (nextVal is smaller than all the elements comes before it)

Tb(n) = 🡪 if array is already sorted  
Tw(n) = 🡪 if array is sorted reversely

* while loops’ worst case 🡪 = n(n-1)/2

Number of comparisons (second condition in while):

* n-1 for the best case 🡪 array is already sorted
* n(n-1)/2 for the worst case 🡪 array is sorted reversely

Number of swaps (shifts):

* 0 shifts for the best case
* n(n-1)/2 for the worst case

Number of copies (lines 3-5-7):

* 2(n-1) for the best case (while loop isn’t executed at all)
* (n+4)(n-1) / 2 for the worst case (while loop is executed at all)

Shift requires movement of only 1 item while an exchange in bubble or selection sort involves 3 copy (references ) operations (temp = a; a = b; b = temp;)

Text, letter

Description automatically generated

**COMPARISON OF QUADRATIC SORTS**

Table

Description automatically generated

Comparison of growth rates:

Table

Description automatically generated

Best performance for most arrays 🡪 INSERTION SORT

* Takes advantage of any partial sorting in the array and uses less costly shifts

Worst performance for most arrays (constant is big) 🡪 BUBBLE SORT

* Unless array is nearly sorted

None of the quadratic sorts are good for large (n > 1000) arrays

Quadratic sorts don’t require any additional space for array being sorted. These are in place sorting operations.

Best sorting algorithms provide nlogn for average case

**SHELL SORT**

We sort with gaps. For example start with gap 100, then 50, 20, 10, 5, 3, 1 etc.

Tüm arrayı sıralamak yerine subarrayleri insertion sort ile sıralıyor, en son tüm array sıralanıyor.

Gap value 3’se 3’ten başlıyoruz. 3. indexteki elemanı 0 ile kıyaslıyoruz, küçükse değiştiriyoruz. 4’ü 1 ile, 5’i 2 ile, 6’yı 3 ve 0 ile …

Gap value 1’se 🡪 normal insertion sort

* 100 boşlukta küçük boyutlu arrayleri sıralıyoruz 🡪 Insertion sort is good for small size array.
* Boşluk küçüldükçe subsequence büyüyor, ama array neredeyse sıralanmış oluyor 🡪 Insertion sort is good for almost sorted array

7-sorted 🡪 elements that are 7 positions away are sorted between each other.

Graphical user interface, text, application, email

Description automatically generated



Text

Description automatically generated

General analysis of shell sort is an open research problem

Performance depends on how the decreasing sequence of values for gap is chosen

If successive powers of 2 are used for gap, performance is O()

* i.e. , , …,

If successive values for gap are based on Hibbard’s sequence:

* - 1 (i.e. 31, 15, 7, 3, 1)

It can be proven that the performance is O()

Above algorithm selects the initial value of gap as n/2 and then divides by 2.2 and truncates the result

* Emprical studies show that the performance is O() or even O()

Text

Description automatically generatedText, letter

Description automatically generated

For gap = 1, it is regular insertion sort.

But nlogn is a lot better than

**MERGE SORT**

We merge 2 sorted sequences and get another bigger sorted sequence.

Diagram

Description automatically generated

A ve B’nin başını belirten 2 index tutarız (both 0), küçük olanı outputa ekleriz, eklediğimiz indexi 1 arttırırız.

If there are n elements in both sequences (2n elements in output sequence):

* number of comparisons
  + BEST 🡪 all values in one sequence are smaller than other sequence 🡪 **n**
  + WORST 🡪 all values in both sequences are intermittent (bi birindeki küçük, bi diğerindeki küçük) 🡪 **2n-1** (last element is copied without comparison, that’s why we subtract 1)
* number of copies
  + There will be **2n** copy operations exactly.

So for merge operation ---> T(n) = (n)

If sequence A has n, B has m elements:

* number of comparisons
  + BEST 🡪 min(n, m)
  + WORST 🡪 n + m - 1
* number of copies
  + n+m

Space requirement 🡪 the array cannot be merged in place so additional space usage (n) is required.

Actually it can be merged in place but it requires more time.

Usually in merge sort, we don’t continue until 1 element is left. We do insertion sort when we have 10 elements (for instance).

Text, letter

Description automatically generated

1. Split the array into 2 halves
2. Sort the left array
3. Sort the right array
4. Merge the two

Text

Description automatically generated

T(n) = T(n/2) + T(n/2) + (n/2) + (n/2) + (n) ------> lines 7 and 8 make T(n/2), lines 5 and 6 make   
 (n/2), line 9 makes (n)

T(n) = 2T(n/2) + (n) for n > 1  
T(n) = (1) for n <= 1

**T(n) = (nlogn)**

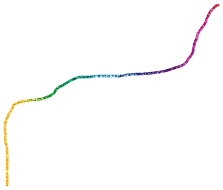
Number of comparisons:

* Cw(n) = 2C(n/2) + n-1 ---> Lines 7-8 creates 2C(n/2) , n-1 comes from line 9’s worst case
* Cw(1) = 0
* Cw(n) = (nlogn)
* -------------------------------
* Cb(n) = 2C(n/2) + n/2
* Cb(1) = 0
* Cb(n) = (nlogn)

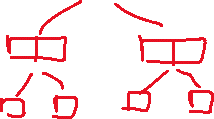
Look page 13 to check where n-1 and n/2 come from. There every subarray is length n, here big array is length n.

Number of moves:

* M(n) = 2M(n/2) + n
* M(1) = 0
* M(n) = (nlogn)

Diagram

Description automatically generated with medium confidence



Height is logn. At each level we perform (n) copying and (n) merging (movement of n elements).

There are logn levels, each level takes (n) time -----> T(n) = (nlogn)

Blue circled arrays are not living at the memory at the same time. A path from root to leaf lives at the same time.

Rainbow path is the path that live at the same time. Expressions are the memory requirements.

n + n/2 + n/4 + n/8 = 2n - 1 ------> maximum extra space required (n + n/2 + … + 2 + 1 = 2n - 1)

Text

Description automatically generated

Advantage of the merge sort compared to other 3(insertion-bubble-selection) sorting algorithms:

For the worst case, merge sort is the best.

Running time of merge sort is always (nlogn). Running time of insertion sort, which is the best of other 3, is between (n) and . You are not sure how long the insertion sort will take. If your problem is time critical, merge sort is better choice obviously.

Disdvantage of the merge sort compared to other 3 sorting algorithms:

For the best case, insertion and bubble sorts are better.

For memory usage, merge sort is the worst.

If array is close to be sorted or array is small, insertion sort is better choice.

In the sort method above, you can use table.length > 20 (for example) as a condition and use insertion sort for the remaining. This makes our sorting algorithm faster (makes the constant smaller in (nlogn)).

**HEAPSORT**

Arraydeki elemanları heap’e atıyoruz (O(nlogn) - insertion O(logn), n tane eleman var), sonra heapten tek tek en küçüğü output array’e koyuyuoruz (O(nlogn) - deletion O(logn), n tane eleman var).

Heap (array) için n memory space lazım. Actually we can build heap on the same input array. First element of the input array will be used as we fill the first element of the heap array…

Heapten outputa ekstra array kullanmadan geçebilir miyiz? Evet. Smallest elemanı heapten alınca heap arrayin son elemanı boşalacak, oraya smallest elemanı koyarız. İkinci smallest elemanı sondan bi önceki indekse koyarız. Böylece array tersten sıralanmış olur. If you want to sort smallest to largest, you can change the priority in the heap (from min heap to max heap).

Min heapten max heape geçiş:

* move the top item to the bottom (arrayin sonuna) of the heap
* reheap, ignore the item moved to the bottom

Heapsort is in place sorting algorithm, only constant extra space is required.

Heapsort is O(nlogn) algorithm.

First Version of a Heapsort Algorithm

Places an array’s data into a heap, then removes each heap item and moves it back into the array

This version of the algorithm requires n extra storage space

A screenshot of a computer

Description automatically generated

If we implement the heap as an array:

* each element removed will be placed at the end of the array, and
* the heap part of the array decreases by one element

Table

Description automatically generated

Array ilk haldeyken (max heap) 89 (top element) 6 (en alt sağdaki element) ile değiştirilir. Sonra array heapify edilir (6 uygun yere getirilir). Böylece son eleman en büyük eleman olur.

In-Place Heapsort Algorithm

A screenshot of a computer

Description automatically generated with medium confidence

Consider the first item to be a heap of one item

Next, consider the general case where the items in array from 0 through n-1 form a heap and the items from n through array.length-1 are not in the heap

Graphical user interface, text, application

Description automatically generated

**Analysis**

Heap is a complete binary tree ----> it has logn levels

Building a heap of size n requires finding the correct location for an item in a heap with logn levels

Each insert or remove is O(logn)

With n items, building a heap is O(nlogn)

No extra storage is needed

Text, letter

Description automatically generated

A black and white document

Description automatically generated with low confidence



Text

Description automatically generated

Merge sort usually a little slower than heapsort bc it requires extra memory space.

Also merge sort uses recursive method so this makes it slower (constant is bigger in merge sort).

**QUICKSORT**

Pivot value seçiyoruz. Pivotun sağı pivottan büyük, solu pivottan küçük olacak şekilde arrayi düzenliyoruz (partitioning). Sol ve sağı sıraladığımızda (pivot aynı yerde kalıyor) merge’e gerek kalmadan array sıralanmış oluyor.

* In merge sort real job (comparisons etc.) is performed during merge operation,
* In quicksort real job is performed during partition operation, during merging (no merge is required actually) we don’t need to do anything since it is already sorted.

We can perform quicksort in place. We don’t need any separate array.

Arbitrarily select the 1st element as the pivot



Partition the elements so that all the elements at the left are smaller than or equal to pivot, all the elements at the right are greater than the pivot.



Pivotu en küçük eleman da seçebilirdin, solunda hiç eleman olmazdı. Şansımıza orta elemanı seçtik.

44 is now in its correct place.

Now apply quicksort recursively to the two subarrays.

Pivot value = 12, we don’t need to do anything, there is no smaller element



Pivot value = 33



🡪 left and right subarrays (23 and 43) have single values so they are sorted

Pivot value = 55, we don’t need to do anything



Pivot value = 64, we don’t need to do anything



Pivot value = 77



Partition, left array (75) is sorted:



Array is sorted:



Algorithm

Indexes first and last are the end points of the array being sorted

Index of the pivot after partitioning is pivIndex

Text, letter

Description automatically generated

If pivot is randomly selected:

* statistically half of the items in the subarray will be less than the pivot and half will be greater

If both subarrays have the same number of elements each time (best case - (nlogn)), there will be logn levels of recursion

Choosing smallest element (and keep choosing smallest element each time 🡪 n | n-1 | n-2 | … | 1 : ) as the pivot is the worst case ()

At each recursion level, the partitioning process involves moving every element to its correct position, n moves

In insertion sort, best case is , worst case is . Insertion sort looks better. In insertion sort, having the worst case has more chance than having the best case. So its average case is .

In quicksort having the best case has more chance than worst case. So its average case is .

Average case 🡪 expected running time

The overhead of recursive calls and the extra run-time stack storage required by these calls makes this version of quicksort a poor performer relative to the quadratic sorts. But average case is good so that’s what makes quicksort quick.

Text, letter

Description automatically generated

If the array is randomly ordered, it doesn’t matter which element is the pivot, for simplicity element with subscript “first” can be picked.



Look for the 1st value starting from left which is greater than the pivot value 🡪 75 (index: up)  
Look for the 1st value starting from right which is less than or equal to the pivot value🡪 33 (index: down)  
Exchange 75 and 33.





Do the same for circled subarray.

🡪 55  
🡪 12  
Swap them:



When you do same:   
 up: 5  
 down: 4

Since 5 is bigger than 4, we stop and exchange down with pivIndex.

Timeline

Description automatically generated with medium confidence

This partitioning process takes linear time: (n)

Text

Description automatically generated

If all the elements are equal in the array, running time will be quadratic because of the less than or “equal to” (up comes to the end, down is also at the end, exchange pivot with end 🡪 (n-1) <--> 0 partitioning, everything at right (no right elements), everything at left is less than or equal to pivot). It is same if we remove equal to (down comes to beginning, up comes to end, it is again 0 <--> (n-1) partitioning).

Best way is to add both sides equal to 🡪 greater than or equal to & less than or equal to  
Downside for this is making several exchange operations

Number of comparisons:

* n+1 (there are n-1 elements (except the pivot), 2 of them in the middle compared twice)
* best case is very similar (n), no need to consider 2 cases

Number of exchanges:

* BEST: 1
* WORST:

T(n) = (n)

Partition is always

Code for partition when pivot is the largest or smallest value:

Graphical user interface, text

Description automatically generated

Text, letter

Description automatically generated

Quicksort is when each split yields one empty subarray, which is the case when the array is presorted (pivot is always smallest) 🡪 T(n) = T(n-1) + n>1

Best case (pivot is at the middle) 🡪 T(n) = 2T(n/2) + 🡪

A better solution is to pick the pivot value in a way that is less likely to lead to a bad split:

* Use 3 references: first, middle, last
* Select the median of these items as the pivot

A picture containing diagram

Description automatically generatedDiagram

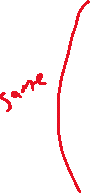
Description automatically generated with medium confidence

  
A picture containing timeline

Description automatically generated Do the partition:

Text, letter

Description automatically generated



We have just changed the pivot selection (selection is still constant). So partition is still linear. Pivot has to be at the beginning to use same partition algorithm lines 3 to 11.

Randomized quick sort: pick random pivot and put it at the beginning OR shuffle the array at the beginning (to make sure the positions are random).

**TESTING THE SORT ALGORITHMS**

Test cases:

* small and large arrays
* arrays in random order
* arrays that are already sorted
* arrays with duplicate values

**COMPARISON OF SORTING ALGORITHMS**

Table

Description automatically generated

Constant on average quicksort is smaller than constant on average heapsort and merge sort. So this makes quicksort quick.

To improve the performance of quicksort, when array gets small other sorting algorithm can be used like insertion sort. Same for merge sort.