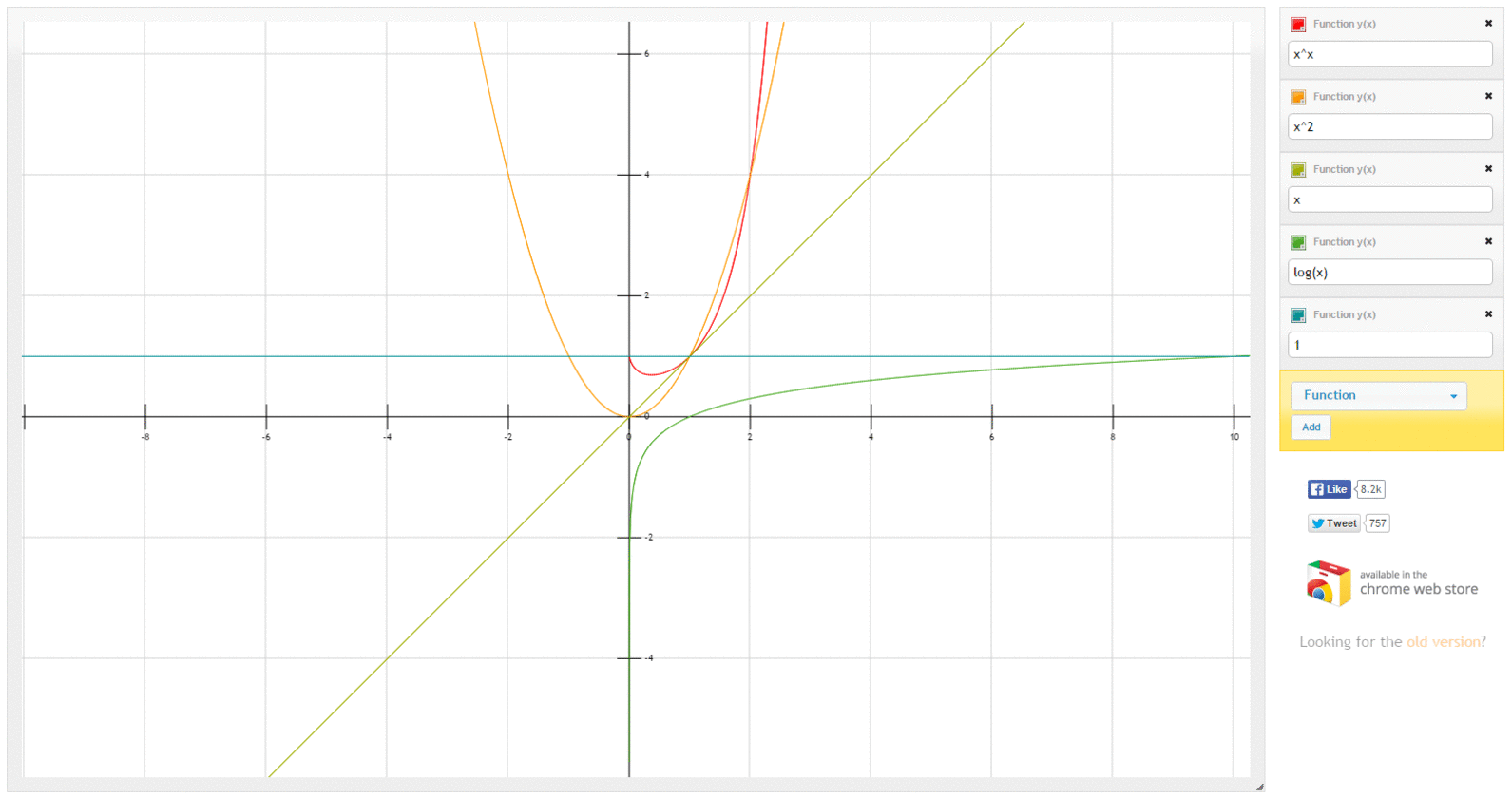
Divide and Conquer

When we need to find certain elements in a list of items, divide and conquer is an algorithmic approach. One example we could consider is trying to find the maximum and minimum elements in a list of integers.

The usual approach would be to iterate through every item, and comparing between every adjacent pair. Hence finding the maximum we would have N - 1 comparisons. Then for the minimum, we would have N-2, hence 2N-3. But we could do better.

Remember the scale of algorithmic performance:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **O(1)** | **O(logn)** | **O(n)** | **O(n2)** | **O(2n)** | **O(nn)** |



**The Idea:**

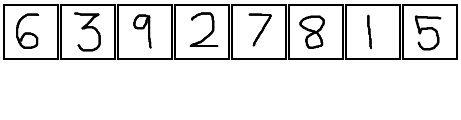
The scheme is to **divide** the input into parts, to then **solve** the parts, and then **combine** the solutions to give the final result. Remember: divide, solve, combine

**An Example:**

Using divide and conquer, we can find the maximum and minimum elements in a sequence of integers.

* Divide the sequence *s* into *s1* and *s2* (we may do this arbitrarily, however for the sake of efficiency, it is usually best to have approximate equal sizes)
* Now we recursively calculate the maximum and minimum of the subsequences of *s1* and of the subsequence *s2*. Call these *max1, min1, max2, min2*

|  |
| --- |
| **function** maxMin(s)  **if** s = [x] **then**  **return** (x, x)  **endif**  **if** s = [x1, x2] **then**  **if** x1 > x2 **then**  **return** (x1, x2)  **else**  **return** (x2, x1)  **endif**  **else**  (s1, s2) = divide(s);  (max1, min1) = maxMin(s1)  (max2, min2) = maxMin(s2)  **return**(max(max1, max2), min(min1, min2))  **endif**  **end function** |



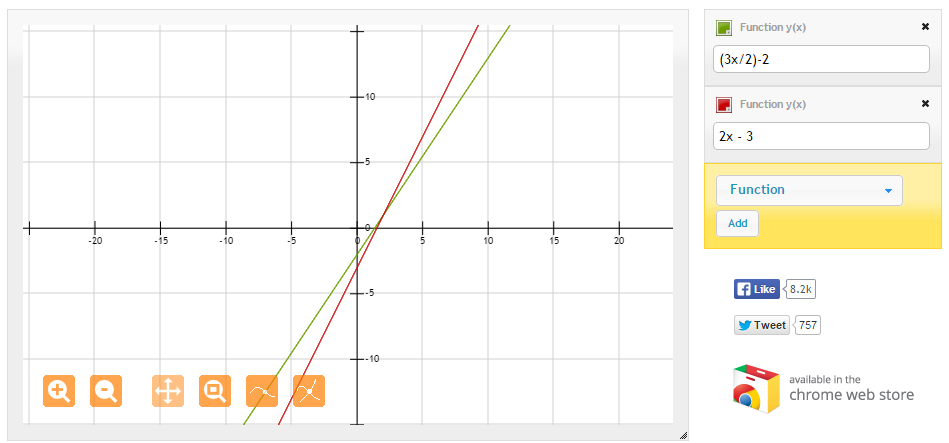
**Complexity**

CN= 2 \* CN/2 + 2

So here we have a recursive complexity. Since at each of the sub-lists, we have two operations, the two comparisons, we have the complexity at each level to be 2. Hence the +2

Now since each of lists have potentially two children, we have 2 \* CN/2where N/2 represents the size of each of the children’s lists.

By solving the above equation algebraically, we get CN = 3N/2 - 2



Although this is linear, it still has a better performance than the original.

**Applications of Divide and Conquer:**

* **Sorting Algorithms** -- Merge sort and quick sort
* **Integer Multiplication** -- Integer multiplication by long multiplication is O(n2), divide and conquer provides O(nlog(n)) solutions
* **Matrix Multiplication** -- Standard is O(n3), divide and conquer can bring this down to O(n2.4)
* **Nearest Neighbour Problems** -- Requires finding Euclidean distances
* **Computational Geometry**

Greedy Methods

As the name suggests, imagine you are short sighted and greedy. This is what the whole algorithm is about. An example to demonstrate this is paying for a bus fare. The greedy approach is to take the largest suitable coin first, and then work down.

If something costs 37 pence, then we would choose

20p: 37 - 20 = 17

10p: 17 - 10 = 7

5p: 7 - 5 = 2

2p: 2 - 2 = 0.

Hence we have chosen 20p + 10p + 5p + 2p. (The lectures seem to forget we have 2p coins in the UK).

**The Greedy Method is not perfect:**

Although in the spirit of the UK currency, greedy does a pretty good job. It’s not always perfect (we say better performance is using less coins). Imagine we were to create some sort of odd currency, where the only coins available were 1p, 6p and 10p. We wanted to buy something for 12p.

10p: 12 - 10 = 2

1p: 2 - 1 = 1

1p: 1 - 1 = 0

Of course, we could have gone for 6p + 6p, which is the perfect solution. For the greedy method, this is impossible to determine.

**Optimisation Problems**

Optimisation is about finding the **best** solution, not just the first solution. We define best in the context of the problem. In the previous example, it was the least number of coins. But in general, we usually talk about maximum and minimum solutions.

Minimum above: 6p + 6p

Maximum above: 1p + 1p + 1p + 1p + 1p + 1p + 1p + 1p + 1p + 1p + 1p + 1p

A “good” greedy strategy will attempt to find a maximum or minimum solution, by maximising or minimising solutions at intermediate stages.

**Greedy Applications**

Greedy applications have various levels of success:

* For some applications, the greedy method is always successful (UK currency problems)
* For some applications, it will solve some instances, but not others (Fake currency example)
* For some applications, it gives suboptimal solutions, but is still a good approximation
* “” “” “” greedy strategy is not applicable, since it won’t produce anything useful

There are some problems which have greedy solutions:

* Path finding algorithms, such as Dijkstra's algorithm
* Job scheduling algorithms
* Finding spanning trees of a graph
* Knapsack problems

Dynamic Programming

With the coin example above, we saw that Greedy can often fail to find the best solution. An alternative approach, “Dynamic Programming” will always produce the optimal solution.

Imagine we have N coin types, and we number these 1, … N. We say the value of each of the coin types is vi.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **i** | 1 | 2 | 3 | 4 | 5 |  |
| **vi** | 1 | 2 | 5 | 10 | 20 |  |

Suppose we have enough coins of each type to produce an optimal solution.

**The “Obvious” Property**

Define s to be the sum we need to make.

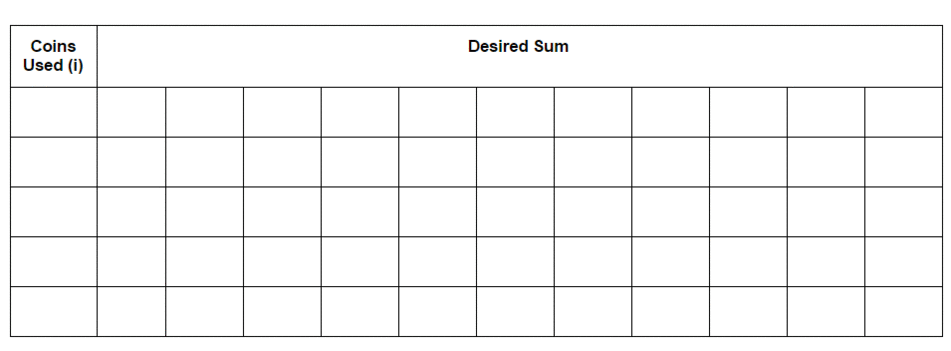
Define c(i, s) to be the number of coins from types 1 through i required to make the sum s.

In other words, c is the number of coins we have as a solution.

|  |
| --- |
| **c(i + 1, s)** = min( c(i, s), c(i, s - vi+1) + 1, … , c(i, s-(k \* vi+1) + k)  *where (k+1)vi+1 > s*  **c(i, s)** = ∞ *if the value of s cannot be made with the coins 1 … 1* |

Although this looks a bit odd, with the aid of a table and example, we can easily see these variables in action.

c(i, s) is represented by each of the cells which are not bold, and is the minimum number of coins, given i (the coins available), and s (the desired sum).

Along the first row, we see that it is impossible to get the sum 1, using the coin 9. Hence referring to above, we see that

|  |
| --- |
| **c(i, s)** = ∞ *if the value of s cannot be made with the coins 1 … 1* |

Hence we set everything which cannot be set to 9.

Along the second row, we have 1 = 1, 2 = 1 + 1 (so two coins), … , 9 = 9 (1 coin), 10 = 9 + 1 (2 coins) etc.

But, there is more to this than first meets the eye! If you look closely, when we set 9, we found the optimal solution for 9. Since we can never represent a cost with less than 1 coin (ignoring cheques and credit cards), we can safely say that this solution will not change. Hence we save this, and when we try to evaluate the next row, we only have to calculate 10 possibilities. Row 3, we calculate 9, and so forth. As you can imagine, on large data sets, this can save a fair bit of time.

**Applications of Dynamic Programming**

* Some path finding algorithms, such as Floyd’s algorithm for the all-nodes shortest path problem
* Longest common subsequence
* Knapsack problems, especially the 0 / 1 Knapsack problem
* Constructing optimal search trees
* The travelling salesperson problem

=