Data Structures and Sorting

Assignment 1

COSC262

Dion Woolley

University of Canterbury

10th April 2014

Contents

[1. Abstract 2](#_Toc384967333)

[2. Problem 1, Data Structures 3](#_Toc384967334)

[2.1 BST Tree insertion 3](#_Toc384967335)

[2.2 AVL Tree Insert 4](#_Toc384967336)

[2.3 AVL Tree Delete root twice 6](#_Toc384967337)

[2.4 2-3 Tree Insert 6](#_Toc384967338)

[3. Problem 2, Computer experiments 8](#_Toc384967339)

[3.1 Heapsort, Quicksort, Mergesort, Radix-10 sort 8](#_Toc384967340)

[3.2 Double Quicksort 11](#_Toc384967341)

[3.3 General Radix-r sort 12](#_Toc384967342)

[3.4 Quick-radix sort 14](#_Toc384967343)

[3.5 Discussion of pervious sorting experiments. 15](#_Toc384967344)

[4. Appendices 17](#_Toc384967345)

# Abstract

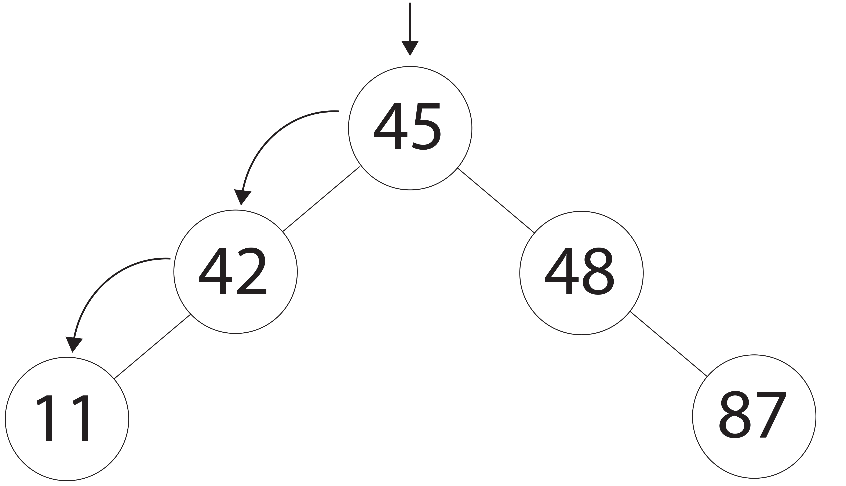
This report compares different “fast” sorting algorithms, Heapsort, Quicksort, Mergesort, General Radix-r sort, including Radix-10 sort and a Quick-radix sort hybrid which uses a divide and conquer method with Radix base 2.

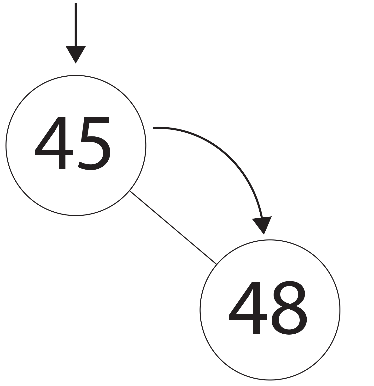
Python 3.3 has been used for the code in this report.

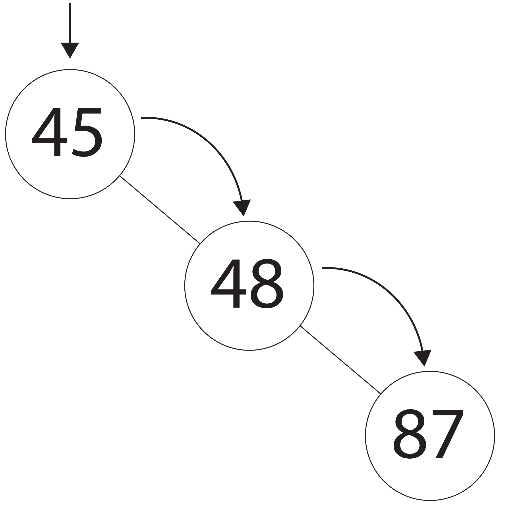
# Problem 1, Data Structures

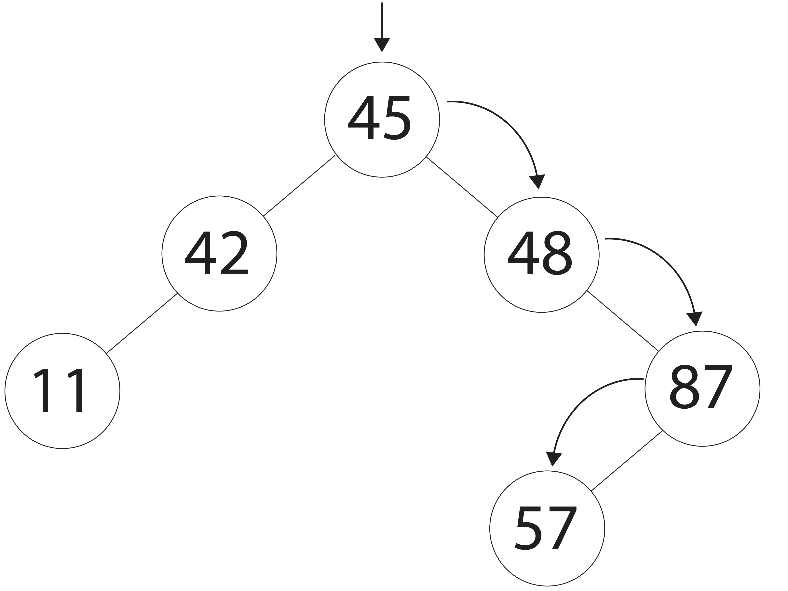
List of data used = [45, 48, 87, 42, 11, 57, 32, 89, 17, 81]

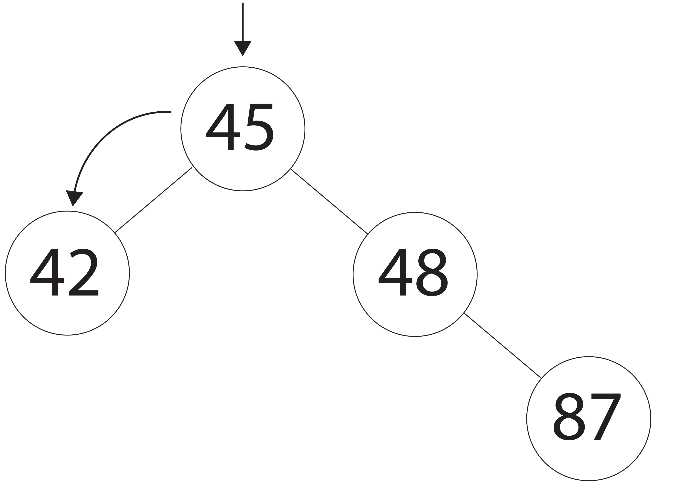
### BST Tree insertion

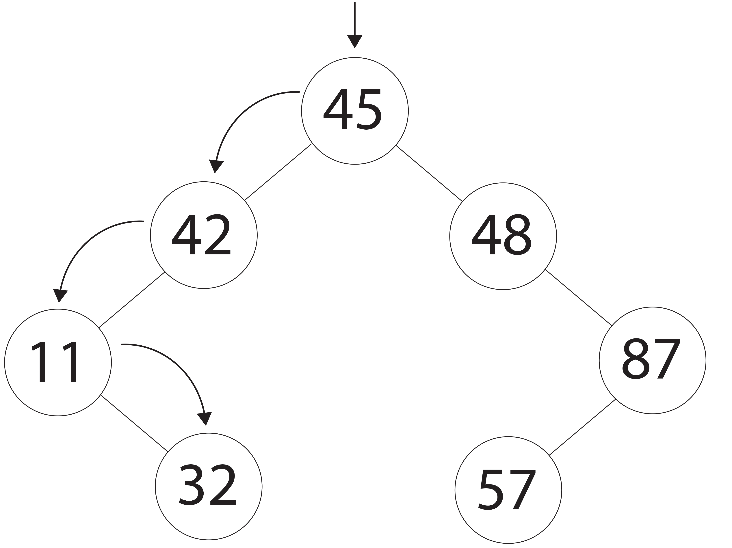
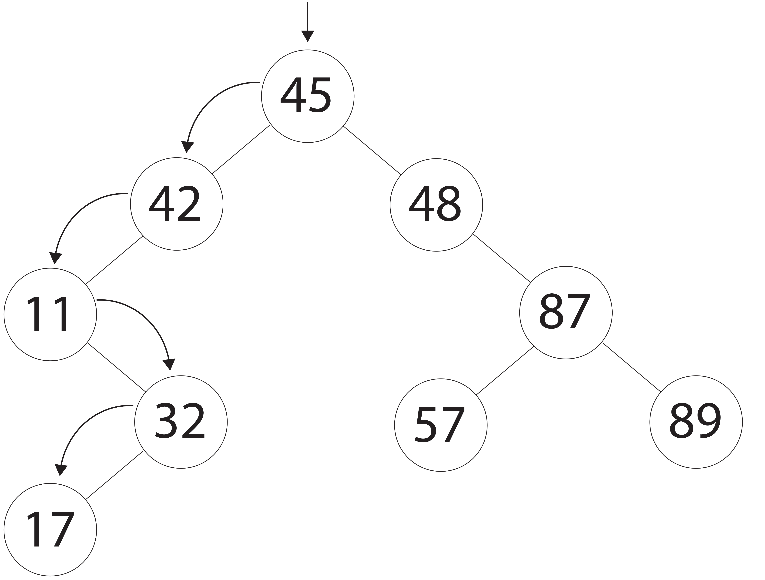


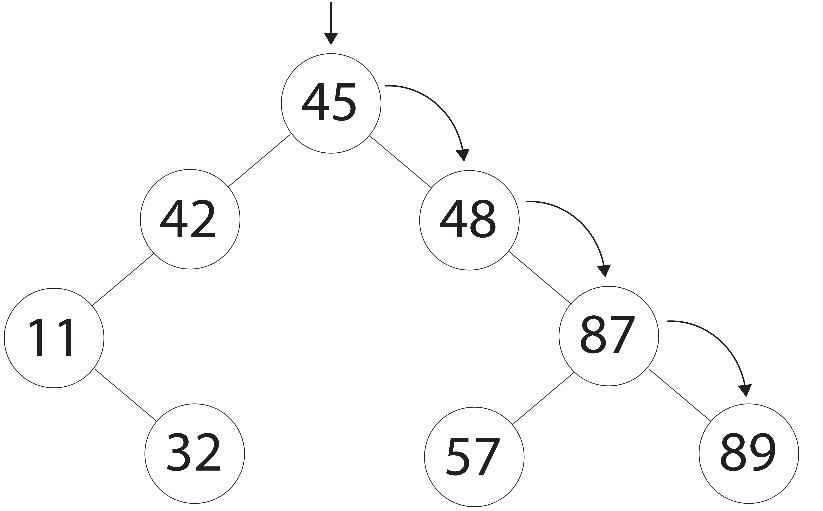
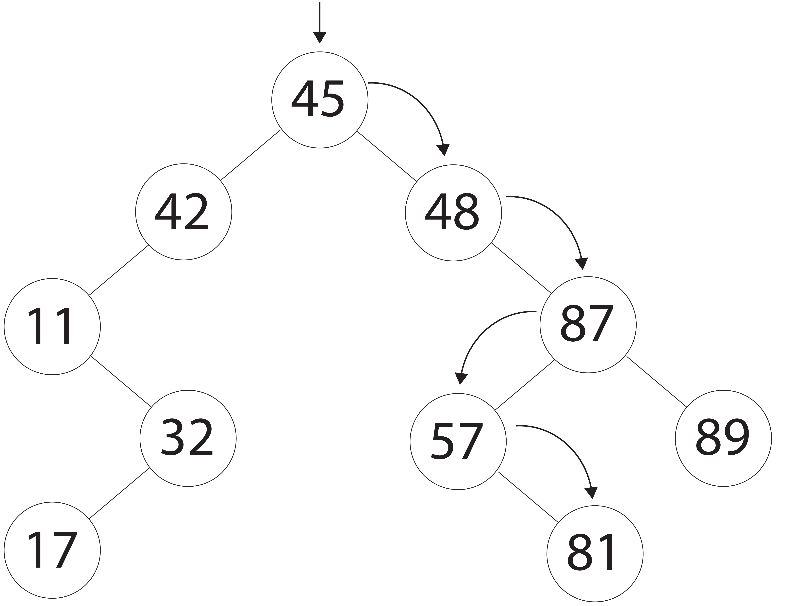




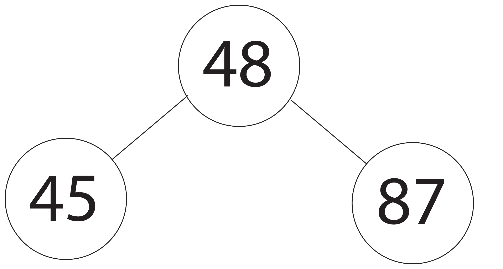


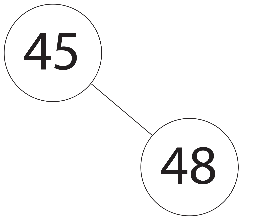


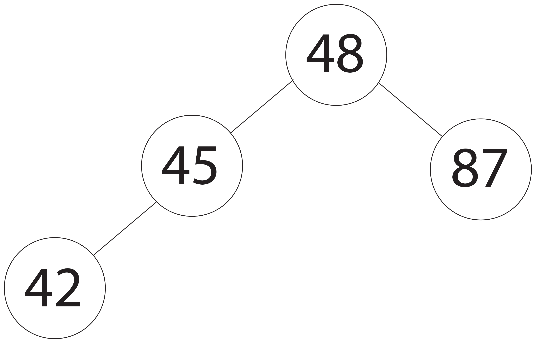


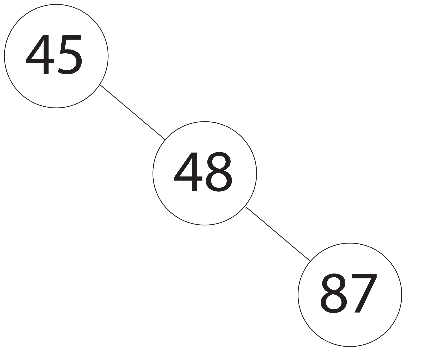


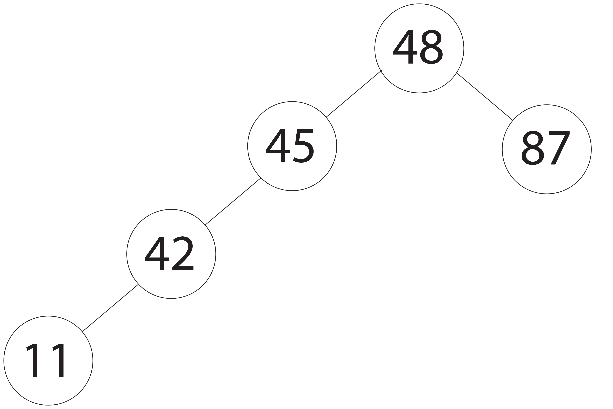
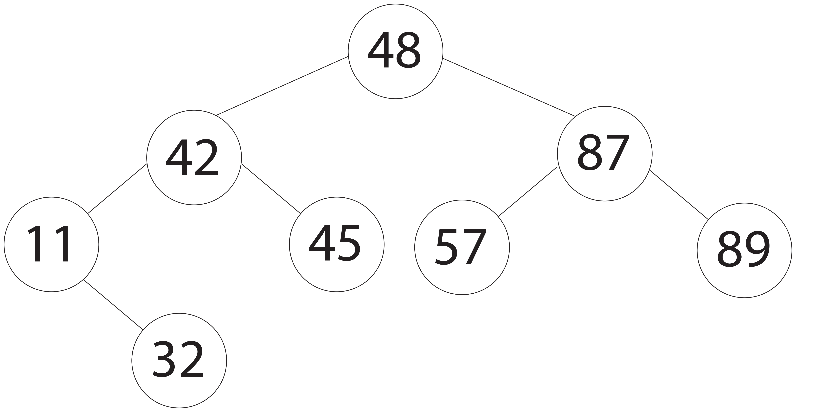
### AVL Tree Insert

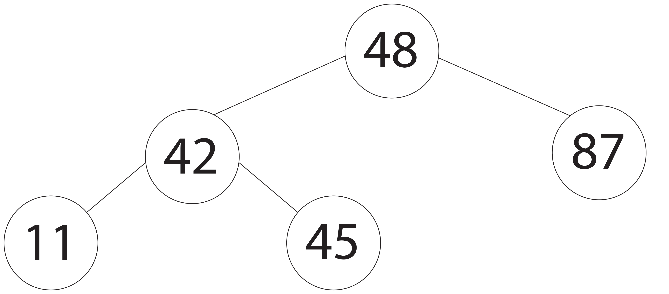
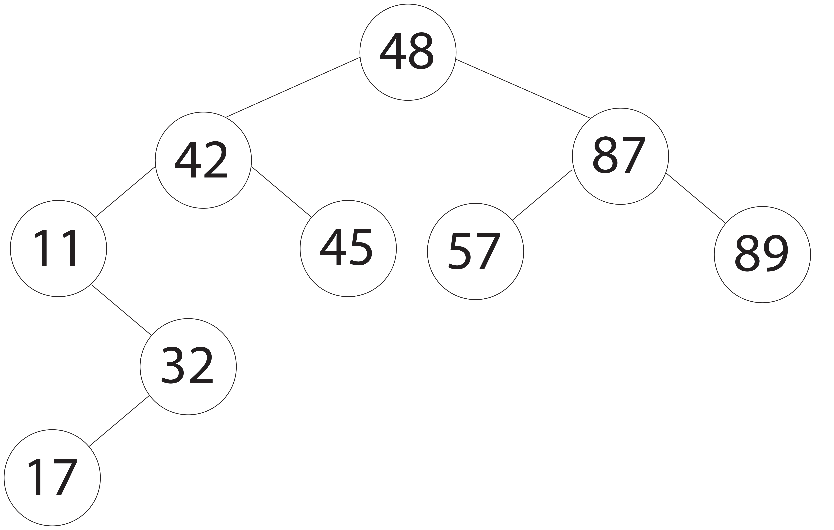


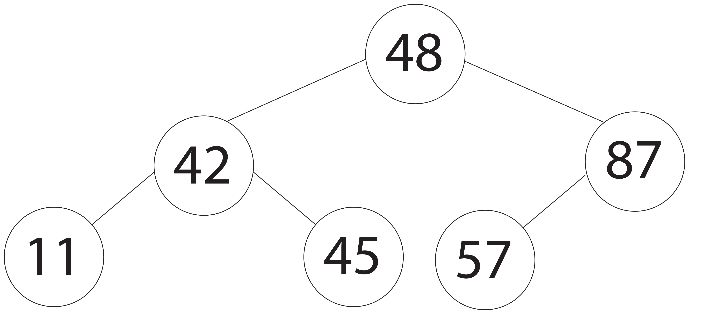


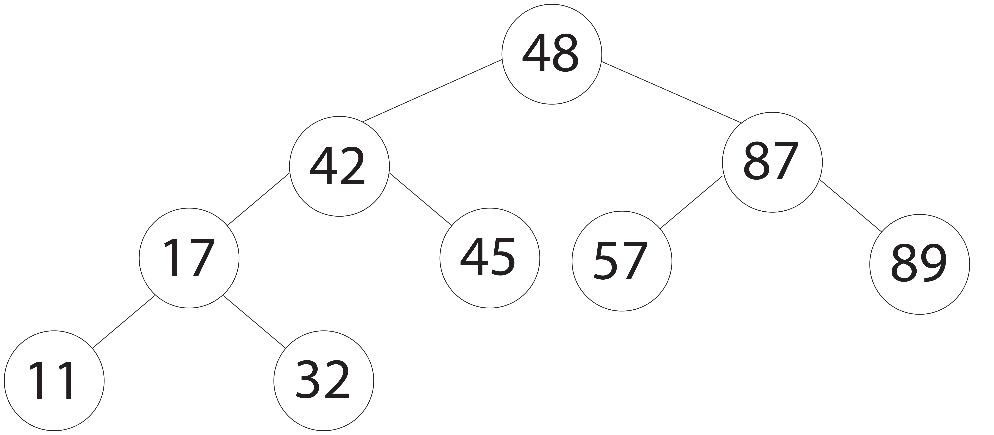


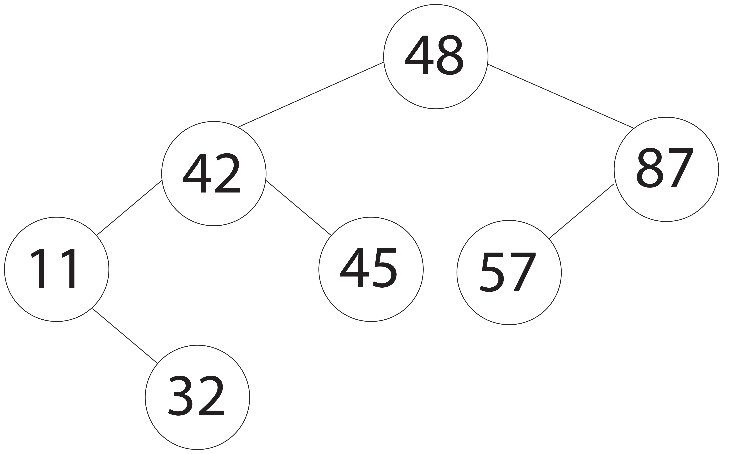


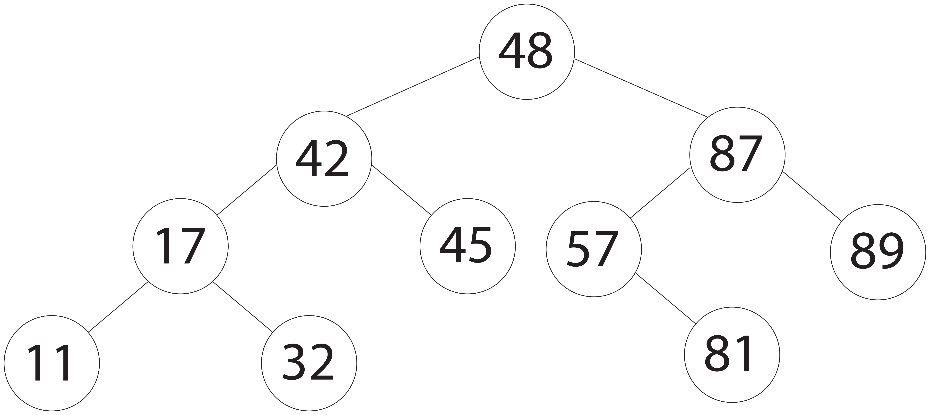






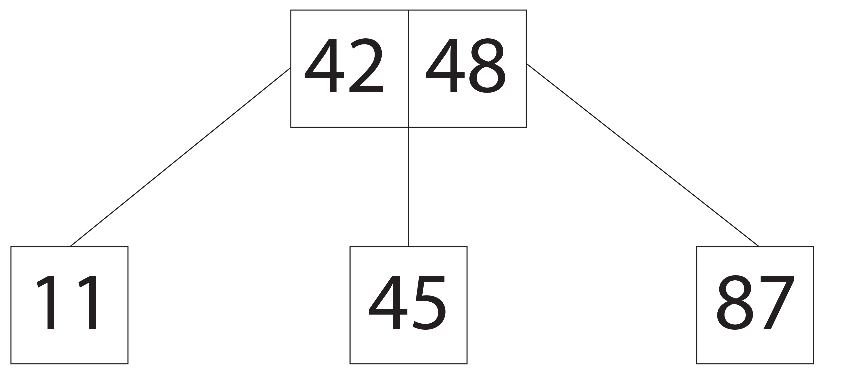
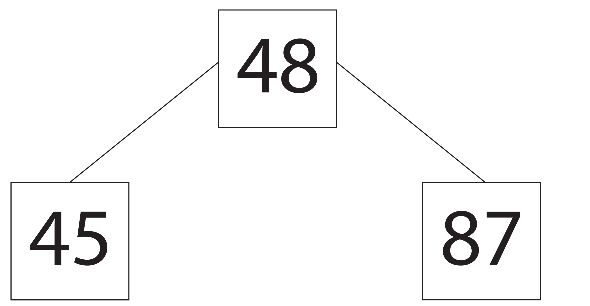


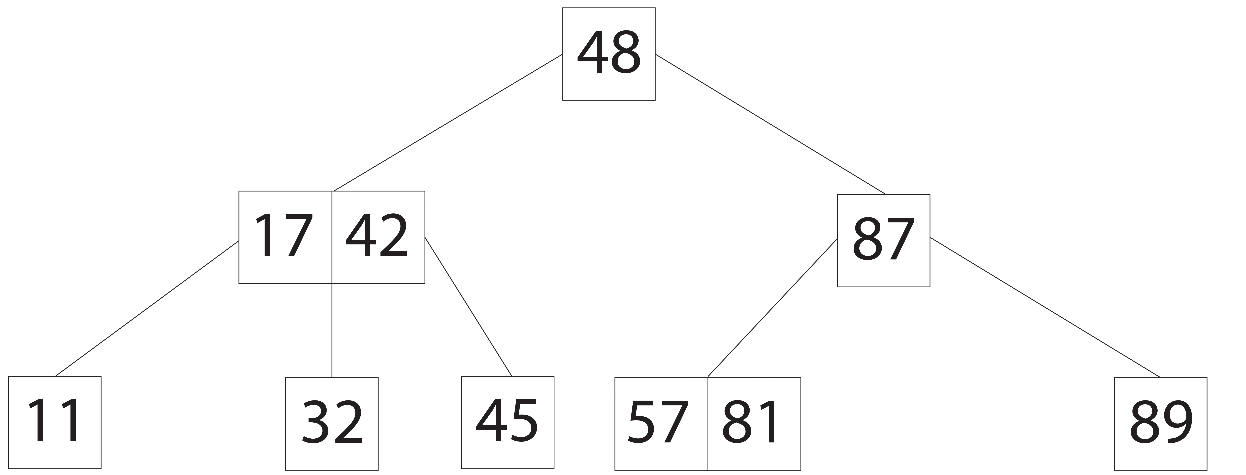
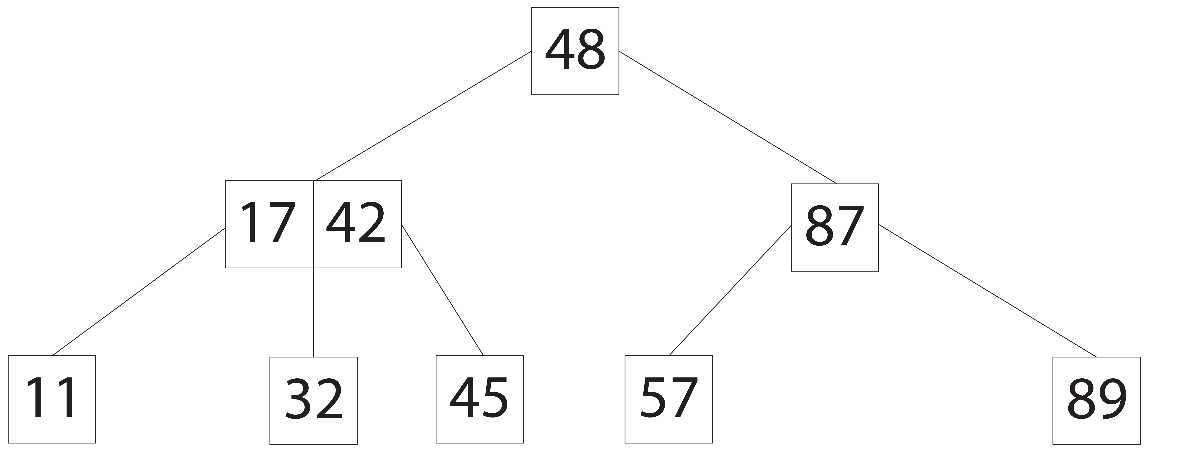
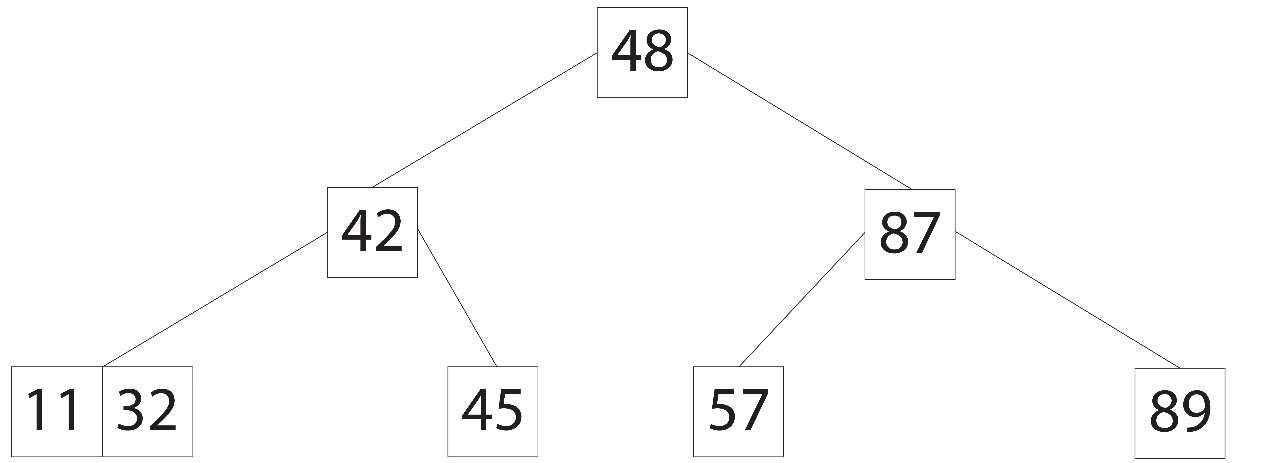
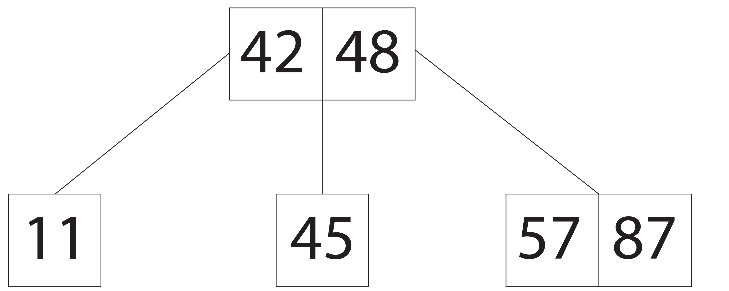
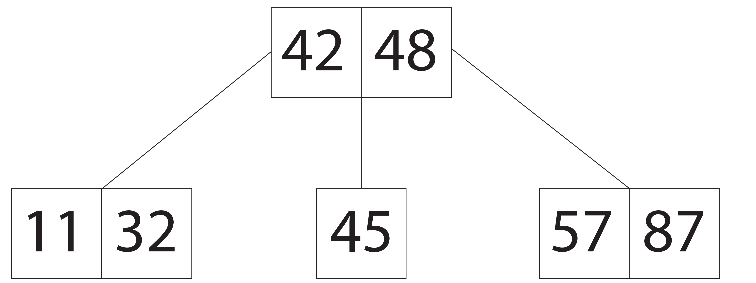




### AVL Tree Delete root twice

### 2-3 Tree Insert





# Problem 2, Computer experiments

The size of the list of unsorted randomised integers used for the experiment ranged from 1,000 – 100,000. The minimum random integer generated is 1. The maximum generated integer is proportional to the size of the list, where the list size = n, the maximum generated integer = n // 5. This was implemented to give a better representation of real world data, as setting the maximum value to a set value for all list sizes would cause unrealistic results for quicksort, as a larger list would have more duplicate numbers, which could be partially sorted. This can result in a decrease in Quicksort’s performance as Quicksort’s worst performance is O(N2) when sorting an already sorted list. The Quicksort used in this experiment does not use the median of three method to generate its pivot, and if it did it would reduce Quicksort’s worst case of O(N2). The average of ten trials was calculated for the performance of each of the sorting algorithms for every size of the list n.

Note: not all data recorded is shown in the tables of data, as there would be too much to display in this report. All data however is present in the graphs.

### Heapsort, Quicksort, Mergesort, Radix-10 sort

Note: Radix-10 sort does not have key comparisons that can be measured.

Table 1. Number of comparisons for Heapsort, Mergesort, and Quicksort.





Figure 1. Graph showing the number of key comparisons between Heapsort, Quicksort and Mergesort.



Table 2. CPU computation time for Heapsort, Mergesort, Quicksort, Radix-10 sort and Quick-radix sort

Figure 2. Graph showing the CPU computation time between Heapsort, Quicksort, Mergesort, Radix-10 sort and Quick-radix sort.

### Double Quicksort

Note: the size of list n could only be measured up 9000, as anything over this size would overflow the stack in Python.



Table 3. Number of comparisons for Quicksort and Double Quicksort.



Figure 3. Graph Number of comparisons for Quicksort and Double Quicksort.



Table 4. CPU computation time for Quicksort and Double Quicksort.



Figure 4. Graph CPU computation time for Quicksort and Double Quicksort.

### General Radix-r sort

Note: the r values measured were 2, 3, 4, 5, 10 …… 100, in steps of 5 from 10 onwards through to and including 100.

Table 5. CPU computation time for general Radix-r sort.





Figure 5. Graph showing the CPU computation time of general Radix-r sort.

### Quick-radix sort

Table 6. CPU computation time for Quick-radix sort and its related sorting methods.



Figure 6. Graph showing the CPU computation time of Quick-radix sort and its related sorting methods.

### Discussion of pervious sorting experiments.

#### Discussion of 3.1, Heapsort, Quicksort, Mergesort, Radix-10 sort

In 3.1 comparing Heapsort, Quicksort, Mergesort and Radx-10 sort algorithms, the Graph Figure 2 shows that Raidx-10 sort performs the fastest followed by Quicksort, then Heapsort and Mergesort, which have almost the exact same CPU computation time. These results are expected as the order of complexity for Heapsort, Mergesort and Quicksort is O(N log(N) + K), but it is known that Heapsort and Mergesort have a higher constant K than Quicksort, so Quicksort is naturally faster. Radix-10 sort, the fastest algorithm, has an order of complexity of O(KN). From the graph data we can assume that the K constant for Radix-10 sort is relatively small. In the graph Figure 1 we can see that Mergesort has a considerably lower number of comparisons than Heapsort, even they both have a very similar computation time as shown in graph Figure 2. This is because Mergesort requires O(2N) amount of memory as it makes a copy of the data to be sorted, which means that Mergesort requires double the memory access processing done by the CPU, when compared to Heapsort which uses O(N) amount of memory.

#### Discussion of 3.2, Double Quicksort

In 3.2 two iterations of the Quicksort algorithm are run on a list of unordered random integers. The first iteration of Quicksort sorts the list, the second iteration attempts to sort an already sorted list. The Quicksort implemented in this experiment uses the first item of the partitioned list as the pivot. Since the second iteration of Quicksort is attempting to sort an already sorted list, this will produce Quicksort’s worst case order of complexity of O(N2) as shown by the graphs Figures 3 and 4. To remedy this, the Quicksort median of three method could be implemented, with the median of three considerably reducing Quicksort’s worst case when sorting an already sorted list. Other options would be to use Introsort (Quicksort, Heapsort hybrid). Introsorts best and worst case is always O(N log(N)).

#### Discussion of 3.3, General Radix-r sort

In 3.3 Radixsort of general r was run to determine the best r value to achieve the fastest performing Radixsort. From graph Figure 5 we see that r values of 30, 50 and 100 perform the fastest, with r value of 100 performing slightly better. Although this is an improvement over an r value of 10, in a real world application the performance difference most likely would not be noticeable. An r value of 2 was found to perform the worst. This result is as expected as the Radixsort algorithm has to run a lot more iterations when working in lower base numbers, compared to base 10.

#### Discussion of 3.4, Quick-radix sort

In 3.4 a Quick-radix hybrid sort was implemented, which essentially uses the divide and conquer method on a radix sort of base 2. It was found that the hybrid performed faster than Quicksort but not quite as fast as Radix-10 sort, as shown in graph Figure 6. In comparison Quicksort and Radix-2 sort performed relatively similar to each other. From the graph Figure 6 it would seem that Quick-radix sort could be an alternative fast sorting algorithm than the other traditional O(N log(N)) sorting algorithms. Although there is not enough data collected to determine the real world application of Quick-radix sort as its order of worst case cannot be extrapolated, more data collection and testing would need to be conducted to determine if it is viable as a reliable sorting algorithm in a complex enterprise environment. More testing of this algorithm would also need to be conducted in a lower level programming language such as C, to determine if the results found using Python are comparable to using C, as Python’s methods of handling data may skew the results. The only obvious drawback to this sorting method is that you are required to know the maximum length in base 2 of the maximum integer you could possibly sort. This is fine if you know the max value, however if you did not, in which case you may have to set the value to an extremely high integer or you would have to check every integer at least once to determine the highest value, then convert that integer into binary and determine its length in binary. Setting an extremely high value could mean that the Quick-radix sort may conduct more iterations of its internal recursion algorithm then is necessary to sort the list. This is a major reason why this sorting method may not by suitable for real world use.

# Appendices

File: file\_output.py

\_\_author\_\_ = 'Dion'

import experiment

def avg(list\_of\_elements):

return sum(list\_of\_elements) / len(list\_of\_elements)

def main():

file = open("results.txt", "w")

for n in range(1000, 9001, 1000):

heapsort\_results, quicksort\_results, doublequicksort\_results, mergesort\_results, radix2sort\_results, \

radix3sort\_results, radix4sort\_results, quickradixsort\_results, radixsort\_results, radix\_max\_base = \

experiment.main(n)

file.write("\n Results for " + str(n) + " list size \n CPU time \n")

file.write(str(avg(heapsort\_results[0])) + '\n')

file.write(str(avg(quicksort\_results[0])) + '\n')

file.write(str(avg(doublequicksort\_results[0])) + '\n')

file.write(str(avg(mergesort\_results[0])) + '\n')

file.write(str(avg(quickradixsort\_results)) + '\n')

file.write(str(avg(radix2sort\_results)) + '\n')

file.write(str(avg(radix3sort\_results)) + '\n')

file.write(str(avg(radix4sort\_results)) + '\n')

for i in range(0, (radix\_max\_base // 5)):

file.write(str(avg(radixsort\_results[i])) + '\n')

file.write("\n Comparisons \n")

file.write(str(int(avg(heapsort\_results[1]))) + '\n')

file.write(str(int(avg(quicksort\_results[1]))) + '\n')

file.write(str(int(avg(doublequicksort\_results[1]))) + '\n')

file.write(str(int(avg(mergesort\_results[1]))) + '\n')

print("finished " + str(n))

for n in range(10000, 100001, 5000):

heapsort\_results, quicksort\_results, doublequicksort\_results, mergesort\_results, radix2sort\_results, \

radix3sort\_results, radix4sort\_results, quickradixsort\_results, radixsort\_results, radix\_max\_base = \

experiment.main(n)

file.write("\n Results for " + str(n) + " list size \n CPU time \n")

file.write(str(avg(heapsort\_results[0])) + '\n')

file.write(str(avg(quicksort\_results[0])) + '\n')

file.write(str(avg(doublequicksort\_results[0])) + '\n')

file.write(str(avg(mergesort\_results[0])) + '\n')

file.write(str(avg(quickradixsort\_results)) + '\n')

file.write(str(avg(radix2sort\_results)) + '\n')

file.write(str(avg(radix3sort\_results)) + '\n')

file.write(str(avg(radix4sort\_results)) + '\n')

for i in range(0, (radix\_max\_base // 5)):

file.write(str(avg(radixsort\_results[i])) + '\n')

file.write("\n Comparisons \n")

file.write(str(int(avg(heapsort\_results[1]))) + '\n')

file.write(str(int(avg(quicksort\_results[1]))) + '\n')

file.write(str(int(avg(doublequicksort\_results[1]))) + '\n')

file.write(str(int(avg(mergesort\_results[1]))) + '\n')

print("finished " + str(n))

file.close()

main()

File: experiment.py

# This is a template for running experiments.

# Each program is run once. You should run each program five times or

# more and take the average time.

import random

from time import clock

import heapsort

import quicksort

import mergesort

import quickradixsort

import radix\_r\_sort

import change\_base

def main(n):

# main program

min = 1

max = n // 5

radix\_max\_base = 100

heapsort\_results = [[], []]

quicksort\_results = [[], []]

doublequicksort\_results = [[], []]

mergesort\_results = [[], []]

radix2sort\_results = []

radix3sort\_results = []

radix4sort\_results = []

quickradixsort\_results = []

radixsort\_results = [[] for \_ in range(radix\_max\_base // 5)]

for i in range(10):

data = []

for i in range(0, n + 1):

data += [random.randint(min, max)]

## Experiment on heapsort ##

a = []

for i in range(0, n + 1):

a += [data[i]]

heapt, heapc = heapsort.main(n, a)

heapsort\_results[0].append(heapt)

heapsort\_results[1].append(heapc)

## Experiment on quicksort ##

a = []

for i in range(0, n + 1):

a += [data[i]]

quickt, quickc, dquickt, dquickc = quicksort.main(n, a)

quicksort\_results[0].append(quickt)

quicksort\_results[1].append(quickc)

doublequicksort\_results[0].append(dquickt)

doublequicksort\_results[1].append(dquickc)

## Experiment on mergesort ##

a = []

for i in range(0, n + 1):

a += [data[i]]

merget, mergec = mergesort.main(n, a)

mergesort\_results[0].append(merget)

mergesort\_results[1].append(mergec)

##Experiment on Radix-2 sort##

base = 2

a = []

for i in range(0, n + 1):

a += [data[i]]

radix2t = clock()

x = bin(max)[2:]

sorted\_a = radix\_r\_sort.main(a, base, len(x))

radix2sort\_results.append(clock() - radix2t)

assert sorted\_a[1:] == sorted(sorted\_a[1:]), "List not sorted, error!!!!!!!!!!"

##Experiment on Radix-3 sort##

base = 3

a = []

for i in range(0, n + 1):

a += [data[i]]

radix3t = clock()

num\_max\_len = len(list(change\_base.change\_base(max, base)))

sorted\_a = radix\_r\_sort.main(a, base, num\_max\_len)

radix3sort\_results.append(clock() - radix3t)

assert sorted\_a[1:] == sorted(sorted\_a[1:]), "List not sorted, error!!!!!!!!!!"

##Experiment on Radix-4 sort##

base = 4

a = []

for i in range(0, n + 1):

a += [data[i]]

radix4t = clock()

num\_max\_len = len(list(change\_base.change\_base(max, base)))

sorted\_a = radix\_r\_sort.main(a, base, num\_max\_len)

radix4sort\_results.append(clock() - radix4t)

assert sorted\_a[1:] == sorted(sorted\_a[1:]), "List not sorted, error!!!!!!!!!!"

##Experiment on Radix-r sort with stepping of 5##

counter = 0

for i in range(5, radix\_max\_base + 1, 5):

base = i

a = []

for i in range(0, n + 1):

a += [data[i]]

radixt = clock()

num\_max\_len = len(list(change\_base.change\_base(max, base)))

sorted\_a = radix\_r\_sort.main(a, base, num\_max\_len)

radixsort\_results[counter].append(clock() - radixt)

assert sorted\_a[1:] == sorted(sorted\_a[1:]), "List not sorted, error!!!!!!!!!!"

counter += 1

##Experiment on Quick-Radix sort##

a = []

for i in range(0, n + 1):

a += [data[i]]

quickradixt = clock()

x = bin(max)[2:]

length = len(x)

sorted\_a = quickradixsort.main(a, length)

quickradixsort\_results.append(clock() - quickradixt)

assert sorted\_a[1:] == sorted(sorted\_a[1:]), "List not sorted, error!!!!!!!!!!"

return heapsort\_results, quicksort\_results, doublequicksort\_results, mergesort\_results, radix2sort\_results,\

radix3sort\_results, radix4sort\_results, quickradixsort\_results, radixsort\_results, radix\_max\_base

File: heapsort.py

# This is heapsort

# sort() sorts array a in descending order

# sift(p,q) heapifies array a from position p to q

# heap is a max-heap, that is, maximum at the root

from time import clock

c = 0

def swap(i, j, a): # This swaps a[i] and a[j]

w = a[i]

a[i] = a[j]

a[j] = w

def siftup(p, q, a): # This is to heapify a when a[p] is wrong

global c

y = a[p] # a[p] is saved to y

j = p

k = 2 \* p

isheap = False

while k <= q and not isheap:

z = a[k]

if k < q:

c += 1

if z < a[k + 1]: # Choose the smaller child

k += 1

z = a[k]

c += 1

if y >= z:

isheap = True

else:

a[j] = z

j = k

k = 2 \* j

a[j] = y # y settles down at position j

def build\_heap(n, a):

for i in reversed(range(1, int(n / 2 + 1))):

siftup(i, n, a)

def sort(n, a):

build\_heap(n, a)

for i in reversed(range(2, n + 1)):

swap(1, i, a) # swap a[1] and a[i]

siftup(1, i - 1, a)

# {main program}

def main(n, a):

b = c

t = clock()

sort(n, a)

t = clock() - t

assert a[1:] == sorted(a[1:]), "List not sorted, error!!!!!!!!!!"

return t, c – b

File: quicksort.py

# This is quicksort

# sort(left,right) sorts array a from position left to right

# partition(left,right) partitions array a from position left to right

# with pivot x=a[left], and returns m such that after partition

# a[left .. m-1] <= x=a[m] <= a[m+1 .. right]

from time import clock

c = 0

import threading

import sys

threading.stack\_size(134217728)

sys.setrecursionlimit(10 \*\* 8)

def sort(left, right, a):

if left < right:

m = partition(left, right, a)

sort(left, m - 1, a)

sort(m + 1, right, a)

def partition(left, right, a): # x is the pivot

global c

x = a[left]

i = left

j = right + 1 # i goes right, j goes left

while i < j:

j -= 1

if i == j:

break

c += 1

while a[j] >= x: # while a[j]>=x,

j -= 1 # j goes left

if i == j:

break

c += 1

if i == j:

break

a[i] = a[j]

i += 1

if i == j:

break

c += 1

while a[i] <= x: # while a[i]<=x,

i += 1 # i goes right

if i == j:

break

c += 1

if i == j:

break

a[j] = a[i] # a[i] is copied to a[j]

a[i] = x # pivot x settles down at i

return i

def main(n, a):

b = c

# {main program}

t = clock()

sort(1, n, a)

t = clock() - t

assert a[1:] == sorted(a[1:]), "List not sorted, error!!!!!!!!!!"

qc = c - b

if n <= 9000:

d = c

t2 = clock()

sort(1, n, a)

t2 = clock() - t2

assert a[1:] == sorted(a[1:]), "List not sorted, error!!!!!!!!!!"

dqc = c - d

else:

t2 = 0

dqc = 0

return t, qc, t2, dqc

File: mergesort.py

# This is mergesort

# mergesort(p,q) sorts array a from position p to q

from time import clock

c = 0

def mergesort(p, q, a, b):

if p < q:

m = int((p + q) / 2)

mergesort(p, m, a, b)

mergesort(m + 1, q, a, b)

merge(p, m + 1, q + 1, a, b)

# merge(p,m,q) merges array a from position p to m

# and position m+1 to q

def merge(p, r, q, a, b):

global c # c is comparison counter

i = p

j = r

k = p

while i < r and j < q:

c += 1

if a[i] <= a[j]:

b[k] = a[i]

i += 1

else:

b[k] = a[j]

j += 1

k += 1

while i < r:

b[k] = a[i]

i += 1

k += 1

while j < q:

b[k] = a[j]

j += 1

k += 1

for k in range(p, q):

a[k] = b[k]

def main(n, a):

d = c

#{ main program }

b = []

for i in range(0, n + 1):

b += [0]

t = clock()

mergesort(1, n, a, b)

t = clock() - t

assert a[1:] == sorted(a[1:]), "List not sorted, error!!!!!!!!!!"

return t, c – d

File: radix\_r\_sort.py

\_\_author\_\_ = 'Dion'

def main(a, base, max\_len):

buckets = []

for i in range(0, base):

buckets.append([])

i = 0

while i < max\_len:

for k in a[1:]:

x = (k // base \*\* i) % base

buckets[x].append(k)

a = a[:1]

for j in buckets:

r, j[:] = j[:], []

a.extend(r)

i += 1

return a

File: quickradixsort.py

\_\_author\_\_ = 'Dion'

def sort(a, max\_len, current\_len, result):

if current\_len < 0:

return result.extend(a)

else:

left = []

right = []

for i in a:

if (i >> current\_len) % 2 == 0:

left.append(i)

else:

right.append(i)

sort(left, max\_len, current\_len - 1, result)

sort(right, max\_len, current\_len - 1, result)

return result

def main(a, max\_len):

# {main program}

result = sort(a[1:], max\_len, current\_len=max\_len, result=[])

return result

File: change\_base.py

\_\_author\_\_ = 'Dion'

import \_collections

def change\_base(num, base):

new\_num = \_collections.deque([])

current = num

while current != 0:

remainder = current % base

new\_num.appendleft(remainder)

current = current // base

return new\_num

def check\_length(num, base, max\_length):

new\_num = change\_base(num, base)

if len(new\_num) < max\_length:

n = max\_length - len(new\_num)

for \_ in range(0, n):

new\_num.appendleft(0)

return list(new\_num)

def main(list\_of\_nums, base, max\_num, quickradix):

new\_num\_list = []

if quickradix is True:

max\_length = len(change\_base(max\_num, base))

for i in list\_of\_nums:

new\_num\_list.append(check\_length(i, base, max\_length))

return new\_num\_list, max\_length

else:

max\_length = len(change\_base(max\_num, base))

for i in list\_of\_nums:

new\_num\_list.append(list(change\_base(i, base)))

return new\_num\_list, max\_length

def revert\_base(list\_of\_nums, base):

org\_nums = []

for i in list\_of\_nums:

org\_num = 0

power = 0

for j in i:

if j != 0:

org\_num += j \* (base \*\* power)

power += 1

else:

power += 1

org\_nums.append(org\_num)

return org\_nums