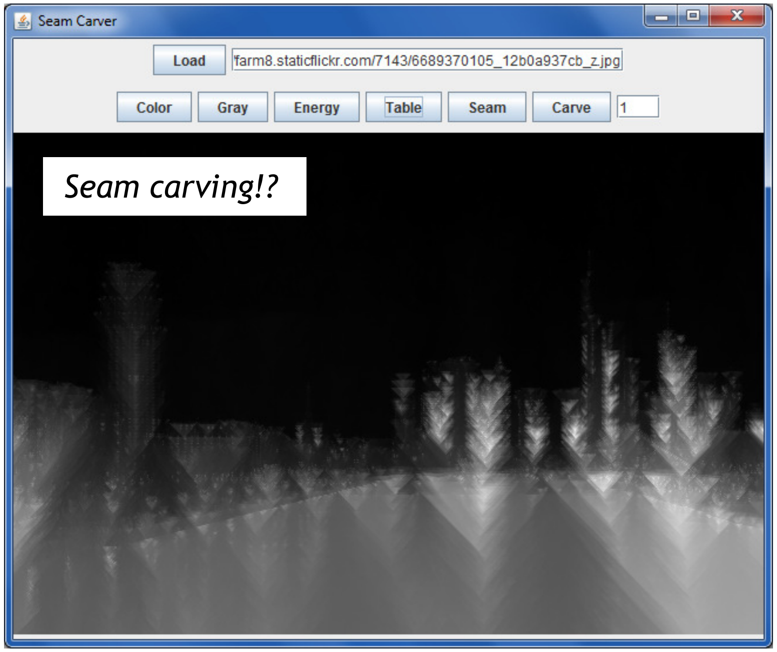
CS 60 today!

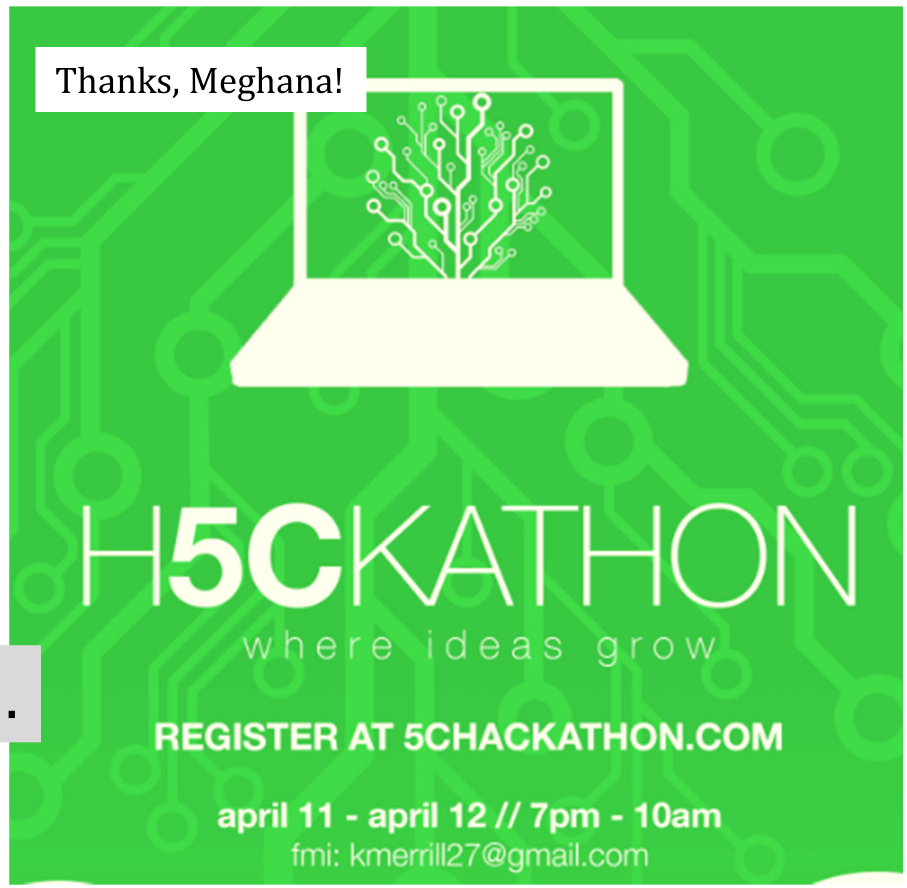


hw8 due Tues., 4/8

(1) BSTNode

(2) min-change (dyn prog)

(3) Floyd-Warshall!



hw9 ~ due Mon. 4/14

hw10 ~ due Mon. 4/21

hw11 ~ due Fri. 5/2

Ex. Cr. project…

Final exam ~ due Thurs. 5/15

Schedule…

Dynamic Programming

(1) Express a problem recursively

(2) ***Do as little as possible***

Who is this talking about?

Dynamic Programming

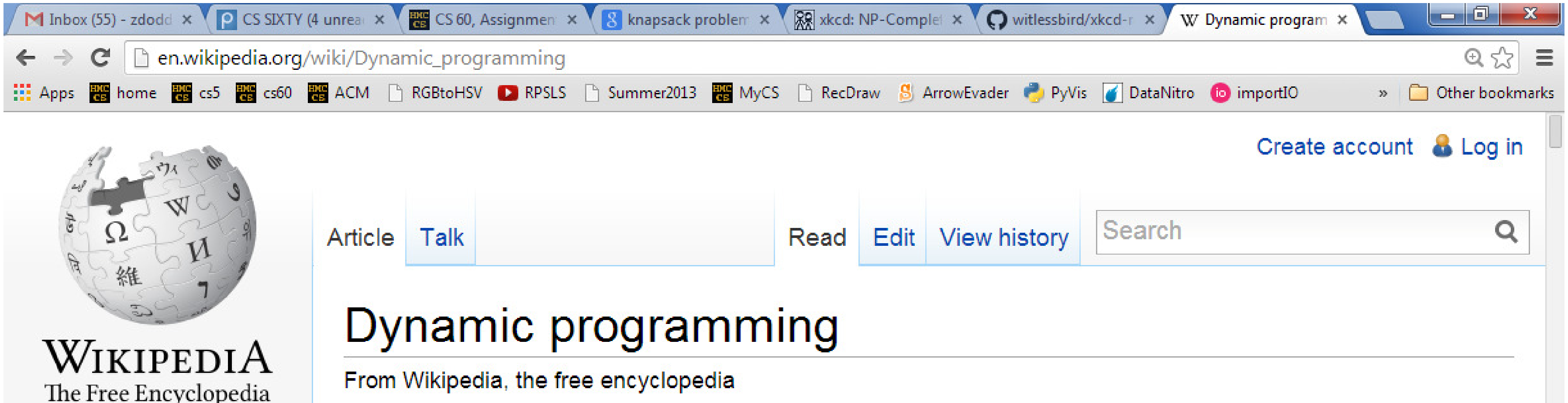
(1) Express a problem recursively

(2) Do as little as possible

for the computer!

If you can make a small set of recursive calls
one-time each, **storing the results in a table –**
you can gain huge big-O and real-time speedups!

Inbox (55) - zdodd
CS SIXTY (4 unrea
CS 60, Assignmen
knapsack problem
xkcd: NP-Complet
witlessbird/xkcd-r
Dynamic program
en.wikipedia.org/wiki/Dynamic_programming
Apps
home
cs5
cs60
ACM
RGBtoHSV
RPSLS
Summer2013
MyCS
RecDraw
ArrowEvader
PyVis
DataNitro
importIO
Other bookmarks
Create account
Log in
Wikipedia
The Free Encyclopedia
Article
Talk
Read
Edit
View history
Search
Dynamic programming
From Wikipedia, the free encyclopedia
For the programming paradigm, see Dynamic programming language.
In mathematics, computer science, economics, and bioinformatics, dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems[1] and optimal substructure (described below). When applicable, the method takes far less time than naive methods that don't take advantage of the subproblem overlap (like depth-first search).
The idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often when using a more naive method, many of the subproblems are generated and solved many times. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored or "memo-ized": the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems grows exponentially as a function of the size of the input.
Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop
Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page
Picture.java
Show all downloads...



> What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying I thought, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is its impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

The word *dynamic* was chosen by Bellman to capture the time-varying aspect of the problems, and because it sounded impressive.[3] The word *programming* referred to the use of the method to find an optimal *program*, in the sense of a military schedule for training or logistics. This usage is the same as that in the phrases *linear programming* and *mathematical programming*, a synonym for mathematical optimization.[4]

DP ~ ***Fibonacci***

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Use-it-or-lose-it is a ***conceptual*** guide:

```
fib(0) = 1
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```

```
fib(10)
```

DP ~ ***Fibonacci***

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Use-it-or-lose-it is a ***conceptual*** guide:

```
fib(0) = 1
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```

`fib(10)`

`fib(9)` `fib(8)`

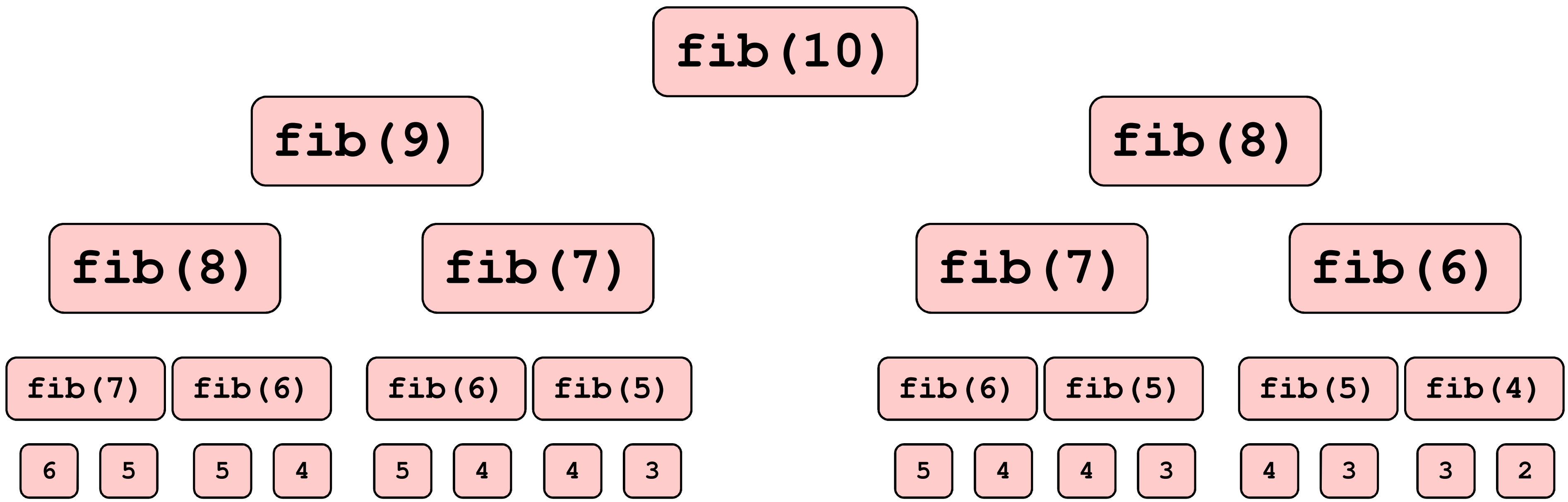
DP ~ *Fibonacci*

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Use-it-or-lose-it is a ***conceptual*** guide:

big-O?

```
fib(0) = 1
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```



but it's way too ***inefficient*** *for large problems!*

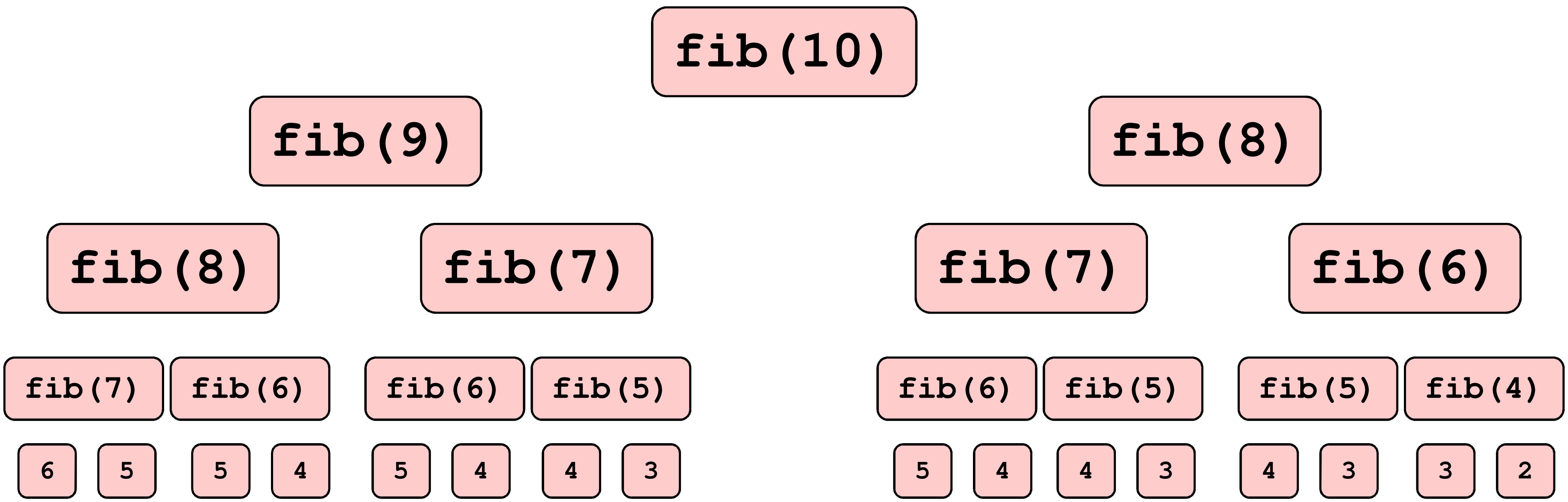
DP ~ *Fibonacci*

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Use-it-or-lose-it is a ***conceptual*** guide:

$O(2^N)$

```
fib(0) = 1
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```



*Is there an identifiable source of the **inefficiency** here?*

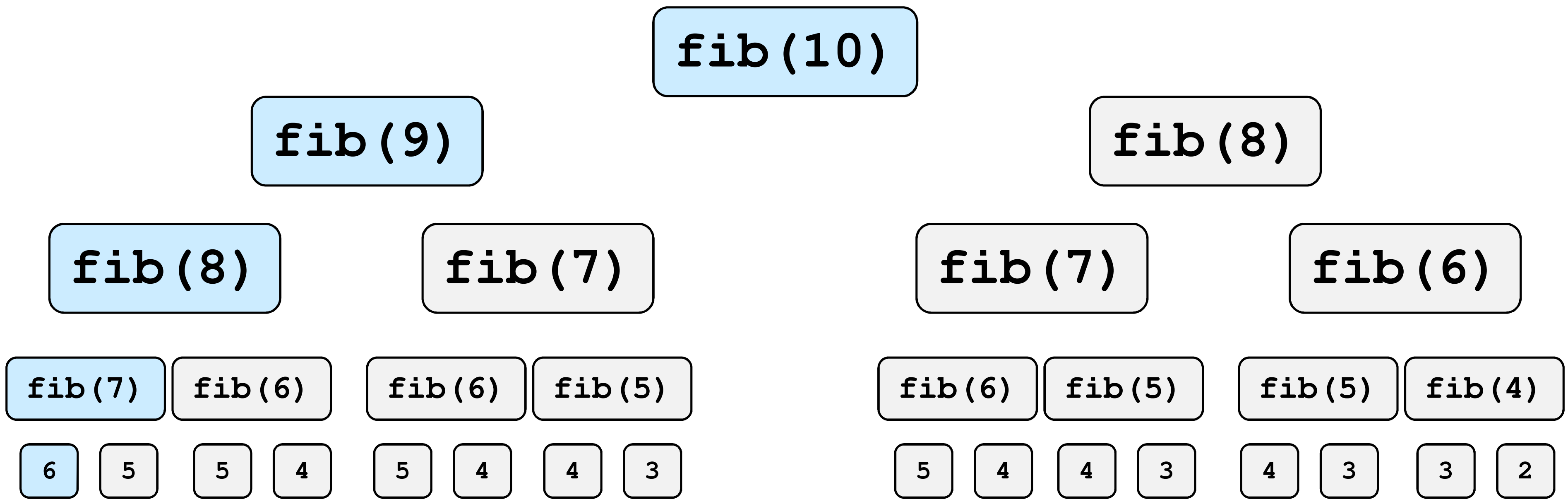
DP ~ *Fibonacci*

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Use-it-or-lose-it is a ***conceptual*** guide:

What's the big-O if we make each call to fib ***only once!?***

```
fib(0) = 1
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```



inefficiency *here ==* ***too many repeated calls!***

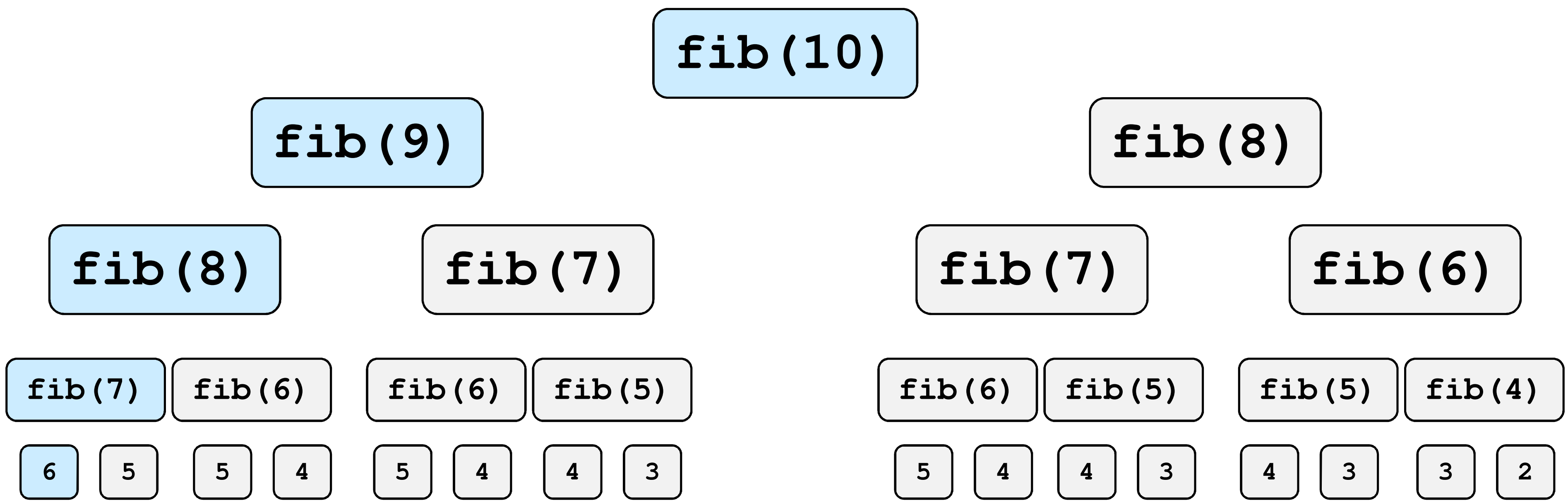
DP ~ *Fibonacci*

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Use-it-or-lose-it is a ***conceptual*** guide:

What's the big-O if we make each call to fib ***only once!?***

```
fib(0) = 1
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```



inefficiency *here* == ***too many repeated calls!***

DP ~ ***Fibonacci***

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Use-it-or-lose-it is a ***conceptual*** guide:

```
fib(0) = 1
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```

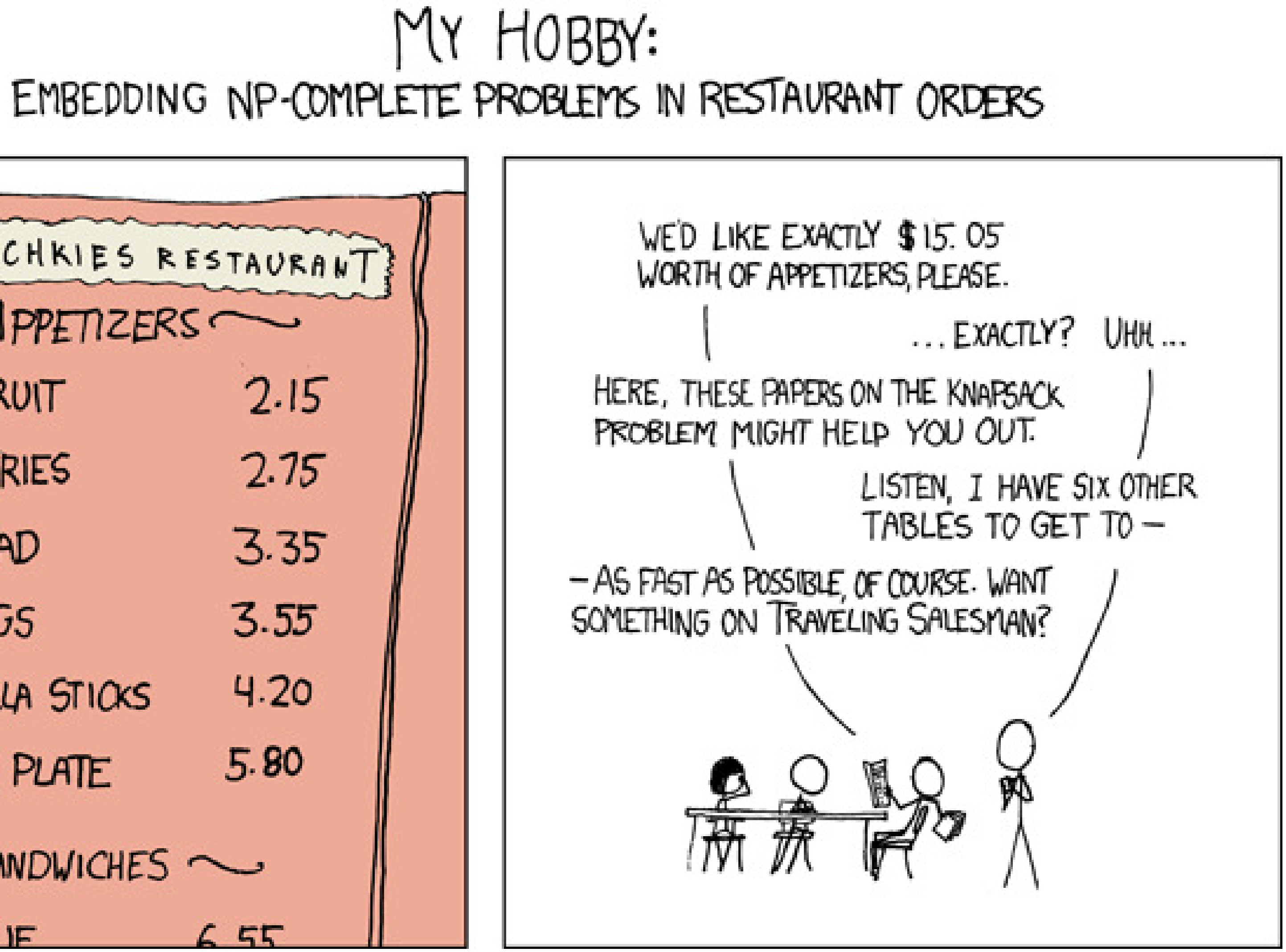
`fib(10)`

and ***dynamic programming*** makes a table of all possible calls:

0	1	2	3	4	5	6	7	8	9	10

...can make ***some*** exponential run-times into polynomial ones

Knapsack...



MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO —
— AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

Knapsack ***contents****?*

Knapsack...

Suppose *a friend* can consume a candy **weight/cost** of ***13***.

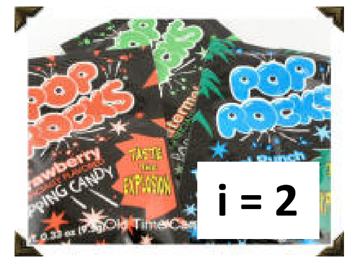
Each candy has different "**values**"

How can you maximize their ***candy-value*** experience?

$v_0 = 100$
$w_0 = 2$

$v_1 = 120$
$w_1 = 3$

$v_2 = 230$
$w_2 = 5$



$v_3 = 560$
$w_3 = 7$

$v_4 = 675$
$w_4 = 9$

Knapsack...

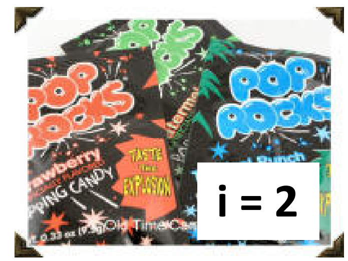
First, think recursively...

`max_val( 13 ) =`

$v_0 = 100$
$w_0 = 2$

$v_1 = 120$
$w_1 = 3$

$v_2 = 230$
$w_2 = 5$



$v_3 = 560$
$w_3 = 7$

$v_4 = 675$
$w_4 = 9$

Try this: use it or use it or use it or use it or use it or lose it

Let's see it in action, too...

Knapsack...

First, think recursively...

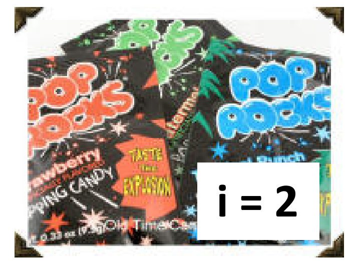
```
max_val( 13 ) =
```

$$\max \begin{cases} \text{max\_val(capacity-1)} \\ \underset{\text{over all possible i}}{\underline{\text{max\_val(capacity-}w_i\text{)} + v_i}} \end{cases}$$

$v_0 = 100$
$w_0 = 2$

$v_1 = 120$
$w_1 = 3$

$v_2 = 230$
$w_2 = 5$



$v_3 = 560$
$w_3 = 7$

$v_4 = 675$
$w_4 = 9$

What's the big-O runtime of this algorithm, as stated?

Knapsack with **DP**

$v_0 = 100$
$w_0 = 2$

$v_1 = 120$
$w_1 = 3$

$v_2 = 230$
$w_2 = 5$

$v_3 = 560$
$w_3 = 7$

$v_4 = 675$
$w_4 = 9$

Minimize work!

Create a table of possible recursive calls.

Compute the results of each one *bottom-up*.

No need for the overhead of the function calls!

CAPACITY

Fill in this table from 0 up to the actual capacity of interest...

0	1	2	3	4	5	6	7	8	9	10	11	12	13

T[cap] = the maximum value obtainable for the capacity cap

Is this table indexed by value or weight?

Does this table hold values or weights?

Which values are "easy" to fill in?

How do we fill in each cell?

(1) Fill in this table of best-results for the knapsack problem:

Name(s) ____________

CAPACITY

0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	100											

VALUE

Fill in this table from 0 up to the actual capacity of interest…

Each cell has **T[cap]**, the maximum ***value*** obtainable for the capacity **cap**

$v_0 = 100$
$w_0 = 2$

$v_1 = 120$
$w_1 = 3$

$v_2 = 230$
$w_2 = 5$

$v_3 = 560$
$w_3 = 7$

$v_4 = 675$
$w_4 = 9$

(2) Estimate the big-O runtime of this table-filling algorithm, in terms of N, the number of items available.

What else does the big-O runtime depend on?

Quiz

(3) How could you *also* keep track of the items to take using a 2nd table with *one item per cell?*

CAPACITY

0	1	2	3	4	5	6	7	8	9	10	11	12	13

One Item

Knapsack!

Why are these important for ***knapsacking*** at HMC ?

hw8pr2: **Change**

we want the ***fewest*** coins that will produce this total

1c is always available

these are the denominations (can use each as many times as we want)

change(27, [1,5,10,25])

3

(1, 1, 25)

Minimizing change...

fewest coins for a desired total

You'll write this two times...

recursively: in Racket

dynamic programming: in Java

Minimizing change...

fewest coins for a desired total

we want the ***fewest*** coins that will produce this total

1c is always available

these are the denominations (can use each as many times as we want)

change(27, [1,5,10,25])

3

(1, 1, 25)

change(27, [1,5,10])

change(18, [1,5,10,12])

change(18, [1,5,9,10,12])

we want the ***fewest*** coins that will produce this total

out of these denominations (we can use as many times as we want)

in Racket: *recursion use-it-or...-or-lose-it*

change(18, [1,5,9,10,12])

in Java: *dynamic programming!*

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fill in this table from 0 up to the actual amount of interest...

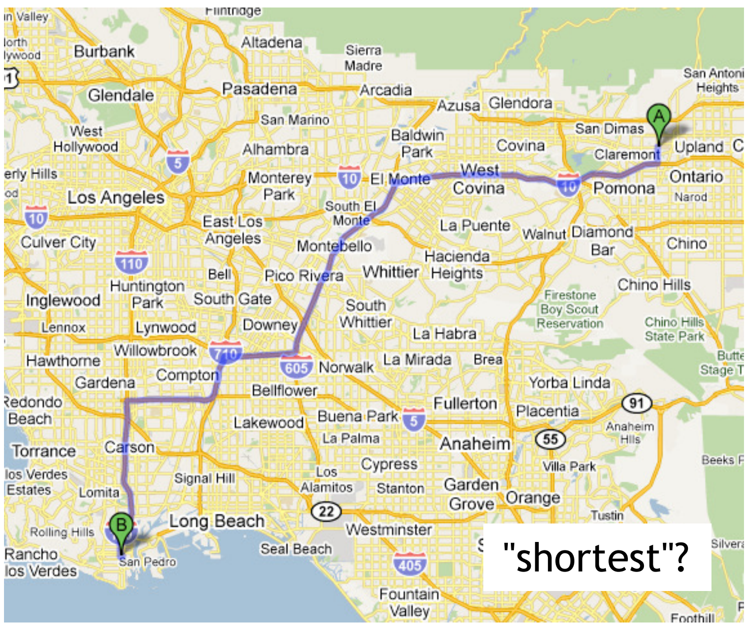
Each top cell has **Min[amt]**, the minimum # of coins that can create **amt**

16	17	18

Try filling out this table by hand *– this is the DP algorithm!*

Shortest paths!

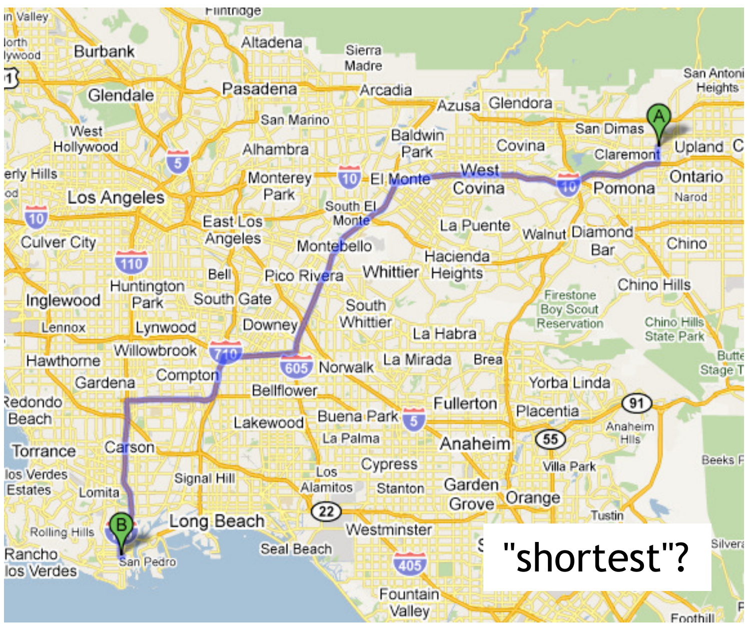
Suppose we'd like the shortest route from any city to any other.

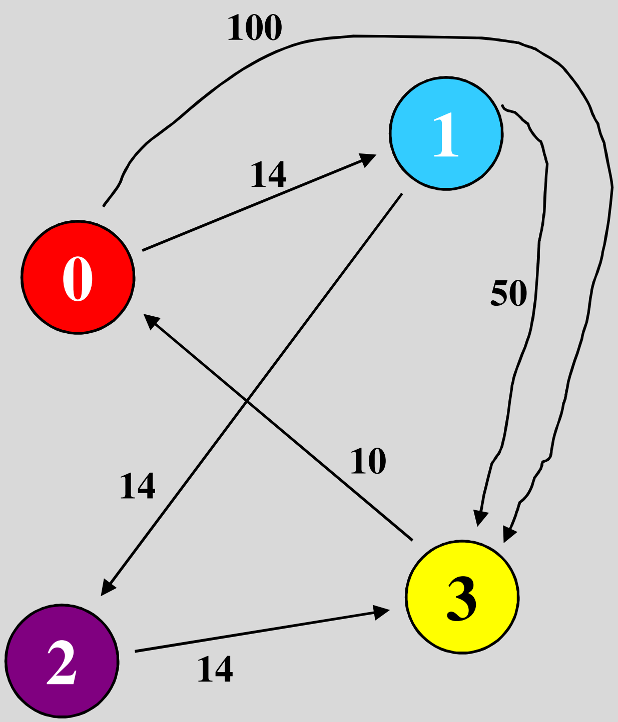


didn’t we solve this already?

Shortest paths!

Suppose we'd like the shortest route from any city to any other.



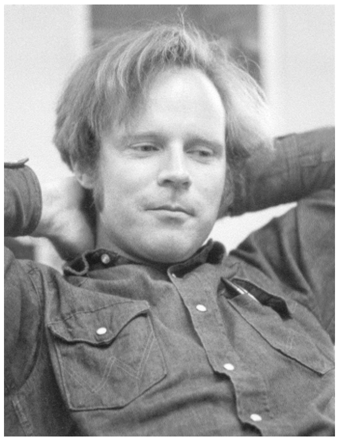


Computationally, many path-planners represent their maps as **GRAPHS**

Racket: min-path

What's the big-O running time?

hw8pr3: Floyd-Warshall



Robert Floyd

From Wikipedia, the free encyclopedia

For other persons named Robert Floyd, see Robert Floyd (disambiguation).

Robert W Floyd (June 8, 1936 – September 25, 2001) was an eminent computer scientist.

Born in New York, Floyd finished school at age 14. At the University of Chicago, he received a Bachelor's degree in liberal arts in 1953 (when still only 17) and a second Bachelor's degree in physics in 1958.

Becoming a computer operator in the early 1960s, he began publishing many noteworthy papers and was appointed an associate professor at Carnegie Mellon University by the time he was 27 and became a full professor at Stanford University six years later. He obtained this position without a Ph.D.

His contributions include the design of Floyd's algorithm, which effici...

All-pairs shortest paths!

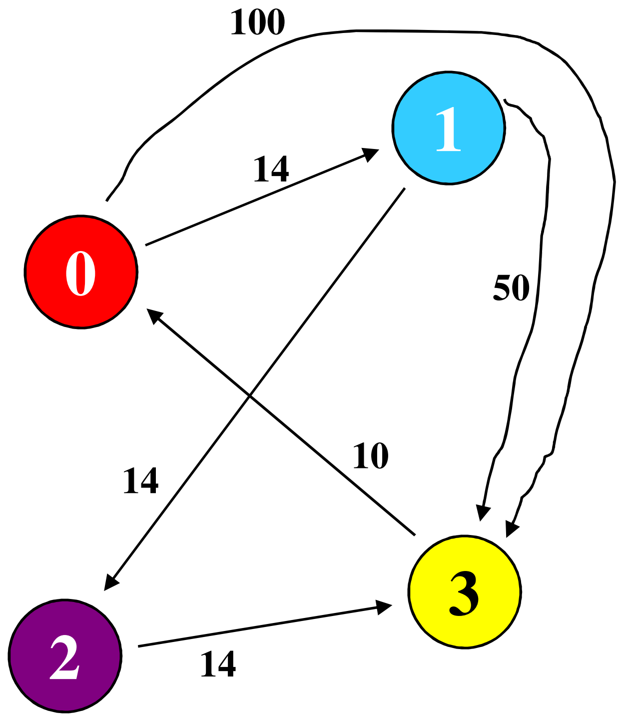
Warshall's algori...

There is an interes... ...losure algorithm, now known as Warshall's algorithm ... ...gue at Technical Operations bet a bottle of rum on who first could determine ... this algorithm always works. Warshall came up with his proof overnight, winning the bet and the rum, which he shared with the loser of the bet. Because Warshall did not like sitting at a desk, he did much of his creative work in unconventional places such as on a sailboat in the Indian Ocean or in a Greek lemon orchard.

Idea: start with an adjacency matrix

What's going on here?

0	14	inf	100
inf	0	14	50
inf	inf	0	14
10	inf	inf	0



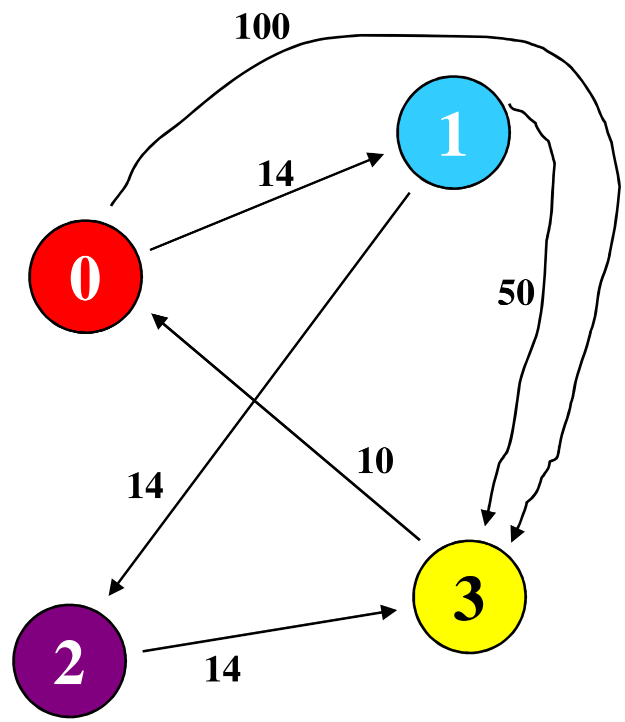
Insight: consider waypoints 1 at a time

dst
"to"

src
"from"

	0	1	2	3
0	0	14	inf	100
1	inf	0	14	50
2	inf	inf	0	14
3	10	inf	inf	0

intermediate nodes:



Step 1

check each ***src*** to each ***dst*** THROUGH 0

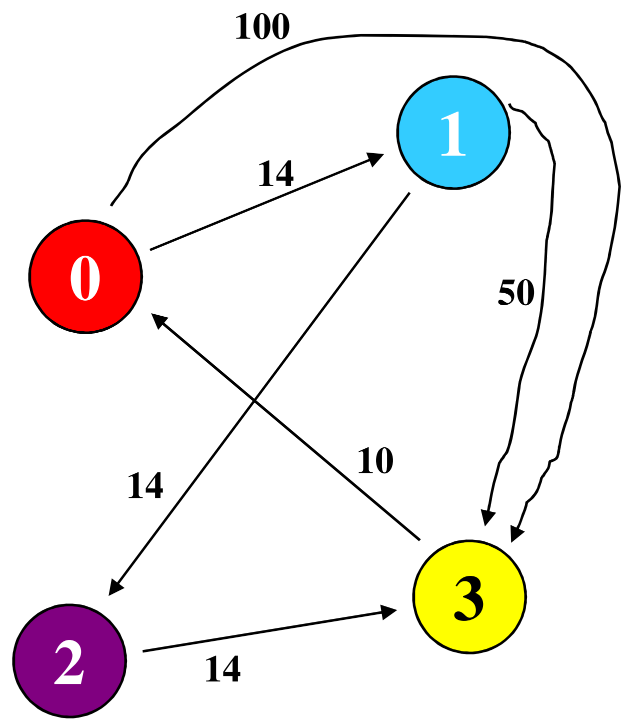
1 entry will change – which?

dst
"to"

src
"from"

src \ dst	0	1	2	3
0	0	14	inf	100
1	inf	0	14	50
2	inf	inf	0	14
3	10	inf	inf	0

intermediate nodes: 0



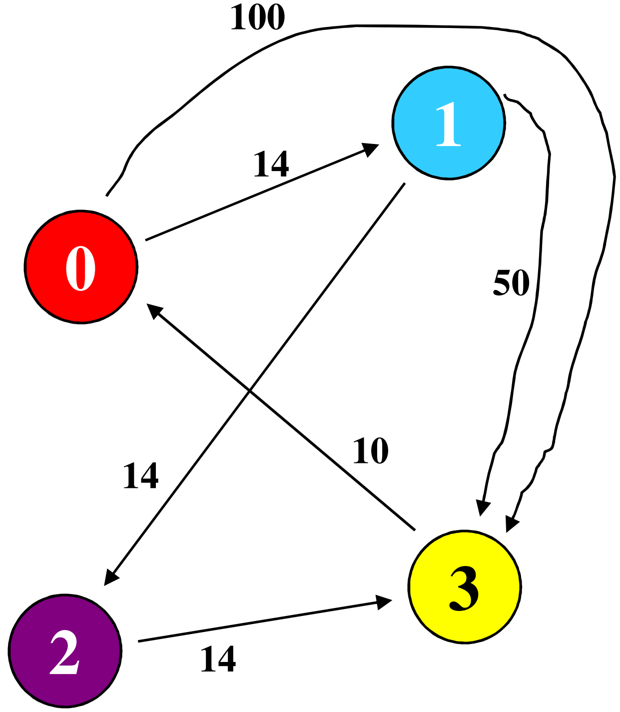
Step 1 check each ***src*** to each ***dst*** THROUGH 0

dst
"to"

src
"from"

src \ dst	0	1	2	3
0	0	14	inf	100
1	inf	0	14	50
2	inf	inf	0	14
3	10	24	inf	0

intermediate nodes: 0



Step 2

check each ***src*** to each ***dst*** THROUGH 1

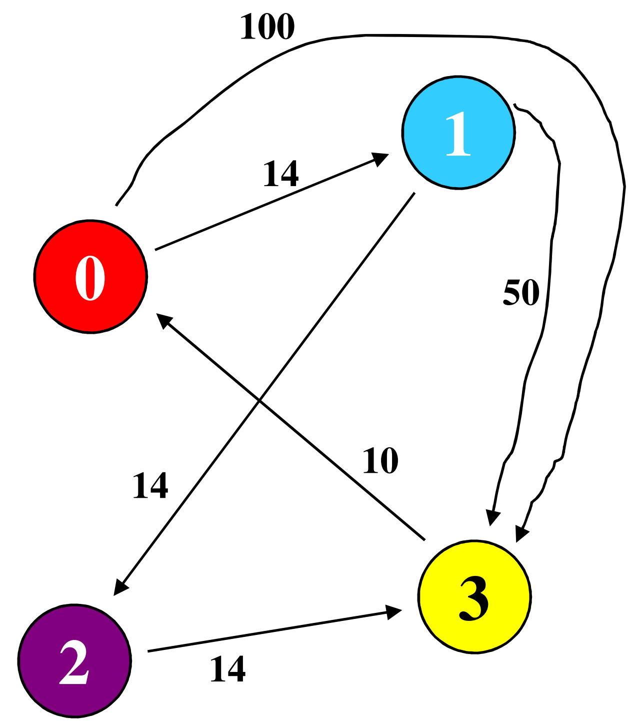
3 entries will change – which?

dst "to"

src "from"

	0	1	2	3
0	0	14	inf	100
1	inf	0	14	50
2	inf	inf	0	14
3	10	24	inf	0

intermediate nodes: 0 1



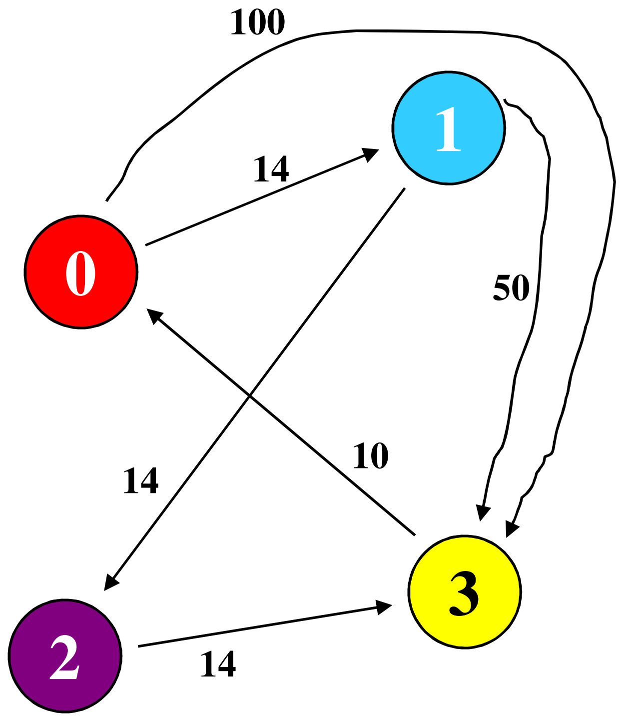
Step 2 check each ***src*** to each ***dst*** THROUGH 1

dst
"to"

src
"from"

	0	1	2	3
0	0	14	28	64
1	inf	0	14	50
2	inf	inf	0	14
3	10	24	38	0

intermediate nodes: 0 1



Step 3 check each ***src*** to each ***dst*** THROUGH 2

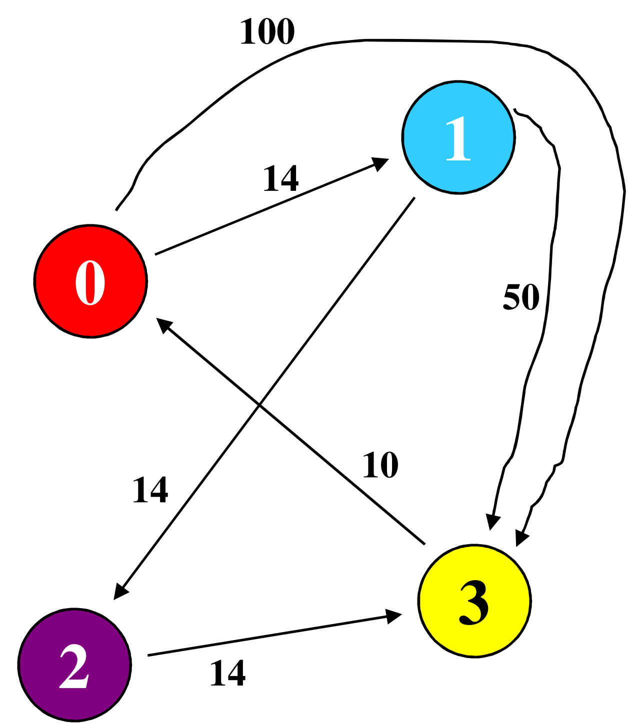
2 entries will change – which?

dst
"to"

src
"from"

	0	1	2	3
0	0	14	28	64
1	inf	0	14	50
2	inf	inf	0	14
3	10	24	38	0

intermediate nodes: 0 1 2



Step 3 check each ***src*** to each ***dst*** THROUGH 2

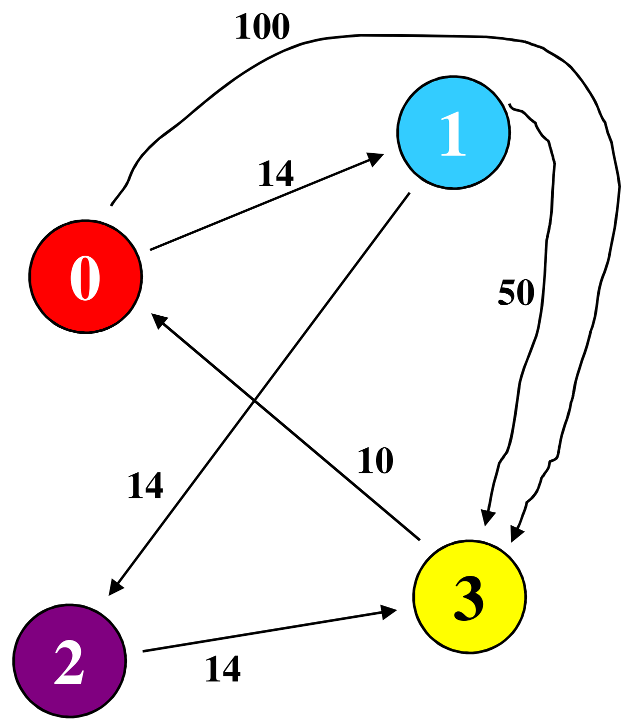
2 entries will change – which?

dst "to"

src "from"

	0	1	2	3
0	0	14	28	42
1	inf	0	14	28
2	inf	inf	0	14
3	10	24	38	0

intermediate nodes: 0 1 2



Done!

We now have all of the shortest-distances between all (src, dst) pairs!

What's the big-O runtime?

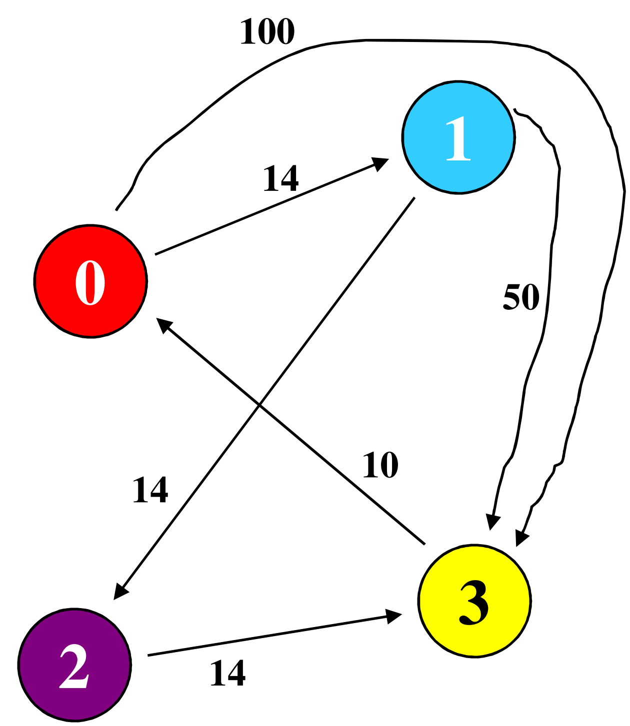
And what about the ***paths***?

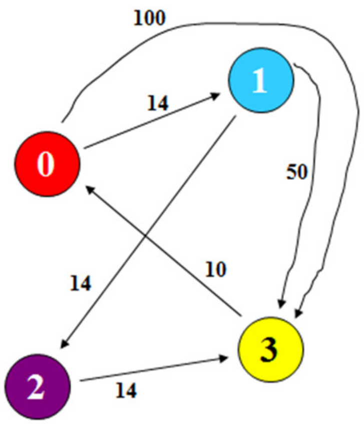
dst "to"

src "from"

src \ dst	0	1	2	3
0	0	14	28	42
1	38	0	14	28
2	24	38	0	14
3	10	24	38	0

intermediate nodes: 0 1 2 3





We now have all of the shortest-distances between all (src, dst) pairs!

dst "to"

src "from"

src \ dst	0	1	2	3
0	0	14	28	42
1	38	0	14	28
2	24	38	0	14
3	10	24	38	0

intermediate nodes: 0 1 2 3

What's the big-O runtime?

And what about the ***paths***?

0	0	0	0
1	1	1	1
2	2	2	2
3	3	3	3

How could we keep track of the paths as we go...?!

FW is popular *in practice*…

Kevin Burke to me show details 10/27/10 Repl

I think the key takeaway from the toll problem is, *if Floyd Warshall doesn't solve your problem, clearly you didn't run Floyd Warshall enough times.*

I'd like to submit the revised ACM problem solving approach:

Old
Try Floyd Warshall
If that doesn't work consider other algorithms

New
Try Floyd Warshall
Try running Floyd Warshall n times
If that still doesn't solve your problem then consider other algorithms

Thanks, Kevin!

May your weekend have $O(2^N)$ runtime!

Join us for LAC hours Friday!

Step 4

check each ***src*** to each ***dst*** THROUGH 3

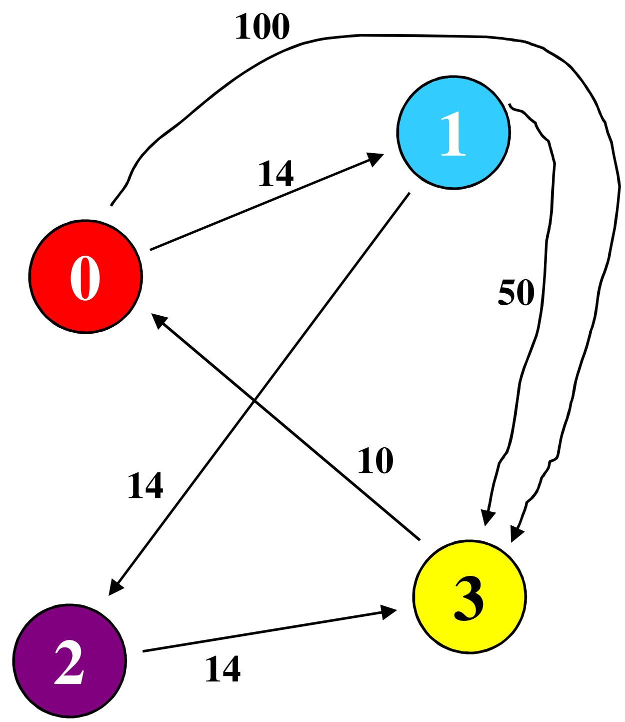
3 entries will change – which?

dst
"to"

src
"from"

src \ dst	0	1	2	3
0	0	14	28	42
1	inf	0	14	28
2	inf	inf	0	14
3	10	24	38	0

intermediate nodes: 0 1 2 3



Done! O( ) ? the path?

Demo!

dst
"to"

src
"from"

	1	2	3	4
1	0	14	28	42
2	38	0	14	28
3	24	38	0	14
4	10	24	38	0

4 intermediate node(s)
1 2 3 4

```
T[k][src][dst]
```

lose k

```
         T[k-1][src][dst]
= min {
         T[k-1][src][k] +
         T[k-1][k][dst]
```

use k

Whole algorithm!

Step 2

check each ***src*** to each ***dst*** THROUGH 2

3 entries will change – which?

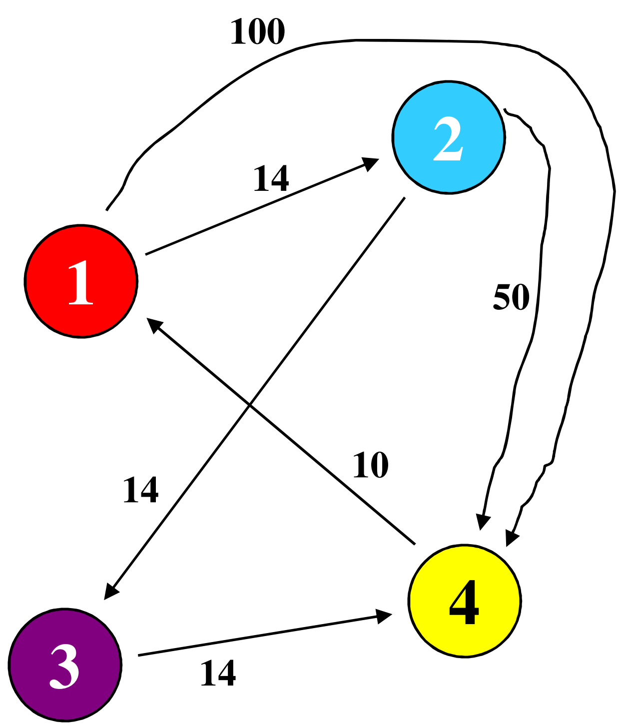
dst
"to"

src
"from"

	1	2	3	4
1	0	14	inf	100
2	inf	0	14	50
3	inf	inf	0	14
4	10	24	inf	0

1 intermediate node(s)

1



Step 3 check each ***src*** to each ***dst*** THROUGH 3

2 entries will change – which?

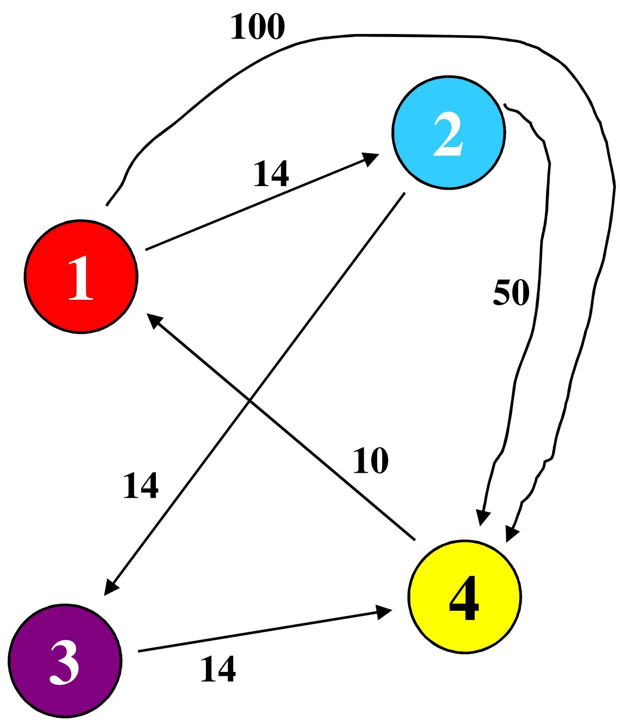
dst
"to"

src
"from"

	1	2	3	4
1	0	14	28	64
2	inf	0	14	50
3	inf	inf	0	14
4	10	24	38	0

2 intermediate node(s)

1 2



Step 3 check each *src* to each *dst* THROUGH 4

3 entries will change – which?

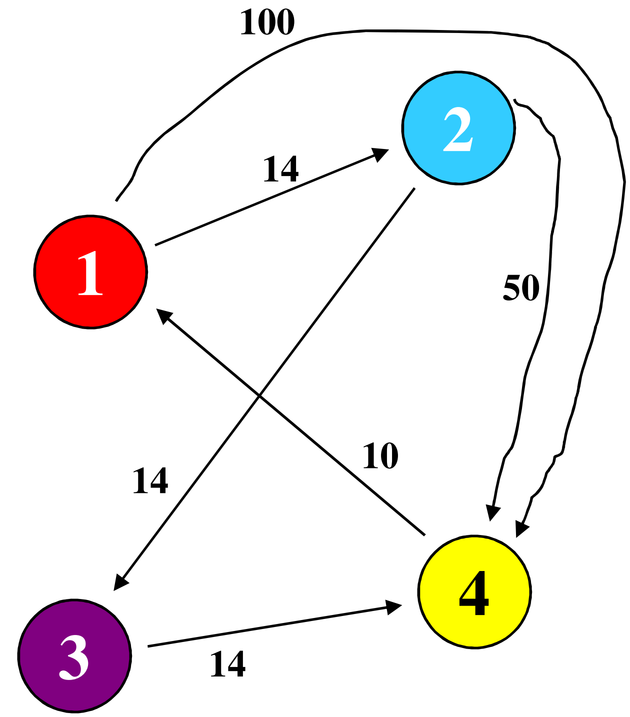
dst
"to"

src
"from"

	1	2	3	4
1	0	14	28	42
2	inf	0	14	28
3	inf	inf	0	14
4	10	24	38	0

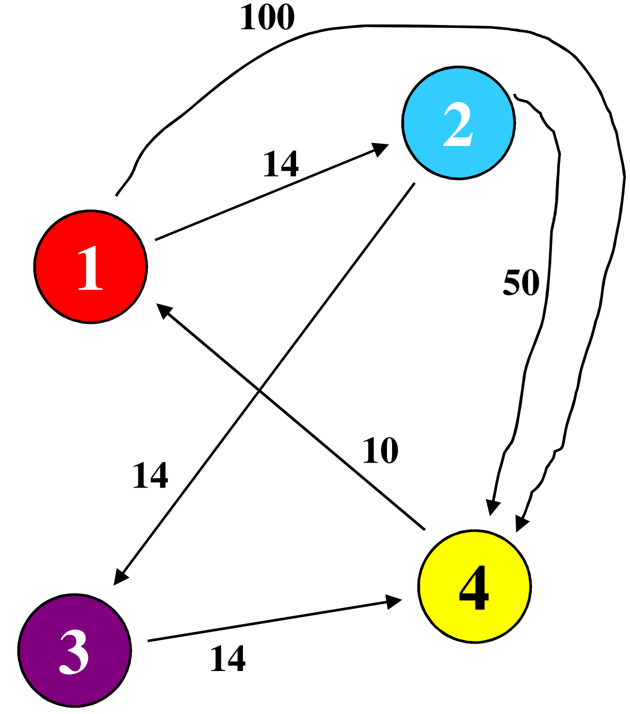
3 intermediate node(s)

1 2 3



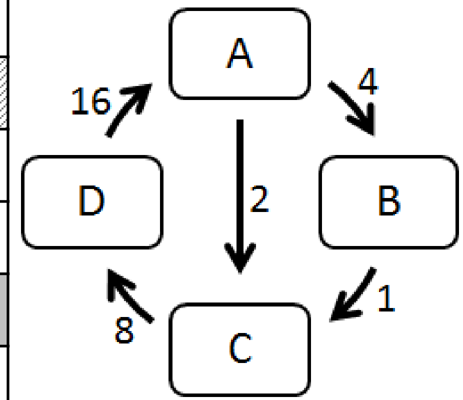
`map2.txt`

0	14	28	42
38	0	14	28
24	38	0	14
10	24	38	0



map1.txt

Edge	Cost	Through A	Cost	Through B	Cost	Through C	Cost	Through D	Cost
A → A	0	A → A → A	0	A → B → A	0	A → C → A	0	A → D → A	0
A → B	4	A → A → B	4	A → B → B	4	A → C → B	4	A → D → B	4
A → C	2	A → A → C	2	A → B → C	2	A → C → C	2	A → D → C	2
A → D	∞	A → A → D	∞	A → B → D	∞	[illegible]	10	A → D → D	10
B → A	∞	B → A → A	∞	[illegible]	[illegible]	[illegible]	∞	B → D → A	25
B → B	0	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	B → D → B	0
B → C	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	1	B → D → C	1
B → D	[illegible]	[illegible]	[illegible]	[illegible]	∞	B → C → D	9	B → D → D	9
C → A	∞	[illegible]	∞	C → B → A	∞	C → C → A	∞	C → D → A	24
C → B	∞	C → A → B	∞	C → B → B	∞	C → C → B	∞	C → D → B	28
C → C	0	C → A → C	0	C → B → C	0	C → C → C	0	C → D → C	0
C → D	8	C → A → D	8	C → B → D	8	C → C → D	8	C → D → D	8
D → A	16	D → A → A	16	D → B → A	16	D → C → A	16	D → D → A	16
D → B	∞	D → A → B	20	D → B → B	20	D → C → B	20	D → D → B	20
D → C	∞	D → A → C	18	D → B → C	18	D → C → C	18	D → D → C	18
D → D	0	D → A → D	0	D → B → D	0	D → C → D	0	D → D → D	0



SOLUTION

	Y → **X** → **X** or **X** → **X** → Y	No change
	X → **X** or **X** → Y → **X**	No change (already 0)

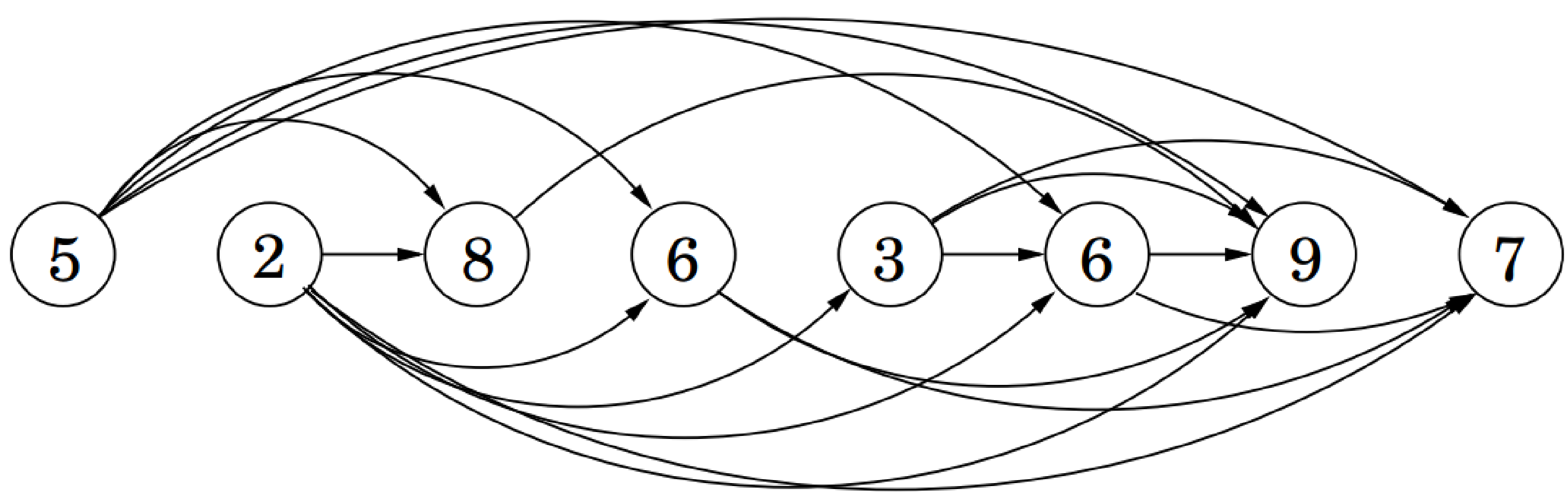
MISS example

Length of max path to get to this node.

1	1	2	2	2	3	4	4

Parents?

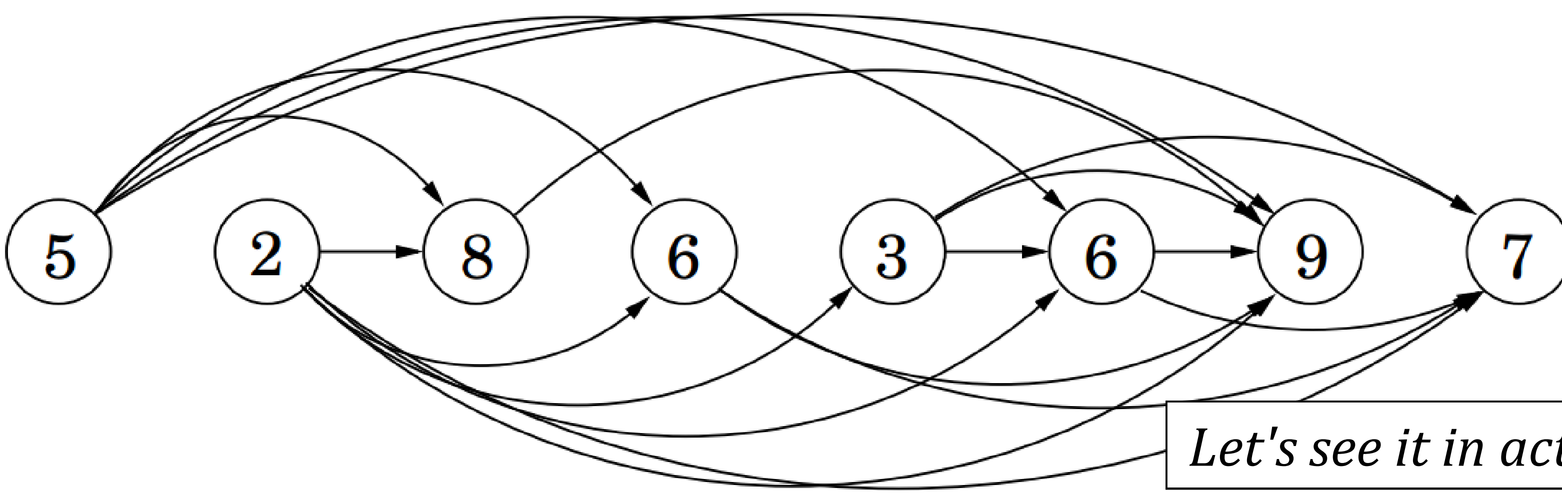
5 2 8 6 3 6 9 7



MISS example

Length of max path to get to this node.

	1	1	2	2	2	3	4	4
Parents	-1	-1	0	0	1	4	5	5
	5	2	8	6	3	6	9	7
indices	0	1	2	3	4	5	6	7



Let's see it in action...

hw11 problem 3: big-O

(Part 3) big-O analysis

All functions are equal; some functions
are more equal than others…

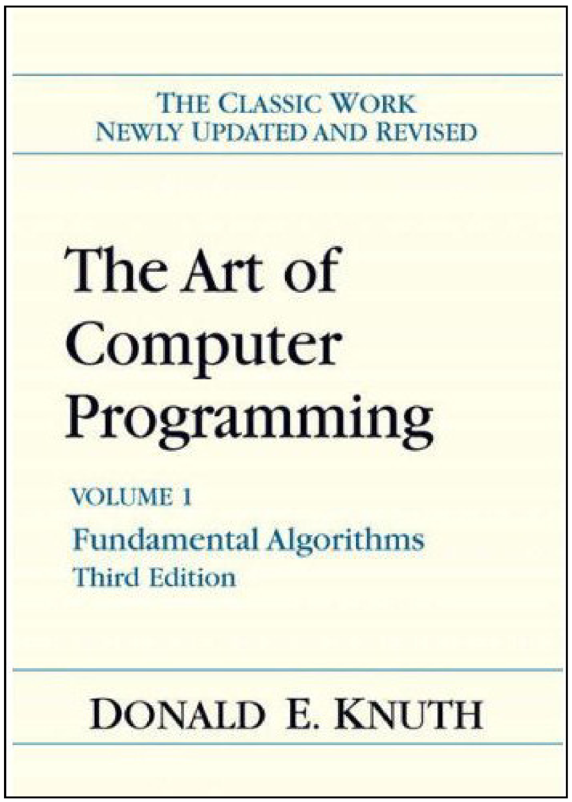
George big-Orwell

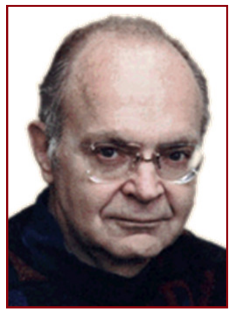
hw11 problem 3: big-O

(Part 3) big-O analysis

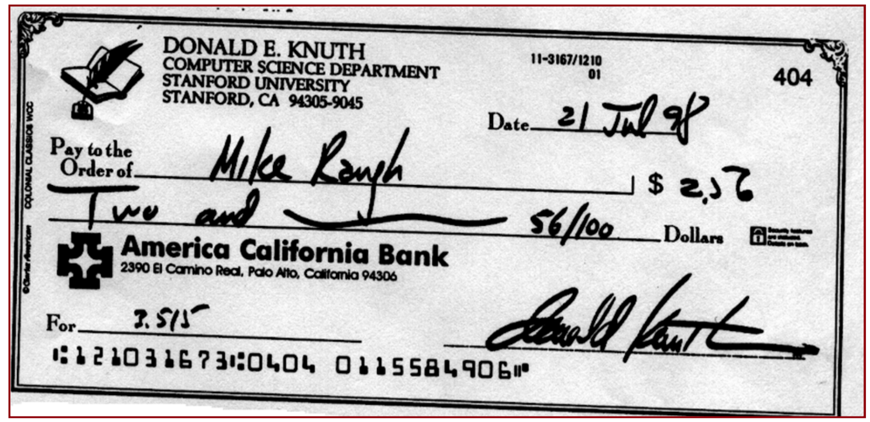
All functions are equal; some functions
are more equal than others…

George big-Orwell





today's CS 60 sponsor
Don Knuth (75)



nearly flawless… at a relatively small price

Considered an expert at writing compilers, Knuth started to write a book about compiler design in 1962, and soon realized that the scope of the book needed to be much larger. In June 1965, Knuth finished the first draft of what was originally planned to be a single volume of twelve chapters. His hand-written manuscript was 3,000 pages long: he had assumed that about five hand-written pages would translate into one printed page, but his publisher said instead that about 1½ hand-written pages translated to one printed page. This meant the book would be approximately 2,000 pages in length. At this point, the plan was changed: the book would be published in seven volumes, each with just one or two chapters. Due to the growth in the material, the plan for Volume 4 has since expanded to include Volumes 4A, 4B, 4C, and possibly 4D.