# A "naive" sorting algorithm
# (also called the Selection Sort)

```
# Suppose that we have a list
nlist = [10, 8, 9, 6, 7, 5, 4, 2, 3, 1]

and we want this list sorted in ascending order. We can simply call the
"sort" method
that each list object has. It will do the job.

Remember nlist.sort() will (a) sort the list, and (b) nlist will change
(i.e., now be sorted).
```

# How does Python do the sort? No doubt, it uses an efficient "sorting"
algorithm.

# Let's try to write a "naive" sorting algorithm. This "naive" sorting
algorithm is really
# the "selection sort" [you are selecting the next minimum from the rest of
the sublist to
# put it in it's proper place at each step].

# The code below is a bit larger than that on page 444 of your text, but
because I am
# calling the "minimum()" function each time, it may be simpler to follow.
It also makes
# explaining the analysis a bit easier.

```
There are len(nlist) numbers in the list.

1. Get the minimum of nlist[0:len(nlist)]. Swap it with the number in
position 0
2. " " " " " nlist[1:len(nlist)]. " " " " 1
......
......
```
n-2. Get the min of nlist[n-2:len(nlist)]. Swap it with the number in position (n-2).
n-1. We are done. The number in position (n-1) is the largest.

Let's write out our own "min" function to do the job, even though we know Python has
its own "min method".

Why?

Because when we write it ourselves, we can clearly see how many comparisons are
made by the algorithm when the list size is n. This will help our analysis.''

def sublistMin(ns sublist, low, high):
    # it returns the position of the minimum element
    smallest = ns sublist[low]  # at first assume the first number is
    index = low  # it's index (i.e., position) is
    for j in range(low+1, high):
        if (ns sublist[j] < smallest):
            smallest = ns sublist[j]  # found smaller number; so update
            index = j

    print("min: ", smallest," ",ns sublist[low:high])
    return index  # return position of smallest number in ns sublist

def swap(somelist,i,j):
    # swap elements i and j of somelist
    if (somelist[i] != somelist[j]):
        temp = somelist[i]
        somelist[i] = somelist[j]
        somelist[j] = temp

def naiveSort(somelist):
    for j in range(len(somelist)):
        minIndex = sublistMin(somelist, j , len(somelist))  # get smallest
        swap(somelist,minIndex,j)  # swap smallest with
# One more obvious efficiency is to stop the for-loop in naiveSort when second to the last element has been swapped. It means the largest number is at the end of list. So no need to touch it.

```python
    print(" swap:",somelist[minIndex],somelist[j]," ,somelist[j:len(somelist)]

    return(somelist)
```

def main():

    nlist =  [10, 8, 9, 6, 7, 5, 4, 2, 3, 1]  # remember lists can be mutated (changed)

    save_nlist = nlist.copy()

    print(" Sorting using Python's sort method for lists \n")

    nlist.sort()

    print("nlist:",nlist," is now sorted; it has changed")
    print("_____________________________________________________")

    print("\n\n Sorting using our own naiveSort() which calls a min() function repeatedly \n")

    print(" ")
    print(save_nlist)
    print(" ")

    result = naiveSort(save_nlist)

    print("\n",result)
    print(" ")
    print("_____________________________________________________")
    print("\n *** Analysis ***\n")

    print("sublistMin(list,low,high) requires (high - low) comparisons to complete")
    print(" ")
    print("Step 1: we call sublistMin(list,1,n) ---> so (n-1) comparisons, but 1st element is place right")
    print(" ")
    print("Step 2: we call sublistMin(list,2,n) ---> so (n-2) comparisons, but 2nd element is place right")
    print(" ")
print(" ..... ")
print(" ")
print("Step(n-1): we call sublistMin(list,n-1,n) ---> so (1) comparison, but now (n-1)-th and n-th")
print(" elements are placed right. Now the entire list is sorted")
print(" ")
print(" algorithm's work, f(n) = 1 + 2 + 3 + 4 + ....... (n-1) = (n-1)n/2")
print(" ")
print(" and so f(n) <= c * n^2, for some constant c")
print(" ")
print(" making this an O(n^2) algorithm")

print("___________________________________________________________")

main()

#__________________ Quicksort algorithm _________________________________
#2.py

# This is a famous algorithm. The analysis on this one is a little bit more involved and so
# we'll only just focus on understanding how it works. An example is best...

nlist: 28 12 4 19 22 7 65 87 3 99 21 32 1 83 17 79 81 33

Let's say the list goes from index 0 to len(nlist)-1

Step1: Choose a "pivot" element. You can choose it randomly. But we'll just use nlist[0],
i.e., whenever we have a sublist, our pivot will be the *first* element

Step 2: Grab hold of the pivot element and move it through the list (in one pass) so that
it ends up in it's "right place". That means, when you are done with this pass,
all the elements to the left of the pivot are smaller,
though they need not be in order;
all the elements to the right of the pivot are larger,
though they need not be in order.

nlist: 12 4 19 22 7 3 21 1 17 28 65 87 99 32 83 79 81 33
Observe: 28 is where it should be if the list is sorted. No need to touch it any more.

The numbers to the left are in the same old unsorted order, but all are smaller than 28.

The numbers to the right are in the same old unsorted order, but all are larger than 28.

Step 1 and Step 2 make up the "workhorse" of this algorithm. They are done by a function called Split(), which selects a pivot and makes a left and right sublist after putting the pivot in its sorted place.

Step 3: We now have a left sublist and a right sublist (just like with binary search; but in the case of binary search we recursively went *either* left or right .... but now we will go recursively *both ways* since both sublists are unsorted.)

Call the Quicksort on left sublist

12  4   19  22  7  3  21   1   17

and call the Quicksort on the right sublist

65   87   99    32    83   79    81   33

which means you repeat Step 1 and Step 2 on each sublist. Each will now generate 2 recursive calls to Quicksort, making up 4 calls at the next level

[remember the "binary tree" discussion in class?

level 0 -------> 1 node (original list)

level 1 -------> 2 child nodes (original breaks up into two sublists)

level 2-------> each again has 2 child nodes, so 4 nodes at this level

.... etc etc.]

So Quicksort's recursive calls fan out in this tree
Analysis: It requires \( f(n) = c \cdot n \cdot \log(n) \) comparisons and so is an \( O(n \cdot \log(n)) \) algorithm.

Because \( n \cdot \log(n) < n^2 \) it is quite a bit faster than the previously described Selection sort.''

def swap(list, i, j):
    print("swapping ", list[i], list[j])
    if (list[i] != list[j]):
        temp = list[i]
        list[i] = list[j]
        list[j] = temp

def Split(nlist, low, high):
    pivot = nlist[low]
    pivotindex = low
    j = low + 1
    while (j <= high):
        if (pivot > nlist[j]):
            pivotindex = pivotindex + 1
            swap(nlist, j, pivotindex)  # pivot is bigger; move it to the right by swapping
        j = j + 1
    swap(nlist, low, pivotindex)
    print("Split says the pivot element ", pivot, " must go to position ", pivotindex)
    print("Split returns: ", nlist)
    return(pivotindex)

def QuickSort(nlist, low, high):
    if (low < high):
splitposition = Split(nlist, low, high)

print("______________________________ ")

print(nlist[low:splitposition])

print("\n Left sublist recursive call")
QuickSort(nlist, low, splitposition - 1) # it will keep recurring this call until it bottoms

print(nlist[splitposition+1:high+1]) # and only then will it print this line, and

print("\n Right sublist recursive call")
QuickSort(nlist, splitposition + 1, high) # then keep recursing through this call.

def main():

    nlist = [28, 12, 4, 19, 22, 7, 65, 87, 3, 99, 21, 32, 1, 83, 17, 79, 81, 33]
    low = 0
    high = len(nlist) - 1

    print(low, high)

    print("original list nlist", nlist)

    QuickSort(nlist, low, high)

    print("\n\n")
    print("Quicksort returns: ", nlist)

main()

#___________ Merge Sort_______________________________________________
#3.py

# This is similar to the Quicksort, and is also an O(nlogn) algorithm. It is useful when the
# list is very, very large (e.g., list of names) and is stored on disk, because it works by
# creating sublists and doing a lot of "ordered copying" (or "merging").
# Let's do the example on page 449 of your text, to save you sometime.
... node 0

nlist: 3 1 4 1 5 9 2 6
(eight numbers in the list)

step 1: split into "equal-sized" (roughly) sublists

3 1 4 1
5 9 2 6
(2 sublists; each sublist is of size 4)

step 2: split into "equal-sized" (roughly) sublists again. Stop only when can split no more.

3 1        4 1
5 9       2 6
(4 sublists; each sublist is of size 2)

Step 3: split into "equal-sized" (roughly) sublists again.

3 1 4 1 5
9 2 6
(8 sublists; each of size 1; we can't split any more. Now we start to do the "merging' part).

...

# Now please pay attention to the order in which things are done. We are splitting recursively
# and when we write code one statement comes after the other

# Really the call will be "MergeSort", but since we are really splitting lists into roughly
# equal pieces, it will look like this:

... 3, 1, 4, 1, 5, 9, 2, 6
3,1,4,1 Split (the left split is working recursively)

3,1 Split (left split is still recursing)

3 1 Split (left split has bottomed; we have two sublists [3] and [1])

Now Merge is called to output and ordered sublist [1 3]. Merge is the last statement in the MergeSort. So Python goes up one level in the recursion.

Where are we now? We had left off at 4,1

4,1 Split (right split off the left recurses)

4 1 Split (right split has bottomed; we have two sublists [4] and [1])

Merge is called again, and gives [ 1  4] as sorted sublist and the "Merge()" has to run after every pair of sublists is found

Merge is called and gives [1, 1, 3, 4]

Where are we now? We have finished recursing the left side, and now have to recurse the right side. We have only one sublist -- the left one. Recursing the right side will give the right sublist and then we can call Merge().

5, 9, 2, 9 Split (recurse right)

5, 9 Split (left split off the right recurses)

5 9 bottoms

Merge is called and gives sublist [5, 9]

2, 6 Split (right split off the right recurses)

2 6 bottoms
Merge is called and gives sublist [2, 6]

Where are we now? We have two sublists [5, 9] and [2, 6].

Whenever a pair of sublists is ready Merge() is called. It now gives [2, 5, 6, 9]

We are back at the second level of the tree, just under the root node with two sublists:

[1, 1, 3, 4] made earlier
[2, 5, 6, 9] just made

We have two sublists. Merge() is called, and outputs the final list.

[1, 1, 2, 3, 4, 5, 6, 9]

Note: you can trace this output below, via my print statements.''

# The only question really is this: given two sublists (see above), how do we
# "merge" them so that the result list is sorted. Yes, each sublist is sorted to begin with. But
# when we put them together to make a bigger list, how to ensure it is sorted?

# Answer: by using an index for each of the two lists and copying the smaller of the numbers
# pointed to by the indexes into the output list.

# Note that the sublists may not be equal-sized. We need only roughly equal.

#------------------ The merge() function merges two sorted lists; output is a sorted list ------------------

def iMerge(firstlist, secondlist, resultlist):
    # the iMerge function works iteratively

        # firstlist and secondlist are sorted; merge them and create a new list resultlist that
        # is also sorted.

        # Imagine my left index finger points to the start of the **first list**

            findex = 0

            #Imagine my right index finger points to the start of the **second list**

        #
sindex = 0

# Imagine we use one of your fingers to point to the start of the
**result list** which is now empty

rindex = 0

# Now all we have to do is to copy the numbers from firstlist and
secondlist into
# resultlist, but in ascending (sorted) order

# How far to copy? We need the limits

flimit = len(firstlist)
slimit = len(secondlist)

while (findex < flimit) and (sindex < slimit):

    if (firstlist[findex] < secondlist[sindex]):  # number in first
        resultlist[rindex] = firstlist[findex]
        findex = findex + 1
        rindex = rindex + 1
        print(" copied from 1")

    else:                                           #
        otherwise, number in second list is smaller
        resultlist[rindex] = secondlist[sindex]
        sindex = sindex + 1
        rindex = rindex + 1
        print("copied from 2")

    # When the above loop is done, we are here. It means one of the two
    # exhausted and there are (sorted) numbers left over in the other list.

    # So simply copy these numbers (they are sorted anyway) into the
resultlist

    # Only one of the following two loops will run.

    while (findex < flimit) : # if we enter this loop, it means secondlist
        was exhausted
        resultlist[rindex] = firstlist[findex]
        findex = findex + 1
        rindex = rindex + 1

    while (sindex < slimit) : # if we enter this loop, it means firstlist
        was exhausted
resultlist[rindex] = secondlist[sindex]
sindex = sindex + 1
rindex = rindex + 1

# We are done. Both sorted lists were copied into resultlist in such
a (simple) way that
# resultlist is sorted.

#----------------- Now comes the recursive merge sort -------------------
# This is a "divide and conquer strategy" just like Quicksort. Both give you
# tree structure.
# If there are n elements in the list, it takes log n steps to build the
tree. If the work at each level
# is not a constant, but related to n, we get an n log n algorithm, making
mergesort an
# O(n log n) algorithm. Just like Quicksort.

def MergeSort(nlist):
    # sort nlist
    n = len(nlist)
    if (n > 1):
        # as long as there is sorting work (i.e., two or
        # more numbers to be sorted)
        # so the base case of the recursion is when n ==
        # 1; this function simply
        # returns, doing nothing
        m = n // 2
        # get a roughly equal split size for each sublist
        firstlist = nlist[ :m]  # slicing goes from 0 ...... (m-1)
        secondlist = nlist[m: ]  # slicing goes from m to (n-1)

        #Now the old recursion trick.
        MergeSort(firstlist)  # recursively sort the first sublist
        MergeSort(secondlist)  # recursively sort the second sublist

        # Now the firstsublist and second sublist have been sorted [really,
        # all we did was to keep
        # spitting nlist repeatedly as we walked down the tree; in the end
        # we arrive at the leaves
        # of the tree, as in the above example. Each of the final sublists
        # has just one number.

        # The next step is to back up the recursion, going in the reverse
order in which we came
    # down (we have to retrace all the function calls), merging sublists
as we go up to
    # make the big resultlist which will be sorted.

    # So all of the sorting work is really doing by the "iMerge()"
algorithm, just like the
    # "iSplit()" function did all the work in QuickSort()

    iMerge(firstlist, secondlist, nlist)  # sorted numbers get
    # put back in (i.e., nlist
    get to be the resultlist)
    print(firstlist, secondlist)
    print(nlist)
    print(" ")

def wait():
    x = input();

def main():
    nlist = [28, 12, 4, 19, 22, 7, 65, 87, 3, 99, 21, 32, 1, 83, 17, 79, 81, 33]

    print("Original list",nlist)
    print(" ")
    MergeSort(nlist)
    print("Sorted list",nlist)

    wait()

    nlist = [3, 1, 4, 1, 5, 9, 2, 6]

    print("Original list",nlist)
    print(" ")
    MergeSort(nlist)
    print("Sorted list",nlist)

main()

# __________________________ Towers of Hanoi ___________________________

#4.py

#Problem Statement:
...
There are n discs (like plates), numbered 1 through n. They all have
different sizes.
Disc 1 is the smallest. Disc n is the largest.

Each disc has a hole in its center, so you can stack discs on a pole.

There are three poles. When discs are stacked on a pole, the rule is

"No larger disc can sit on top of a smaller disc"

Task: Move the n discs (all sitting now on a "source" pole) to a "destination" pole, using the third pole as a temporary pole.

1. Only one disc may be moved at a time.
2. You can only place discs on the three poles; no place else.
3. A larger disc can never be placed on top of a smaller disc.

Here is an example of how we can solve the problem with just 3 discs

S = smallest disc, M = medium disc, L = largest disc

Move the discs from Pole 1 to Pole 3

S
M
L
x       x       x
Pole 1   Pole 2   Pole 3'''

# In the following notation, (L, M, S) means largest is at the bottom and smallest is at the top ...''

Start: (L,M,S) is on Pole 1. Want to get (L,M,S) on Pole 3. Pole 2 and Pole 3 are empty.

1. Move S from Pole 1 to Pole 3:       (L,M)     (-)         (S)
2. Move M from Pole 1 to Pole 2:      (L)         (M)        (S)
3. Move S from Pole 3 to Pole 2:       (L)         (M,S)     (-)
4. Move L from Pole 1 to Pole 3:       (-)          (M,S)     (L)
5. Move S from Pole 2 to Pole 1:       (S)         (M)        (L)
6. Move M from Pole 2 to Pole 3:      (S)         (-)          (L,M)
7. Move S from Pole 1 to Pole 3:       (-)           (-)         (L,M,S)

# Look at how we solved the problem for n = 3

# We first solved a subproblem by (see steps 1-3) by moving the top n-1
(that is two) discs
# from Pole 1 to Pole 2

# Then we moved the n-th disc to Pole 3 (see step 4). Just one disc.

# Then we solved another subproblem (see steps 5-7) by moving the (n-1)
# discs from Pole 2 to Pole 3.

# This is the essence of RECURSION. Break the n-problem into a (simple)
# partial
# solution and then one or more (n-1)-subproblems. This is "divide and
# conquer".

# Notation: Pole 1 = "source", Pole 2 = "temporary", Pole 3 = "destination"

def wait():
    x = input()

def MoveDiscs(n, source, temporary, destination):
    if (n == 1):
        #base case of the recursion. It's where the recursion bottoms
        print("Move disc from ", source, " to ", destination)
    else:
        MoveDiscs(n-1, source, destination, temporary)
        MoveDiscs(1, source, temporary, destination)
        MoveDiscs(n-1, temporary, source, destination)

def Hanoi(n):
    # The Hanoi Problem moves these n discs from a Source Pole to a Destination Pole
    MoveDiscs(n, "1", "2", "3")  # Pole 1 is source, Pole 2 is "temp", Pole 3 is "destination"

print("\nWe'll try it with 3 discs, to verify \n")
Hanoi(3)
print("\nYou'll notice it took 7 moves, which is 2^3 - 1 \n")
wait()

print("\nNow pay attention to how the number of moves simply explodes with 4 discs \n")
Hanoi(4)
print("\nYou'll notice it took 15 moves, which is 2^4 - 1 \n")
```python
wait()
print("\n Now pay attention to how the number of moves explodes more with 5 discs \n")
Hanoi(5)
print("\n You'll notice it took 31 moves, which is 2^5 - 1 \n")
wait()

print("Thus n discs require 2^n - 1 moves")
print("which makes for an O(2^n) algorithm; this is called exponential complexity")

# Note that with work = number of comparisions or moves
# log n  < n  < n log n < n^2  < 2^n

# so we can count operations (comparisions, moves) and tell very quickly which
# algorithm is a good algorithm, independently of the speed of the computer that you
# run your program on. That is why the "Analysis of Algorithms" is be important.
```

From: 
http://courses.cs.purdue.edu/ - Computer Science Courses

Permanent link: 
http://courses.cs.purdue.edu/cs17700:fall15:week16_examples1

Last update: 2015/12/08 00:05