**Searching & Sorting**

**Bubble** **Sort** Pseudo-code

Mylist 🡨 [23, 42, 4, 16, 8, 15]

End 🡨 length of Mylist – 1

While True

Swapped 🡨 -1

For i = 0 to i = End

If Mylist[i] > Mylist[i + 1]

Temp 🡨 mylist[i]

Mylist[i] 🡨 Mylist[i + 1]

Mylist[i + 1] 🡨 Temp

Swapped 🡨 i

If Swapped = -1

Break

**Bubble** **Sort** [Python] Code

myList = [23, 42, 4, 16, 8, 15]

end=len(myList)-1

while True:

swapped=-1

for i in range(0,end):

if myList[i]>myList[i+1]:

temp=myList[i]

myList[i]=myList[i+1]

myList[i+1]=temp

swapped=i

if swapped == -1:

break

print (myList)

**Insertion** **Sort** Pseudo-code

array = [23, 42, 4, 16, 8, 15]

for i = 1 to n - 1

element = array[i]

j = i

while (j > 0 and array[j - 1] > element):

array[j] = array[j - 1]

j = j - 1

array[j] = element

print(array)

*https://www.youtube.com/watch?v=DFG-XuyPYUQ&ab\_channel=CS50*

**Insertion** **Sort** [Python] Code

array = [23, 42, 4, 16, 8, 15]

for i in range(1, len(array)):

element = array[i]

j = i

while (j > 0 and array[j - 1] > element):

array[j] = array[j - 1]

j = j - 1

array[j] = element

print(array)

**Sort Comparison**

Having implemented the bubble & insertion sorts using Python, below are how ‘long’ each algorithm took to execute on several different sized data sets.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sort Method | Number of Elements | | | |
| 1,000 | 10,000 | 100,000 | 1,000,000 |
| Bubble Sort | 0.391695976 |  |  |  |
| Insertion Sort | 0.175899982 |  |  |  |

My Conclusions/Notes

The insertion sort is far faster than the bubble sort due to the smaller amount of comparisons and the lack of a final overall check on the numbers. When the list is short there appears to be a small difference however as more elements are added the difference in time taken to sort the lists grows further apart.

**Linear** **Search** Pseudo-code

Mylist 🡨 [23, 42, 4, 16, 8, 15]

Wanted 🡨 4

For i , n in Mylist

If n = wanted

Print found in position i

Break

Else

Pass

**Linear** **Search** [Python] Code

mylist = [23, 42, 4, 16, 8, 15]

wanted = 15

for i, n in enumerate(mylist):

if n == wanted:

print("Found in position:", i)

break

else:

pass

**Binary** **Search** Pseudo-code

Function binarySearch(a, value, left, right)

While left <= right

Mid := floor((right-left)/2) + left

If a[mid] = value

Return mid

If value < a[mid]

Right := mid - 1

Else

Left := mid + 1

Return not found

*http://www.codecodex.com/wiki/Binary\_search*

**Binary** **Search** [Python] Code

def nlower(n):

length = int(len(myList[: n]))

if length % 2 == 0:

y = int(len(myList[: n]) /2)

n = myList[: n][y]

elif length % 2 != 0:

y = int((len(myList[: n]) - 1) /2)

n = myList[: n][y]

return n

n = nfind(x)

found = 'n'

while found == 'n':

if myList[n] == wanted:

print("The number is in position:", n)

found = 'y'

elif myList[n] > wanted:

n = nlower(n)

elif myList[n] < wanted:

n = nhigher(n)

wanted = int(input("enter number to find: "))

myList = [1,2,3,4,5,6,7,8,9,10,11]

x = len(myList)

global myList

global wanted

def nfind(length):

length = int(length)

if length % 2 == 0:

n = int(length / 2)

elif length % 2 != 0:

n = int((length - 1) /2)

return n

def nhigher(n):

x = len(myList)

length = int(len(myList[n:x]))

if wanted < 10:

if length % 2 == 0:

y = int(len(myList[n:x]) /2)

n = myList[n:x][y - 2]

elif length % 2 != 0:

y = int((len(myList[n:x]) - 1) /2)

n = myList[n : x][y - 2]

else:

if length % 2 == 0:

y = int(len(myList[n:x]) /2)

n = myList[n:x][y - 1]

elif length % 2 != 0:

y = int((len(myList[n:x]) - 1) /2)

n = myList[n : x][y - 1]

return n

def nlower(n):

length = int(len(myList[: n]))

if length % 2 == 0:

y = int(len(myList[: n]) /2)

n = myList[: n][y]

elif length % 2 != 0:

y = int((len(myList[: n]) - 1) /2)

n = myList[: n][y]

return n

n = nfind(x)

found = 'n'

while found == 'n':

if myList[n] == wanted:

print("The number is in position:", n)

found = 'y'

elif myList[n] > wanted:

n = nlower(n)

elif myList[n] < wanted:

n = nhigher(n)

**Search Comparison**

Having implemented the linear & binary searches using Python, below are how ‘long’ each algorithm took to execute on several different sized data sets.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sort Method | Number of Elements | | | |
| 1,000 | 10,000 | 100,000 | 1,000,000 |
| Linear Search | 0.00047802925 |  |  |  |
| Binary Search |  |  |  |  |

My Conclusions/Notes

The binary search is faster than the linear search for large lists. In some circumstances the linear search appears to be faster due to the wanted number being close to the front of the list. The binary search is also dependent on the list being sorted first which can add on time to the time taken for the program to work.

Question: Is it fair to compare the linear search and the binary search – should you include the time taken to sort a dataset before you can use a binary search?

I don’t think it is fair to compare to compare linear and binary search however if you are going to compare them then I think you should include the time taken to