21. Sorting a List

Topics:

- Selection Sort
- Merge Sort
Our examples will highlight the interplay between functions and lists.
Sorting a List of Numbers

Before:

\[ x \rightarrow \begin{array}{cccccccc}
50 & 40 & 10 & 80 & 20 & 60
\end{array} \]

After:

\[ x \rightarrow \begin{array}{cccccccc}
10 & 20 & 40 & 50 & 60 & 80
\end{array} \]
We Will First Implement the Method of Selection Sort

At the Start:

x -->  

High-Level:

for k in range(len(x)-1)
    Swap x[k] with the smallest value in x[k:]
Selection Sort: How It Works

Before:

\[ x \rightarrow \begin{bmatrix} 50 & 40 & 10 & 80 & 20 & 60 \end{bmatrix} \]

Swap \( x[0] \) with the smallest value in \( x[0:] \)
Selection Sort: How It Works

Before:

\[
\begin{array}{c}
\text{x} \rightarrow \begin{bmatrix} 50 & 40 & 10 & 80 & 20 & 60 \end{bmatrix}
\end{array}
\]

Swap x[0] with the smallest value in x[0:]

After:

\[
\begin{array}{c}
\text{x} \rightarrow \begin{bmatrix} 10 & 40 & 50 & 80 & 20 & 60 \end{bmatrix}
\end{array}
\]
Selection Sort: How It Works

Before:

\[ x \rightarrow \begin{array}{cccccc}
10 & 40 & 50 & 80 & 20 & 60 \\
\end{array} \]

Swap \( x[1] \) with the smallest value in \( x[1:] \)
Selection Sort: How It Works

**Before:**

\[
\begin{array}{c}
x \rightarrow \\
10 \quad 40 \quad 50 \quad 80 \quad 20 \quad 60
\end{array}
\]

Swap \(x[1]\) with the smallest value in \(x[1:]\)

**After:**

\[
\begin{array}{c}
x \rightarrow \\
10 \quad 20 \quad 50 \quad 80 \quad 40 \quad 60
\end{array}
\]
Selection Sort: How It Works

Before:

\[
x \rightarrow \begin{bmatrix} 10 & 20 & 50 & 80 & 40 & 60 \end{bmatrix}
\]

Swap \(x[2]\) with the smallest value in \(x[2:]\)
Selection Sort: How It Works

Before:

\[
x \rightarrow \begin{array}{cccccc}
10 & 20 & 50 & 80 & 40 & 60 \\
\end{array}
\]

Swap x[2] with the smallest value in x[2:]

After:

\[
x \rightarrow \begin{array}{cccccc}
10 & 20 & 40 & 80 & 50 & 60 \\
\end{array}
\]
Selection Sort: How It Works

Before:

\[ x \rightarrow \begin{array}{cccccc}
10 & 20 & 40 & 80 & 50 & 60 \\
\end{array} \]

Swap \( x[3] \) with the smallest value in \( x[3:] \)
Selection Sort: How It Works

Before:

\[
\begin{array}{c}
x \rightarrow \begin{array}{cccccccc}
10 & 20 & 40 & 80 & 50 & 60 \\
\end{array}
\end{array}
\]

Swap \(x[3]\) with the smallest value in \(x[3:]\)

After:

\[
\begin{array}{c}
x \rightarrow \begin{array}{cccccccc}
10 & 20 & 40 & 50 & 80 & 60 \\
\end{array}
\end{array}
\]
Selection Sort: How It Works

Before:

\[ x \rightarrow \begin{array}{cccccc}
10 & 20 & 40 & 50 & 80 & 60
\end{array} \]

Swap \( x[4] \) with the smallest value in \( x[4:] \)
Selection Sort: How It Works

Before:

\[
\begin{array}{c}
\text{x} \\
\rightarrow
\end{array}
\begin{array}{cccccccc}
10 & 20 & 40 & 50 & 80 & 60
\end{array}
\]

Swap \(x[4]\) with the smallest value in \(x[4:]\)

After:

\[
\begin{array}{c}
\text{x} \\
\rightarrow
\end{array}
\begin{array}{cccccccc}
10 & 20 & 40 & 50 & 60 & 80
\end{array}
\]
Selection Sort: Recap

<table>
<thead>
<tr>
<th>50</th>
<th>40</th>
<th>10</th>
<th>80</th>
<th>20</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>40</td>
<td>50</td>
<td>80</td>
<td>20</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>50</td>
<td>80</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>40</td>
<td>80</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>40</td>
<td>50</td>
<td>80</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>80</td>
</tr>
</tbody>
</table>
The Essential Helper Function: Select(x,i)

def Select(x,i):
    """ Swaps the smallest value in x[i:] with x[i]
    """

    PreC: x is a list of integers and i is an in that satisfies 0<=i<\text{len}(x)"

Does not return anything and it has a list argument
How Does it Work?

The calling program has a list. E.g.,

```
a  ---  0  ---  50
   1  ---  40
   2  ---  10
   3  ---  80
   4  ---  20
   5  ---  60
```
How Does it Work?

The calling program executes `Select(a,0)` and control passes to `Select`.

```
 a  -->

0  ---->  50
1  ---->  40
2  ---->  10
3  ---->  80
4  ---->  20
5  ---->  60
```
How Does Select Work?

- Nothing new about the assignment of 0 to i.
- But there is no assignment of the list a to x.
- Instead x now refers to the same list as a.

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>60</td>
<td></td>
</tr>
</tbody>
</table>
```
How Does Select Work?

If inside Select we have
\[ t = x[0]; \ x[0] = x[2]; \ x[2] = t \]
it is as if we said
\[ t = a[0]; \ a[0] = a[2]; \ a[2] = t \]
How Does Select Work?

It changes the list $a$ in the calling program. We say $x$ and $a$ are aliased. They refer to the same list.
Let's Assume This Is Implemented

def Select(x,i):
    """ Swaps the smallest value in x[i:] with x[i]"

PreC: x is a list of integers and i is an in that satisfies 0<=i<len(x)"""
<table>
<thead>
<tr>
<th>After this:</th>
<th>The list ( a ) looks like this</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>50  40  10  80  20  60</td>
</tr>
<tr>
<td>Select((a,0))</td>
<td>10  40  50  80  20  60</td>
</tr>
<tr>
<td>Select((a,1))</td>
<td>10  20  50  80  40  60</td>
</tr>
<tr>
<td>Select((a,2))</td>
<td>10  20  40  80  50  60</td>
</tr>
<tr>
<td>Select((a,3))</td>
<td>10  20  40  50  80  60</td>
</tr>
<tr>
<td>Select((a,4))</td>
<td>10  20  40  50  60  80</td>
</tr>
<tr>
<td>Select((a,5))</td>
<td>10  20  40  50  60  80</td>
</tr>
</tbody>
</table>
In General We Have This

def SelectionSort(a):
    n = len(a)
    for k in range(n):
        Select(a,k)
Next Problem

Merging Two Sorted Lists into a Single Sorted List
Example

\[ x \rightarrow \begin{array}{cccc} 12 & 33 & 35 & 45 \end{array} \]

\[ y \rightarrow \begin{array}{cccc} 15 & 42 & 55 & 65 \end{array} \]

\[ z \rightarrow \begin{array}{cccccccc} 12 & 15 & 33 & 35 & 42 & 45 & 55 & 65 & 75 \end{array} \]

\[ x \text{ and } y \text{ are input} \]
\[ \text{They are sorted} \]
\[ z \text{ is the output} \]
Merging Two Sorted Lists

ix and iy keep track of where we are in x and y
Merging Two Sorted Lists

Do we pick from x? $x[ix] \leq y[iy]$  ???

$x \rightarrow \begin{array}{cccc} 12 & 33 & 35 & 45 \end{array}$  

$ix: 0$

$y \rightarrow \begin{array}{cccc} 15 & 42 & 55 & 65 & 75 \end{array}$  

$iy: 0$

$z \rightarrow \begin{array}{c} [ ] \end{array}$
Merge

Yes. So update ix
Merge

x→ [12, 33, 35, 45]

y→ [15, 42, 55, 65, 75]

z→ [12]

Do we pick from x?  \( x[\text{ix}] \leq y[\text{iy}] \)  ???
No. So update iy
Merge

Do we pick from x? \( x[ix] \leq y[iy] \)
Merge

x-> 12 33 35 45

y-> 15 42 55 65 75

z-> 12 15 33

ix: 1
iy: 1

Yes. So update ix
Merge

Do we pick from x?  \( x[ix] \leq y[iy] \)  ???

x→ 12 33 35 45

y→ 15 42 55 65 75

z→ 12 15 33

ix: 2
iy: 1
Yes. So update ix
Merge

Do we pick from x?  \( x[i_x] \leq y[i_y] \)  ???
Merge

\[ x \rightarrow \begin{array}{c|c|c|c}
12 & 33 & 35 & 45 \\
\end{array} \]

\[ y \rightarrow \begin{array}{c|c|c|c|c}
15 & 42 & 55 & 65 & 75 \\
\end{array} \]

\[ z \rightarrow \begin{array}{c|c|c|c|c}
12 & 15 & 33 & 35 & 42 \\
\end{array} \]

No. So update iy...
Merge

$$x \rightarrow \begin{array}{c}
12 \\
33 \\
35 \\
45 \\
\end{array}$$

$$y \rightarrow \begin{array}{c}
15 \\
42 \\
55 \\
65 \\
75 \\
\end{array}$$

$$z \rightarrow \begin{array}{c}
12 \\
15 \\
33 \\
35 \\
42 \\
\end{array}$$

$$ix: 3$$

$$iy: 2$$

Do we pick from $$x$$? $$x[ix] \leq y[iy]$$ ???
Merge

x→  12  33  35  45

y→  15  42  55  65  75

z→  12  15  33  35  42  45

ix:  3
iy:  2

Yes. So update ix.
Merge

x→ 12 33 35 45

y→ 15 42 55 65 75

z→ 12 15 33 35 42 45

ix: 4
iy: 2

Done with x. Pick from y
Merge

\[ x \rightarrow \begin{array}{ccccc} 12 & 33 & 35 & 45 \\ \end{array} \]

\[ y \rightarrow \begin{array}{ccccc} 15 & 42 & 55 & 65 & 75 \\ \end{array} \]

\[ z \rightarrow \begin{array}{ccccccc} 12 & 15 & 33 & 35 & 42 & 45 & 55 \\ \end{array} \]

So update iy
Merge

\[ x \rightarrow 12 \hspace{10pt} 33 \hspace{10pt} 35 \hspace{10pt} 45 \]

\[ y \rightarrow 15 \hspace{10pt} 42 \hspace{10pt} 55 \hspace{10pt} 65 \hspace{10pt} 75 \]

\[ z \rightarrow 12 \hspace{10pt} 15 \hspace{10pt} 33 \hspace{10pt} 35 \hspace{10pt} 42 \hspace{10pt} 45 \hspace{10pt} 55 \]

\text{ix: 4}

\text{iy: 3}

\text{Done with x. Pick from y}
So update iy.
Merge

\[ x \rightarrow [12, 33, 35, 45] \]
\[ y \rightarrow [15, 42, 55, 65, 75] \]
\[ z \rightarrow [12, 15, 33, 35, 42, 45, 55, 65] \]

\[ \text{i}_{x}: 4 \]
\[ \text{i}_{y}: 4 \]

\[ \text{Done with } x. \text{ Pick from } y \]
Merge

x→ 12 33 35 45

y→ 15 42 55 65 75

z→ 12 15 33 35 42 45 55 65 75

ix: 4
iy: 4

Update iy
All Done
The Python Implementation...
def Merge(x, y):
    n = len(x); m = len(y);
    ix = 0; iy = 0; z = []
    for iz in range(n+m):
        if ix >= n:
            z.append(y[iy]); iy+=1
        elif iy >= m:
            z.append(x[ix]); ix+=1
        elif x[ix] <= y[iy]:
            z.append(x[ix]); ix+=1
        elif x[ix] > y[iy]:
            z.append(y[iy]); iy+=1
    return z

x-list exhausted  y-list exhausted  x-value smaller  y-value smaller
def Merge(x, y):
    n = len(x); m = len(y);
    ix = 0; iy = 0; z = []
    for iz in range(n+m):
        if ix>=n:
            z.append(y[iy]); iy+=1
        elif iy>=m:
            z.append(x[ix]); ix+=1
        elif x[ix] <= y[iy]:
            z.append(x[ix]); ix+=1
        elif x[ix] > y[iy]:
            z.append(y[iy]); iy+=1
    return z

len(x)+len(y) is the total length of the merged list

x-list exhausted  y-list exhausted  x-value smaller  y-value smaller
def Merge(x,y):
    u = list(x)  # Make copies of the Incoming lists
    v = list(y)
    z = []
    while len(u)>0 and len(v)>0 :
        if u[0]<= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
    z.extend(u)
    z.extend(v)
    return z
```python
def Merge(x, y):
    u = list(x)
    v = list(y)
    z = []
    while len(u) > 0 and len(v) > 0:
        if u[0] <= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
    z.extend(u)
    z.extend(v)
    return z
```

Implementation Using Pop

Build z up via repeated appending
def Merge(x, y):
    u = list(x)
    v = list(y)
    z = []
    while len(u) > 0 and len(v) > 0:
        if u[0] <= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
    z.extend(u)
    z.extend(v)
    return z

Implementation Using Pop

Every “pop” reduces the length by 1. The loop shuts down when one of u or v is exhausted.
def Merge(x, y):
    u = list(x)
    v = list(y)
    z = []
    while len(u) > 0 and len(v) > 0:
        if u[0] <= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
    z.extend(u)
    z.extend(v)
    return z
def Merge(x, y):
    u = list(x)
    v = list(y)
    z = []
    while len(u) > 0 and len(v) > 0:
        if u[0] <= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
    z.extend(u)
    z.extend(v)
    return z

Implementation Using Pop
Add what is left in u.
OK if u is the empty list
def Merge(x,y):
    u = list(x)
    v = list(y)
    z = []
    while len(u)>0 and len(v)>0 :
        if u[0]<= v[0]:
            g = u.pop(0)
        else:
            g = v.pop(0)
        z.append(g)
    z.extend(u)
    z.extend(v)
    return z

Add what is left in v.
OK if v is the empty list
MergeSort

Binary Search is an example of a “divide and conquer” approach to problem solving.

A method for sorting a list that features this strategy is MergeSort
**Motivation**

You are asked to sort a list but you have two “helpers”: H1 and H2.

**Idea:**

1. Split the list in half and have each helper sort one of the halves.

2. Then merge the two sorted lists into a single larger list.

This idea can be repeated if H1 has two helpers and H2 has two helpers.
Subdivide the Sorting Task

HEMG BKAQ FLP DRCJ N

HEMG BKAQ FL P DRCJ N
Subdivide Again

<table>
<thead>
<tr>
<th>H</th>
<th>E</th>
<th>M</th>
<th>G</th>
<th>B</th>
<th>K</th>
<th>A</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>L</td>
<td>P</td>
<td>D</td>
<td>R</td>
<td>C</td>
<td>J</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>H</th>
<th>E</th>
<th>M</th>
<th>G</th>
<th>B</th>
<th>K</th>
<th>A</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>L</td>
<td>P</td>
<td>D</td>
<td>R</td>
<td>C</td>
<td>J</td>
<td>N</td>
</tr>
</tbody>
</table>
And Again

H E M G B K A Q F L P D R C J N

H E M G B K A Q F L P D R C J N
And One Last Time
And Merge Again

EGHM     ABKQ     DFLP     CJNR
EHGM     BKAQ     FLDP     CRJN
And Again

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>E</th>
<th>G</th>
<th>H</th>
<th>K</th>
<th>M</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
<td>F</td>
<td>J</td>
<td>L</td>
<td>N</td>
<td>P</td>
<td>R</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>H</td>
<td>M</td>
<td>A</td>
<td>B</td>
<td>K</td>
<td>Q</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>L</td>
<td>P</td>
<td>C</td>
<td>J</td>
<td>N</td>
<td>R</td>
</tr>
</tbody>
</table>
And One Last Time
Done!

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
</table>
Let's write a function to do this making use of

```python
def Merge(x, y):
    """ Returns a float list that is the merge of sorted lists x and y.

    PreC: x and y are lists of floats that are sorted from small to big.
    """
```
8 Merges Producing length-2 lists
Handcoding the n = 16 case

\[
\begin{align*}
A_0 &= \text{Merge}(a[0], a[1]) \\
A_1 &= \text{Merge}(a[2], a[3]) \\
A_2 &= \text{Merge}(a[4], a[5]) \\
A_3 &= \text{Merge}(a[6], a[7]) \\
A_4 &= \text{Merge}(a[8], a[9]) \\
A_5 &= \text{Merge}(a[10], a[11]) \\
A_6 &= \text{Merge}(a[12], a[13]) \\
A_7 &= \text{Merge}(a[14], a[15])
\end{align*}
\]
4 Merges Producing Length-4 lists
Handcoding the $n = 16$ case

B0 = Merge(A0, A1)
B1 = Merge(A2, A3)
B2 = Merge(A4, A5)
B3 = Merge(A6, A7)
2 Merges Producing Length-8 Lists
Handcoding the $n = 16$ case

$C_0 = \text{Merge}(B_0, B_1)$
$C_1 = \text{Merge}(B_2, B_3)$
1 Merge Producing a Length-16 List
All Done!

D0 = Merge(C0,C1)

For general n, it can be handled using recursion.
Recursive Merge Sort

def MergeSort(a):
    n = length(a)
    if n==1:
        return a
    else:
        m  = n/2
        u0 = list(a[:m])
        u1 = list(a[m:]):
        y0 = MergeSort(u0)
        y1 = MergeSort(u1)
        return Merge(y0,y1)

A function can call itself!
Back To Merge Sort
def MergeSort(a):
    n = length(a)
    if n==1:
        return a
    else:
        m  = n/2
        u0 = list(a[:m])
        u1 = list(a[m:]
        y0 = MergeSort(u0)
        y1 = MergeSort(u1)
        return Merge(y0,y1)

Recursive Merge Sort
A function can call Itself!
A Sorted List is produced at each ":". Let's look at the order in which lists are sorted.
A Schematic

A Sorted List is produced at each “:”. Let’s look at the order in which lists are sorted.
A Schematic

A Sorted List is produced at each “:” Let's look at the order in which...
A Sorted List is produced at each "::". Let's look at the order in which
A Sorted List is produced at each “:”  Let’s look at the order in which
A Sorted List is produced at each “:“  Let’s look at the order in which
A Sorted List is produced at each “:”  Let’s look at the order in which
A Schematic

A Sorted List is produced at each “::” Let’s look at the order in which lists are sorted.
A Sorted List is produced at each “:”. Let’s look at the order in which
A Sorted List is produced at each “:” Let’s look at the order in which lists are sorted.
A Sorted List is produced at each “:” Let’s look at the order in which lists are sorted.
A Sorted List is produced at each “:“ Let’s look at the order in which lists are sorted.
A Schematic

A Sorted List is produced at each “:” Let’s look at the order in which lists are sorted.
A Sorted List is produced at each “:” Let’s look at the order in which lists are sorted.
A Sorted List is produced at each “:” Let’s look at the order in which lists are sorted.
A Schematic

All Done!
Some Conclusions

Infinite recursion (like infinite loops) can happen so careful reasoning is required.

Will we reach the “base case”?

In MergeSort, a recursive call always involves a list that is shorter than the input list. So eventually we reach the len(a)==1 base case.