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:
\[ x \rightarrow 50 \ 40 \ 10 \ 80 \ 20 \ 60 \]

After:
\[ x \rightarrow 10 \ 20 \ 40 \ 50 \ 60 \ 80 \]

We Will First Implement the Method of Selection Sort

At the Start:
\[ x \rightarrow 50 \ 40 \ 10 \ 80 \ 20 \ 60 \]

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:
\[ x \rightarrow 50 \ 40 \ 10 \ 80 \ 20 \ 60 \]

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

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

Before:

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

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

After:

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

Selection Sort: How It Works

Before:

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

Swap \( x[2] \) with the smallest value in \( x[2:] \)

After:

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

Selection Sort: How It Works

Before:

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

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

After:

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

Before:

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

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

Selection Sort: Recap

After:

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

The Essential Helper Function: \( \text{Select}(x,i) \)

```python
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)"
    Does not return anything and it has a list argument
```

How Does it Work?

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

\[ a \rightarrow [0 \rightarrow 50, 1 \rightarrow 40, 2 \rightarrow 10, 3 \rightarrow 80, 4 \rightarrow 20, 5 \rightarrow 60] \]

How Does it Work?

The calling program executes \( \text{Select}(a,0) \) and control passes to \( \text{Select} \)
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.

```
  a ---> 0 ----> 50
        1 ----> 40
        2 ----> 10
        3 ----> 80
        4 ----> 20
        5 ----> 60
```

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
```

```
  a ---> 0 ----> 50
        1 ----> 40
        2 ----> 10
        3 ----> 80
        4 ----> 20
        5 ----> 60
```

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

```
  a ---> 0 ----> 10
        1 ----> 40
        2 ----> 50
        3 ----> 80
        4 ----> 20
        5 ----> 60
```

Let's Assume This Is Implemented

```
def Select(x, i):
    """ Swaps the smallest value in x[i:] with x[i]"
    t = x[i]; x[i] = x[i+1]; x[i+1] = t

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

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

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

Merging Two Sorted Lists

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

Do we pick from \( x \)?

\[ x[ix] \leq y[iy] \text{ ???} \]

Merge

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

Yes. So update \( ix \)

Do we pick from \( x \)?

\[ x[ix] \leq y[iy] \text{ ???} \]
Merge

x -> 12 33 35 45  
  iy: 0
y -> 15 42 55 65 75  
  iy: 1
z -> 12 15

No. So update iy

Merge

x -> 12 33 35 45  
  iy: 1
y -> 15 42 55 65 75  
  iy: 1
z -> 12 15

Do we pick from x? x[ix] <= y[iy] ??

Merge

x -> 12 33 35 45  
  iy: 1
y -> 15 42 55 65 75  
  iy: 1
z -> 12 15 33

Yes. So update ix

Merge

x -> 12 33 35 45  
  iy: 2
y -> 15 42 55 65 75  
  iy: 1
z -> 12 15 33

Do we pick from x? x[ix] <= y[iy] ??

Merge

x -> 12 33 35 45  
  iy: 2
y -> 15 42 55 65 75  
  iy: 1
z -> 12 15 33 35

Yes. So update ix

Merge

x -> 12 33 35 45  
  iy: 3
y -> 15 42 55 65 75  
  iy: 1
z -> 12 15 33 35

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 35 42

No. So update iy...

Merge

x→ 12 33 35 45
y→ 15 42 55 65 75
z→ 12 15 33 35 42

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 35 42 45

Yes. So update ix.

Merge

x→ 12 33 35 45
y→ 15 42 55 65 75
z→ 12 15 33 35 42 45

Done with x. Pick from y

Merge

x→ 12 33 35 45
y→ 15 42 55 65 75
z→ 12 15 33 35 42 45

Done with x. Pick from y

Merge

x→ 12 33 35 45
y→ 15 42 55 65 75
z→ 12 15 33 35 42 45 55

Update iy
Merge

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

So update iy.

x: 4  
y: 3  
z: 5

Merge

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

Done with x. Pick from y

Merge

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

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
        else:
            z.append(y[iy]); iy+=1
    return z

Build z up via repeated appending

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.

---

**Implementation Using Pop**

```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
```

- Make copies of the incoming lists.
- Every "pop" reduces the length by 1. The loop shuts down when one of u or v is exhausted.
- g gets the popped value and it is appended to z.
- Add what is left in u. OK if u is the empty list.
**Implementation Using Pop**

```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
```

**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**

```
| E | M | C | R | K | A | Q | F | L | P | D | R | C | J | N
```

```
| E | M | C | R | K | A | Q | F | L | P | D | R | C | J | N
```

**Subdivide Again**

```
| E | M | C | R | K | A | Q | F | L | P | D | R | C | J | N
```

```
| E | M | C | R | K | A | Q | F | L | P | D | R | C | J | N
```

**And Again**

```
| E | M | C | R | K | A | Q | F | L | P | D | R | C | J | N
```

```
| E | M | C | R | K | A | Q | F | L | P | D | R | C | J | N
```
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

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.
    """

Handcoding the n =16 case

A0 = Merge(a[0], a[1])
A1 = Merge(a[2], a[3])
A2 = Merge(a[4], a[5])
A3 = Merge(a[6], a[7])
A4 = Merge(a[8], a[9])
A5 = Merge(a[10], a[11])
A6 = Merge(a[12], a[13])
A7 = Merge(a[14], a[15])

Handcoding the n =16 case

B0 = Merge(A0, A1)
B1 = Merge(A2, A3)
B2 = Merge(A4, A5)
B3 = Merge(A6, A7)
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

For general \( n \), it can be handled using recursion.

All Done!

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

Recursive Merge Sort

\[
\begin{align*}
def \text{MergeSort}(a): \\
& n = \text{length}(a) \\
& \text{if } n=1: \\
& \quad \text{return } a \\
& \text{else:} \\
& \quad m = n/2 \\
& \quad u_0 = \text{list}(a[0:m]) \\
& \quad u_1 = \text{list}(a[m:]) \\
& \quad y_0 = \text{MergeSort}(u_0) \\
& \quad y_1 = \text{MergeSort}(u_1) \\
& \quad \text{return } \text{Merge}(y_0, y_1)
\end{align*}
\]

Back To Merge Sort

Recursive Merge Sort

\[
\begin{align*}
def \text{MergeSort}(a): \\
& n = \text{length}(a) \\
& \text{if } n=1: \\
& \quad \text{return } a \\
& \text{else:} \\
& \quad m = n/2 \\
& \quad u_0 = \text{list}(a[0:m]) \\
& \quad u_1 = \text{list}(a[m:]) \\
& \quad y_0 = \text{MergeSort}(u_0) \\
& \quad y_1 = \text{MergeSort}(u_1) \\
& \quad \text{return } \text{Merge}(y_0, y_1)
\end{align*}
\]

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.
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 $\text{len}(a)=1$ base case.