Programming Dictionary

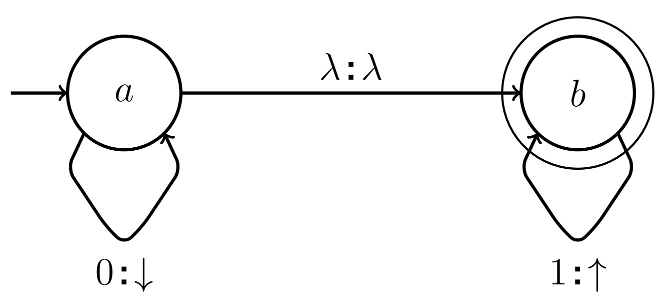
By Charles Cook

Begun on Dec. 6th, 2018

Contents

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
$\downarrow$	Push (onto the stack)
$\uparrow$	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*

```
void swap(a, b) {
        temp = a;
        a = b;
        b = temp;
}

void swapNoTemp(a, b) {
        a += b; // a = a + b
        b -= a; // b = b - (a + b) = -a
        b *= -1; // b = a
        a -= b; // a = a + b - a = b
}
```

2.1.2 *swaptest.c*

```
#include <stdio.h>
void swap(int *a, int *b) {
        *a += *b;
        *b -= *a;
        *b *= -1;
        *a -= *b;
}

int main() {
        int x, y, z;
        x = 10;
        y = 13;
        z = 2;

        swap(&x, &y);
        swap(&x, &z);
        swap(&y, &z);
        printf("x: &d\ny: &d\nz: &d\n", x, y, z);

        return 0;
}
```

2.1.3 *swaptest.cpp*

```cpp
#include <iostream>
using namespace std;

void swap(int *a, int *b) {
        *a += *b;
        *b -= *a;
        *b *= -1;
        *a -= *b;
}

int main() {
        int x, y;
        x = 13;
        y = 29;

        cout << x << ", " << y << "\n";
        swap(x, y);
        cout << x << ", " << y << "\n";

        return 0;
}
```

2.2 Reverse an array with no extra space (pseudocode)

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

2.3 Reverse a doubly linked list (pseudocode)

```cpp
struct Node {
        int data;
        struct Node *next;
        struct Node *prev;
};

void reverseDLL(head, tail) {
        struct Node tempH = head;
        struct Node tempT = tail;

        while (tempH -> next != tempT -> prev && tempH != tempT) {
                swap(tempH -> data, tempT -> data);
        }
}
```

2.4 Reverse a doubly linked list recursively (pseudocode)

```
void reverseDLL_Recursive(head, tail) {
        if (head -> next != tail -> prev && head != tail) {
                swap(head -> data, tail -> data);
                reverseDLL_Recursive(head -> next, tail -> prev);
        }
}
```

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:

```
head:
 [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:

```
 pp
  |
  V
head:
 [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).**

```
 pp -+
     |
head:V
 [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)**

```
 pp --------+
            |
head:       V
 [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)`

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

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

`[9] -> next = *pp;`

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

`*pp = [9];`

```
 pp ---------------+
                   |
head:              V
 [2] -> [4] -> [7] -+ [10] -> [13]
                   /  ^
                  V   |
                 [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 $O(n^2)$.

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*

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

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

```
| ?- ordered([1,2,4,8,16]).
      1    1  Call: ordered([1,2,4,8,16]) ?
      2    2  Call: 1<2 ?
      2    2  Exit: 1<2 ?
      3    2  Call: ordered([2,4,8,16]) ?
      4    3  Call: 2<4 ?
      4    3  Exit: 2<4 ?
      5    3  Call: ordered([4,8,16]) ?
      6    4  Call: 4<8 ?
      6    4  Exit: 4<8 ?
      7    4  Call: ordered([8,16]) ?
      8    5  Call: 8<16 ?
      8    5  Exit: 8<16 ?
      9    5  Call: ordered([16]) ?
      9    5  Exit: ordered([16]) ?
      7    4  Exit: ordered([8,16]) ?
      5    3  Exit: ordered([4,8,16]) ?
      3    2  Exit: ordered([2,4,8,16]) ?
      1    1  Exit: ordered([1,2,4,8,16]) ?

true ?

yes
{trace}
| ?-
```

2.10 Labelled Path (with cost) in Prolog

2.10.1 ASCII-art chart of my path

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

2.10.2 *charliesPath.pl*

```prolog
link(a, b, 1).
link(b, c, 3).
link(c, d, 11).
link(c, e, 5).
link(d, f, 7).
link(e, g, 13).
link(e, h, 19).

path(Origin, Destination, Distance) :-
        link(Origin, Destination, Distance)
.
path(Origin, Destination, TotalDistance) :-
        link(Origin, Intermediate, DistanceA),
        path(Intermediate, Destination, DistanceB),
        TotalDistance is DistanceA + DistanceB
.
```

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)

```
int numberOfNodes(head) {
	int n = 1;
	temp = head;

	while (temp -> next != NULL) {
		n += 1;
		temp = temp -> next;
	}

	return n;
}
```

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

```
int numberOfNodesQuick(head) {
	int n = 1;
	temp = head;

	while (temp -> next != NULL && temp -> next -> next != NULL) {
		n += 2;
		temp = temp -> next -> next;
	}

	if (temp -> next != NULL) {
		n += 1;
	}

	return n;
}
```

3.3 Return middle node of linked list (pseudocode)

```
node middleNode(head, int length) {
	temp = head;

	for (int i = 0; i < length / 2; i++) {
		temp = temp -> next;
	}

	return temp;
}
```

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

It is possible to run a reachability function on a directed graph / linked list map that runs in polynomial time. For this, we need an array, identified as $marked$, and a stack, identified as $reached$. Starting from the root node of travel, push this node onto $reached$. Then, begin a loop which will continue until $reached$ is empty, or until the desired node to reach gets pushed to $reached$. In each pass, if the top element in $reached$ has children, pop it off and append its address a.k.a. indentifier to $marked$. Then, push all of its child nodes onto $reached$, except those already in $marked$. This condition prevents a cyclic map from running the algorithm forever.

As the loop proceeds, eventually all ultimate child nodes will be pushed to the stack. If the top node at the start of a pass has no children, then pop it off and start a new pass of the loop. If only ultimate children are in the stack, and none of them are the desired node, then $reached$ will be emptied, and a $false$ value can be returned. If the node-to-be-reached is ever pushed to $reached$ however, then a $true$ value can be returned. With the $marked$ array, no node will ever be pushed to $reached$ twice, which makes this a P-class algorithm.

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

```
concat([], L, L).
concat([X|L1], L2, [X|L3]) :- concat(L1, L2, L3).
reverselist([], []).
reverselist([Head|Tail], Reversed) :-
        reverselist(Tail, ReversedTail),
        concat(ReversedTail, [Head], Reversed)
.
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.

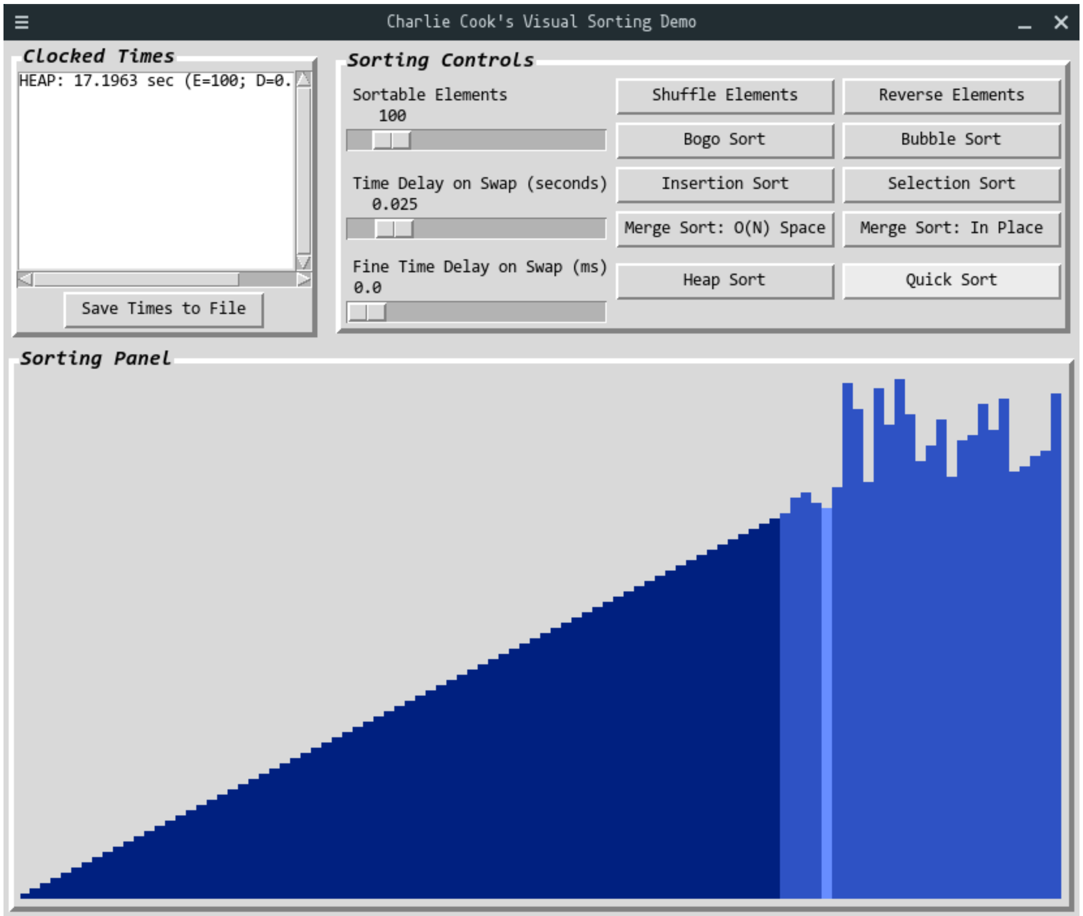
4.2.2 reverselist()

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

```python
import tkinter as tk
import tkinter.simpledialog as sdg
import tkinter.messagebox as mbx
import tkinter.font as tkf
import tkinter.filedialog as fdg
import random as rnd
import platform as pt
import time

if pt.system() == "Linux":
    id = "~"
elif pt.system() == "Windows":
    id = "C:\\"

base = tk.Tk()
base.title("Charlie Cook's Visual Sorting Demo")
base.resizable(False, False)

# Fonts
fontSmall = tkf.Font(family = "Consolas", size = 8)
fontNormal = tkf.Font(family = "Consolas", size = 10)
fontNormUnd = tkf.Font(family = "Consolas", size = 10, underline = 1)
fontNormBold = tkf.Font(family = "Consolas", size = 10, weight = "bold")
fontLarge = tkf.Font(family = "Consolas", size = 12)
fontLargeBold = tkf.Font(family = "Consolas", size = 12, weight = "bold")
fontSubtitle = tkf.Font(family = "Consolas", size = 12, slant = "italic")
fontTitle = tkf.Font(family = "Consolas", size = 12, weight = "bold", slant = "italic")
fontBigTitle = tkf.Font(family = "Consolas", size = 16, weight = "bold", slant = "italic")
fontName = tkf.Font(family = "Consolas", size = 16, weight = "bold")

frameTimes = tk.LabelFrame(base, text = "Clocked Times", bd = 4, relief = "raised", font = fontTitle)
frameTimes.grid(row = 0, column = 0, padx = 4, pady = 4, sticky = "n")

listboxTimes = tk.Listbox(frameTimes, width = 30, height = 10,  font = fontNormal)
listboxTimes.grid(row = 0, column = 0)

scrollVTimes = tk.Scrollbar(frameTimes, orient = "vertical", command = listboxTimes.yview)
scrollHTimes = tk.Scrollbar(frameTimes, orient = "horizontal", command = listboxTimes.xview)
listboxTimes.configure(yscrollcommand = scrollVTimes.set, xscrollcommand = scrollHTimes.set)
scrollVTimes.grid(row = 0, column = 1, sticky = "ns")
scrollHTimes.grid(row = 1, column = 0, columnspan = 2, sticky = "we")

def clockTime(start, end, algo):
    listboxTimes.insert("end", algo + ": " + str(round(end - start, 4)) + " sec (E=" + \
        str(elements.get()) + "; D=" + str(sleepTime.get() + sleepTimeFine.get() / 1000) + \
        " sec)" \
    )

def saveTimes():
    savefile = fdg.asksaveasfilename(parent = base, title = "Select or Enter a file to save to:",
        initialdir = id, filetypes = (("Text Files","*.txt"),("All Files","*.*")))

    if type(savefile) is str and len(savefile) > 0:
        file = open(savefile, "w")

        for clock in listboxTimes.get(0, "end"):
            file.write(clock + "\n")

        file.close()

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

```
frameControls = tk.LabelFrame(base, text = "Sorting Controls", bd = 4, relief = "raised", font = fontTitle)
frameControls.grid(row = 0, column = 1, padx = 4, pady = 4)

frameMain = tk.LabelFrame(base, text="Sorting Panel", bd = 4, relief = "raised", font = fontTitle)
frameMain.grid(row = 1, column = 0, columnspan = 2, padx = 4, pady = 4)

frameScreen = tk.Canvas(frameMain, width = 800, height = 400)
frameScreen.pack(padx = 4, pady = 4)

elementHeights = list(range(1, 11))
elementColorCoding = {"indicated": 0, "sortedBorder": -1, "sortedSide": "none"}

def processColor(element):
    colorNormal = "#2E52C4"
    colorIndicated = "#678DFF"
    colorSorted = "#002080"
    return colorIndicated if element == elementColorCoding["indicated"] else \
        colorSorted if element <= elementColorCoding["sortedBorder"] and \
            elementColorCoding["sortedSide"] == "left" or \
        element >= elementColorCoding["sortedBorder"] and \
            elementColorCoding["sortedSide"] == "right" \
        else colorNormal

def clearElements():
    for el in frameScreen.find_all():
        frameScreen.delete(el)

def updateElements(strNewElements):
    newElements = int(strNewElements)
    clearElements()

    global elementHeights

    if newElements == 0:
        newElements = elements.get()
    else:
        elementHeights = list(range(1, newElements + 1))
        elementColorCoding["indicated"] = -1
        elementColorCoding["sortedBorder"] = -1
        elementColorCoding["sortedSide"] = "none"
        swaps.set(0)
        comparisons.set(0)

    elWidthUnit = round(800 / newElements, 2)
    elHeightUnit = round(400 / newElements, 2)

    for i in range(newElements):
        frameScreen.create_rectangle(elWidthUnit * i, 400, elWidthUnit * (i + 1), 400 - elHeightUnit * elementHeights[i], fill = processColor(i), width = 0)
        #fill = ("#678DFF" if ??? else "#2E52C4")

    frameScreen.update_idletasks()

elements = tk.IntVar()
scaleElements = tk.Scale(frameControls, label = "Sortable Elements", resolution = 10, from_ = 10, to = 800, length = 200, orient = "horizontal", variable = elements, command = updateElements, font = fontNormal)
scaleElements.grid(row = 0, column = 0, rowspan = 2, padx = 2, pady = 2)
updateElements(0)
```

5.1.3 Delay Sliders, Swap, Shuffle, and Reverse

```python
sleepTime = tk.DoubleVar()
sleepTimeFine = tk.DoubleVar()
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)
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)
scaleSleep.grid(row = 2, column = 0, rowspan = 2, padx = 2, pady = 2)
scaleSleepFine.grid(row = 4, column = 0, rowspan = 2, padx = 2, pady = 2)

swaps = tk.IntVar()
#labelSwaps = tk.Label(frameControls, textvariable = swaps, width = 6, anchor = "e", bd = 2, relief = "ridge", padx = 4, pady = 2, font = fontNormal)
comparisons = tk.IntVar()
#labelComparisons = tk.Label(frameControls, textvariable = comparisons, width = 6, anchor = "e", bd = 2, relief = "ridge", padx = 4, pady = 2, font = fontNormal)

#tk.Label(frameControls, text = "Swaps:", font = fontNormal).grid(row = 5, column = 1, padx = 2, pady = 2, sticky = "w")
#tk.Label(frameControls, text = "Comparisons:", font = fontNormal).grid(row = 5, column = 2, padx = 2, pady = 2, sticky = "w")
#labelSwaps.grid(row = 5, column = 1, padx = 2, pady = 2, sticky = "e")
#labelComparisons.grid(row = 5, column = 2, padx = 2, pady = 2, sticky = "e")

def swap(elA, elB, doDelay = True):
    if elA == elB:
        return None

    elementHeights[elA] += elementHeights[elB]
    elementHeights[elB] -= elementHeights[elA]
    elementHeights[elB] *= -1
    elementHeights[elA] -= elementHeights[elB]
    if doDelay:
        updateElements(0)
        time.sleep(sleepTime.get() + sleepTimeFine.get() / 1000)

def shuffleElements():
    rnd.shuffle(elementHeights)
    elementColorCoding["indicated"] = -1
    elementColorCoding["sortedBorder"] = -1
    elementColorCoding["sortedSide"] = "none"
    swaps.set(0)
    comparisons.set(0)
    updateElements(0)

def reverseElements():
    elementColorCoding["indicated"] = -1
    elementColorCoding["sortedBorder"] = -1
    elementColorCoding["sortedSide"] = "none"
    swaps.set(0)
    comparisons.set(0)

    for i in range(len(elementHeights) // 2):
        swap(i, elements.get() - (1 + i), doDelay = False)

    updateElements(0)
```

5.1.4 Bubble & Insertion Sort

```
def bubbleSort():
    elementColorCoding["sortedSide"] = "right"
    elementColorCoding["sortedBorder"] = elements.get()
    swaps.set(0)
    comparisons.set(0)
    start = time.time()

    for i in range(elements.get() - 1):
        localSwaps = 0

        for j in range(elements.get() - i - 1):
            elementColorCoding["indicated"] = j + 1
            if elementHeights[j] > elementHeights[j + 1]:
                swap(j, j + 1)
                localSwaps += 1
                swaps.set(swaps.get() + 1)

            comparisons.set(comparisons.get() + 1)

        elementColorCoding["sortedBorder"] = j + 1

        if localSwaps == 0:
            break

    clockTime(start, time.time(), "BBL")

def insertionSort():
    elementColorCoding["sortedSide"] = "left"
    elementColorCoding["sortedBorder"] = -1
    swaps.set(0)
    comparisons.set(0)
    start = time.time()

    for i in range(1, elements.get()):
        j = i - 1
        while j >= 0 and elementHeights[j] > elementHeights[j + 1]:
            elementColorCoding["indicated"] = j
            swap(j + 1, j)
            j -= 1
            swaps.set(swaps.get() + 1)
            comparisons.set(comparisons.get() + 1)

        elementColorCoding["sortedBorder"] = i + 1

    clockTime(start, time.time(), "INS")
```

5.1.5 Selection Sort & Merge Functions

```
def selectionSort():
    elementColorCoding["sortedSide"] = "left"
    elementColorCoding["sortedBorder"] = -1
    swaps.set(0)
    comparisons.set(0)

    def minIndex(firstIndex):
        min = firstIndex

        for i in range(firstIndex, elements.get()):
            if elementHeights[i] < elementHeights[min]:
                min = i

        return min

    start = time.time()

    for i in range(1, elements.get()):
        m = minIndex(i)

        while m >= i and elementHeights[m] < elementHeights[m - 1]:
            elementColorCoding["indicated"] = m - 1
            swap(m, m - 1)
            m -= 1
            swaps.set(swaps.get() + 1)
            comparisons.set(comparisons.get() + 1)

        elementColorCoding["sortedBorder"] = i - 1

    clockTime(start, time.time(), "SLC")

def merge(baseLeft, lengthLeft, baseRight, lengthRight):
    localArray = elementHeights[baseLeft : baseLeft + lengthLeft + lengthRight]
    localLeft = 0
    localRight = lengthLeft

    for k in range(baseLeft, baseLeft + lengthLeft + lengthRight):
        elementColorCoding["indicated"] = k

        if localRight == lengthLeft + lengthRight or (localLeft < lengthLeft and localArray[localLeft] <
            localArray[localRight]):
            elementHeights[k] = localArray[localLeft]
            localLeft += 1
        else:
            elementHeights[k] = localArray[localRight]
            localRight += 1

        elementColorCoding["sortedBorder"] = k

        updateElements(0)
        time.sleep(sleepTime.get() + sleepTimeFine.get() / 1000)

def mergeInPlace(baseLeft, lengthLeft, baseRight, lengthRight):
    for i in range(baseRight + lengthRight - 1, baseRight - 1, -1):
        elementColorCoding["indicated"] = i
        j = baseLeft + lengthLeft - 1

        while j > baseLeft and elementHeights[j - 1] > elementHeights[i]:
            swap(j, j - 1, doDelay = True)
            j -= 1

        if elementHeights[j] > elementHeights[i]:
            swap(j, i)

    elementColorCoding["sortedBorder"] = baseRight + lengthRight - 1
```

5.1.6 Merge & Heap Sort

```python
def mergeSort(base, length, mergeFunc = merge):
    elementColorCoding["sortedSide"] = "left"
    elementColorCoding["sortedBorder"] = -1

    if length == elements.get():
        start = time.time()

    if length > 1:
        lengthLeft = length // 2
        lengthRight = length - lengthLeft
        baseLeft = base
        baseRight = base + lengthLeft

        mergeSort(baseLeft, lengthLeft, mergeFunc)
        mergeSort(baseRight, lengthRight, mergeFunc)

        mergeFunc(baseLeft, lengthLeft, baseRight, lengthRight)
        if length == elements.get():
            clockTime(start, time.time(), "MGON" if mergeFunc == merge else "MGIP")
```

```python
def heapify(head, heapSize):
    left = 2 * head + 1
    right = 2 * head + 2
    largest = head

    if (left < heapSize and elementHeights[left] > elementHeights[head]):
        largest = left
    if (right < heapSize and elementHeights[right] > elementHeights[largest]):
        largest = right
    if largest != head:
        elementColorCoding["indicated"] = head
        swap(largest, head)

def buildHeap(length):
    for i in range(length // 2, -1, -1):
        heapify(i, length)

def heapSort():
    elementColorCoding["sortedSide"] = "right"
    elementColorCoding["sortedBorder"] = elements.get()
    start = time.time()
    buildHeap(elements.get())

    for i in range(elements.get() - 1, 0, -1):
        elementColorCoding["sortedBorder"] = i
        swap(0, i)
        buildHeap(i)

    clockTime(start, time.time(), "HEAP")
```

5.1.7 Bogo & Quick Sort

```
def bogoSort():
    if elements.get() > 10:
        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.")
        return None
    else:
        mbx.showinfo("Notice", "Only the fine time delay will be used in this sorting run.")

    sorted = False
    start = time.time()
    while not sorted:
        rnd.shuffle(elementHeights)
        updateElements(0)
        time.sleep(sleepTimeFine.get() / 1000)

        broke = False
        for i in range(elements.get() - 1):
            if elementHeights[i] > elementHeights[i + 1]:
                broke = True
                break

        if not broke:
            sorted = True
    clockTime(start, time.time(), "BGO")

def qsPartition(left, right):
    elementColorCoding["indicated"] = right
    pivot=elementHeights[right]
    i = left

    for j in range(left, right):
        if elementHeights[j] < pivot:
            if i != j:
                swap(i, j)

            i += 1

    swap(i, right)
    elementColorCoding["sortedBorder"] = i
    return i

def quickSort(left, right):
    elementColorCoding["sortedSide"] = "left"
    elementColorCoding["sortedBorder"] = left - 1

    if left == 0 and right == elements.get() - 1:
        start = time.time()

    if left < right:
        pivot = qsPartition(left, right)
        quickSort(left, pivot - 1)
        quickSort(pivot + 1, right)

    if left == 0 and right == elements.get() - 1:
        clockTime(start, time.time(), "QCK")
```

5.1.8 Control Buttons

```
buttonShuffle = tk.Button(frameControls, text = "Shuffle Elements", bd = 2, width = 20, command = shuffleElements, font = fontNormal)
buttonReverse = tk.Button(frameControls, text = "Reverse Elements", bd = 2, width = 20, command = reverseElements, font = fontNormal)
buttonBubble = tk.Button(frameControls, text = "Bubble Sort", bd = 2, width = 20, command = bubbleSort, font = fontNormal)
buttonInsertion = tk.Button(frameControls, text = "Insertion Sort", bd = 2, width = 20, command = insertionSort, font = fontNormal)
buttonSelection = tk.Button(frameControls, text = "Selection Sort", bd = 2, width = 20, command = selectionSort, font = fontNormal)
buttonMerge = tk.Button(frameControls, text = "Merge Sort: O(N) Space", bd = 2, width = 20, command = lambda: mergeSort(0, elements.get()), font = fontNormal)
buttonMergeIP = tk.Button(frameControls, text = "Merge Sort: In Place", bd = 2, width = 20, command = lambda: mergeSort(0, elements.get(), mergeInPlace), font = fontNormal)
buttonHeap = tk.Button(frameControls, text = "Heap Sort", bd = 2, width = 20, command = heapSort, font = fontNormal)
buttonQuick = tk.Button(frameControls, text = "Quick Sort", bd = 2, width = 20, command = lambda: quickSort(0, elements.get() - 1), font = fontNormal)
buttonBogo = tk.Button(frameControls, text = "Bogo Sort", bd = 2, width = 20, command = bogoSort, font = fontNormal)

buttonShuffle.grid(row = 0, column = 1, padx = 2, pady = 2)
buttonReverse.grid(row = 0, column = 2, padx = 2, pady = 2)

buttonBogo.grid(row = 1, column = 1, padx = 2, pady = 2)
buttonBubble.grid(row = 1, column = 2, padx = 2, pady = 2)

buttonInsertion.grid(row = 2, column = 1, padx = 2, pady = 2)
buttonSelection.grid(row = 2, column = 2, padx = 2, pady = 2)

buttonMerge.grid(row = 3, column = 1, padx = 2, pady = 2)
buttonMergeIP.grid(row = 3, column = 2, padx = 2, pady = 2)

buttonHeap.grid(row = 4, column = 1, padx = 2, pady = 2)
buttonQuick.grid(row = 4, column = 2, padx = 2, pady = 2)

base.mainloop()
```

6 Puzzles

6.1 Eight Balls

You are given eight balls. They are visually 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.