Design and Analysis of Algorithms

Lecture 01 – Algorithms, Programming, Data Abstraction and Core Data Types

Algorithms

An algorithm is a method to solve an information based problem.

With an algorithm specified you can create an implementation that will solve problems of the same type.

A data structure is a method to store the information content of the problem, and any other necessary information created about the problem.

With an algorithm, a data structure one can create a computer program to automate the problem solving process.

Algorithms

Here is a typical problem.

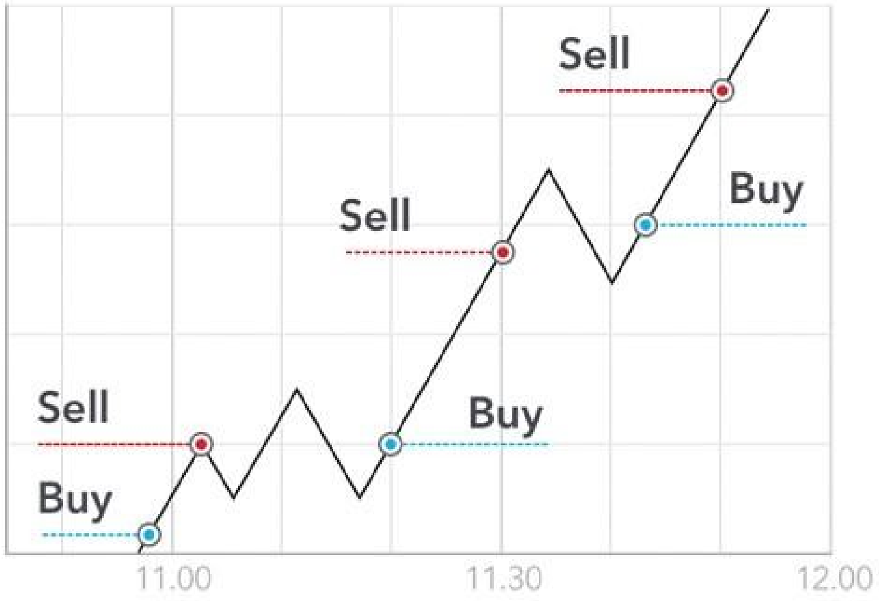
When should I buy or sell a stock?

Common strategy is to buy once you make sure the stock's price is on the rise, and to sell once you make a decent profit.

The idea is that many people follow this strategy, so that once they are satisfied with their profits, they will start selling and create a fall in the price.

But **how** do you make sure the stock's price is on the rise?

That is a problem to solve and the solution is you algorithm.



Algorithms

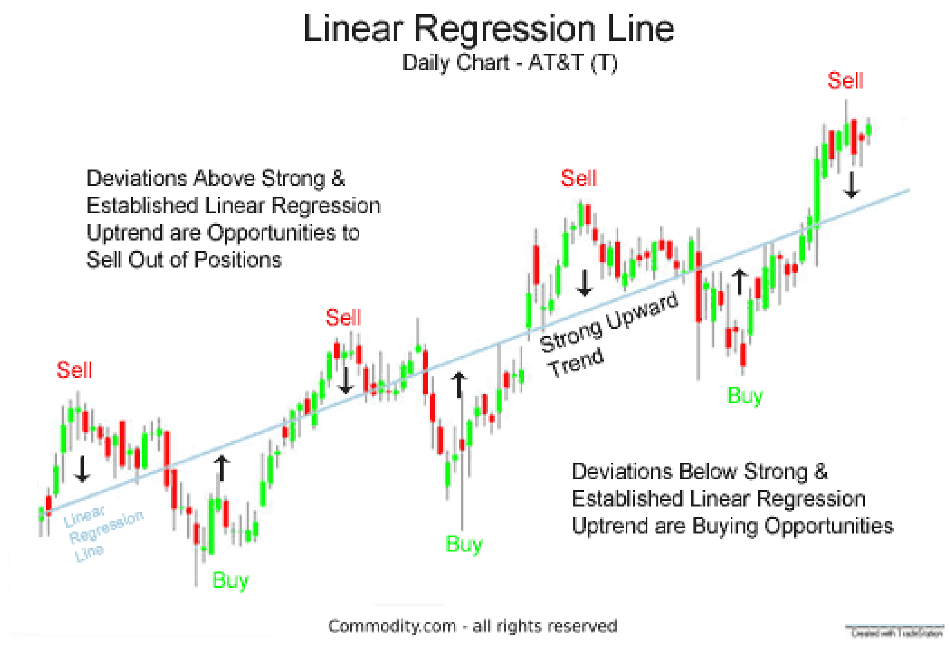
Here is a typical problem.

Using statistical regression, a very basic tool in statistics, one can set up a trend line that shows if the trend is upwards or downwards.

If you can specify to a program how to compute a trend line, and check if the slope is upwards (ie. positive), then you can make sure that the price is on the rise.

The specification is now your algorithm, and your code (ie. in Java) that does the computation is your program.

Without algorithms, computer programs are limited to very trivial tasks.



Algorithms

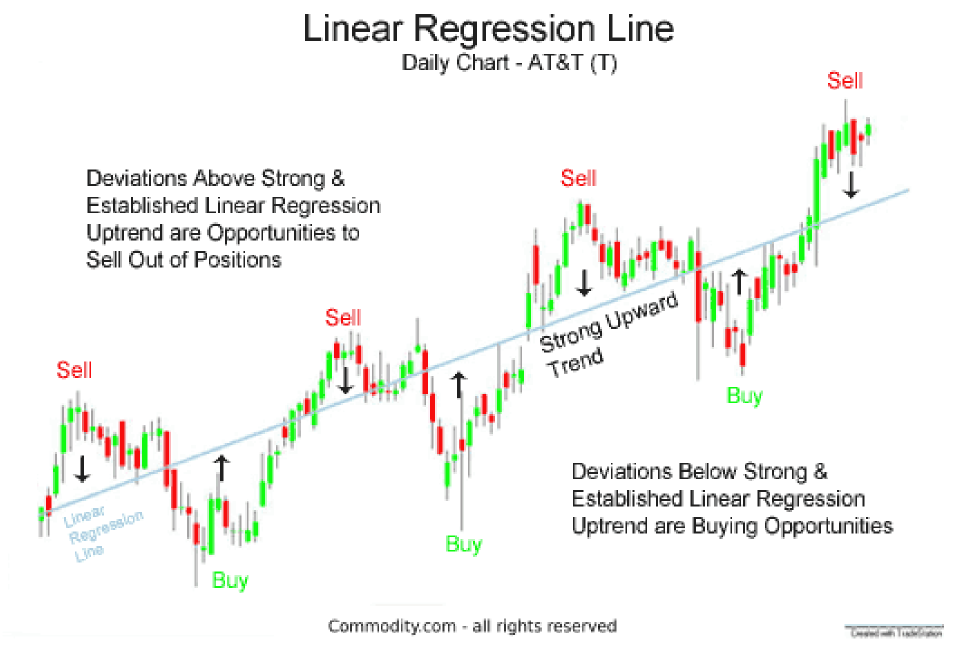
Here is a typical problem.

Algorithms are designed and validated using mathematics. Code is verified against the algorithm.

Validation means the algorithm solves the intended problem. Verification means the code correctly implements the algorithm.

Validation and verification are commonly called **testing**.

Testing usually requires sample inputs and outputs for which we know the relationship.



Algorithms

Many problems are information based because we solve these problems by abstracting them into information.

- Abstraction is a simple yet powerful feature of the human brain.
- It allows us to understand only those parts of the whole that we consider relevant for our purpose.

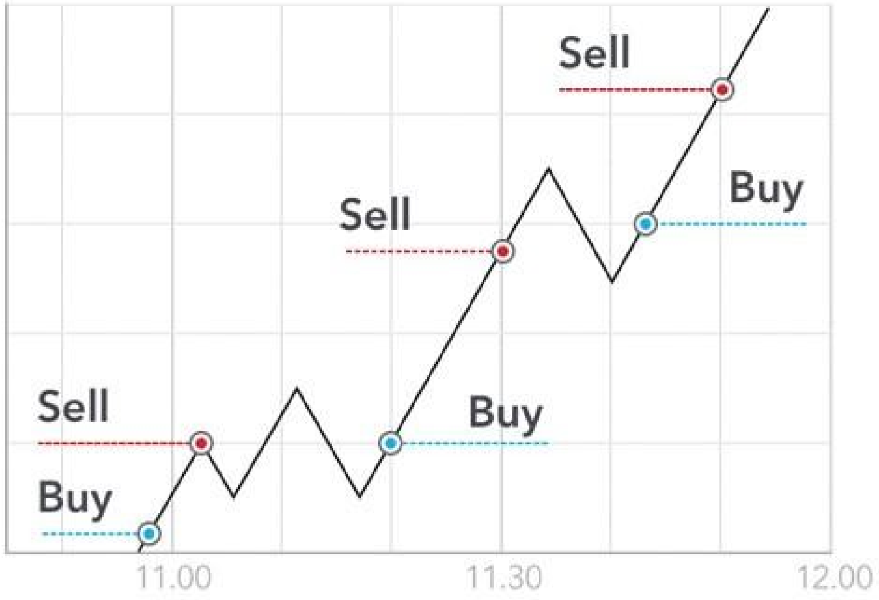
To understand each type of algorithm, we should understand the type of problem addressed and the information content (ie. the abstraction) of the problem.

- Then we compare our new problems' abstractions with the older problems we have studied.
- We decide if they are similar. If they are similar, we try to use the solution to the older problem.

Algorithms

An example problem.

- We are worried about **mosquito population**.
- One way to combat mosquitoes is to spray everyplace with a poisonous gas.
  - This gas kills the eggs laid by currently alive mosquitoes.
  - If we spray continuously for many days, all eggs laid will die, and the existing mosquitoes will just outlive their lifespan.
  - If we spray for a limited number of days, the number of mosquitoes will start dropping even after we stop spraying.
- However, this gas harms also other living things such as humans.
  - So we should minimize the use of gas to only when necessary.
  - Controlling but not destroying the mosquito population should be sufficient.
- If only we knew when to **start** spraying, and also when to **stop**... Just as we know then to **buy** and also when to **sell**.



Algorithms

All algorithms consume data.

In the cases of both stocks and mosquitoes, the data is a time series. It is abstracted as **a linear series of numbers**, with **an index** that allows us to understand with number occurred before another.

Mathematical representation of time series also contains an index.

How to express that with a computer?

A series of numbers is basically stored in a block of computer memory.

We should know **how large a single number is**, so that we can identify and pull individual numbers from that memory. We should also know **how to interpret the bits in the pulled part of memory**. These two make up **a data type**.

We should also know **how to process a wholesome** of these numbers. That is the part left for a data structure (to be covered some time later).

Algorithms

Computers really work only on whole numbers.

But we can extend on how to interpret the exact same bits (ie. 1’s and 0’s) to wok with real numbers as well.

Beware, real numbers are not exactly represented. Instead, we have single and double precision floating point numbers.

As the number of bits in an computer architecture increase, we get better precision.

Sometimes you end up with errors related to 1:1.000.000th of your expected exact result.

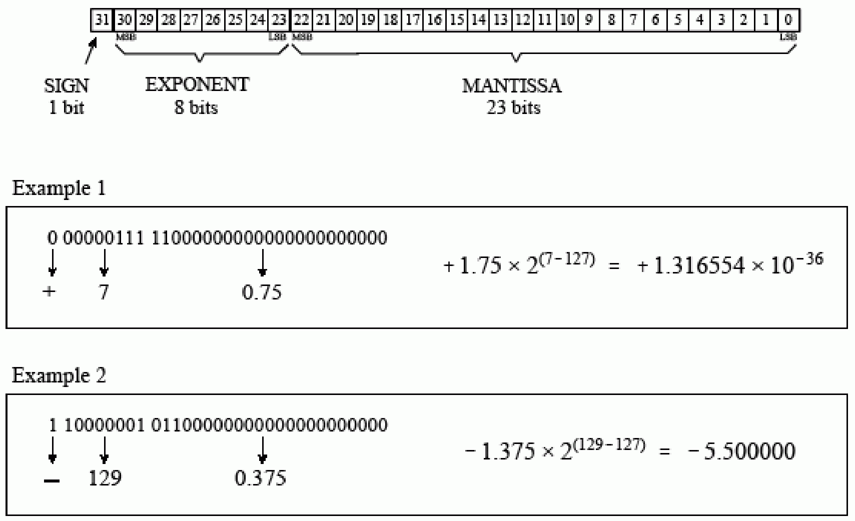
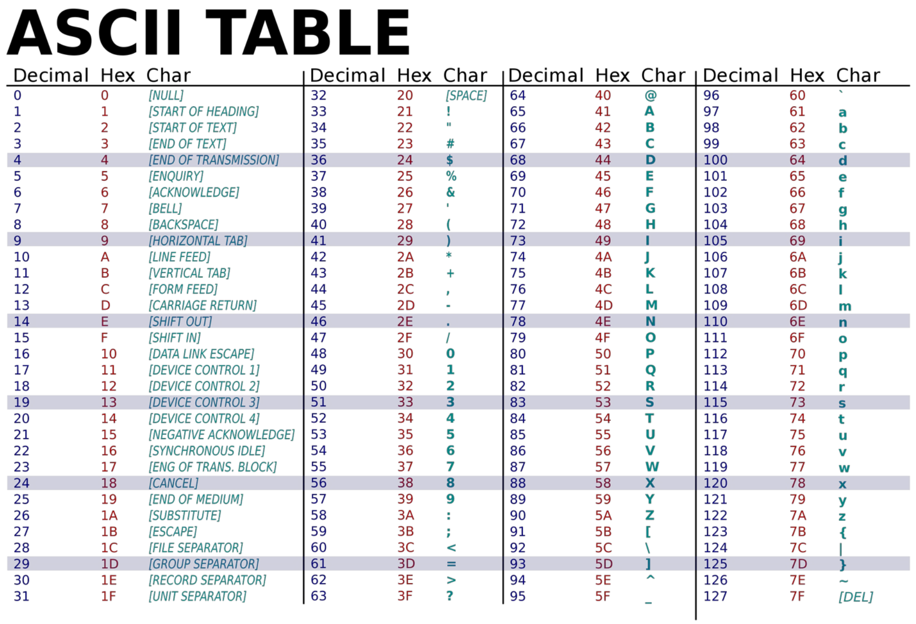


FIGURE 4-2
Single precision floating point storage format. The 32 bits are broken into three separate parts, the sign bit, the exponent and the mantissa. Equations 4-1 and 4-2 shows how the represented number is found from these three parts. MSB and LSB refer to “most significant bit” and “least significant bit,” respectively.

Algorithms

Computers really work only on whole numbers.

- We can also extend this to even non-numbers.
  - Characters are just encoded so that particular whole numbers, when interpreted as a character, correspond to the index in the encoding table.
  - Unfortunately there are as many encoding tables as human languages.
  - As Americans commercialized computers in 1970s, they had the choice for the default table (ie. ASCII table) and default language (ie. English).
  - UNICODE standard uses a much large table, so that all languages fit in the same table.



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Algorithms

Computers really work only on whole numbers.

- But we can extend on how to interpret the exact same bits (ie. 1’s and 0’s) to wok with real numbers as well.
- Beware, real numbers are not exactly represented. Sometimes you end up with errors related to 1:1.000.000th of your expected exact result.
- We can also extend this to even non-numbers.
- Characters are just encoded so that particular whole numbers, when interpreted as a character, correspond to the index in the encoding table.

Programming

Let’s recall (or list) what we <u>absolutely</u> need to know about programming

- Core Data Types (int, float, double, char, strings and boolean, the concept of signed/unsigned, the concept of zero-initialization, the importance of initial values, enumerations, the concept of a structure, the concept of a typedef, the concept of a class and an object)
- Use of References (and or pointers, the concept of null, null not necessarily being zero, Java always using objects and references)
- Core Operations (assignment operator, basic arithmetic, modulus, use of parentheses, comparisons, boolean arithmetic)
- Branching (If statements, use of else, nested if statements, switches)
- Looping (The structure of a loop, checking before or after, while loop, do-while loop, for loop)

Programming

Let's recall (or list) what we <u>absolutely</u> need to know about programming (cnt'd)

- Arrays (The most basic data structure, always knowing the size, iterating from one end to the other, use of calculated indexes, not going out of bounds, expressing a circle as an array, taking a strip out of an array, for each loops, copying an array, re-sizing an array, deep copies vs shallow copies)
- Functions (calling a function, return values, call by value vs call by reference/pointer, recursion)
- Using Modules (importing libraries, full names of functions with libraries)
- Modular Program Design (designing libraries for functions and classes)
- Having heard of (but not necessarily fully understood) some design principles such as these: https://dzone.com/articles/10-coding-principles-every-programmer-should-learn

Programming

We will use Java as it has enough language features for us, and yet it is relatively easier to learn at a basic level.

- Java is <u>**old**</u>. It was released in 1996.
- Current version is Java 16, released in 2021.
- Therefore there is a great deal of material out there. Some of them are dated.
- However, Java is always backwards-compatible. So even if you use an older material, what you learn will be working OK. But you <u>**might be missing some newer and easier way**</u> to do things.
- Anything from 2005 on would be sufficient for you.

If you have learned R, Matlab, or Python the you will be very quick to learn Java.

- Remember that Java requires you to be more systematic than Python would, but not as rigid as R or Matlab.
- Java was designed on experiences gained using C++ which is another great language, more than a decade older than Java, and also still under development and heavy use.

Programming

Useful textbooks:

Core Java SE 9 for the Impatient. Horstmann, C. Addison-Wesley. 2017.

Head First Java: A Brain-Friendly Guide. 2nd Edition, Slerra, K. and Bates, B. O'Reily. 2005. (3rd Edition will be released in 2022)

Effective Java. 3rd Edition. Bloch, J. Addison-Wesley. 2017.

Recommended Online Sources (Mostly blogs, or simpler tutorials)

https://blogs.oracle.com/java/

https://www.baeldung.com/

https://mkyong.com/

https://howtodoinjava.com/

http://tutorials.jenkov.com/

(Back to) Algorithms

Suppose we have an algorithm on paper, and we know how to implement it with a programming language such as Java.

- Are we done?
- Is our algorithm fast enough?
- How much memory does it consume?
- Can our program run effectively on available computers?
- If not, we should improve our algorithm.

Algorithms

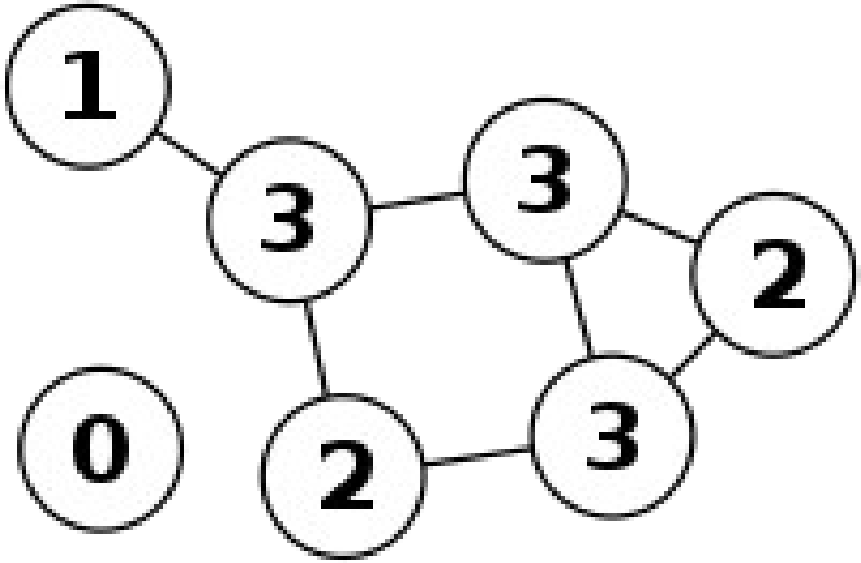
Some basic problems

- Count how many nodes each node is connected to.
- Discover items (nodes) not connected to anybody else (ie. connection count is zero).
- Find, if exists, any path connecting two given nodes.

What is the basic operation in solving these problems?

- Verifying that there exists a direct connection between two particular nodes.
- Counting the connections.
- Checking if any of a node's connected neighbors, have a path towards the other given node. (An indirect connection). No need to find the path.

How should we show the information content?



Algorithms

How should we show the information content?

A matrix showing all possible direct connections between two nodes. A 1 indicates a connection, a 0 indicates none.

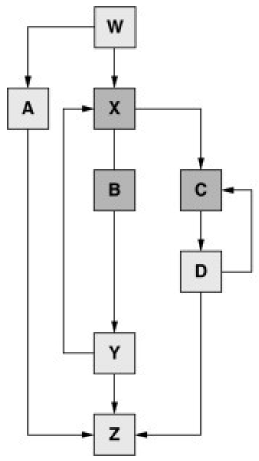
For the time being let's disregard the directions of the arrows.

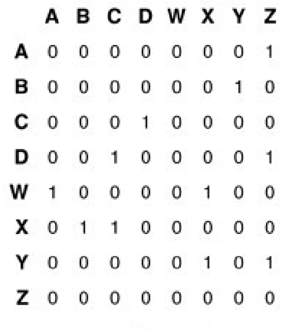
The basic operation in solving these problems, expressed as the information content?

Verifying that there exists a connection between two particular nodes. - Checking the particular entry in the matrix.

Counting the connections. - Summing up the numbers in a row.

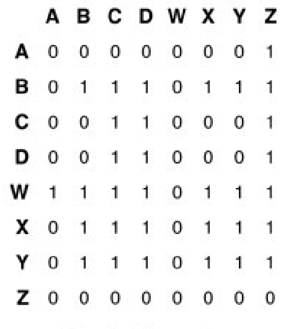
Checking if any of a node's connected neighbors, have a path towards the other given node. - For each 1-entry in the row, **construct its Minor matrix**, and redo the same for the minor matrix.





	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	1	0
C	0	0	0	1	0	0	0	0
D	0	0	1	0	0	0	0	1
W	1	0	0	0	0	1	0	0
X	0	1	1	0	0	0	0	0
Y	0	0	0	0	0	1	0	1
Z	0	0	0	0	0	0	0	0

Connectivity matrix



	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	1	1	1	0	1	1	1
C	0	0	1	1	0	0	0	1
D	0	0	1	1	0	0	0	1
W	1	1	1	1	0	1	1	1
X	0	1	1	1	0	1	1	1
Y	0	1	1	1	0	1	1	1
Z	0	0	0	0	0	0	0	0

Reachability matrix

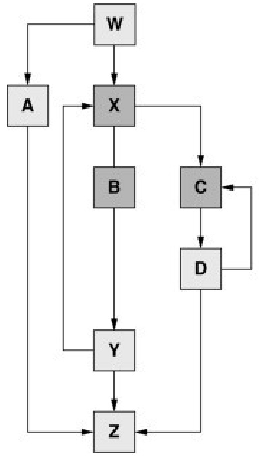
Algorithms

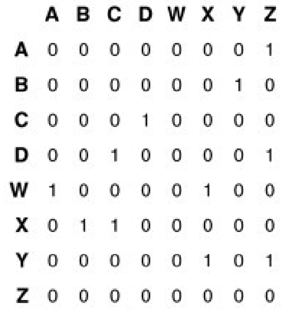
The information content, expressed as a matrix is our <u>abstract data structure</u>.

- Algorithms are easier to design with appropriate data structures.
- Efficiency of operations in algorithms are based on **<u>how abstract data structures are actually implemented</u>** in programming languages.

Implementing the matrix as a table (a two dimensional array) makes

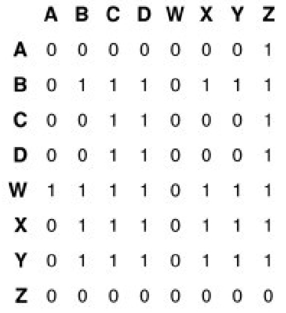
- The first two operations really easy.
- The third operation is conceptually easy to explain, but not necessarily straightforward to code.
- See some straightforward Java effort at minors, co-factors and determinants. – https://danhalesprogramming.medium.com/matrix-operations-in-java-determinants-ca7c8022944a
- Let's **<u>discuss</u>** the amount of memory and the number of steps for the third operation.





	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	1	0
C	0	0	0	1	0	0	0	0
D	0	0	1	0	0	0	0	1
W	1	0	0	0	0	1	0	0
X	0	1	1	0	0	0	0	0
Y	0	0	0	0	0	1	0	1
Z	0	0	0	0	0	0	0	0

Connectivity matrix



	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	1	1	1	0	1	1	1
C	0	0	1	1	0	0	0	1
D	0	0	1	1	0	0	0	1
W	1	1	1	1	0	1	1	1
X	0	1	1	1	0	1	1	1
Y	0	1	1	1	0	1	1	1
Z	0	0	0	0	0	0	0	0

Reachability matrix

Algorithms

Let’s go back multiple steps behind and discuss how we receive the data for the data structure?

- Do we read it from a file? If so, how is it formatted? Do we read one by one?
- Does the user or any other program enter the connectedness information (ie. links) one by one?

Let’s propose another structure. This time it is an array.

- Let connected[i] show the starting point for the string of items containing item “i.”

Algorithms

Let’s go back multiple steps behind and discuss how we receive the data for the data structure?

- Do we read it from a file? If so, how is it formatted?
- Does the user or any other program enter the connectedness information (ie. links) one by one?

Let’s propose another structure. This time it is an array.

- Let connected[i] show the starting point for the string of items containing item “i.”
- This is easier to construct. When we connect item j to item i, and trying to find out connected[j] for the first time, we ask the item it is being connected to (ie. item I) so that connected[i] is copied over to connected[j].

Algorithms

Let connected[i] show the starting point for the string of items containing item “i.”

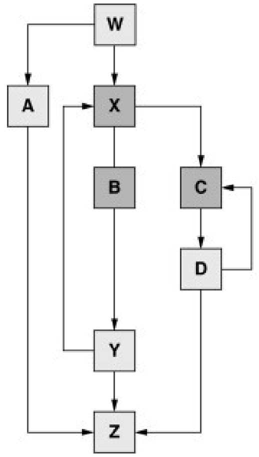
connected[W] is W (obviously.) Also connected[A] is W and connected[X] is W.

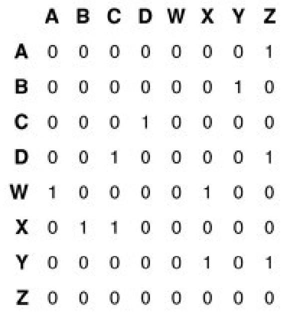
connected[B], and connected[C], and connected[Y] should be taken from X, so all three become W.

connected[D] is taken from C, so becomes W.

connected[Z] is taken from Y, so becomes W.

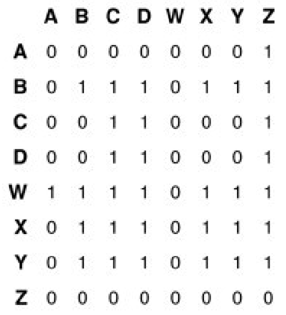
All items have W because it is the starting point.





	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	1	0
C	0	0	0	1	0	0	0	0
D	0	0	1	0	0	0	0	1
W	1	0	0	0	0	1	0	0
X	0	1	1	0	0	0	0	0
Y	0	0	0	0	0	1	0	1
Z	0	0	0	0	0	0	0	0

Connectivity matrix



	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	1	1	1	0	1	1	1
C	0	0	1	1	0	0	0	1
D	0	0	1	1	0	0	0	1
W	1	1	1	1	0	1	1	1
X	0	1	1	1	0	1	1	1
Y	0	1	1	1	0	1	1	1
Z	0	0	0	0	0	0	0	0

Reachability matrix

Algorithms

All items have W as their connected[…] value because it is the starting point.

- Verifying that there exists a direct connection between two particular nodes. - Not available.
- Counting the connections. - Not available.
- Checking for an indirect path. – Very easy. Checking that their connected[…] entries are the same. A **<u>single</u>** step needed! Much less memory too.

So if your data structure fits your problem, you are efficient.

Algorithms

Some data structures are efficient with static data. However, real-world data changes a lot.

Suppose we delete a single connection between two nodes. What do we do with the data structure?

- Matrix implementation – Find the entry, make it a zero. Single step.
- Array implementation – Re-create whole data structure. As many steps as there are nodes.

Over time the array implementation may become a burden in dynamic case.

- Highways between cities vs network connections between computers.

Algorithms

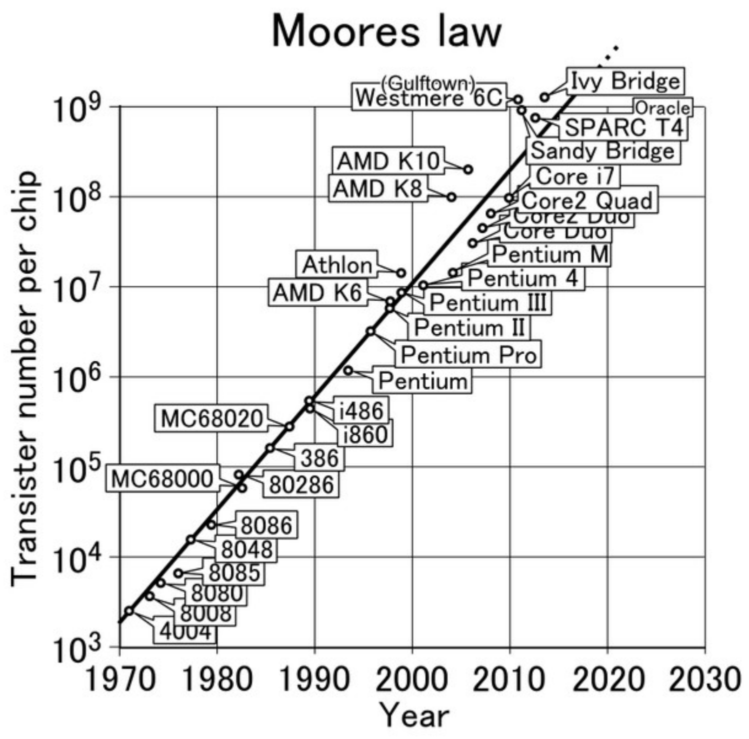
Can we not just get a better computer?

The computing capacity of processors roughly double every 18 months (Moore’s Law).

Memory chips are manufactured using the same production technology of processors. So we can double memory sizes every 18 months or so as well.

So we could get much better computers every 18 months.

Unfortunately our problems become larger and catch up with the expansion in computing capacity. Why?

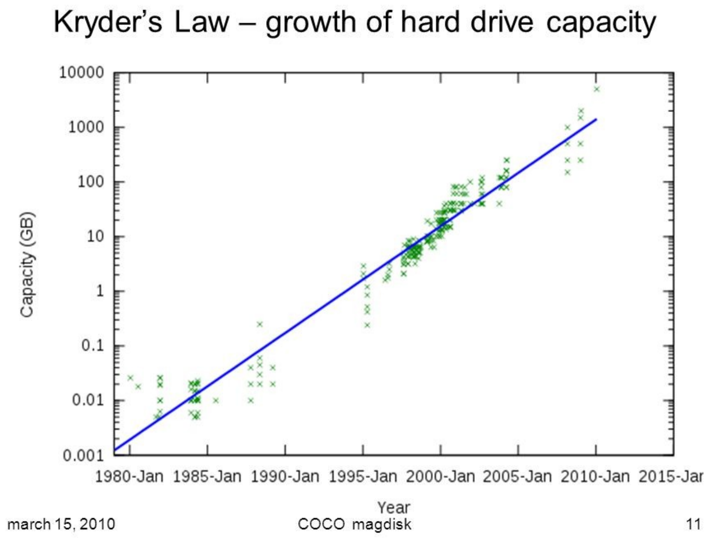


Algorithms

Can we not just get a better computer? (cnt'd)

The storage capacity of disks also double (every 13 months or so).

As the amount of data we store increase, the amount of data we want to process increases as well.

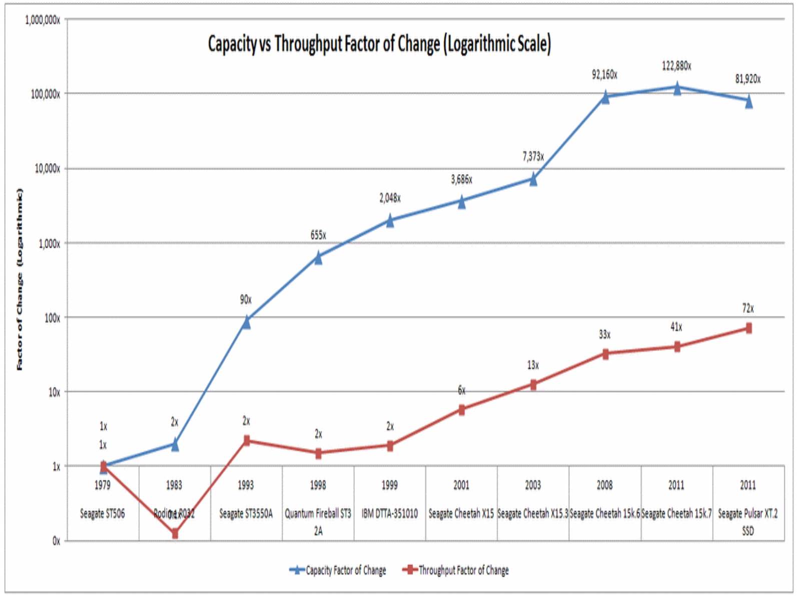


Algorithms

Can we not just get a better computer? (cnt’d)

Increased capacity in storage does drive our need to process more information, but **<u>throughput</u>** in accessing that larger sized information does not increase that fast!

Therefore we can <u>never</u> assume we will get a better computer, <u>unless we know our problem’s size will remain the same.</u>



Algorithms

May’s Law.

Software efficiency halves every 18 months, compensating Moore's Law.

In ubiquitous systems, halving the instructions executed can double the battery life and big data sets bring big opportunities for better software and algorithms:

Reducing the number of operations from N x N to N x log(N) has a dramatic effect when N is large … for N = 30 billion, this change is as good as 50 years of technology improvements.

Algorithms

We need to measure the efficiency of algorithms.

The number of computational steps to conduct a single operation. - This is called the **time metric**. More complex algorithms require more steps and take up more time.

The amount of memory per item/node used to store the date structure. - This is called the **memory metric**. Some data structures are memory inefficient at large sizes.

The traditional view is that to be better at one of these two, we sacrifice performance from the other.

The amount of power used to conduct a single operation. - This is called the **power metric**. It is conceptually related to the time because it is the CPU that uses most power. However in some cases, it is easier to visualize in terms of wattage.

- This is becoming particularly popular in mobile devices and in cases where the power usage occurs in a far-site, not necessarily observed or experienced by the end user.

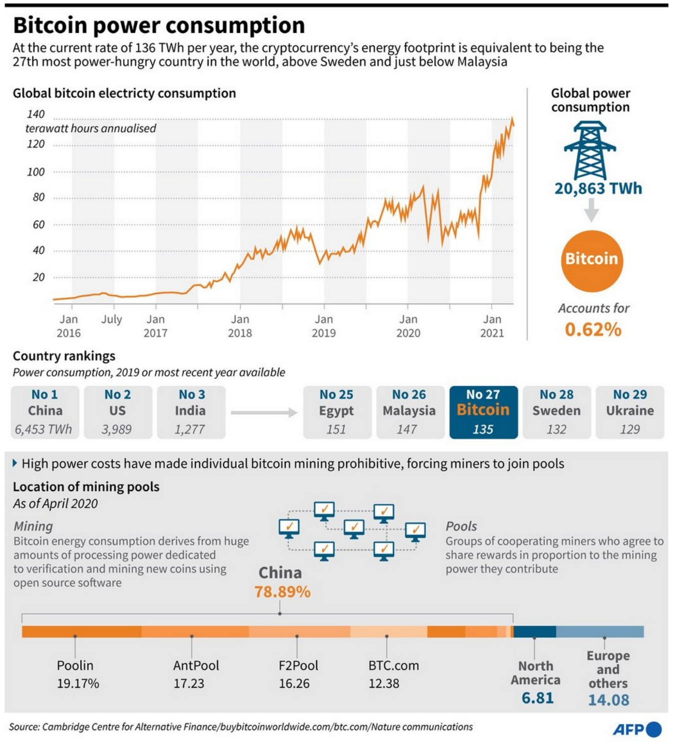
Algorithms

Case Study: Bitcoin

Bitcoin uses unique cryptographic methods to make sure that it is **statistically improbable to delete or modify a transaction in a distributed ledger.**

This comes at a significant algorithmic cost, in particular in terms of power consumed.

First of all, in order to provide this high level of security a large number of computers are required to be kept online, therefore there is a fixed cost, even when there are no transactions. This fixed cost increases with the size of the Bitcoin network (ie. people who own Bitcoin).



Algorithms

Case Study: Bitcoin

Second, in order to record even one transaction, more than half of all Bitcoin owners should acknowledge the transaction.

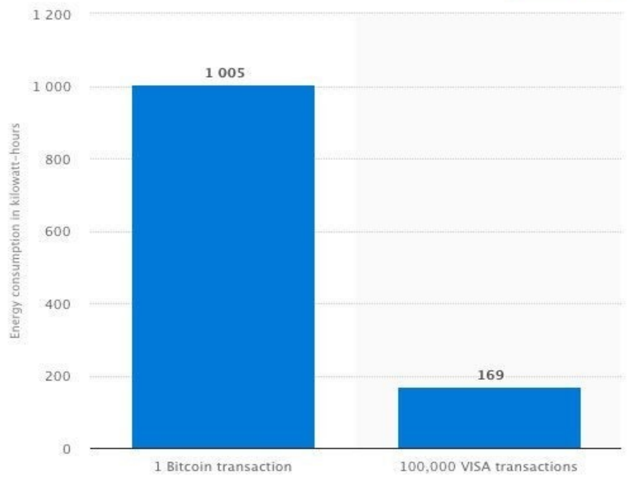
This requires power-hungry cryptographic operations on millions of computers and mobile devices.

Hence a money transfer on Bitcoin required roughly 600.000 times more power than a VISA payment. To better illustrate, a large mall consumes 1GWh of power in a day. So does a single Bitcoin transaction.

Is the Bitcoin algorithm power-efficient? No.

Are there any other cryptocurrencies which are power-efficient? Yes.

This statistic shows average energy consumption per transaction for Bitcoin compared to VISA as of 2018. According to the source, one Bitcoin transaction consumes about 1005 kilowatt-hours of energy.



© Statista 2018

Algorithms

Back to the connectedness.

Our problem is to find a balance between

- Knowing the path how nodes are connected.
- Knowing the end result immediately whether two nodes are indirectly connected.

Let’s try to balance with a third data structure. Another array.

- Let path[i] be the “parent” of node i, which is defined as the node that appeared before itself.
- So if node j is connected to node i, and node i appeared first, than path[j] is simply “i.”

Algorithms

Let connected[i] show the starting point for the string of items containing item "i."

path[W] is none (obviously.)

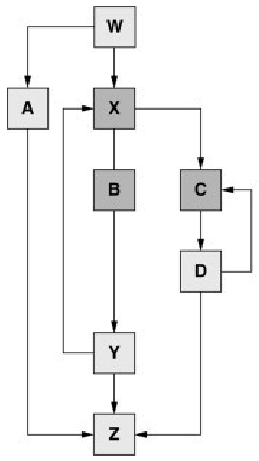
Also path[A] and path[X] is W.

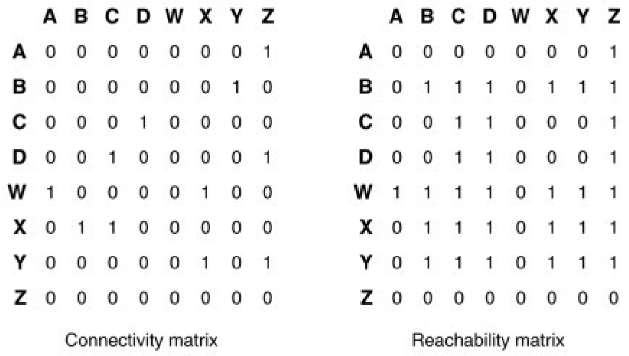
path[B], path[C], and path[Y] are X.

path[D] is C.

path[Z] is either A or Y, depending on our order of tracing. Let's say it is A.

```
        W  A  X B C Y D Z
Path [  -  W W X X X C A]
```





	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	1	0
C	0	0	0	1	0	0	0	0
D	0	0	1	0	0	0	0	1
W	1	0	0	0	0	1	0	0
X	0	1	1	0	0	0	0	0
Y	0	0	0	0	0	1	0	1
Z	0	0	0	0	0	0	0	0

Connectivity matrix

	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	1	1	1	0	1	1	1
C	0	0	1	1	0	0	0	1
D	0	0	1	1	0	0	0	1
W	1	1	1	1	0	1	1	1
X	0	1	1	1	0	1	1	1
Y	0	1	1	1	0	1	1	1
Z	0	0	0	0	0	0	0	0

Reachability matrix

Algorithms

W A X B C Y D Z

Path [- W W X X X C A]

Let’s discuss how to solve our three problems with this data structure.

- Verifying that there exists a direct connection between two particular nodes.
  - If one of them is the other’s parent? Yes, easily done.
  - Otherwise? There could be other paths as well. See the case of Y and Z. Z uses A as its parent, but Y is also Z’s parent.
  - How to solve this?
- Counting the direct connections.
  - Count nodes that show you as your parent. Add 1 for your parent (if it exists)
- Checking for an indirect connection.
  - Try to work backwards from both nodes and see if they meet at any common parent.
  - Example. Y and A are connected because they both trace back to W.

Algorithms

Let’s try to balance with a fourth data structure. Another array.

Let path[i] be the “list of parents” of node i, which is defined as any nodes that appeared before itself.

So if node j is connected to node i, and node i appeared first, than path[j] is simply “i.”

Also if node k is connected to nodes i and j, and both nodes appeared first, than path[k] is simply “i,j.”

	W	A	X	B	C	Y	D	Z
Path [	-	“W”	“W”	“X”	“X”	“B,X”	“C”	“A,Y”]

Algorithms

	W	A	X	B	C	Y	D	Z
Path [	**-**	**“W”**	**“W”**	**“X”**	**“X”**	**“B,X”**	**“C”**	**“A,Y”]**

Let’s discuss how to solve our three problems with this data structure.

- Verifying that there exists a direct connection between two particular nodes.
  - If one of them is among the other’s parents? Yes, easily done.
  - See that both pairs A and Z, and A and Y are easy to spot here.
  - However, **<u>we switch from simply checking an identity relationship to checking a set membership operation.</u>** We will call this **<u>a search operation</u>**. So the fixed cost of our operation is larger.
- Counting the direct connections.
  - Count nodes that show you as your parent. Add 1 for your parent (if it exists).
- Checking for an indirect connection.
  - Try to work backwards from both nodes and see if they meet at any common parent.
  - We will need to work for “all” parents.

Algorithms

Our final take as the fourth algorithm ends up as

- A data structure that stores N entries for N nodes (array).
- Each entry is a list of node entries, size is anything from 0 to N-1. So **total memory consumption is similar to but less than that of a matrix**.

Fourth algorithm, in terms of operations.

- Verifying that there exists a direct connection between two particular nodes.
  - Either one or two operations of searching among the list of node entries. (Why two?)
- Counting the direct connections.
  - For each other node, one search among the list of node entries. So basically N-1 searches.
- Checking for an indirect connection.
  - Check is there is a common parent (intersection of lists is non-empty). If yes, connected.
  - If not, repeat for one set of parents vs the other set of parents.
  - That is, produce many of the same problem with size smaller. How many of the same problems? The Cartesian product.
  - Example. Try to check if Y and D are connected.

Algorithms

Fourth algorithm, if the connection pattern changes.

We change the entries on the effected nodes only.

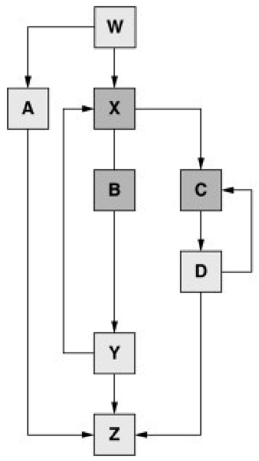
W A X B C Y D Z

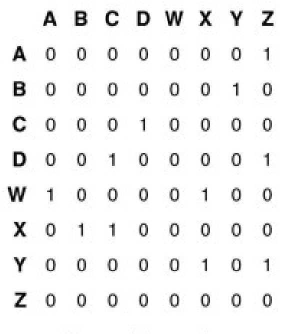
Path[- “W” “W” “X” “X” “B,X” “C” “A,Y,D”]

Let’s cut the connection between B and X.

W A X B C Y D Z

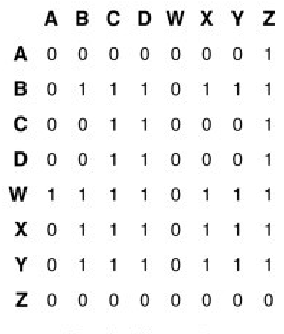
Path [- “W” “W” **-** “X” “X” “C” “A,Y,D”]





	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	1	0
C	0	0	0	1	0	0	0	0
D	0	0	1	0	0	0	0	1
W	1	0	0	0	0	1	0	0
X	0	1	1	0	0	0	0	0
Y	0	0	0	0	0	1	0	1
Z	0	0	0	0	0	0	0	0

Connectivity matrix



	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	1	1	1	0	1	1	1
C	0	0	1	1	0	0	0	1
D	0	0	1	1	0	0	0	1
W	1	1	1	1	0	1	1	1
X	0	1	1	1	0	1	1	1
Y	0	1	1	1	0	1	1	1
Z	0	0	0	0	0	0	0	0

Reachability matrix

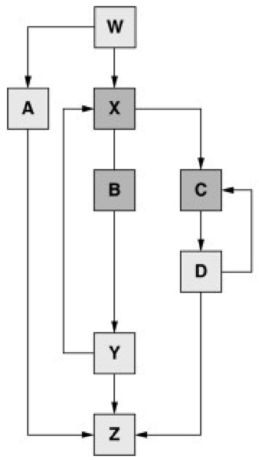
Algorithms

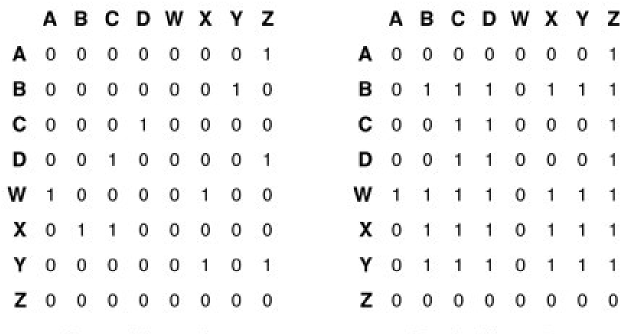
Another idea.

How about using the data structures from the second algorithm (which is excellent at the third problem) and the fourth algorithm (which is good at the first two algorithms).

The figure illustrates this in matrix form.

- The connectivity matrix is used to solve the first and second problem.
- The reachability matrix is used to solve the third problem.
- If there is a change in connections, we have to update the matrices. Connectivity matrix is easy to update, reachability is not.





	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	1	0
C	0	0	0	1	0	0	0	0
D	0	0	1	0	0	0	0	1
W	1	0	0	0	0	1	0	0
X	0	1	1	0	0	0	0	0
Y	0	0	0	0	0	1	0	1
Z	0	0	0	0	0	0	0	0

Connectivity matrix

	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	1	1	1	0	1	1	1
C	0	0	1	1	0	0	0	1
D	0	0	1	1	0	0	0	1
W	1	1	1	1	0	1	1	1
X	0	1	1	1	0	1	1	1
Y	0	1	1	1	0	1	1	1
Z	0	0	0	0	0	0	0	0

Reachability matrix

Algorithms

Important takeaway

To improve an algorithm we propose a data structure.

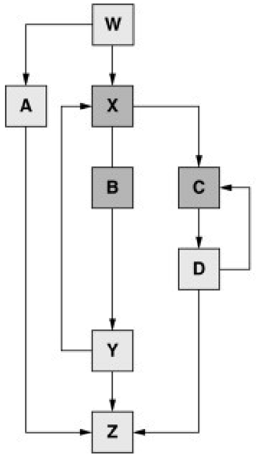
To improve a data structure we propose another algorithm.

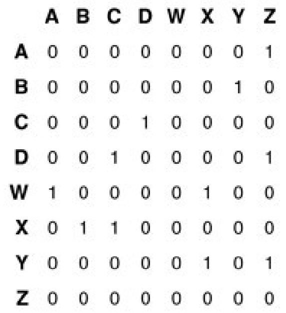
Here is an idea:

Repeat searches for the third problem's solution are the largest problem. We cannot simply avoid them.

Can we statistically minimize the number of repeat searches?

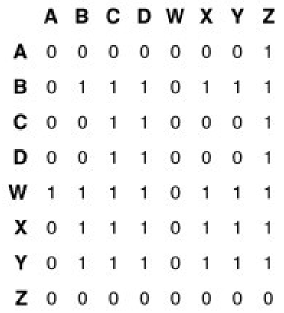
The idea is to change the parent we have chosen, so that part of the data structure gets re-structured and we end up with less depth, and less repeat searches.





	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	1	0
C	0	0	0	1	0	0	0	0
D	0	0	1	0	0	0	0	1
W	1	0	0	0	0	1	0	0
X	0	1	1	0	0	0	0	0
Y	0	0	0	0	0	1	0	1
Z	0	0	0	0	0	0	0	0

Connectivity matrix



	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	1	1	1	0	1	1	1
C	0	0	1	1	0	0	0	1
D	0	0	1	1	0	0	0	1
W	1	1	1	1	0	1	1	1
X	0	1	1	1	0	1	1	1
Y	0	1	1	1	0	1	1	1
Z	0	0	0	0	0	0	0	0

Reachability matrix

Algorithms

Change the parent:

W A X B C Y D Z

Path[- "W" "W" "X" "X" "B,X" "C" "A,Y,D"]

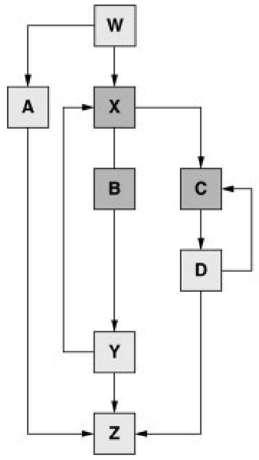
Can we choose something else as starting point?

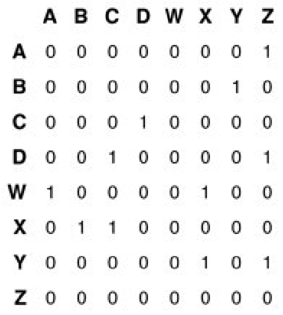
- Depth from W → A side. W → A → Z = 2
- Depth from W → X side.
  - X → B → Y → Z = 4
  - X → C → D → Z = 4
- Average distance = 3.33.

Maybe choose X as the starting point.

- Depth from X → W side. W → A → Z = 3
- Depth from X → B side. B → Y → Z = 3
- Depth from X → C side. C → D → Z = 3
- Average distance to root is 3.

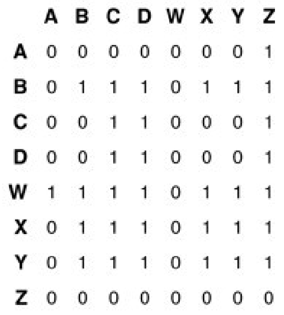
With this particular example, there is no significant effect. But as the structure grows, the effect of balancing becomes more significant.





	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	0	0	0	0	0	1	0
C	0	0	0	1	0	0	0	0
D	0	0	1	0	0	0	0	1
W	1	0	0	0	0	1	0	0
X	0	1	1	0	0	0	0	0
Y	0	0	0	0	0	1	0	1
Z	0	0	0	0	0	0	0	0

Connectivity matrix



	A	B	C	D	W	X	Y	Z
A	0	0	0	0	0	0	0	1
B	0	1	1	1	0	1	1	1
C	0	0	1	1	0	0	0	1
D	0	0	1	1	0	0	0	1
W	1	1	1	1	0	1	1	1
X	0	1	1	1	0	1	1	1
Y	0	1	1	1	0	1	1	1
Z	0	0	0	0	0	0	0	0

Reachability matrix

Algorithms

Another idea.

As long as the answers to our questions are answered the same, do we care about how the structure is represented?

Let's try to minimize depth by moving structures, and connecting them artificially to the root.

- Verifying that there exists a direct connection between two particular nodes. - Not answered correctly.
- Counting the direct connections. - Not answered correctly.
- Checking for an indirect connection. - Much faster.

Where to break?

- Break connection midway in the longest path from top to bottom.
- Go on until the depths vary by 1.
- You will end up with a very flat structure.

```
      W  A    X   B   C    Y    D   Z
Path[-  "W" "W" "X" "X" "B,X" "C" "A,Y,D"]
```

Y,Z eventually leads upwards to W. Let's connect Y to W. Disconnect Y from X.

Do the same to D,Z as well. Connect D to W. Disconnect D from C.

```
     W   A   X   B   C   Y          D     Z
Path[-"W""W""X""X""B,X,W""C,W" "A,Y,D" ]
```

Depths from W.

- A,Z=2
- Y,Z=2
- D,Z=2
- X,B=2
- X,C=2
- Average = 2.

Algorithms

Exercise:

Draw an arbitrary graph of at least 15 nodes.

Create the data structures for second, third, fourth, and fifth approaches.

For individual data structures and for the combined (second and fourth together)

- Classify and calculate the average number of steps for each of the three operations.
- Observe if there is any need to balance.

Core Abstract Data Types

All data structures are similar in the way that

- They have a method to add data/items
- They have a method to remove data/items
- They have a method to access an item, particular item or not.

Details change on the data structure

- Some data structures support other operations, based on what abstract data type they represent.

Core Abstract Data Types

Implementation Dependent

The behavior of these data structures are dependent on the implementation.

Arrays

Indexes are built-in to the structure. Therefore indexed access is very fast. You can also play neat tricks with indexes.

Arrays are created as fixed-sized. This both and advantage and a disadvantage.

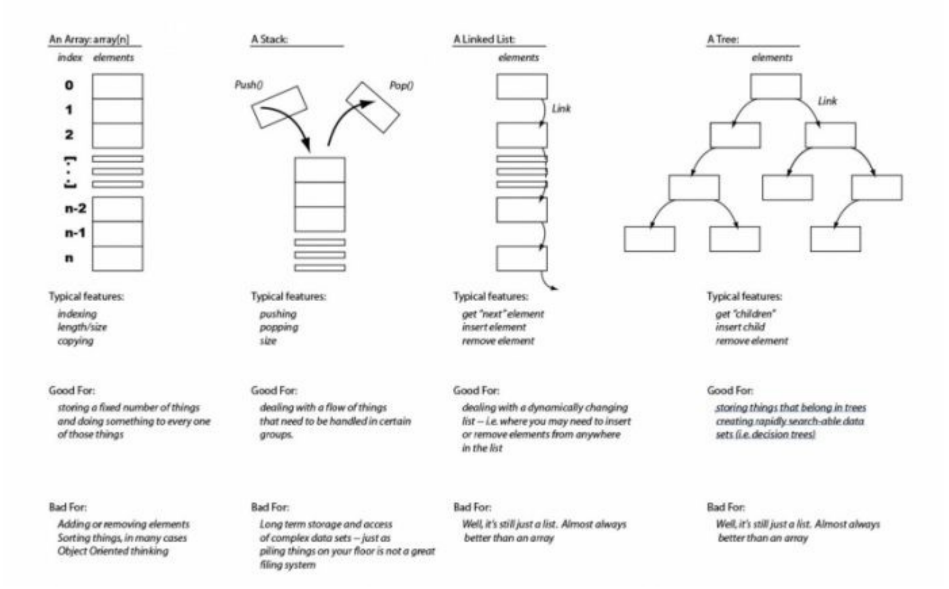
Operations like slicing are easy to define, but not always easy to implement.

Dependent on size-related details. Resizing arrays is an operation that may consume too much memory.

Arrays are often used to implement **stacks**. But stacks are abstract, so that they are not implementation dependent.

Arrays can be multiple-dimensioned. This allows construction of simple **tables**.

- One particular type is that of a **hash table**, where the index onto which you enter an item is determined by the hash generated by a hash function.
- The hash function processes the content of the item, and returns an integer hash value which is then used as the index.
- You might be required to implement your own hash function to be able to use a hash table.



	An Array: array[n]	A Stack:	A Linked List:	A Tree:
Typical features:	*indexing* *length/size* *copying*	*pushing* *popping* *size*	*get "next" element* *insert element* *remove element*	*get "children"* *insert child* *remove element*
Good For:	*storing a fixed number of things and doing something to every one of those things*	*dealing with a flow of things that need to be handled in certain groups.*	*dealing with a dynamically changing list – i.e. where you may need to insert or remove elements from anywhere in the list*	*storing things that belong in trees* *creating rapidly search-able data sets (i.e. decision trees)*
Bad For:	*Adding or removing elements* *Sorting things, in many cases* *Object Oriented thinking*	*Long term storage and access of complex data sets – just as piling things on your floor is not a great filing system*	*Well, it's still just a list. Almost always better than an array*	*Well, it's still just a list. Almost always better than an array*

Core Abstract Data Types

Implementation Dependent

The behavior of these data structures are dependent on the implementation.

Linked Structures

These structures contain two part nodes. One part of each node is the data, and the other part is a link that points to another node. Links can be implemented in many ways.

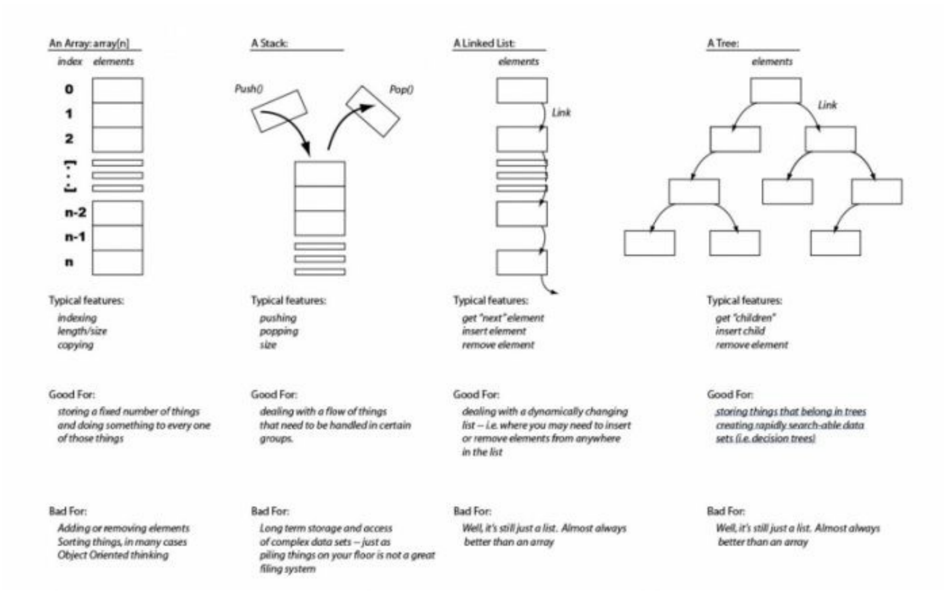
A **Linked List** is a linear, ordered structure similar to an array. An index does not exist, but can be virtually created. The major advantage is easy re-sizing.

Linked lists are used often to implement linear abstract types such as stacks and queues.

Combining arrays and linked lists, enables one to create simple **key-value stores** or **dictionaries**.

A **Tree** is a non-linear, ordered structure. The order is maintained through choosing a “side” to place a new node. Trees are very efficient in search operations as long as their balance is maintained.

There are many types of trees.



An Array: array[n]	A Stack:	A Linked List:	A Tree:
Typical features: *indexing, length/size, copying*	Typical features: *pushing, popping, size*	Typical features: *get "next" element, insert element, remove element*	Typical features: *get "children", insert child, remove element*
Good For: *storing a fixed number of things and doing something to every one of those things*	Good For: *dealing with a flow of things that need to be handled in certain groups.*	Good For: *dealing with a dynamically changing list – i.e. where you may need to insert or remove elements from anywhere in the list*	Good For: *storing things that belong in trees creating rapidly search-able data sets (i.e. decision trees)*
Bad For: *Adding or removing elements Sorting things, in many cases Object Oriented thinking*	Bad For: *Long term storage and access of complex data sets – just as piling things on your floor is not a great filing system*	Bad For: *Well, it's still just a list. Almost always better than an array*	Bad For: *Well, it's still just a list. Almost always better than an array*

Core Abstract Data Types

Implementation independent - Queues and Stacks

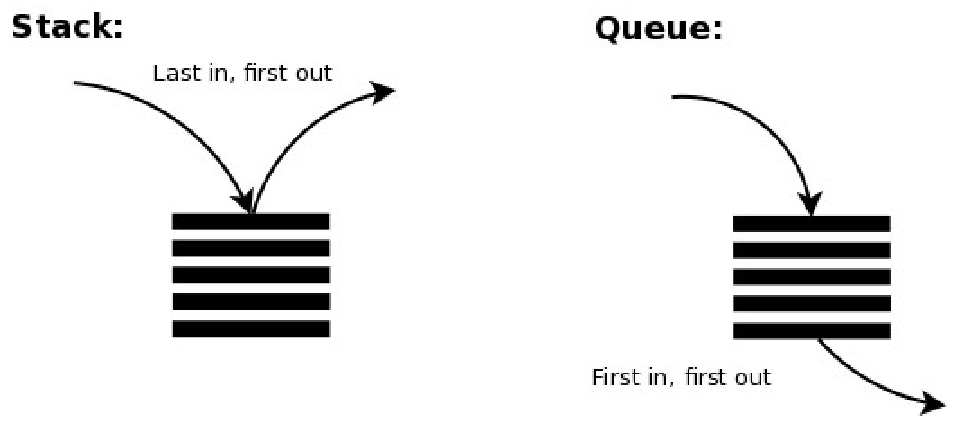
These structures are very abstract in their operations. The implementation details do not effect their behavior at all.

A **queue** is a data structure where the order in which the items are inserted is important. In a queue, items are inserted from the "back" and removed from "the front," effectively creating a First in First Out (FIFO) scheme.

The simplest way to implement a queue is that of an array.

A **stack** is another data structure where the order is again important. However, in a stack, the items are inserted from the "top" and removed from "the top," effectively creating a Last in First Out (LIFO) scheme.

The operations in these data structures are implementation independent. That means, however you implement these in code, the operations are still the same.



Core Abstract Data Types

Implementation independent – Bags and Sets

These structures are also very abstract in their operations, but they support a wider variety of operations than stacks and queues.

A **bag** (or a multiset) is a collection of items, where the order of insertion is completely irrelevant. Because the order is irrelevant, sorting the contents of a bag is impossible.

The same item can be inserted multiple times into a bag.

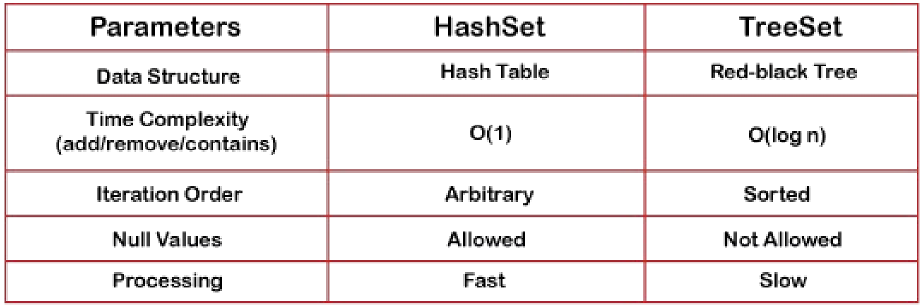
A **set** is the equivalent of a mathematical set. Compared to a bag, it is a special bag which allows insertion of an item just once.

Sets support operations such as unions, intersections and differences. Checking if a set is a subset of the other is also common.

A set that cannot change its contents anymore is called a **frozen set**.

Both data structures rely heavily on **search operations**.

Bags and sets are usually implemented using **trees** or **hash tables**. Note that a mapping relationship is necessary in order to use a mapping. A tree has no such constraint.



Parameters	HashSet	TreeSet
Data Structure	Hash Table	Red-black Tree
Time Complexity (add/remove/contains)	O(1)	O(log n)
Iteration Order	Arbitrary	Sorted
Null Values	Allowed	Not Allowed
Processing	Fast	Slow

HashSet Vs TreeSet

Core Abstract Data Types

Implementation independent – Mappings

A **mapping** (or map for short) represents a mapping relationship between values of two types. These types are often referred to as the **key type** and the **value type**.

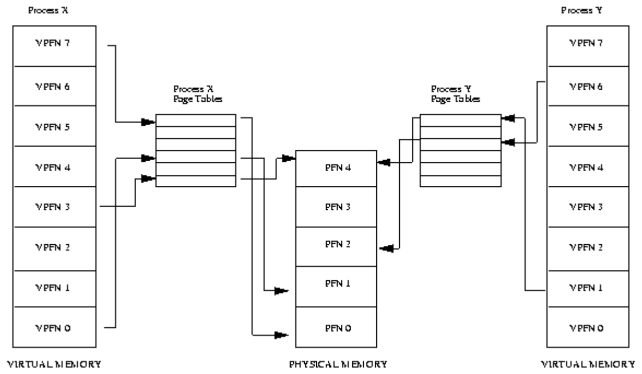
Value pairs are inserted, and using only a particular key, one can search if the corresponding value is stored in the mapping.

Mappings are **usually implemented as tables**, utilizing an array of key values and linked lists corresponding to each array entry.

Mappings are usually 1-1 but multiple values per key is also possible.

Example. Odd and even numbers used as keys. In this example, there are two categories. The mapping would result in 0 or 1 as the array index, and that would choose which linked list the value would be inserted into.

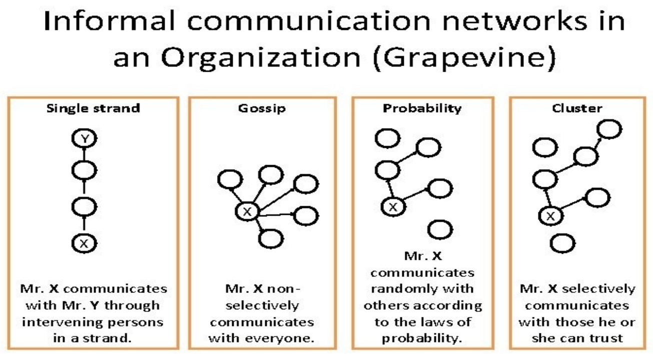
Example. Linux kernel manages the entire memory using maps. Portions of the virtual memory assigned to a process ae actually in diverse parts of the physical memory. A mapping exists between these two.



Core Abstract Data Types

An Example – Gossips

- Gossips propagate through informal networks.
  - In the case of formal communication, there is a structure to who communicates with who.
  - In the case of a gossip, a person does not select who to communicate. She just communicates with everybody.
  - So gossip propagates much faster than formally communicated information.
- On the other hand, gossips are unreliable information. Formal communication is more reliable. Another criteria of reliability is who the information is coming from.
  - Question. If you believe the gossip coming from a trusted friend, would you be willing to spread it?


Informal communication networks in an Organization (Grapevine)
Single strand
Y
X
Mr. X communicates with Mr. Y through intervening persons in a strand.
Gossip
X
Mr. X non-selectively communicates with everyone.
Probability
X
Mr. X communicates randomly with others according to the laws of probability.
Cluster
X
Mr. X selectively communicates with those he or she can trust

Core Abstract Data Types

An Example – Gossips

- If you wanted to simulate gossip propagation.
  - What would you be interested in? How would you measure it?
  - What kind of data structure would you need?
  - What kind of rules (algorithm) would you develop?