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

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:

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

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:

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

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:

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

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

After:

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

Selection Sort: Recap

The Essential Helper Function:

\[ \text{def Select}(x, i) \]

```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 \(0 \leq i < \text{len}(x)\)"
    
    Does not return anything and it has a list argument
```

How Does it Work?

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

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

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.

0
---
50
---
10
---
40
---
20
---
60
---
30
---
80
---
40
---
20
---
50
---
80
---
60
---
70
---
90
---
100
---

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.

0
---
50
---
10
---
40
---
20
---
60
---
30
---
80
---
40
---
20
---
50
---
80
---
60
---
70
---
90
---
100
---

Let's Assume This Is Implemented

```python
def Select(x, i):
    """ Swaps the smallest value in x[i:] with x[i]"
    t = x[i]; x[i] = x[2]; x[2] = t

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

In General We Have This

```python
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

\[
\begin{array}{c}
\text{x-} & 12 & 33 & 35 & 45 \\
\text{y-} & 15 & 42 & 55 & 65 & 75 \\
\end{array}
\]

\( x \) and \( y \) are input

They are sorted

\( z \) is the output

\[
\begin{array}{c}
\text{z-} & 12 & 15 & 33 & 35 & 42 & 45 & 55 & 65 & 75 \\
\end{array}
\]

Merging Two Sorted Lists

\[
\begin{array}{c}
\text{x-} & 12 & 33 & 35 & 45 \\
\text{y-} & 15 & 42 & 55 & 65 & 75 \\
\text{z-} & [] \\
\end{array}
\]

\( \text{ix:} \) 0

\( \text{iy:} \) 0

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

Merging Two Sorted Lists

\[
\begin{array}{c}
\text{x-} & 12 & 33 & 35 & 45 \\
\text{y-} & 15 & 42 & 55 & 65 & 75 \\
\text{z-} & [] \\
\end{array}
\]

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

Merge

\[
\begin{array}{c}
\text{x-} & 12 & 33 & 35 & 45 \\
\text{y-} & 15 & 42 & 55 & 65 & 75 \\
\text{z-} & 12 \\
\end{array}
\]

Yes. So update \( \text{ix} \)

Merge

\[
\begin{array}{c}
\text{x-} & 12 & 33 & 35 & 45 \\
\text{y-} & 15 & 42 & 55 & 65 & 75 \\
\text{z-} & 12 \\
\end{array}
\]

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

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

Yes. So update ix

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

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

Yes. So update ix.

Done with x. Pick from y

Update iy
The Python Implementation...

```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
        else:
            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

Implementation Using Pop

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

Implementation Using Pop

def Merge(x, y):
    u = list(x)  # Build z up via repeated appending
    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

def Merge(x, y):
    u = list(x)  # Every "pop" reduces the length by 1. The loop shuts down when one of u or v is exhausted
    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

def Merge(x, y):
    u = list(x)  # g gets the popped value and it is appended to z
    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

def Merge(x, y):
    u = list(x)  # Add what is left in u.
    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

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

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

And Again
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.
    """
```

Handcoding the \( n = 16 \) case

\[
\begin{align*}
A0 &= \text{Merge}(a[0], a[1]) \\
A1 &= \text{Merge}(a[2], a[3]) \\
A2 &= \text{Merge}(a[4], a[5]) \\
A3 &= \text{Merge}(a[6], a[7]) \\
A4 &= \text{Merge}(a[8], a[9]) \\
A5 &= \text{Merge}(a[10], a[11]) \\
A6 &= \text{Merge}(a[12], a[13]) \\
A7 &= \text{Merge}(a[14], a[15])
\end{align*}
\]

Handcoding the \( n = 16 \) case

\[
\begin{align*}
B0 &= \text{Merge}(A0, A1) \\
B1 &= \text{Merge}(A2, A3) \\
B2 &= \text{Merge}(A4, A5) \\
B3 &= \text{Merge}(A6, A7)
\end{align*}
\]
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!

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

For general $n$, it can be handled using recursion.

Recursive Merge Sort

```python
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!
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 \( \vdash \) 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.