CSE 361 HW-1

Daniel Dyla

September 15, 2016

Contents

1 Zero Sum Implementations **1**
- 1.1 Brute Force Approach . . . 1
  - 1.1.1 Correctness . . . 2
  - 1.1.2 Runtime . . . 2
- 1.2 Smarter Approach . . . 2
  - 1.2.1 Correctness . . . 3
  - 1.2.2 Runtime . . . 3
- 1.3 Smartest Approach . . . 3
  - 1.3.1 Correctess . . . 3
  - 1.3.2 Runtime . . . 4

2 Compare Asymptotic Runtimes **4**
- 2.1 Random Arrays . . . 4
- 2.2 Guaranteed Worst Case . . . 4

3 Graph Comparison **5**

1 Zero Sum Implementations

1.1 Brute Force Approach

```
def zerosum_brute(a):
  for i in range(len(a)-1):
    for j in range(i+1, len(a)):
      if a[i] == -a[j]:
        return((i,j))

  return (-1, -1)
```

1.1.1 Correctness

The algorithm terminates because the for loops have finite bounds. The outer loop considers every element except the last. The inner loop checks each element against each element to its right. If the two considered elements sum to zero we return early. If every pair is considered without sucess we return failure.

1.1.2 Runtime

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} n-1-(i+1)-1 \tag{1}$$

$$= \sum_{i=0}^{n-2} n-i-1 \tag{2}$$

$$= \sum_{i=0}^{n-2} n-1-\sum_{i=0}^{n-2} i \tag{3}$$

$$= (n-2-0+1)(n-1) - \frac{(n-1)(n-2)}{2} \tag{4}$$

$$= \Theta(n^2) \tag{5}$$

1.2 Smarter Approach

```
def zerosum_smarter(a):
  a = sorted(a)
  l = 0
  r = len(a)-1
  while (l < r):
    s = a[l]+a[r]
    if s == 0:
      return a[l], a[r]
    elif s < 0:
      l += 1
    elif s > 0:
      r -= 1
  return None,None
```

1.2.1 Correctness

The algorithm terminates because s can only be positive, negative, or zero. If s is 0 the algorithm returns. If s is negative l increments, if s is positive r decrements. Eventually l must be equal to r.

If the algorithm terminates early, a match is found and returned. If s is positive, the element at r is more positive than the element at l is negative. Moving l will only make s further from zero so r must be moved. Conversely, if s is negative, the element at l is more negative than the element at r is positive and r must move. If the pointers point at the same element then no other matches need to be checked.

1.2.2 Runtime

The sort runs in $\Theta(n \lg n)$. In the worst case, $l = r$.

$$\begin{aligned}
\Theta(n \lg n) + (l - 0) + ((n - 1) - r) &= \Theta(n \lg n) + l + ((n - 1) - l) && (6)\\
&= \Theta(n \lg n) + l + n - 1 - l && (7)\\
&= \Theta(n \lg n) + n - 1 && (8)\\
&= \Theta(n \lg n) + \Theta(n) && (9)\\
&= \Theta(n \lg n) && (10)
\end{aligned}$$

1.3 Smartest Approach

In the worst case the whole array must be read so no solution can be better than $\Theta(n)$

```
def zerosum_smartest(a):
  s = {}
  for i,e in enumerate(a):
    if -e in s:
      return i, s[-e]
    else:
      s[e] = i
  return None
```

1.3.1 Correctess

The algorithm terminates because the for loop has a defined bound.

The algorithm checks the set of "seen elements" for its compliment. If the complement is found, the program terminates with a match. If the entire array has been "seen" and no matches were found, there are no matches.

1.3.2 Runtime

The for loop checks, at worst, each element once. There are n elements in the array. The insertion and checking of the hash table is a constant time operation.

$$\sum_{i=0}^{n-1} 1 = (n-1) - 0 + 1 \tag{11}$$

$$= \Theta(n) \tag{12}$$

2 Compare Asymptotic Runtimes

2.1 Random Arrays

All algorithms as written above were tested in a python 2 environment with random int arrays with elements from -1M to +1M of size n from 10 to 1000000. The runtimes are shown below. Runtimes are in milliseconds.

Table 1: Runtimes in milliseconds for random arrays of -1M to +1M

n	10	100	1000	10000	100000	1000000
brute	.008	.490	35.619	209.345	219.296	248.246
smarter	.003	.039	.358	3.064	41.668	677.971
smartest	.002	.022	.176	.332	.367	.468

Interestingly, the "smarter" algorithm actually does the worst by far for lists with many elements. I suspect this is because the brute algorithm found matches and returned early, where the "smarter" algorithm had to completely sort the list before beginning processing it.

Here is the result of another test using linear values from 10000 to 100000 and plotted. Data is not the same as the table.

2.2 Guaranteed Worst Case

In order to test worst case (no matches) I generated the same array sizes but used only positive numbers. This guarantees that every incarnation of the algorithm runs in its worst possible time. Runtimes are in milliseconds.

Table 2: Runtimes in milliseconds for random arrays of 0 to +1M

n	10	100	1000	10000	100000	1000000
brute	.018	.455	42.74	3952.48		
smarter	.004	.033	.428	4.662	61.941	911.503
smartest	.002	.013	.174	1.54	12.97	127.61

The final two brute cells are left blank because the computation was infeasible on my machine (and I suspect many others as well).

Here is the result as the same linear test from above, but with positive only arrays and smaller sizes of n due to computation

3 Graph Comparison

All 3 algorithms run and plotted on a graph to compare runtimes. Note: tests rerun with much smaller but many more n values to get better graph data.

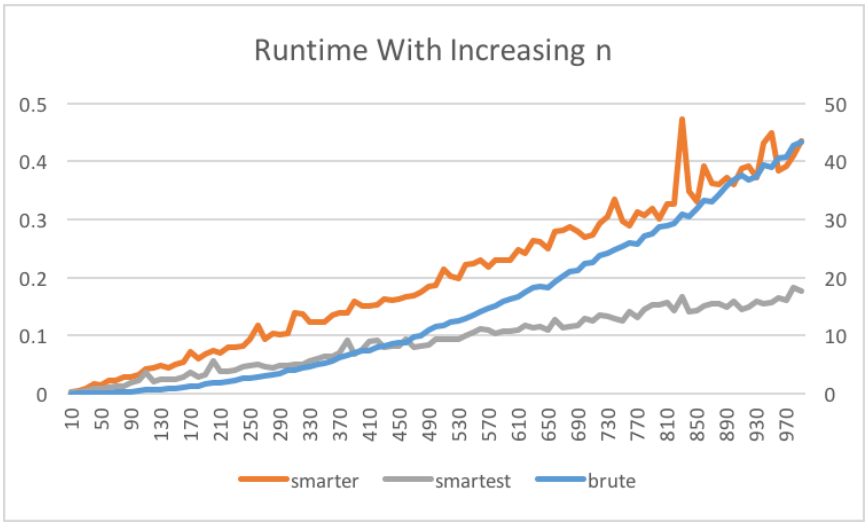

Figure 1: brute algorithm plotted on secondary axis for better fit