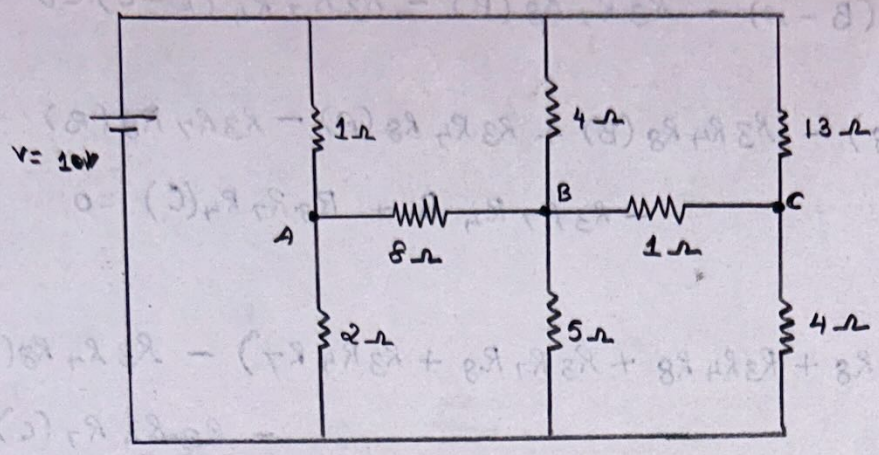


$$\frac{10-A}{1} - \frac{A}{2} - \frac{A-B}{8} = 0$$

$$10 - A - \frac{A}{2} - \frac{A}{8} + \frac{B}{8} = 0$$

$$80 - 8A - 4A - A + B = 0$$

$$\boxed{13A - B = 80}$$

$$\frac{10-C}{13} - \frac{C-B}{1} - \frac{C}{4} = 0$$

$$\frac{40 - 4C}{52} - \frac{52C - 52B}{52} - \frac{13C}{52} = 0$$

$$\boxed{40 = 69C - 52B}$$

$$\frac{10-B}{4} - \frac{B-A}{8} - \frac{B}{5} - \frac{B-C}{1} = 0$$

$$\frac{20 - 3B + A}{8} - \frac{B - 5B + 5C}{5} = 0$$

$$(100 - 15B + 5A) - 48B + 40C = 0$$

$$\boxed{100 = 63B - 5A - 40C}$$

In to canonical form

$$R_2R_7(V-A) - R_1R_2(A-B) - R_1R_7A = 0$$

$$R_2R_7V - AR_2R_7 - AR_1R_2 + BR_1R_2 - AR_1R_7 = 0$$

$$\boxed{V(R_2R_7) = A(R_1R_2 + R_1R_7 + R_2R_7) - B(R_1R_2)}$$

$$R_4 R_7 R_8 (V-B) - R_3 R_4 R_8 (B-A) - R_3 R_7 R_8 (B) - R_3 R_7 R_4 (B-C) = 0$$

$$R_4 R_7 R_8 (V) - B(R_4 R_7 R_8) - R_3 R_4 R_8 (B) + R_3 R_4 R_8 (A) - R_3 R_7 R_8 (B) - R_3 R_7 R_4 B + R_3 R_7 R_4 (C) = 0.$$

$$R_4 R_7 R_8 (V) = B(R_4 R_7 R_8 + R_3 R_4 R_8 + R_3 R_7 R_8 + R_3 R_4 R_7) - R_3 R_4 R_8 (A) - R_3 R_4 R_7 (C)$$

$$\frac{V-C}{R_5} - \frac{C-B}{R_8} - \frac{C}{R_6} = 0$$

$$R_6 R_8 (V-C) - R_5 R_6 (C-B) - R_5 R_8 C = 0$$

$$R_6 R_8 (V) = C(R_6 R_8 + R_5 R_6 + R_5 R_8) - R_5 R_6 (B)$$

Given a set of approximate $x, y \rightarrow$ plane (Best fitting line in least square sense.

$ax+b=y$ (std. line), Compute a & b.

len_seg $\rightarrow x, y$ (row vectors) $\rightarrow$ Coordinate points.

$$ax+b=y$$

$$a.*\begin{pmatrix}\vdots\end{pmatrix} + \begin{pmatrix}b\\ \vdots\end{pmatrix} = \begin{pmatrix}\vdots\end{pmatrix}$$

$$\begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ x_3 & 1 \\ \vdots & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \end{pmatrix} \rightarrow 1 \text{ Sol}$$

$$a.*(x_1\ x_2\ x_3) + b.*(1,1,\dots) = (y_1, y_2 \dots) \rightarrow 2 \text{ sol.}$$

$1\times m \qquad 1\times m \qquad 1\times m$

$$\begin{pmatrix} x_1 & x_2 & x_3 & \dots \\ 1 & 1 & 1 & \dots \end{pmatrix} \begin{pmatrix} a & b \end{pmatrix} = (\dots)_{1\times m}$$

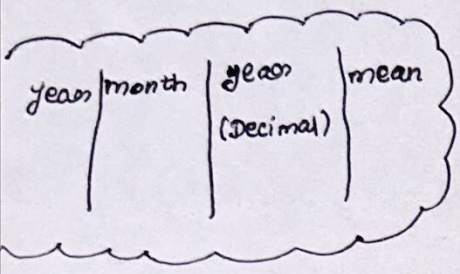
Live Scripts

* Works: only with scripts. → Very useful in create small & impactful projects
* No workspace: Commandline level. (check Command line: Workspace variables)
* can't have i/p & o/p arguments.

Mauna Loa Observatory - Hawaii

* Co_2 Concentration: collected over 50 years. (In atmosphere) – NOAA (org.)

https://www.esrl.noaa.gov/gmd/ccgg/about/co2_measurements.html.



open .xls

```
Co2 = xlsread('CO2 Mauna Loa.xlsx'); → load
year = Co2(:,3); → 3 column alone
Co2 = Co2(:,4); → 4 column alone.
plot(year, Co2, 'g'); grid on; → For grid
title('Monthly Co2 Concentrations');
xlabel('year'); ylabel('Co2 (ppm)');
```

'periodic' – upwards!

Zoom manually

click magnifier → update code → copy paste.

Add description: text.

---

* Live scripts - not a new feature
* we can run normal scripts & even functions → as sections. (%%)

Live script: Lot more formatting freedom – text, images, hyperlinks.

'Format'

'Smooth & plot' – the data

Smooth data → function introduced in 2017

window = 12 → (12 points from the original data would be combined to produce each new point in the smooth data)

* smoothdata (Co2, 'movmean', window)

↳ moving mean (Roughly: moves center of the window from beginning to end).
(method)

```
window = 12;
Smoothed = smoothdata (Co2, 'movmean', window);
Plot (year, Co2, 'g', year, Smoothed, 'K'); grid on;
title (sprintf ('monthly Co2 concentration and %.d- month Smoothing', window));
xlabel ('year'); ylabel ('Co2 (ppm)');
legend ('monthly', 'Smoothed', 'location', 'best');
```

growth: very much like an exponential → try exponential fit!

Matlab's curve fitting toolbox → cftool

model = fittype ('a+b* exp(n *(year - 1958))', 'independent', 'year');

Formula

$a + be^{n(year-1958)}$ → growth (exponential)

year → independent variable. a, b & n → constants determined by MATLAB.

* o/p → model to be fit

fitoptions ('Method', 'NonlinearLeastSquares', 'StartPoint', [0,0,0]);

method for choosing best values of a, b & n

Initial values of a, b, n to use in an iterative method to have sum of minimal squared differences.

---

fit (year, Co2, model, options)

Independ, Dependent → variables

model, option → previous.

```
model = fittype ('a+b*exp(n*(year - 1958))', 'Independent', 'year');
options = fitoptions ('Method', 'NonlinearLeastSquares', 'StartPoint', [0,0,0]);
Co2_fit = fit (year, Co2, model, options);
Co2_model = Co2_fit.a + Co2_fit.b * exp (Co2_fit.n *(year - 1958));
plot (year, Smoothed, 'k', year, Co2_model, 'r--'); grid on;
title ("Co2 Smoothed and Co2 model");
xlabel ("year"); ylabel ("Co2 (ppm)");
legend ('Smoothed', 'Model', 'Location', 'best');
```

model → used to fit

options → helps in designing model

fit → uses model, options to fit the curve

struck 'Co2_model' → uses a, b, n → To calculate values of model for even year. (element of variable year)

'Now we have model → Co2_model' → plot

* Plotting → Smoothed & fitted plots.

'Perfect fit'

* Root square mean error ≈ 2.2 (Avg error: 2 parts per million) → Good fit!

In usual script: second graph - replace first!

But Livescript: No!

'Revisits curve fitting toolbox'!

Extrapolate:

```
future = year (end) : 1/12 : 2050;
Co2_future = Co2_fit.a + Co2_fit.b * exp (Co2_fit.n *
                                          (future - 1958));
plot (year, Co2, 'g', future, Co2_future, 'r--'); grid on;
title ("Co2 measured and projected to the future");
```

xlabel (. . .)

legend (. . . .)

Convert non-live to live script

* Table of Contents: Insert → Table of Contents (Automatically created) – live links.

*

Typeset equation.

Insert → equation

$$Co_2 = a + be^{n(year - 1958)}$$

Highlight — B Bold

CO_2 → Index → $CO_{[2]}$

'Hide code' → No code will be shown in document.

Live script from non-livescript

Publish → html

Save as → .m to .mlx (That's it)

For installed version: Save as → .mlx (open new one)

* 2016 → until livescript not invented!

---

* 2050 → 500ppm ↑
* Act !

Errors handling:

* Behave in unexpected way! – handle them well

* diff (end+1) → out of index.

Run → debug

"Not a good way! when others (know nothing) uses it!"

Not enough info: error message: (long time: forget what's inside the hood)

Solution:

```
function h = my_harmonic(n)
    if ~isscalar(n) || n < 1 || n ~= floor(n)
        error('positive integer input expected...');
    end
    h = 1;
    for ii = 2:n
        h = h + 1/ii;
    end
end
```

```
>> my_harmonic(0)
    error!
>> my_harmonic(2.3)
    error!
>> my_harmonic([2,3])
    error!
```

→ exits entire program!

End of chain: throws an error!

```
function h = harmonic_chain(n)
    h = sub_harmonic(n);
end
function h = sub_harmonic(n)
    h = my_harmonic(n);
end
```

error!

'useless'

"Tells position of stack" – when error is realized.

Assumes a hours of calculation! → Boom! (waste of time!)

Avoid! → Convert bad i/p to good i/p.

```
if ~isscalar(n)
    n = n(1);
```

→ picks first element of array.

* `n=max(1,round(abs(n)));` → round to integers

Exception handling!

* code: designated as being executed only when an exception occurs in that part of the program.

```
try
    h = my_harmonic(n);
catch
    h = [];   → If error do this (catch)
end
```

C++, Java also same keyword

`warning('wrong input provided');` → provide info: location executes afterwards.

```
try
    out = X(r,n) * Y(n,c)
Catch ME
    If isequal(ME.identifier, 'MATLAB:badsubscript')
        [xsize] = size(X); [ysize] = size(Y);
        m1 = sprintf('Accessed X(%d,%d) and Y(%d,%d), but \n',
                                        r,n,n,c);
        m2 = sprintf('Sizes = X(%d,%d) and Y(%d,%d),', xsize,
                                        ysize);
        error([m1,m2])
    end
end
```

'STILL cause error' – detailed error (Catch)

ME ? → when an error is caught → that variable after catch is assigned an object that has info about the error! (Kind of error – thrown),

* message - that would have been printed - if no error

Stored in property of the object.

---

* ME → struct of fields

identifier: name of one of those fields.

↳ Accessing property (field) by dot operator.

Identifier: 'MATLAB:badSubscript' ⟶ By running error (found)!

List of Identifiers: exist but not given by MATLAB:

make errors - find yourself.

>> MException.last

ans =

⟶ displays most recent one!

: Properties

:

:

MException.last ⟶ reference to a struct (object property)

Identifier: '' ⟶ why? (Shows only info for most recently uncaught error)

But now: thrown inside error function → caught! Identifiers '' !

'use meaningful identifiers'

Instead of:

error([m1, m2]); →

error('JMF:element_prod:badSubscript', [m1, m2]);

JMF → string (used) - nothing

↓

message!

Another way – Built in function

```
m1 = ''
m2 = ''
myE = MException('JMF:'' '', [m1, m2]);
throw(MyE);
```

---

* Rethrow → error

rethrow(ME); → error not caught
(So it get caught)

Version 1

```
function out = element_prod(X, Y, r, n, c)
% Element_Prod two element product for matrix mult.
% element_prod(X, Y, r, n, k) = X(r, n) * Y(n, c) {n-index}.
% must be summed over to produce the element at row r &
```

Column c of the product of matrices X & Y.

```
% version 1
out = X(r, n) = Y(n, c);
end
```

matmul.m

```
function C = matmul(A, B)
[rowA, colA] = size(A);
[rowB, colB] = size(B);
if ~isMatrix(A) || ~isMatrix(B)
    error('Function matmul requires matrices...');
elseif colA ~= rowB
    error('Inner dimensions must agree!');
end
C = zeros(rowA, colB);
for ii = 1:rowA
    for jj = 1:colB
        for kk = 1:colA
```

```
C(ii, jj) = C(ii, jj) + element_prod(A, B, ii, jj, kk);
        end
    end
  end
end
```

Product: changing message (version): Annoying!

Known message!

Version 2

```
try
    out = X(r, n) * X(n, c);
catch ME
    if strcmp(ME.identifier, 'MATLAB:badsubscript')
        [xsize] = size(x); [ysize] = size(y);
        m1 = sprintf('Accessed ...')
        m2 = sprintf('...      ')
        error([m1, m2]);
    end
end
```

Particular error caused!
Version: 1
(By running)!

Version 3 : No identifier

```
MyE = ....
throw(MyE)
```

→ exception thrown (prev. case: not thrown as ', identifier)

Rethrow

```
rethrow(ME);   % error - not caught = error.
```

Assertion

* Assumptions: about variables in code — if that violated - MATLAB let you know.

```
function assert_example
    % do some computation
    x = abs(randn)
    % do some more computation
    assert(x >= 0)
    % keep working
end
```

→ helps to exterminate bugs while developing!

So once done - remove/comment it?

Assumption: important to proper behaviour of the code!

Introduce assertion!

we are using abs()

how x will be negative?

do some more → if makes x negative!

o/p

Assertion failed.

'I'm sure the Tigers cages locked or
I'm sure this pool I'm about to dive in head first is deep enough'!

---

save → time (normally don't throw error!
if it does: no worries!
Assertion is there!)

Error handling - Summary!

* Exception: error thrown in designated portion of code
* Handler: runs when error is thrown: instead of error & exit

'try - catch block'

* Identify errors with catchME, MException.last
* Throw errors (detected): throw function
* Rethrow error with rethrow function (undetected)
* assert function to identify incorrect assumptions!

Algorithmic Complexity

* 1 + 1 = 2

1 + 1 + 2 = 3

1 1 2 + 3 = 5

⋮

Recursive call → lot of recursive calls!

fibo(50)

Running forever!

'Busy with recursive calls'!

↓

fibo(50) → fibo(49) + fibo(48)

fibo(49) → 48 47

fibo(48) → 47 46

'Twice' → called!

Use persistent:

```
function [f, cnt] = fibocnt(n)
    persistent Count;
    if isempty(Count)
        Cout = 1;
    else
        Count = Count + 1;
    end
    if n <= 2
        f = 1;
    else
        f = fibocnt(n-2) + fibocnt(n-1);
    end
    cnt = Count;
end.
```

→ Count (retains even if it is local)

Note: A persistant variable can't be an o/p argument!

fibocnt(3) ⟶ 3 calls !

(5) ⟶ 9 calls !

25 → 150049 calls!

35 → 18604978 calls!

"Need to clear – persistant variable" → since it retains!

↓

"Not a elegant solution"

Algorithm: wrong approach!

① Non recursive!

② modify!

```
function f = fibo_last(n)
if n <= 2
    f = ones(1, n);
else
    f = fibo_last(n-1);
    f = [f  f(end-1) + f(end)];
end
end
```

[1 1] ⟶ [1 1 2] ⟶ [1 1 2 3] ⟶ [1 1 2 3 5]

go to deep – then climb

2 ⟶ [1 1]

3 ⟶ [[1 1], 2] ⟶ [1 1 2]

4 ⟶ [1 1 2 3]

30 → 57 rec calls!

Reverse

* V = 1:10;
* V(end:-1:1);

flip(V)

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

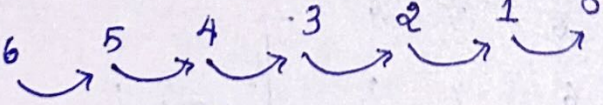
• 'i = 1', swap ①, i → i++, shift

①, 2(i) → i++, shift

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

Swap → a[2] & a[0]

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



---

* time_it only evaluates — functions with no arguments

» timeit(@() my_flip(1:1e4))

↓

function handle to anonymous function!

ans =

0.1402 (Accurate)

* tic toc → Runs multiple times → returns average!

Our algorithm:

1000 → 0.1412
100000 → 14.19
] Factor: 100 (Size)
Time: 100! (Wrong)

20000 → 0.56
40000 → 2.27
] Time ↑ by 4

'Quadratic Relationship'

---

2 nested loops: (n-1) (1 to n-1)

$(n-1)\left(\frac{1}{2}n\right)$ → Average! $\approx \frac{1}{2}n - \frac{1}{2}n^2$

'quadratic'

---

million → 20 minutes

10million → half a day +.

Better solution - Swap!

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

Tada!

$\boxed{\frac{1}{2}n}$ ∴ Swapping!

1 million → 0.0040
10 million → 0.0485
] great (1 Lakh: 14 seconds → previous algo!)

---

flip( ) → matlab inbuilt

```
>> timeit(@() flip(1:1e7))
ans=
    0.0285
```

→ Better than ours!

↓

why? → Low level features [not even implemented in MATLAB] like many built in functions

Improvement - Constant time!

$t(N) \approx C\log(N) \longrightarrow O(\log N)$

$t(N) \approx CN^2 \longrightarrow O(N^2)$

$t(N) \approx CN \longrightarrow O(N)$ → Linear

$t(N) \approx C \longrightarrow O(C)$

Complexity theory/Analysis of Algorithm - Field of Algo..

Any problems: do better than N:

Sum of N positive integers

$Sum(1:100) = 1+2+3+\cdots+100 = 5050$

Carl Friedrich Gauss:

$1 + 100 = 101$
$2 + 99 = 101$
$\vdots$
$50 + 51 = 101$

50 such sums

$50 \times 101 = 5050$

$$\frac{n(n+1)}{2}$$

'Constant time' - Independent of size!

---

* nth Fibonacci formula: Using golden ratio!

* Searching: find(v, 1) → returns indexes

'Binary search' → much faster! $\log_2(\text{Size})$

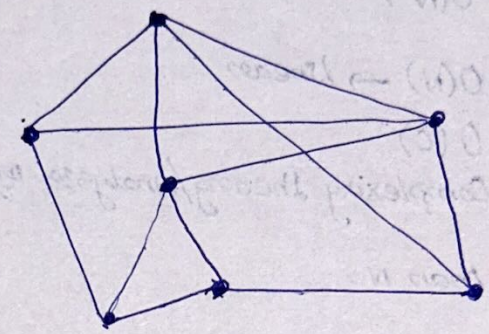
1024 → 10 steps

$t(N) \approx c N \log(N) \longrightarrow O(N \log N)$

Traveling Salesperson problem!

many cities - many paths - shortest path - visit each city & return home!

Simplest worst case: when shortest road connecting each pair of cities doesn't pass through any other cities.



* n! factorial steps!
(See all solutions)

↳ 'Simple solution - catching eye'. (worst than n^2)

Impractical for even 20 digits!

'Best (still known) : 2^n

$t(N) \approx c\, 2^N \Rightarrow O(2^N)$

N^2 = 100 steps, 2^{10} = 1000 steps

2^{20} = million

2^{30} = billion

'Many real world problem': follows this: have no solution!

better than 2^N!

If one can solve this: (Sales man problem)!

Able to solve other problems too - efficiently!

'Fed Ex, DHL, UPS - postal service' - minimize fuel, work etc...

'changing base is just equal as changing constant c'

$t(N) \approx c \log(N)$ → ∴ No bases included.

'No effect on complexity

Matrix multiplication — $O(n^3)$

$M \times M^2 = M^3$
(add) (m×m - o/p size)

∴ I/p → $N = 2M^2$ (I/p = M^2, M×M = ($2M^2$) size)

o/p → M^2 size.

$M^3 = CN^{3/2}$

$t(N) \approx C N^{3/2} \Rightarrow O(N^{3/2})$

1969: Strassen → $N^{2.8}$ (Power) → Complicated steps.

Matlab overfunction → (700 x 700) by (700 by 700)

1.4968 seconds

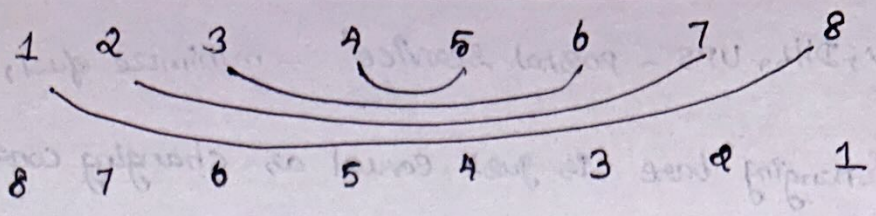
Matlab inbuilt → 0.0044

why? 340 x faster!

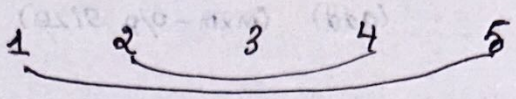
Now best: $M^{2.373}$ - Slower than Current M^3 implementation (Complicated steps)

MATLAB Algo → not written in MATLAB

(uses microcontroller - multi programming)



Skaark = 1	Skaark = 2	Skaark = 3	Skaark = 4	Skaark = 5
end = 8	end = 7	end = 6	end = 5	end = 6
↓	↓	↓	↓	↓
Swap.	Swap	Swap	Swap	If (Skaark > end) return.



Skaark = 1	Skaark = 2	Skaark = 3
end = 5	end = 4	end = 3.

Perev solutions → have whole V → ip array in heap.

Say: 10000 elements → 5000 × 10000 elements → in heap (depth)

∴ 5000 recursion calls!

Solution:

1 2 3 4 5 6

fun [2 3 4 5 6] 1]

fun [(3 4 5 6) 2 1]

∴ only - few elements goes inside recursion.

```
function out reverse (in)
if (length(v) <= 1)
    Out = in;
else
    out = [reverse (in(2:end))  in(1)];
end
```

10	9	8	7	6	5	4	3	2	1
5	4	3	2	1	10	9	8	7	6
2	1	3	5	4	*	6	8	10	9
1	2	3	4	5	6	7	8	9	10

---

len odd →

```
a = IP(1 : len/2);
b = IP(len/2 + 1);
c = IP(len/2 + 1 : end);
V = [c b a]
      reversal
```

5/2 = 2

until length → 1.

end

len even →

```
a = IP(1 : len/2)
b = IP(len/2 + 1 : end);
V = [b a]
     reverse
```

Fibonacci problem

* we used → pass by ref variable to profile (how many calls)
* Another way — use trace → vector → having elements each time — recursive call.

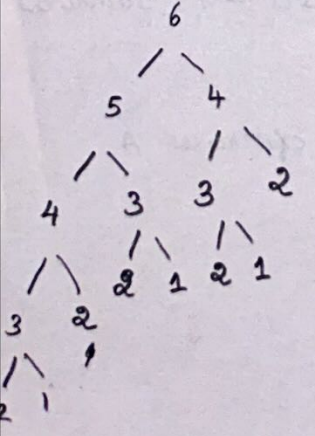
6, []

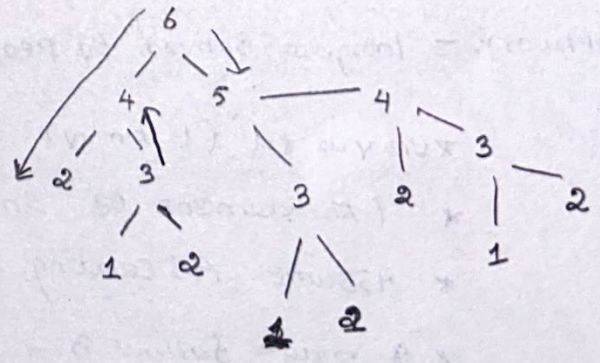
6

①

fibo(n-2) + fibo(n-1)

5

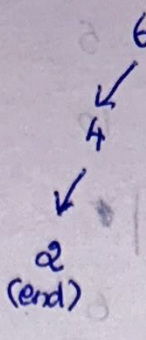


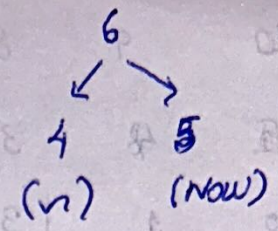


{ 6, 4, 2, 3, 1, 2,

5, 3, 1, 2, 4, 2, 3, 1, 2 }

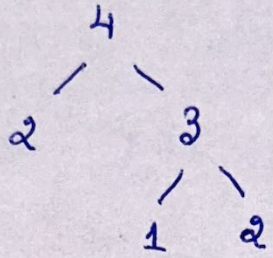
why → since





Continues!

1 1 2 3



4 2 3 1 2

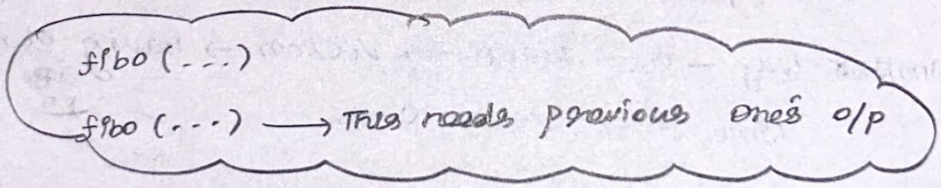
fibo (4, [])

fibo (2, [4]) + fibo (3, [4 2]) ⟶

[4 2]

1 2 [4 2 3 1 2]

[4 2 3 1]



Social network – longest subset of people – each follow each other

* Unique id (1 to N)
* ith element of sn → vector (list of id – i follows)
* Assume: Ascending order.
* A may follow B – B may/may not follow A.

* Profiling - how much time - spent by various parts

* Built in - profiler (function)

  * shows time (most times), how many times each func. called?

  * matlab online - don't have this feature.

Improve algorithm:

* Any unnecessary work.

(same length)

```
function A = rooting_V1(v, w)
    A = zeros(length(v), length(w));
    for ii = 1: length(v)
        for jj = 1: length(w)
            A(ii, jj) = nthroot(v(ii), ii) * nthroot(w(jj), jj);
        end
    end
end
```

$$A(ii,jj) = v(ii)^{1/ii} * w(jj)^{1/jj}$$

$$A(2,3) = v(2)^{1/2} * w(3)^{1/3}$$

* nthroot ⟶ Inbuilt function

* n^2 → Algorithm

```
tic; rooting_V1(rands(1e6, 1, 1e3); rands(1e6, 1, 1e3)); toc;
```

7.9233 seconds! (1000 elements)

$10000 \longrightarrow 100 \times 7.9233 = 870$ seconds

Unnecessary works.

each time calculating $A(ii)^{1/ii} \longrightarrow \therefore$ ii times

Unnecessary!

modify

```
for ii = 1 : length(V)
    x = nthroot(V(ii), ii);        ⟶ 1 calculation! Instead of length(W) times!
    for jj = 1 : length(W)
        A(ii, jj) = x * nthroot(W(jj), jj);
    end
end
```

Now: 4.008896 seconds.

* Is equal $(A_1, A_2) = 1$ (yes)! → Answer of prev, new version same!

---

modify

```
rw = zeros(1, length(W));
for jj = 1 : length(W)
    rw(jj) = nthroot(W(jj), jj);      ⟶ Not runned 1000 times!
for ii = 1 : length(V)                  ∴ Saved
    x = nthroot(V(ii), ii);
    for jj = 1 : length(W)
        A(ii, jj) = x * rw(jj);
    end
end
```

Now: 0.031786 (100 times faster)

* Is equal (A_1, A_3)!

Now: 2000 times instead of million times by already doing nthroot for ii, jj.

---

modify - 1 million multiplications!

Still $-n^2$ algorithm?

```
tic; A3 = rooking_V3(V, m); toc
```

1.500814 seconds (instead of 12 minutes).

→ save time!

Precalculating! → General practice

* Trade off b/w memory & time (But small memory)

one vector of length n → 2 orders of magnitude time save

'Cool'!

* A = zeros(length(v), length(w)); → pre allocation!

'Saves time'! / Some case: Default! (no value : 0) or -1.

without pre allocation!

% A = zeros(length(v), length(w));

↓

410.850459 seconds (7 minutes)

↓

Saved by pre allocation

Profiler

>> profile on

>> A2 = rooting_v2(v, w);

>> profile viewer → window opens. (nth root → 1001000 times) 5.808 seconds with livelink.

rooting_v2 → line by line (time), info!

x = nthroot(v(99), 99); → 0.2% (1000 times)

↓

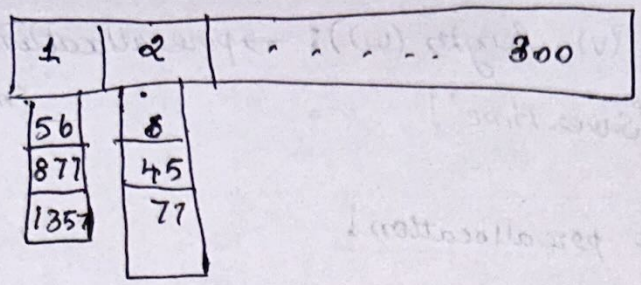
line-time-boo chaos

only in Installed version?

unnecessary work!

*speed up – without extra memory! (avoid unnecessary work)!

ID: 1 to N



which two sets of people – have lots of people in common!

'Cell array'

* 2 & 2998 → Have 11 common people

* load follows

whose

follows 1×3000 399256 bytes cell.

---

* 4 1/2 million pairs to look up

* Intersect ( )
↳ finds Common! (In two vectors)!

worst case: Everybody follows everybody!

4 million pairs

Time save: If we found (say): 1 & 2 : Have 5 persons

Leave any two having less than 5 → SKIP

(save time)

118 seconds → 1.7 seconds

Caution: If every person has atleast 5 → No need for over if case!

'Skip — 'continue' the loop' — avoid unnecessary work!

Focus on Compatationally intensive codes!

Vectorization & other speed ups

* 'Implementation' — makes difference

MATLAB & Implicit looping:

* MATLAB — invented to work with matrices & arrays!
* Incorporate operators, notions — directly in MATLAB.

Implicit looping — everywhere!

---

* add, sub, mul, div → doesn't require any loop (In built)

MATLAB & Implicit looping

* Implicit → By lowlevel methods (Inside there: really fast)
  loops

Any function operates on entire vector → /entire array

'Vectors command' → using Implicit looping

Avoid explicit loops by vectorization.

↓

Translation of code from version uses explicit looping to one that uses a vectors command.

other languages — lack built in support for vectorization — even it has power of Implicit looping!

↓

Dramatic speedups — less prone to errors & easy to debug!

Rooting_ more modified!

```
function A = rooting_v4(v, w)
    A = zeros(length(v), length(w));
    rv = nthroot(v, 1:length(v));
    rw = nthroot(w, 1:length(w));
    for pp = 1:length(v)
        for ss = 1:length(w)
            A(pp, ss) = rv(pp) * rw(ss)
        end
    end
end
```

→ 1.8 seconds!

---

timeit(@() rooting_v3(v, w))

↳ ∴ timeit doesn't take i/p functions!
Anonymous function!

What we are doing: $A(pp, ss) = rv(pp) * rw(ss)$ → matrix multiplication

```
function A = rooting_v5(v, m)
    rv = nthroot(v, 1:length(v));
    rw = nthroot(w, 1:length(w));
    A = rv' * rw;
end
```

0.2050 seconds!

↓

easy to understand & implement!

n×n output

(1×n) × (1×n)

∴

(n×1) × (1×n) =

(n×n)

---

time comparison & profile → using logical indexing.

* Nested loop
* Indexing using logic.

```
function A=small2zero_v1(A,limit)
    for ii=1:size(A,1)          ---> Rows
        for jj=1:size(A,2)      ---> columns
            if A(ii,jj)<limit
                A(ii,jj)=0;     ---> 1.3503 seconds
            end
        end
    end
end
```

```
>> A=sprand(1e6,1e4);
>> timeit(@() small2zero_v1(A,50))
ans=
    1.3503
```

modify

```
function A = small2zero_v2(A,limit)
    A(A<limit)=0;               ---> 0.6662 !
end
```

timeit ---> only expressions are allowed !

```
tic; A(A<50)=0; toc;
```

Profiler

```
>> profile off
>> profile on
>> small2zero_v1(A,50)
>> profile viewer
```

ff ---> 7.65 seconds
end ---> 3.701
end ---> 3.563

why slow — when profiled ?

* MATLAB optimizes the code
* MATLAB converts to translated (compiled)

MATLAB ---> [JIT Compiler (with optimization)] ---> compiled code

Just in time results (Compiler)

↳

* optimization results - guarantees not to change results.

* Improved every year - MATLAB engineers improving it!

* optimization - order of some operations, looping may be changed almost as implicit looping in the vectorized code! (But never be as good as vectorized) but trying to be!

* optimization methods: secret (changable: programmers shouldn't take advantage).

---

* Compiler saves translated in local file. (internal)
* So when (restart MATLAB) - until edit) → no need to compile again!

Say: persistent (clear when restarted!)

---

* Point: can't use profiler to compare the vectorized code with optimized code times!

Profiler: Still useful for non vectorized ones!

---

* Say every element of the array to zero - second element in its own row.

↳ Seems like nested loops only solution (No!)

Built in function:

repmat → replicating matrices (makes copies of matrices & arrays)

```
>> A = randi(99,4,5);
repmat(A(:,2),1,size(A,2))
```

2 column: 71, 55, 99, 11

↓ 1 row, ↓ 5 column

copies