24. 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 -->  50  40  10  80  20  60

After:

x -->  10  20  40  50  60  80
We Will First Implement the Method of Selection Sort

At the Start:

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

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:

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

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

Before:

\[
x \quad \rightarrow \quad 50 \, 40 \, 10 \, 80 \, 20 \, 60
\]

Swap \( x[0] \) with the smallest value in \( x[0:] \)

After:

\[
x \quad \rightarrow \quad 10 \, 40 \, 50 \, 80 \, 20 \, 60
\]
Selection Sort: How It Works

Before:

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

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

**After:**

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

Before:

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

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:

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

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

Before:

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

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

After:

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

Before:

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

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

Before:

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

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

After:

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

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>50</td>
<td>80</td>
<td>40</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>40</td>
<td>80</td>
<td>50</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>40</td>
<td>50</td>
<td>80</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>80</td>
<td></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 in that satisfies 0<=i<len(x)"""
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`.

```
 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.
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.
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)"""
The list `a` looks like this after each select operation:

<table>
<thead>
<tr>
<th>Initialization</th>
<th>After this:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Select(a,0)</code></td>
<td>50 40 10 80 20 60</td>
</tr>
<tr>
<td><code>Select(a,1)</code></td>
<td>10 40 50 80 20 60</td>
</tr>
<tr>
<td><code>Select(a,2)</code></td>
<td>10 20 50 80 40 60</td>
</tr>
<tr>
<td><code>Select(a,3)</code></td>
<td>10 20 40 80 50 60</td>
</tr>
<tr>
<td><code>Select(a,4)</code></td>
<td>10 20 40 50 80 60</td>
</tr>
<tr>
<td><code>Select(a,5)</code></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

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

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

ix and iy keep track of where we are in x and y

z→ [ ]
Merging Two Sorted Lists

Do we pick from \( x \)?

\[
x[ix] \leq y[iy] \quad ???
\]
Yes. So update ix
Do we pick from $x$?  $x[ix] \leq y[iy]$  ???
Merge

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

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

\[ z \rightarrow \begin{pmatrix} 12 & 15 \end{pmatrix} \]

\[ ix: 1 \]

\[ iy: 0 \]

No. So update iy
Merge

x→

| 12 | 33 | 35 | 45 |

y→

| 15 | 42 | 55 | 65 | 75 |

z→

| 12 | 15 |

ix: 1

iy: 1

Do we pick from x? x[ix] ≤ y[iy] ???
Merge

x→ 12 33 35 45

y→ 15 42 55 65 75

z→ 12 15 33

ix: | iy: |
| 1  |
| 1  |

Yes. So update ix
Merge

Do we pick from $x$?

$x[ix] \leq y[iy]$  

$x \rightarrow 12 33 35 45$

$y \rightarrow 15 42 55 65 75$

$z \rightarrow 12 15 33$

$ix: 2$

$iy: 1$
Merge

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

Yes. So update ix
Do we pick from $x$?  

$x[ix] \leq y[iy]$  

???
No. So update iy...
Merge

x\rightarrow 12 \ 33 \ 35 \ 45

y\rightarrow 15 \ 42 \ 55 \ 65 \ 75

z\rightarrow 12 \ 15 \ 33 \ 35 \ 42

ix: 3

iy: 2

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

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

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

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

\[ ix: \begin{array}{c} 4 \end{array} \]

\[ iy: \begin{array}{c} 2 \end{array} \]

Done with x. Pick from y
Merge

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

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

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

So update iy
Merge

\[
\begin{align*}
x & \rightarrow & 12 & 33 & 35 & 45 \\
y & \rightarrow & 15 & 42 & 55 & 65 & 75 \\
z & \rightarrow & 12 & 15 & 33 & 35 & 42 & 45 & 55
\end{align*}
\]

\[\text{ix: 4} \]
\[\text{iy: 3} \]

Done with x. Pick from y
So update iy.
Done with x. Pick from y
Merge

\[
\begin{align*}
x \rightarrow & \quad \begin{array}{cccc}
12 & 33 & 35 & 45 \\
\end{array} \\
y \rightarrow & \quad \begin{array}{cccc}
15 & 42 & 55 & 65 & 75 \\
\end{array} \\
z \rightarrow & \quad \begin{array}{ccccccc}
12 & 15 & 33 & 35 & 42 & 45 & 55 & 65 & 75 \\
\end{array}
\end{align*}
\]

Update iy
Merge

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

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

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

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
```python
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)
    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
Make copies of the Incoming lists
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

Implementation Using Pop

g gets the popped value and it is appended to 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)  # Add what is left in u.
    z.extend(v)  # OK if u is the empty list
    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 v.
OK if v is the empty list
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
Subdivide Again

HEMGKBKAQ

FLPDRCJN

HEMG BKAQ

FLPDRCJN
And Again

HEMG BKAQ

FLPD RCJN

HEMG BK AQ FL PD RC JN
And One Last Time
Now Merge
And Merge Again
And Again
And One Last Time
Done!
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

EGHM ABKQ DFLP CJNR
EH GM BK AQ FL DP CR JN
Handcoding the $n=16$ case

\[ \begin{align*}
B_0 &= \text{Merge}(A_0,A_1) \\
B_1 &= \text{Merge}(A_2,A_3) \\
B_2 &= \text{Merge}(A_4,A_5) \\
B_3 &= \text{Merge}(A_6,A_7)
\end{align*} \]
2 Merges Producing Length-8 Lists
Handcoding the $n=16$ case

\[
\begin{align*}
C_0 &= \text{Merge}(B_0, B_1) \\
C_1 &= \text{Merge}(B_2, B_3)
\end{align*}
\]
1 Merge Producing a Length-16 List
All Done!

$$D_0 = \text{Merge}(C_0, C_1)$$

For general $n$, it can be handled using recursion.
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)
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 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
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 lists are sorted.
A Schematic

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 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.