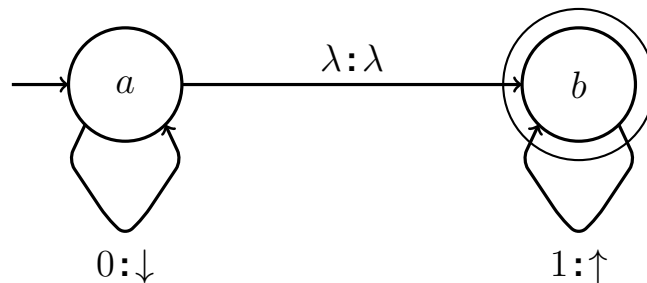# Programming Dictionary

By Charles Cook

Begun on Dec. 6th, 2018

1 Compiler Theoretical Foundations

Push Down Automata for strings of the form 0^n1^n

Key:

Symbol	Meaning
a, b	State Node
0, 1	Symbol to Print
↓	Push (onto the stack)
↑	Pull (off of the stack)
λ	Null operation (no print, push, or pull)

Derivation Examples: 01, 00001111, 0000000011111111.

2 Programming Tools & Languages: First Ten Questions

2.1 Swap Function

2.1.1 Pseudocode for *swap*

```
1  void swap(a, b) {
2         temp = a;
3         a = b;
4         b = temp;
5  }
6
7  void swapNoTemp(a, b) {
8         a += b;  // a = a + b
9         b -= a;  // b = b - (a + b) = -a
10        b *= -1;  // b = a
11        a -= b;  // a = a + b - a = b
12 }
```

2.1.2 *swaptest.c*

```
1  #include <stdio.h>
2  void swap(int *a, int *b) {
3         *a += *b;
4         *b -= *a;
5         *b *= -1;
6         *a -= *b;
7  }
8
9  int main() {
10        int x, y, z;
11        x = 10;
12        y = 13;
13        z = 2;
14
15        swap(&x, &y);
16        swap(&x, &z);
17        swap(&y, &z);
18        printf("x: &d\ny: &d\nz: &d\n", x, y, z);
19
20        return 0;
21 }
```

2.1.3 *swaptest.cpp*

```cpp
1  #include <iostream>
2  using namespace std;
3
4  void swap(int *a, int *b) {
5          *a += *b;
6          *b -= *a;
7          *b *= -1;
8          *a -= *b;
9  }
10
11 int main() {
12         int x, y;
13         x = 13;
14         y = 29;
15
16         cout << x << ", " << y << "\n";
17         swap(x, y);
18         cout << x << ", " << y << "\n";
19
20         return 0;
21 }
```

2.2 Reverse an array with no extra space (pseudocode)

```
1  void reverseArray(array, int length) {
2          for (int i = 0; i < length / 2; i++) {
3                  swap(array[i], array[length - i - 1]);
4          }
5  }
```

2.3 Reverse a doubly linked list (pseudocode)

```
1  struct Node {
2          int data;
3          struct Node *next;
4          struct Node *prev;
5  };
6
7  void reverseDLL(head, tail) {
8          struct Node tempH = head;
9          struct Node tempT = tail;
10
11         while (tempH -> next != tempT -> prev && tempH != tempT) {
12                 swap(tempH -> data, tempT -> data);
13         }
14 }
```

2.4 Reverse a doubly linked list recursively (pseudocode)

```
1 void reverseDLL_Recursive(head, tail) {
2       if (head -> next != tail -> prev && head != tail) {
3               swap(head -> data, tail -> data);
4               reverseDLL_Recursive(head -> next, tail -> prev);
5       }
6 }
```

2.5 Pointer to a Pointer

A use of a pointer-to-a-pointer in C would be to insert an element into a sorted singly-linked list. Take the following list:

```
1 head:
2   [2] -> [4] -> [7] -> [10] -> [13]
```

Say we want to insert the node **[9]** into the list; If we make a pointer-to-pointer, call it **pp**, and make it point to the **next** attribute of each node, we can insert **[9]** as follows:

First, we set the value of **pp** to be the **head** of the list:

*pp = head;

where head is the pointer to the first struct in the list. We can visualize this below:

```
1   pp
2    |
3    V
4 head:
5   [2] -> [4] -> [7] -> [10] -> [13]
```

*pp is **[2]**, and **[2] -> data** is 2, which is less than 9. Until we find a candidate ***pp -> data** that is greater than 9, we keep going by setting **pp** to the address of the next node with ***pp = &(*pp -> next)**.

```
1   pp -+
2       |
3 head:V
4   [2] -> [4] -> [7] -> [10] -> [13]
```

*pp is **[2] -> next** which is **[4]**, **[4] -> data** is 4 which is less than 9, keep going; ***pp = &(*pp -> next)**

```
1   pp --------+
2             |
3 head:       V
4   [2] -> [4] -> [7] -> [10] -> [13]
```

*pp is **[4] -> next** which is **[7]**, **[7] -> data** is 7 which is less than 9, keep
going; ***pp = &(*pp -> next)**

```
1    pp  --------------+
2                      |
3  head:               V
4    [2]  -> [4]  -> [7]  -> [10]  -> [13]
```

Paydirt! ***pp -> data** is 10, which is greater than 9. Now we do the following:

[9] -> next = *pp;

```
1    pp  --------------+
2                      |
3  head:               V
4    [2]  -> [4]  -> [7]  -> [10]  -> [13]
5                            ^
6                            |
7                     [9]  -+
```

*pp = [9];

```
1    pp  --------------+
2                      |
3  head:               V
4    [2]  -> [4]  -> [7]  -+ [10]  -> [13]
5                         /  ^
6                        V   |
7                     [9]  -+
```

The list is still ordered, and **pp** is needed no longer.

2.6 Sorting Algorithm with $O(n)$ Complexity

An optimised Bubble Sort that stops after doing a pass thru an array wherein no
swaps occur will run in $O(n)$ time on an already-sorted list. In a sorted list,
comparing every element with its next element all the way to the end will not
prompt Bubble Sort to perform any swaps at all. When a single pass of the array
is completed and no swaps have occured, a proper Bubble Sort will stop, as no
swaps is a sign of a sorted list.

2.7 Selection Sort Complexity

Consider the worst case of Selection Sort: a completely-reversed array, say
$(6, 5, 4, 3, 2, 1)$. On the first pass, 1 is selected, and has to be moved 5 indices,
$n - 1$. On the second pass, 2 is selected, and has to be moved 4 indices, $n - 2$.
This eventually boils down to $n - 1 + n - 2 + n - 3 + n - 4 + n - 5$ operations; given
$n = 6$, this becomes $6 - 1 + 6 - 2 + 6 - 3 + 6 - 4 + 6 - 5 = 5 + 4 + 3 + 2 + 1 = 5 * 6/2$ (per
Gauss' Formula). If we reintroduce n, we get $\frac{(n-1)*n}{2} = \frac{n^2-n}{2}$. Overall, Selection
Sort's worst case (and average case) boils down to $O(n^2)$.

5

2.8 Insertion Sort Complexity

Consider again $(6, 5, 4, 3, 2, 1)$. On the first pass of Insertion Sort, the 5 is swapped 1 time, $n - 5$. On the second pass, the 4 is swapped 2 times, $n - 4$. Note the pattern. Overall, there are $n-5+n-4+n-3+n-2+n-1$. Recall from the previous problem on Selection Sort that this boils down to $\frac{(n-1)n}{2} = \frac{n^2-n}{2}$. Insertion Sort is also worst-case and average-case $O(n^2)$ complexity.

2.9 *ordered* Predicate in Prolog

2.9.1 *sorted.pl*

```
1  ordered([X]).
2  ordered([X, Y|Z]) :- X < Y, ordered([Y|Z]).
```

2.9.2 Output on $[1, 2, 4, 8, 16]$ with trace-on in GProlog

```
1  | ?- ordered([1,2,4,8,16]).
2          1    1  Call: ordered([1,2,4,8,16]) ?
3          2    2  Call: 1<2 ?
4          2    2  Exit: 1<2 ?
5          3    2  Call: ordered([2,4,8,16]) ?
6          4    3  Call: 2<4 ?
7          4    3  Exit: 2<4 ?
8          5    3  Call: ordered([4,8,16]) ?
9          6    4  Call: 4<8 ?
10         6    4  Exit: 4<8 ?
11         7    4  Call: ordered([8,16]) ?
12         8    5  Call: 8<16 ?
13         8    5  Exit: 8<16 ?
14         9    5  Call: ordered([16]) ?
15         9    5  Exit: ordered([16]) ?
16         7    4  Exit: ordered([8,16]) ?
17         5    3  Exit: ordered([4,8,16]) ?
18         3    2  Exit: ordered([2,4,8,16]) ?
19         1    1  Exit: ordered([1,2,4,8,16]) ?
20
21 true ?
22
23 yes
24 {trace}
25 | ?-
```

2.10 Labelled Path (with cost) in Prolog

2.10.1 ASCII-art chart of my path

```
1                                   +-[11]-> (d) -[7]-> (f)
2 (a) -[1]-> (b) -[3]-> (c) -+
3                                   +-[5]-> (e) -+-[13]-> (g)
4                                                 |
5                                                 +-[19]-> (h)
```

2.10.2 *charliesPath.pl*

```prolog
1 link(a, b, 1).
2 link(b, c, 3).
3 link(c, d, 11).
4 link(c, e, 5).
5 link(d, f, 7).
6 link(e, g, 13).
7 link(e, h, 19).
8
9 path(Origin, Destination, Distance) :-
10        link(Origin, Destination, Distance)
11 .
12 path(Origin, Destination, TotalDistance) :-
13        link(Origin, Intermediate, DistanceA),
14        path(Intermediate, Destination, DistanceB),
15        TotalDistance is DistanceA + DistanceB
16 .
```

2.10.3 Recursion Analysis

Whenever path is called with two locations that are indirectly linked, the first thing that will happen is that all directly linked locations to the Origin will be checked; these are the Intermediate locations. These locations will eached be checked to see if they have a direct link to the Destination, otherwise their directly linked locations will act as new Intermediates and themselves go through the same check.

Once an Intermediate is found that directly connects to the Destination, the recursion will cease and the value of true will be returned, along with the total distance. If all recursion becomes exhausted and no satisfactory Intermediate is found, the value of false will be returned.

3 Programming Tools & Languages: Last Ten Questions

3.1 Number of nodes in a linked list (pseudocode)

```
1  int numberOfNodes(head) {
2         int n = 1;
3         temp = head;
4
5         while (temp -> next != NULL) {
6                 n += 1;
7                 temp = temp -> next;
8         }
9
10        return n;
11 }
```

3.2 Number of nodes in a linked list (pseudocode, $O(\frac{n}{2})$ complexity)

```
1  int numberOfNodesQuick(head) {
2         int n = 1;
3         temp = head;
4
5         while (temp -> next != NULL && temp -> next -> next != NULL) {
6                 n += 2;
7                 temp = temp -> next -> next;
8         }
9
10        if (temp -> next != NULL) {
11                n += 1;
12        }
13
14        return n;
15 }
```

3.3 Return middle node of linked list (pseudocode)

```
1  node middleNode(head, int length) {
2         temp = head;
3
4         for (int i = 0; i < length / 2; i++) {
5                 temp = temp -> next;
6         }
7
8         return temp;
9  }
```

3.4 Return middle node of linked list with two pointers

Declare two pointers and initialize them with the head value. Let the first
pointer be known as the one-jump, and the second as the two-jump. While the
two-jump pointer's next is not equal to NULL and it's next-next is not equal to
NULL, have the one-jump pointer take on the next value one node ahead, and have
the two-jump pointer take on the next value two nodes ahead. When the two-jump
has a next that is NULL, the one-jump will be in the middle.

For example, if a linked list has 9 nodes, the two-jump will go from node 1 to
3 to 5 to 7 to 9, then stop. The one-jump will go from node 1 to 2 to 3 to 4 to
5, then stop. 5 is halfway between node 1 and node 9.

3.5 Circualr singly-linked list check with two pointers

Declare two pointers as before when finding the middle node of a linked list:
one-jump and two-jump. A circular linked list can be detected if you repeatedly
have the one-jump update to its next node, and have the two-jump update to its
next-next node. If ever the nodes ever are equal, i.e. they are pointing to
the same node, then there is a cycle in the list. If there wasn't, the two-jump
would reach a terminal point well ahead of the one-jump.

3.6 Reachability

3.7 Stable vs. Non-Stable Sorting Algorithms

A stable sorting algorithm is one that preserves the order of elements of equal
value from before sorting to after sorting. Insertion Sort is stable, because
when an equal value element is inserted, it will not be able to move beyong another
equal value element, because the citeria for an insertion swap requires the lower
element to be strictly less than the higher element. Selection Sort, in its
simplest implementation, is not stable. If a minimum finding function that uses
a linear search is used, the last instance of an equal value element will become
the first instance, breaking stability.

3.8 Fundamental Data Structures

- Array

- Dictionary (a.k.a. Hash Table, Associative Array)

- Linked List (singly or doubly linked)

- Binary Search Tree (Lefthand children are smaller, righthand are larger)

- Heap Tree (In a maximum heap, all children are smaller)

- Stack

3.9 P vs. NP

The "P" in P vs. NP stands for polynomial. If an algorithm qualifies for a
P-class problem, then the Big-O complexity of the algorithm is modelled by a
polynomial expression; In short, the algorithm runs in polynomial-class time
in reference to the size of the input. Alternatly, if an algorithm is of the
NP-class, its time complexity is non-polynomial, which is ususally worse than
polynomial complexity. For example, there are only NP-class algorithms for solutions
to the Travelling Salesman Problem, which entales travelling to every node in
a graph (every city in a region) and returning to your starting location while
minimizing distance.

3.10 Class vs. Struct

A struct is a composite data type with multiple data fields of any type defined
in the programming language. A class combines the data-ordering of a struct
with associated functions called methods, which manipulate the data ordered by
the class. When a class is instantiated, the instance is referred to as an object.

4 Palindromes in Prolog

4.1 palindrome.pl

```prolog
1 concat([], L, L).
2 concat([X|L1], L2, [X|L3]) :- concat(L1, L2, L3).
3 reverselist([], []).
4 reverselist([Head|Tail], Reversed) :-
5        reverselist(Tail, ReversedTail),
6        concat(ReversedTail, [Head], Reversed)
7 .
8 palindrome(X) :- reverselist(X, X).
```

4.2 Analysis

4.2.1 concat()

The three parameters in concat() are the lefthand list, the righthand list, and the concatenated list. Clearly, the concatenated list is where all the lefthand list's elements are added first, then followed by the elements of the righthand list. As seen in the second statement, To concatenate $[X|L_1]$ with L_2, L_1 and L_2 have to concatenate to L_3, which results in $[X|L_3]$. Eventually, after enough recursive calls, L_1 will be an empty list, in which case the first statement will be invoked, returning L_2 back up the stack.

4.2.2 reverselist()

The two parameters in reverselist() are the original list, $[Head|Tail]$, and the reversed list, $Reversed$. First, a recursive call is issued with $Tail$ and $ReversedTail$ as the parameters. Second, the recursive call is conjucted with a concat() call of $ReversedTail$ with $[Head]$, which results in $Reveresed$. Back to the first call, the $Tail$ will repeatedly be widdled down until it is empty, which will fulfill the statement that an empty list is the reverse of itself. When this happens, the final element will be in $ReversedTail$ in the highest call (stack-wise) will be concatenated to the element before it, and this will cascade down the stack of calls, completing the $Reversed$ list.

4.2.3 palindrome()

A palindrome is any ordered arrangement that is its own reverse. Thus, if a call of reverselist() is issued with X and X as both parameters, and the end result is true, then X is a palindrome.

5 My Project: Visualized Sorting Algorithms Demonstration
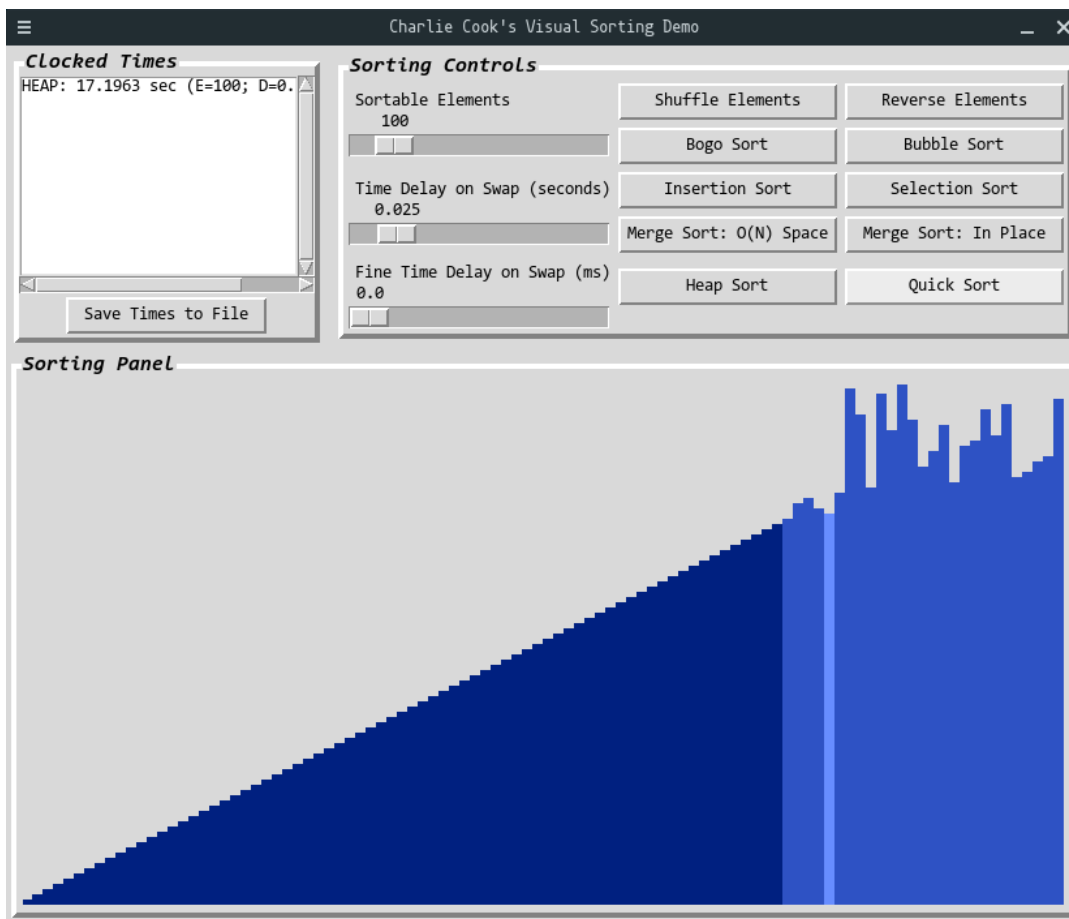
I wrote this demo program in Python, mainly with the Tcl/Tk binding library "tkinter", that implements the following sorting algorithms on a visualized list of numbers represented by rectangles:

- Bogo

- Bubble (optimized)

- Insertion

- Selection

- Merge Sort

    { $O(0)$ extra space a.k.a. In-Place
    { $O(n)$ extra space a.k.a. Overwrite

- Heap

- Quick

You can adjust the amount of elements to sort from 10 to 800 in 10-element increments, introduce a time delay accurate to 10^{-4} seconds, and write recorded sorting time information to a text file. Check out https://github.com/cSquaerd/visualSort for more updates in the future. The code is as follows for Version 1.2:

5.1 ccVisualSort.py

5.1.1 Imports and Time Management

```
1   import tkinter as tk
2   import tkinter.simpledialog as sdg
3   import tkinter.messagebox as mbx
4   import tkinter.font as tkf
5   import tkinter.filedialog as fdg
6   import random as rnd
7   import platform as pt
8   import time
9
10  if pt.system() == "Linux":
11      id = "~"
12  elif pt.system() == "Windows":
13      id = "C:\\"
14
15  base = tk.Tk()
16  base.title("Charlie Cook's Visual Sorting Demo")
17  base.resizable(False, False)
18
19  # Fonts
20  fontSmall = tkf.Font(family = "Consolas", size = 8)
21  fontNormal = tkf.Font(family = "Consolas", size = 10)
22  fontNormUnd = tkf.Font(family = "Consolas", size = 10, underline = 1)
23  fontNormBold = tkf.Font(family = "Consolas", size = 10, weight = "bold")
24  fontLarge = tkf.Font(family = "Consolas", size = 12)
25  fontLargeBold = tkf.Font(family = "Consolas", size = 12, weight = "bold")
26  fontSubtitle = tkf.Font(family = "Consolas", size = 12, slant = "italic")
27  fontTitle = tkf.Font(family = "Consolas", size = 12, weight = "bold", slant = "italic")
28  fontBigTitle = tkf.Font(family = "Consolas", size = 16, weight = "bold", slant = "italic")
29  fontName = tkf.Font(family = "Consolas", size = 16, weight = "bold")
30
31  frameTimes = tk.LabelFrame(base, text = "Clocked Times", bd = 4, relief = "raised", font = fontTitle)
32  frameTimes.grid(row = 0, column = 0, padx = 4, pady = 4, sticky = "n")
33
34  listboxTimes = tk.Listbox(frameTimes, width = 30, height = 10,  font = fontNormal)
35  listboxTimes.grid(row = 0, column = 0)
36
37  scrollVTimes = tk.Scrollbar(frameTimes, orient = "vertical", command = listboxTimes.yview)
38  scrollHTimes = tk.Scrollbar(frameTimes, orient = "horizontal", command = listboxTimes.xview)
39  listboxTimes.configure(yscrollcommand = scrollVTimes.set, xscrollcommand = scrollHTimes.set)
40  scrollVTimes.grid(row = 0, column = 1, sticky = "ns")
41  scrollHTimes.grid(row = 1, column = 0, columnspan = 2, sticky = "we")
42
43  def clockTime(start, end, algo):
44      listboxTimes.insert("end", algo + ": " + str(round(end - start, 4)) + " sec (E=" + \
45          str(elements.get()) + "; D=" + str(sleepTime.get() + sleepTimeFine.get() / 1000) + \
46          " sec)" \
47      )
48
49  def saveTimes():
50      savefile = fdg.asksaveasfilename(parent = base, title = "Select or Enter a file to save to:",
51          initialdir = id, filetypes = (("Text Files","*.txt"),("All Files","*.*")))
52
53      if type(savefile) is str and len(savefile) > 0:
54          file = open(savefile, "w")
55
56          for clock in listboxTimes.get(0, "end"):
57              file.write(clock + "\n")
58
59          file.close()
60
61  buttonSaveTimes = tk.Button(frameTimes, text = "Save Times to File", bd = 2, command = saveTimes, font =
        fontNormal)
    buttonSaveTimes.grid(row = 2, column = 0, columnspan = 2, padx = 2, pady = 2)
```

5.1.2 Base Frames and Element Processing

```
63  frameControls = tk.LabelFrame(base, text = "Sorting Controls", bd = 4, relief = "raised", font =
        fontTitle)
64  frameControls.grid(row = 0, column = 1, padx = 4, pady = 4)
65
66  frameMain = tk.LabelFrame(base, text="Sorting Panel", bd = 4, relief = "raised", font = fontTitle)
67  frameMain.grid(row = 1, column = 0, columnspan = 2, padx = 4, pady = 4)
68
69  frameScreen = tk.Canvas(frameMain, width = 800, height = 400)
70  frameScreen.pack(padx = 4, pady = 4)
71
72  elementHeights = list(range(1, 11))
73  elementColorCoding = {"indicated": 0, "sortedBorder": -1, "sortedSide": "none"}
74
75  def processColor(element):
76      colorNormal = "#2E52C4"
77      colorIndicated = "#678DFF"
78      colorSorted = "#002080"
79      return colorIndicated if element == elementColorCoding["indicated"] else \
80          colorSorted if element <= elementColorCoding["sortedBorder"] and \
81              elementColorCoding["sortedSide"] == "left" or \
82          element >= elementColorCoding["sortedBorder"] and \
83              elementColorCoding["sortedSide"] == "right" \
84          else colorNormal
85
86  def clearElements():
87      for el in frameScreen.find_all():
88          frameScreen.delete(el)
89
90  def updateElements(strNewElements):
91      newElements = int(strNewElements)
92      clearElements()
93
94      global elementHeights
95
96      if newElements == 0:
97          newElements = elements.get()
98      else:
99          elementHeights = list(range(1, newElements + 1))
100         elementColorCoding["indicated"] = -1
101         elementColorCoding["sortedBorder"] = -1
102         elementColorCoding["sortedSide"] = "none"
103         swaps.set(0)
104         comparisons.set(0)
105
106     elWidthUnit = round(800 / newElements, 2)
107     elHeightUnit = round(400 / newElements, 2)
108
109     for i in range(newElements):
110         frameScreen.create_rectangle(elWidthUnit * i, 400, elWidthUnit * (i + 1), 400 - elHeightUnit *
                elementHeights[i], fill = processColor(i), width = 0)
111         #fill = ("#678DFF" if ??? else "#2E52C4")
112
113     frameScreen.update_idletasks()
114
115 elements = tk.IntVar()
116 scaleElements = tk.Scale(frameControls, label = "Sortable Elements", resolution = 10, from_ = 10, to =
        800, length = 200, orient = "horizontal", variable = elements, command = updateElements, font =
        fontNormal)
117 scaleElements.grid(row = 0, column = 0, rowspan = 2, padx = 2, pady = 2)
118 updateElements(0)
```

5.1.3 Delay Sliders, Swap, Shuffle, and Reverse

```
120  sleepTime = tk.DoubleVar()
121  sleepTimeFine = tk.DoubleVar()
122  scaleSleep = tk.Scale(frameControls, label = "Time Delay on Swap (seconds)", resolution = 0.005, from_ =
         0, to = 0.2, length = 200, orient = "horizontal", variable = sleepTime, font = fontNormal)
123  scaleSleepFine = tk.Scale(frameControls, label = "Fine Time Delay on Swap (ms)", resolution = 0.1, from_
         = 0, to = 4.9, length = 200, orient = "horizontal", variable = sleepTimeFine, font = fontNormal)
124  scaleSleep.grid(row = 2, column = 0, rowspan = 2, padx = 2, pady = 2)
125  scaleSleepFine.grid(row = 4, column = 0, rowspan = 2, padx = 2, pady = 2)
126
127  swaps = tk.IntVar()
128  #labelSwaps = tk.Label(frameControls, textvariable = swaps, width = 6, anchor = "e", bd = 2, relief = "
         ridge", padx = 4, pady = 2, font = fontNormal)
129  comparisons = tk.IntVar()
130  #labelComparisons = tk.Label(frameControls, textvariable = comparisons, width = 6, anchor = "e", bd = 2,
         relief = "ridge", padx = 4, pady = 2, font = fontNormal)
131
132  #tk.Label(frameControls, text = "Swaps:", font = fontNormal).grid(row = 5, column = 1, padx = 2, pady =
         2, sticky = "w")
133  #tk.Label(frameControls, text = "Comparisons:", font = fontNormal).grid(row = 5, column = 2, padx = 2,
         pady = 2, sticky = "w")
134  #labelSwaps.grid(row = 5, column = 1, padx = 2, pady = 2, sticky = "e")
135  #labelComparisons.grid(row = 5, column = 2, padx = 2, pady = 2, sticky = "e")
136
137  def swap(elA, elB, doDelay = True):
138      if elA == elB:
139          return None
140
141      elementHeights[elA] += elementHeights[elB]
142      elementHeights[elB] -= elementHeights[elA]
143      elementHeights[elB] *= -1
144      elementHeights[elA] -= elementHeights[elB]
145      if doDelay:
146          updateElements(0)
147          time.sleep(sleepTime.get() + sleepTimeFine.get() / 1000)
148
149  def shuffleElements():
150      rnd.shuffle(elementHeights)
151      elementColorCoding["indicated"] = -1
152      elementColorCoding["sortedBorder"] = -1
153      elementColorCoding["sortedSide"] = "none"
154      swaps.set(0)
155      comparisons.set(0)
156      updateElements(0)
157
158  def reverseElements():
159      elementColorCoding["indicated"] = -1
160      elementColorCoding["sortedBorder"] = -1
161      elementColorCoding["sortedSide"] = "none"
162      swaps.set(0)
163      comparisons.set(0)
164
165      for i in range(len(elementHeights) // 2):
166          swap(i, elements.get() - (1 + i), doDelay = False)
167
168      updateElements(0)
```

5.1.4 Bubble & Insertion Sort

```
170  def bubbleSort():
171      elementColorCoding["sortedSide"] = "right"
172      elementColorCoding["sortedBorder"] = elements.get()
173      swaps.set(0)
174      comparisons.set(0)
175      start = time.time()
176
177      for i in range(elements.get() - 1):
178          localSwaps = 0
179
180          for j in range(elements.get() - i - 1):
181              elementColorCoding["indicated"] = j + 1
182              if elementHeights[j] > elementHeights[j + 1]:
183                  swap(j, j + 1)
184                  localSwaps += 1
185                  swaps.set(swaps.get() + 1)
186
187              comparisons.set(comparisons.get() + 1)
188
189          elementColorCoding["sortedBorder"] = j + 1
190
191          if localSwaps == 0:
192              break
193
194      clockTime(start, time.time(), "BBL")
195
196  def insertionSort():
197      elementColorCoding["sortedSide"] = "left"
198      elementColorCoding["sortedBorder"] = -1
199      swaps.set(0)
200      comparisons.set(0)
201      start = time.time()
202
203      for i in range(1, elements.get()):
204          j = i - 1
205          while j >= 0 and elementHeights[j] > elementHeights[j + 1]:
206              elementColorCoding["indicated"] = j
207              swap(j + 1, j)
208              j -= 1
209              swaps.set(swaps.get() + 1)
210              comparisons.set(comparisons.get() + 1)
211
212          elementColorCoding["sortedBorder"] = i + 1
213
214      clockTime(start, time.time(), "INS")
```

5.1.5 Selection Sort & Merge Functions

```
216  def selectionSort():
217      elementColorCoding["sortedSide"] = "left"
218      elementColorCoding["sortedBorder"] = -1
219      swaps.set(0)
220      comparisons.set(0)
221
222      def minIndex(firstIndex):
223          min = firstIndex
224
225          for i in range(firstIndex, elements.get()):
226              if elementHeights[i] < elementHeights[min]:
227                  min = i
228
229          return min
230
231      start = time.time()
232
233      for i in range(1, elements.get()):
234          m = minIndex(i)
235
236          while m >= i and elementHeights[m] < elementHeights[m - 1]:
237              elementColorCoding["indicated"] = m - 1
238              swap(m, m - 1)
239              m -= 1
240              swaps.set(swaps.get() + 1)
241              comparisons.set(comparisons.get() + 1)
242
243          elementColorCoding["sortedBorder"] = i - 1
244
245      clockTime(start, time.time(), "SLC")
246
247  def merge(baseLeft, lengthLeft, baseRight, lengthRight):
248      localArray = elementHeights[baseLeft : baseLeft + lengthLeft + lengthRight]
249      localLeft = 0
250      localRight = lengthLeft
251
252      for k in range(baseLeft, baseLeft + lengthLeft + lengthRight):
253          elementColorCoding["indicated"] = k
254
255          if localRight == lengthLeft + lengthRight or (localLeft < lengthLeft and localArray[localLeft] <
                  localArray[localRight]):
256              elementHeights[k] = localArray[localLeft]
257              localLeft += 1
258          else:
259              elementHeights[k] = localArray[localRight]
260              localRight += 1
261
262          elementColorCoding["sortedBorder"] = k
263
264          updateElements(0)
265          time.sleep(sleepTime.get() + sleepTimeFine.get() / 1000)
266
267  def mergeInPlace(baseLeft, lengthLeft, baseRight, lengthRight):
268      for i in range(baseRight + lengthRight - 1, baseRight - 1, -1):
269          elementColorCoding["indicated"] = i
270          j = baseLeft + lengthLeft - 1
271
272          while j > baseLeft and elementHeights[j - 1] > elementHeights[i]:
273              swap(j, j - 1, doDelay = True)
274              j -= 1
275
276          if elementHeights[j] > elementHeights[i]:
277              swap(j, i)
278
279      elementColorCoding["sortedBorder"] = baseRight + lengthRight - 1
```

5.1.6 Merge & Heap Sort

```python
281  def mergeSort(base, length, mergeFunc = merge):
282      elementColorCoding["sortedSide"] = "left"
283      elementColorCoding["sortedBorder"] = -1
284
285      if length == elements.get():
286          start = time.time()
287
288      if length > 1:
289          lengthLeft = length // 2
290          lengthRight = length - lengthLeft
291          baseLeft = base
292          baseRight = base + lengthLeft
293
294          mergeSort(baseLeft, lengthLeft, mergeFunc)
295          mergeSort(baseRight, lengthRight, mergeFunc)
296
297          mergeFunc(baseLeft, lengthLeft, baseRight, lengthRight)
298          if length == elements.get():
299              clockTime(start, time.time(), "MGON" if mergeFunc == merge else "MGIP")
```

```python
303  def heapify(head, heapSize):
304      left = 2 * head + 1
305      right = 2 * head + 2
306      largest = head
307
308      if (left < heapSize and elementHeights[left] > elementHeights[head]):
309          largest = left
310      if (right < heapSize and elementHeights[right] > elementHeights[largest]):
311          largest = right
312      if largest != head:
313          elementColorCoding["indicated"] = head
314          swap(largest, head)
315
316  def buildHeap(length):
317      for i in range(length // 2, -1, -1):
318          heapify(i, length)
319
320  def heapSort():
321      elementColorCoding["sortedSide"] = "right"
322      elementColorCoding["sortedBorder"] = elements.get()
323      start = time.time()
324      buildHeap(elements.get())
325
326      for i in range(elements.get() - 1, 0, -1):
327          elementColorCoding["sortedBorder"] = i
328          swap(0, i)
329          buildHeap(i)
330
331      clockTime(start, time.time(), "HEAP")
```

5.1.7 Bogo & Quick Sort

```
333  def bogoSort():
334      if elements.get() > 10:
335          mbx.showwarning("Warning!", "Bogo Sort's time complextity is n-factorial. It cannot in good faith
                    be run on a list larger than 10 elements.")
336          return None
337      else:
338          mbx.showinfo("Notice", "Only the fine time delay will be used in this sorting run.")
339
340      sorted = False
341      start = time.time()
342      while not sorted:
343          rnd.shuffle(elementHeights)
344          updateElements(0)
345          time.sleep(sleepTimeFine.get() / 1000)
346
347          broke = False
348          for i in range(elements.get() - 1):
349              if elementHeights[i] > elementHeights[i + 1]:
350                  broke = True
351                  break
352
353          if not broke:
354              sorted = True
355      clockTime(start, time.time(), "BGO")
356
357  def qsPartition(left, right):
358      elementColorCoding["indicated"] = right
359      pivot=elementHeights[right]
360      i = left
361
362      for j in range(left, right):
363          if elementHeights[j] < pivot:
364              if i != j:
365                  swap(i, j)
366
367              i += 1
368
369      swap(i, right)
370      elementColorCoding["sortedBorder"] = i
371      return i
372
373  def quickSort(left, right):
374      elementColorCoding["sortedSide"] = "left"
375      elementColorCoding["sortedBorder"] = left - 1
376
377      if left == 0 and right == elements.get() - 1:
378          start = time.time()
379
380      if left < right:
381          pivot = qsPartition(left, right)
382          quickSort(left, pivot - 1)
383          quickSort(pivot + 1, right)
384
385      if left == 0 and right == elements.get() - 1:
386          clockTime(start, time.time(), "QCK")
```

5.1.8 Control Buttons

```
388  buttonShuffle = tk.Button(frameControls, text = "Shuffle Elements", bd = 2, width = 20, command =
         shuffleElements, font = fontNormal)
389  buttonReverse = tk.Button(frameControls, text = "Reverse Elements", bd = 2, width = 20, command =
         reverseElements, font = fontNormal)
390  buttonBubble = tk.Button(frameControls, text = "Bubble Sort", bd = 2, width = 20, command = bubbleSort,
         font = fontNormal)
391  buttonInsertion = tk.Button(frameControls, text = "Insertion Sort", bd = 2, width = 20, command =
         insertionSort, font = fontNormal)
392  buttonSelection = tk.Button(frameControls, text = "Selection Sort", bd = 2, width = 20, command =
         selectionSort, font = fontNormal)
393  buttonMerge = tk.Button(frameControls, text = "Merge Sort: O(N) Space", bd = 2, width = 20, command =
         lambda: mergeSort(0, elements.get()), font = fontNormal)
394  buttonMergeIP = tk.Button(frameControls, text = "Merge Sort: In Place", bd = 2, width = 20, command =
         lambda: mergeSort(0, elements.get(), mergeInPlace), font = fontNormal)
395  buttonHeap = tk.Button(frameControls, text = "Heap Sort", bd = 2, width = 20, command = heapSort, font =
         fontNormal)
396  buttonQuick = tk.Button(frameControls, text = "Quick Sort", bd = 2, width = 20, command = lambda:
         quickSort(0, elements.get() - 1), font = fontNormal)
397  buttonBogo = tk.Button(frameControls, text = "Bogo Sort", bd = 2, width = 20, command = bogoSort, font =
         fontNormal)
398
399  buttonShuffle.grid(row = 0, column = 1, padx = 2, pady = 2)
400  buttonReverse.grid(row = 0, column = 2, padx = 2, pady = 2)
401
402  buttonBogo.grid(row = 1, column = 1, padx = 2, pady = 2)
403  buttonBubble.grid(row = 1, column = 2, padx = 2, pady = 2)
404
405  buttonInsertion.grid(row = 2, column = 1, padx = 2, pady = 2)
406  buttonSelection.grid(row = 2, column = 2, padx = 2, pady = 2)
407
408  buttonMerge.grid(row = 3, column = 1, padx = 2, pady = 2)
409  buttonMergeIP.grid(row = 3, column = 2, padx = 2, pady = 2)
410
411  buttonHeap.grid(row = 4, column = 1, padx = 2, pady = 2)
412  buttonQuick.grid(row = 4, column = 2, padx = 2, pady = 2)
413
414  base.mainloop()
```

6 Puzzles

6.1 Eight Balls

You are given eight balls. They are visaully identical, but one weighs more than the other seven, which themselves are all of equal weight. You have a mechanical balance as well, but can only make two measurements to determine the heavy ball. What measurements do you make?

Let the balls be represened by the set $S = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8,\}$. Let a measurement be the output of the function $f(A, B) = \max(\sum a_i, \sum b_i)$, where A and B are subsets of S, and a_i and b_i are elements of A and B respectivly, and $\sum$ is an operation to compute the total weight of all balls in a set. If the set of balls in A weigh more than those in B, then $f(A, B) = A$ and vice versa. If the sets are of equal weight, then $f(A, B) = \emptyset$.

To find which ball in S is heaviest, we first observe $f(A = \{b_1, b_2, b_3\}, B = \{b_4, b_5, b_6\})$. If result is A, then we know the heaviest ball is in A, at which point we measure any two balls from A, for example $f(A = \{b_1\}, B = \{b_3\})$. Here, a result of A or B will reveal the heaviest ball, and a tie will indicate that b_2, the ball not weighed again, is heaviest. This method follows for if $B = \{b_4, b_5, b_6\}$ is the result in the first measurement as well.

However, if the first measurement yields a result of $\emptyset$, then the first six balls are not the heaviest, leaving b_7 and b_8. Simply checking $f(A = \{b_7\}, B = \{b_8\})$ at this point will reveal the heaviest ball.

6.2 Prisoners and Hats Dilemma

Three prisoners are given three of four hats to put on. Of the four hats, two are red, and two are blue. One hat is not put on and hidden from the prisoners. The prisoners A, B, and C are lined up such that A cannot see anyone, B can see A, and C can see both B and A. None of the prisoners can see their own hats, nor can they talk to the other prisoners. They each must guess the color of their hat. If any one of them are correct, they will all be released, and if any one of them are wrong, they will all be sent to solitary confinement for life. How can all of the prisoners go free?

First, consider C, who can see two of the hats. If the hats are the same color, both blue or red, then C can say the opposite of what they see. However, if C sees both one red and one blue hat, then they cannot be sure what their own hat is. At this point, if C doesn't jump at guessing first, then B should recognize that they have the color opposite of A, and say so. In either case, as long as the prisoners have thought this thru, A has to do nothing, B has to wait to see if C answers, and C has to just observe the hats of A and B.