Human Agency in a Digital World

/ Understand technology and make it work for you

< MARCUS FONTOURA

an imprint of Microsoft

1st pass

31,	**27,**	101,	1,	–1,	0
27,	**31,**	**101,**	1,	–1,	0
27,	31,	101,	1,	–1,	0
27,	31,	1,	**101,**	**–1,**	0
27,	31,	1,	–1,	101,	0
27,	31,	1,	–1,	0,	101

2nd pass

27,	**31,**	1,	–1,	0,	101
27,	**31,**	**1,**	–1,	0,	101
27,	1,	**31,**	**–1,**	0,	101
27,	1,	–1,	**31,**	**0,**	101
27,	1,	–1,	0,	31,	101

3rd pass

27,	**1,**	**–1,**	0,	31,	101
1,	**27,**	**–1,**	o,	31,	101
1,	–1,	**27,**	**0,**	31,	101
1,	–1,	0,	27,	31,	101

4th pass

1,	**–1,**	0,	27,	31,	101
–1,	**1,**	**0,**	27,	31,	101
–1,	0,	1,	27,	31,	101

5th pass

–1,	**0,**	1,	27,	31,	101
–1,	0,	1,	27,	31,	101

Figure 1. All the bubble sort passes required to sort the full list. The comparisons in each step are highlighted. During each pass, we "bubble" the next largest element towards the end of the list. In this case we just need 5 passes to sort the full list, as the list was partially sorted with -1 and 0 in the correct relative order.

```
1     public int[] SortList()
2     {
3           var n = list.Length;
4
5           for (int i = 0; i < n - 1; i++)
6              for (int j = 0; j < n - i - 1; j++)
7                 if (list[j] > list[j + 1])
8                 {
9                    var tempVar = list[j];
10                   list[j] = list[j + 1];
11                   list[j + 1] = tempVar;
12                }
13
14    return list;
15    }
```

Figure 2. Bubble sort algorithm described in C#. The variable *j* identifies the location in the list, also referred to as the index, of the element we are comparing with its adjacent neighbor in index *j +1*. The value of the element in index *j* in the list is *list[j].* Similarly, *list[j + 1]* is the value for index *j + 1*. The fragment in lines 9 to 11 swap the list elements using variable tempVar as a temporary buffer. We store the old value of *list[j]* in this temporary variable. Then, we assign the value of *list[j+1]* to *list[j]*. If we hadn't saved *list[j]'s* value in the temporary variable, it would have been lost after line 10. Finally, in line 11 we assign the old value of *list[j]* to *list[j + 1],* effectively swapping the values. The *"for"* instructions in lines 5 and 6 control how many times we do the comparisons and swaps. We affectionately call these instructions for loops.

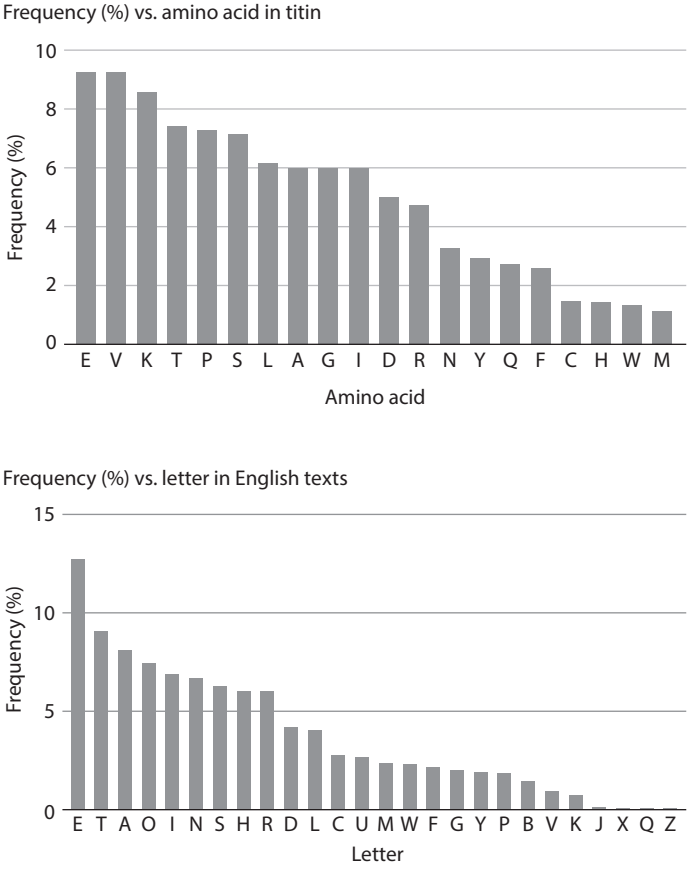

Figure 3. Distribution of amino acids in titin proteins and letters in English texts.

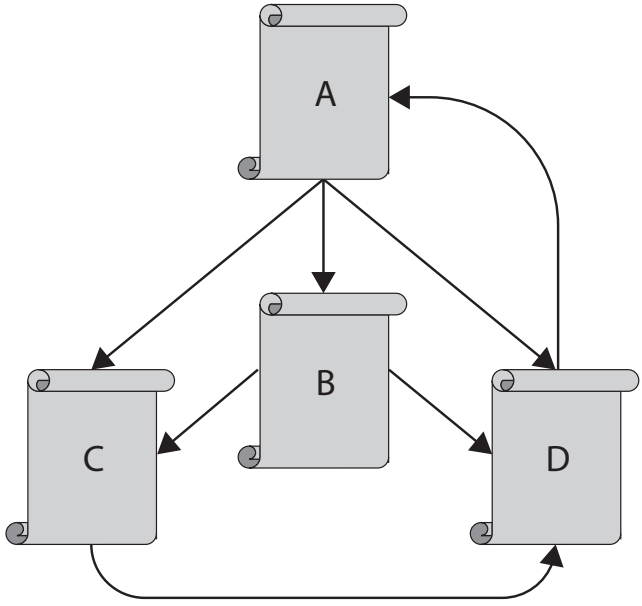

Figure 4. Example of four pages, *A*, *B*, *C*, and *D*, with hyperlinks amongst them.

```
1  static int[] FindIntersection(int[] array1, int[] array2)
2  {
3      List<int> intersection = new List<int>();
4      int i = 0, j = 0;
5
6      while (i < array1.Length && j < array2.Length)
7      {
8          if (array1[i] < array2[j])
9          {
10             i++; // Move cursor in array1
11         }
12         else if (array1[i] > array2[j])
13         {
14             j++; // Move cursor in array2
15         }
16         else
17         {
18             // Found a common element
19             intersection.Add(array1[i]);
20             i++;
21             j++;
22         }
23     }
24     return intersection.ToArray();
25 }
```

Figure 5. Intersection algorithm described in C#. Variables *i* and *j* are the indices of the two cursors for the lists in *array1* and *array2,* respectively. The output is stored in variable *intersection*. Line 19 adds the match that we found, when *array1[i]* is equal to *array2[j],* to the output list.

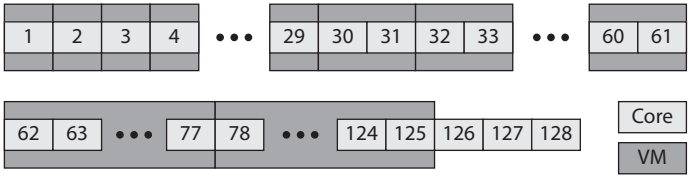

Figure 6. A possible allocation of VMs with one core, two cores, 16 cores, and 48 cores. The last three cores, numbered 126, 127, and 128, are left idle.

VMsallocated	Cores used	Memory used	Storage used
option 1: size-1, size-1, size-2	4	6GB	100GB
option 2: size-1, size-1, size-4	10	20GB	260GB
option 3: size-2, size-4	10	18GB	240GB

Table 1. Three possible allocations for the four requested VMs.

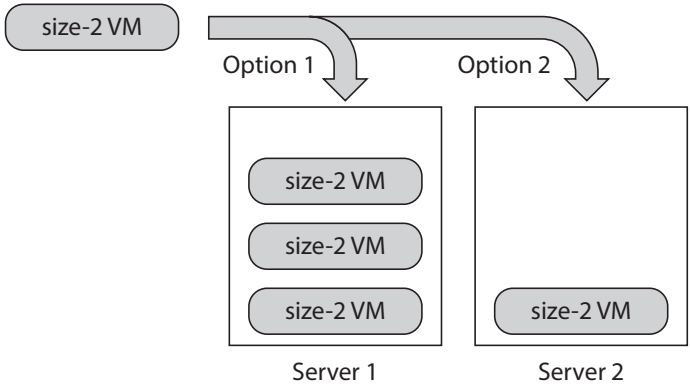

Figure 7. Tradeoff between two possible allocation options.

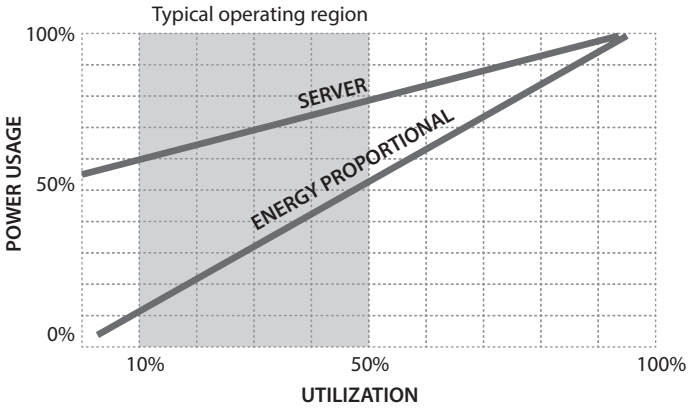

Figure 8. This figure is adapted from *The Case for Energy-Proportional Computing*.[15] It compares a typical, energy-efficient server with a hypothetical energy proportional machine. The server consumes about half of its power even when doing very little work. At high utilization, it starts behaving more efficiently in its power consumption.

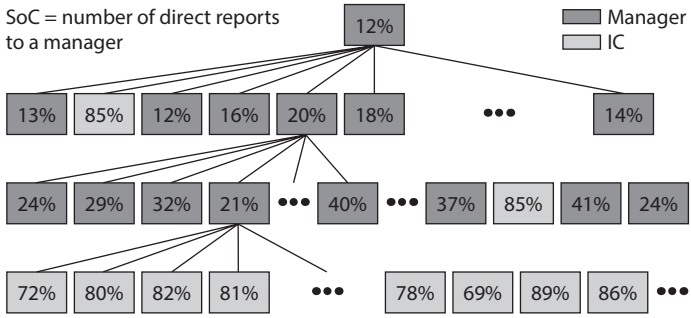

Figure 9. Example of hierarchical organization with four layers. The numbers inside the boxes represent the percentages VPW per employee. VPW increases as we go down the hierarchy, with ICs, in general, having much higher values than managers.

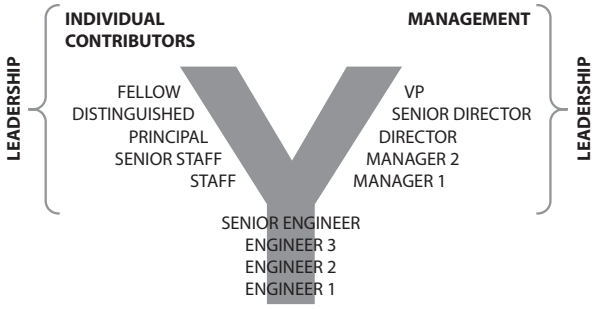

Figure 10. Y-career ladder. The initial branches are IC-only. The leadership tracks start at the base of the Y, with parallel tracks for ICs and managers.

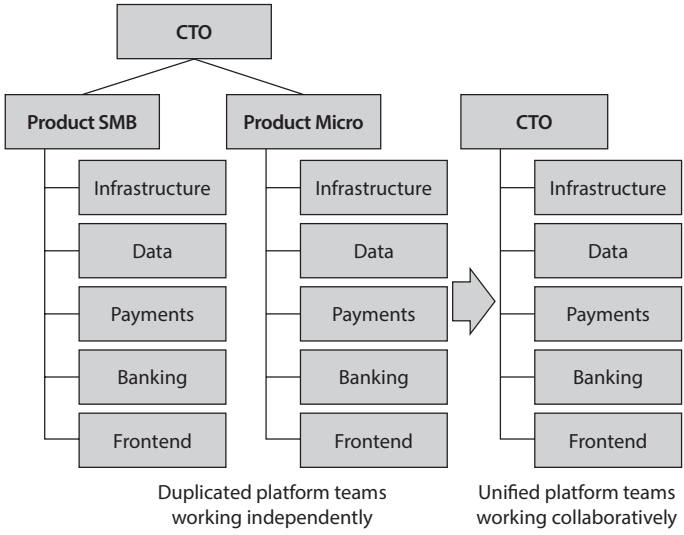

Figure 11. The organizational change at Stone transformed a product-oriented structure into a platform-oriented structure.

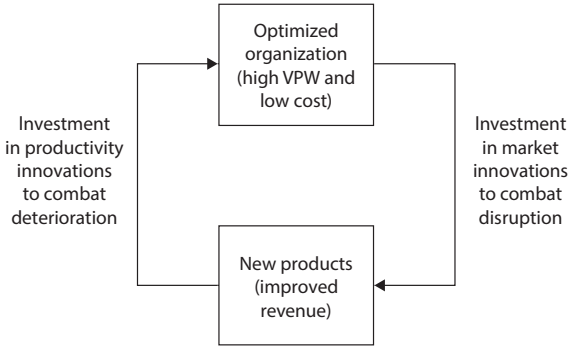

Figure 12. The virtuous cycle of innovation. Innovation in productivity improves the bottom line while market innovations improve the top line. Continued reinvestments of these gains combat deterioration and disruption. By investing in innovation, the company will attract better leaders, both managers and ICs, who will further increase productivity and produce novel ideas.

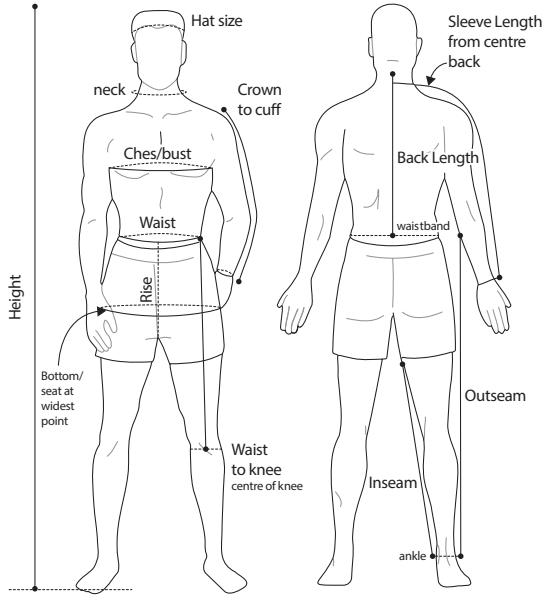

Figure 13. A typical tuxedo measurement chart downloaded from the web.

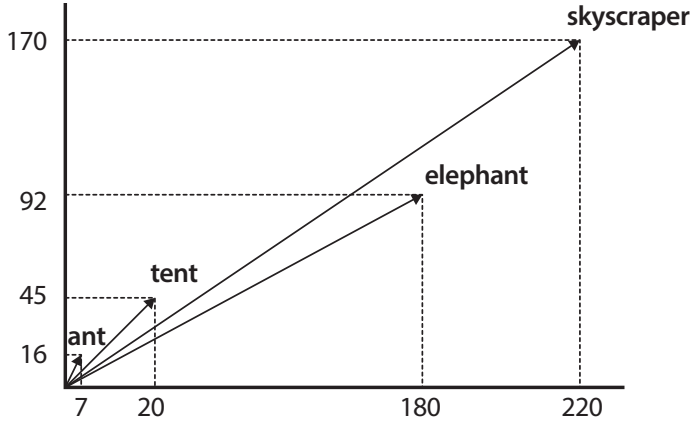

Figure 14. A few embeddings. *E(word)* is the embedding vector for "word." In this example,
E(elephant) is *(180, 92)*, *E(skyscraper)* is *(220, 170)*, *E(ant)* is *(7, 16)*, and *E(tent)* is *(20, 45)*.

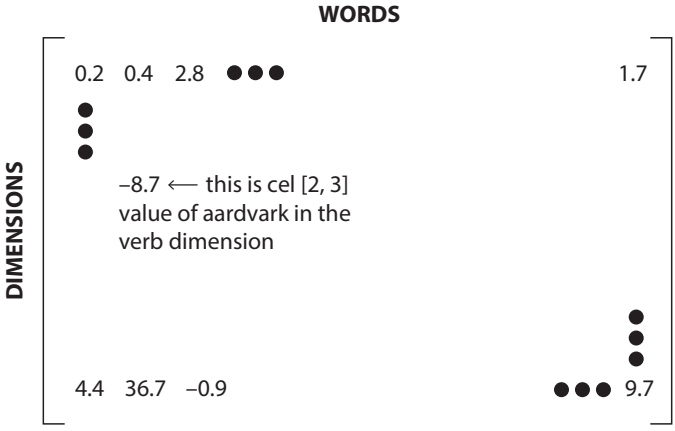

Figure 15. Embedding matrix in which the columns represent the embedding vectors for all words in the vocabulary. The size of the matrix is *1000* × *100*, as we have one thousand words and one hundred dimensions.

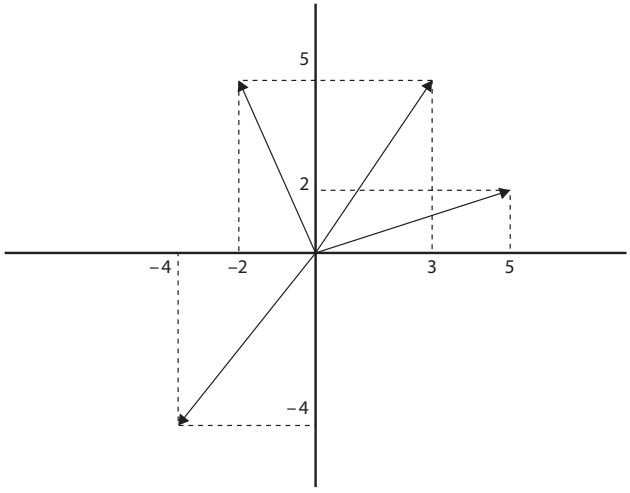

Figure 16. Dot product examples. The dot product of *(3, 5)* and *(5, 2)* is *3 × 5 + 5 × 2 = 25* (correlated, vectors in the same direction). The dot product of *(5, 2)* and *(-2, 5)* is *5 × -2 + 2 × 5 = 0* (unrelated, vectors are orthogonal). The dot product of *(5, 2)* and *(-4, -4)* is *5 × -4 + 2 × -4 = -28* (opposing, vectors are in opposing directions).

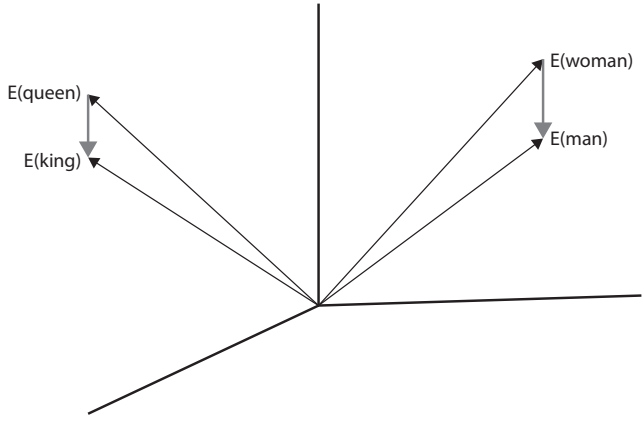

Figure 17. This is a 3D projection of the high-dimensional space of embeddings. The difference between the embeddings of "queen" and "king" is similar to the difference to the embeddings of "women" and "men."

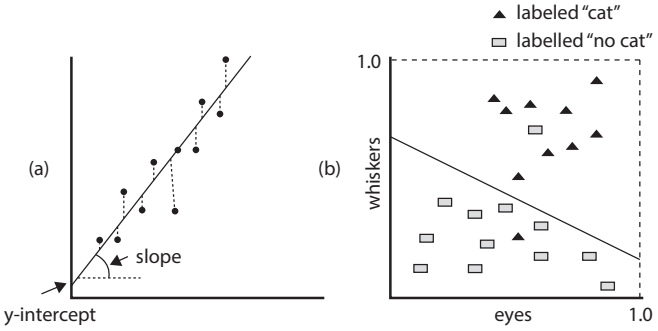

Figure 18(a). AI model to predict future growth of a service. (b) AI model to classify videos as "cat" and "no cat" using two simple features, the presence of whiskers and cat eyes.

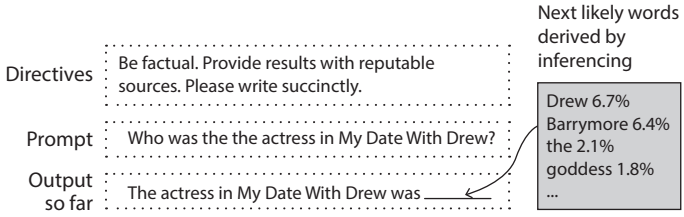

Figure 19. The inferencing algorithm produces a list of next likely tokens that will be appended to the output, one by one, until the full output is generated. After each token is selected, we invoke a new round of inferencing, producing a new list of next likely tokens.

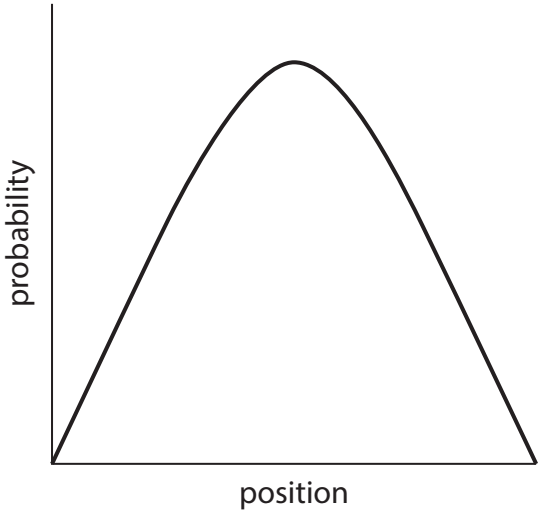

Figure 20. Sample wave function, which indicates the probability of the particle being at each position at a given time.

Criteria	Classical digital computer	Quantum computer
Information carrier	bit	qubit
Speed	picoseconds, or 0.000001 microseconds	10s to 1000s of microseconds
Error rate	10^{-15}	10^{-6} now, aiming for 10^{-18} in a few years
Scale	billions of bits per chip	10s of qubits now, aiming for thousands in a few years

Table 2. Comparison of classical digital and quantum computers. Take this with a grain of salt, since we still don't have a scalable quantum computer and these numbers may change drastically as we make progress in their development.

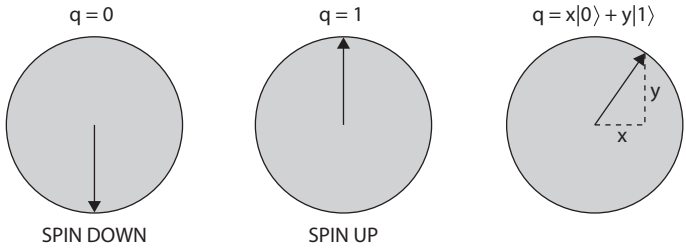

Figure 21. Qubits implementation using electron spins. Spin down means qubit q is 0. Spin up means it is 1. Unlike classical digital computers, qubits can be in a superposition state. In that case, reading the qubit would return 0 with probability x^2 and 1 with probability y^2.

5		9				4		
7		8	3		4	9		
6		1				7	3	
4	6	2	5					
3	8	5	7	2		6	4	9
1		7	4		8	2		
2			1					4
		3		4			8	7
	7			5	3			6

Figure 22. Sample sudoku board with 40 starting positions. We have 9^{41} combinations left.

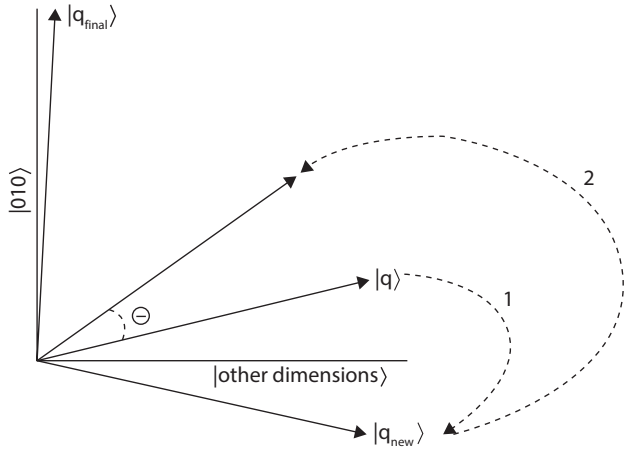

Figure 23. This figure shows the first step of Grover's algorithm. We start with the uniform superposition state $|q\rangle$. Each step of the algorithm involves two vector reflections. The first one reflects $|q\rangle$ over the hyperplane formed by the other dimensions, giving us $|q_{new}\rangle$. The second reflection moves $|q_{new}\rangle$ over the original uniform superposition state. These two reflections combined, move is an angle Θ closer to the final qubit $|010\rangle$. As Θ is approximately $\frac{1}{\sqrt{6}}$ in this example, after $\sqrt{6}$ steps will be aligned with $|q_{final}\rangle$.

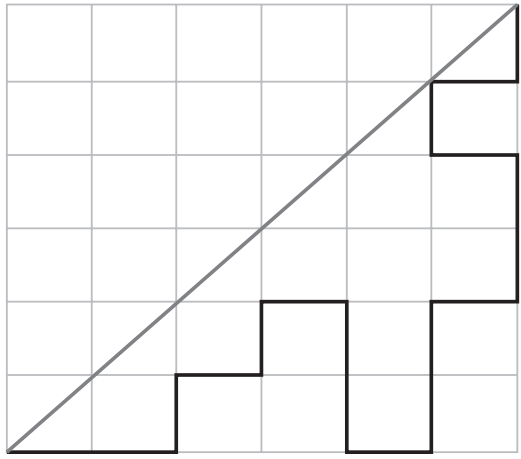

Figure 24. Illustration of a search algorithm in a regular digital computer, which only has access to the cathetus, and in a quantum computer, which can go through the diagonals. Being hand-wavy, this is where the speedup from $O(N)$ to $O(\sqrt{N})$ steps come from.