Algorithm and Object-Oriented Programming for Modeling

Part 5: Dynamic Programming

MSDM 5051, Yi Wang (王一), HKUST

What’s dynamic programming (動態規劃)?

(Bellman 1953)

RICHARD BELLMAN ON THE BIRTH OF DYNAMIC PROGRAMMING

STUART DREYFUS

University of California, Berkeley, IEOR, Berkeley, California 94720, dreyfus@ieor.berkeley.edu

What follows concerns events from the summer of 1949, when Richard Bellman first became interested in multistage decision problems, until 1955. Although Bellman died on March 19, 1984, the story will be told in his own words since he left behind an entertaining and informative autobiography, *Eye of the Hurricane* (World Scientific Publishing Company, Singapore, 1984), whose publisher has generously approved extensive excerpting.

During the summer of 1949 Bellman, a tenured associate professor of mathematics at Stanford University with a developing interest in analytic number theory, was consulting for the second summer at the RAND Corporation in Santa Monica. He had received his Ph.D. from Princeton in 1946 at the age of 25, despite various war-related activities during World War II—including being assigned by the Army to the Manhattan Project in Los Alamos. He had already exhibited outstanding ability both in pure mathematics and in solving applied problems arising from the physical world. Assured of a successful conventional academic career, Bellman, during the period under consideration, cast his lot instead with the kind of applied mathematics later to be known as operations research. In those days applied practitioners were regarded as distinctly second-class citizens of the mathematical fraternity. Always one to enjoy controversy, when invited to speak at various university mathematics department seminars, Bellman delighted in justifying his choice of applied over pure mathematics as being motivated by the real world's greater challenges and mathematical demands.

Following are excerpts, taken chronologically from Richard Bellman's autobiography. The page numbers are given after each. The excerpt section titles are mine. These excerpts are far more serious than most of the book, which is full of entertaining anecdotes and outrageous behaviors by an exceptionally *human* being.

Stuart Dreyfus

BELLMAN'S INTRODUCTION TO MULTISTAGE DECISION PROCESS PROBLEMS

"I was very eager to go to RAND in the summer of 1949 . . . I became friendly with Ed Paxson and asked him what RAND was interested in. He suggested that I work on multistage decision processes. I started following that suggestion" (p. 157).

CHOICE OF THE NAME DYNAMIC PROGRAMMING

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. 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, let's kill two birds with one stone. Let's 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 it's 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" (p. 159).

EARLY ANALYTICAL RESULTS

"The summer of 1951 was old-home-week. Sam Karlin and Hal Shapiro were at RAND.

RICHARD BELLMAN ON THE BIRTH OF DYNAMIC PROGRAMMING

STUART DREYFUS

University of California, Berkeley, IEOR, Berkeley, California 94720, dreyfus@ieor.berkeley.edu

What follows concerns events from the summer of 1949, when Richard Bellman first became interested in multistage decision problems, until 1955. Although Bellman died on March 19, 1984, the story will be told in his own words since he left behind an entertaining and informative autobiography, *Eye of the Hurricane* (World Scientific Publishing Company, Singapore, 1984), whose publisher has generously approved extensive excerpting.

During the summer of 1949 Bellman, a tenured associate professor of mathematics at Stanford University with a developing interest in analytic number theory, was consulting for the second summer at the RAND Corporation in Santa Monica. He had received his Ph.D. from Princeton in 1946 at the age of 25, despite various war-related

what RAND was interested in. He suggested that I work on multistage decision processes. I started following that suggestion" (p. 157).

CHOICE OF THE NAME DYNAMIC PROGRAMMING

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the

兰德公司 (RAND Corporation)

智库

兰德公司是美国的一所智库。在其成立之初主要为美国军方提供调研和情报分析服务。其后组织逐步扩展，并为其他政府以及盈利性团体提供服务。虽名称冠有“公司”，但实际上是登记为非营利组织。

RICHARD BELLMAN ON THE BIRTH OF DYNAMIC PROGRAMMING

STUART DREYFUS

University of California, Berkeley, IEOR, Berkeley, California 94720, dreyfus@ieor.berkeley.edu

What follows concerns events from the summer of 1949, when Richard Bellman first became interested in multistage decision problems, until 1955. Although Bellman died on March 19, 1984, the story will be told in his own words since he left behind an entertaining and informative autobiography, *Eye of the Hurricane* (World Scientific Publishing Company, Singapore, 1984), whose publisher has generously approved extensive excerpting.

During the summer of 1949 Bellman, a tenured associate professor of mathematics at Stanford University with a developing interest in analytic number theory, was consulting for the second summer at the RAND Corporation in Santa Monica. He had received his Ph.D. from Princeton in 1946 at the age of 25, despite various war-related activities during World War II—including being assigned by the Army to the Manhattan Project in Los Alamos. He had already exhibited outstanding ability both in pure mathematics and in solving applied problems arising from the physical world. Assured of a successful conventional academic career, Bellman, during the period under consideration, cast his lot instead with the kind of applied mathematics later to be known as operations research. In those days applied practitioners were regarded as distinctly second-class citizens of the mathematical fraternity. Always one to enjoy controversy, when invited to speak at various university mathematics department seminars, Bellman delighted in justifying his choice of applied over pure mathematics as being motivated by the real world's greater challenges and mathematical demands.

Following are excerpts, taken chronologically from Richard Bellman's autobiography. The page numbers are given after each. The excerpt section titles are mine. These excerpts are far more serious than most of the book, which is full of entertaining anecdotes and outrageous behaviors by an exceptionally *human* being.

Stuart Dreyfus

BELLMAN'S INTRODUCTION TO MULTISTAGE DECISION PROCESS PROBLEMS

"I was very eager to go to RAND in the summer of 1949 . . . I became friendly with Ed Paxson and asked him what RAND was interested in. He suggested that I work on multistage decision processes. I started following that suggestion" (p. 157).

CHOICE OF THE NAME DYNAMIC PROGRAMMING

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. 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, let's kill two birds with one stone. Let's 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 it's 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" (p. 159).

EARLY ANALYTICAL RESULTS

"The summer of 1951 was old-home-week. Sam Karlin and Hal Shapiro were at RAND.

story will be told
n entertaining and
Hurricane (World
re, 1984), whose
sive excerpting.
, a tenured asso-
d University with
theory, was con-
AND Corporation
D. from Princeton
rious war-related
ng being assigned
in Los Alamos.
ility both in pure
lems arising from
ssful conventional
eriod under con-
kind of applied
tions research. In
arded as distinctly
fraternity. Always
to speak at vari-
eminars, Bellman
d over pure math-
rld's greater chal-

CHOICE OF THE NAME DYNAMIC PROGRAMMING

"I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes.

"An interesting question is, 'Where did the name, dynamic programming, come from?' The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. 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, let's kill two birds with one stone. Let's take a word that has an

D. from Princeton
arious war-related
ng being assigned
in Los Alamos.
bility both in pure
lems arising from
ssful conventional
period under con-
e kind of applied
tions research. In
arded as distinctly
fraternity. Always
to speak at vari-
eminars, Bellman
ed over pure math-
orld's greater chal-

nologically from
page numbers are
s are mine. These
f the book, which
rageous behaviors

Secretary of Defense, and he actually had a pathological fear and hatred of the word, research. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. 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, let's kill two birds with one stone. Let's 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 it's 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" (p. 159).

What’s dynamic programming (動態規劃)?
Unfortunately, it’s a bad name. Doesn’t tell what’s the algorithm.

There’s something programming (planning).

But something like “reduce, try and memorize” is perhaps better.
Let’s see what it actually is.

Example: Fibonacci numbers

Example: calculate Fibonacci numbers.

$F_1 = F_2 = 1, F_n = F_{n-1} + F_{n-2}$. How to calculate F_n?

```
def fib_1(n):
    return fib_1(n-1) + fib_1(n-2) if n > 2 else 1
```

```
fib[1] := 1
fib[2] := 1
fib[n_] := fib[n - 1] + fib[n - 2]
```

BTW: Mathematica

Example: calculate Fibonacci numbers.

$F_1 = F_2 = 1, F_n = F_{n-1} + F_{n-2}$. How to calculate F_n?

```python
def fib_1(n):
    return fib_1(n-1) + fib_1(n-2) if n > 2 else 1
```

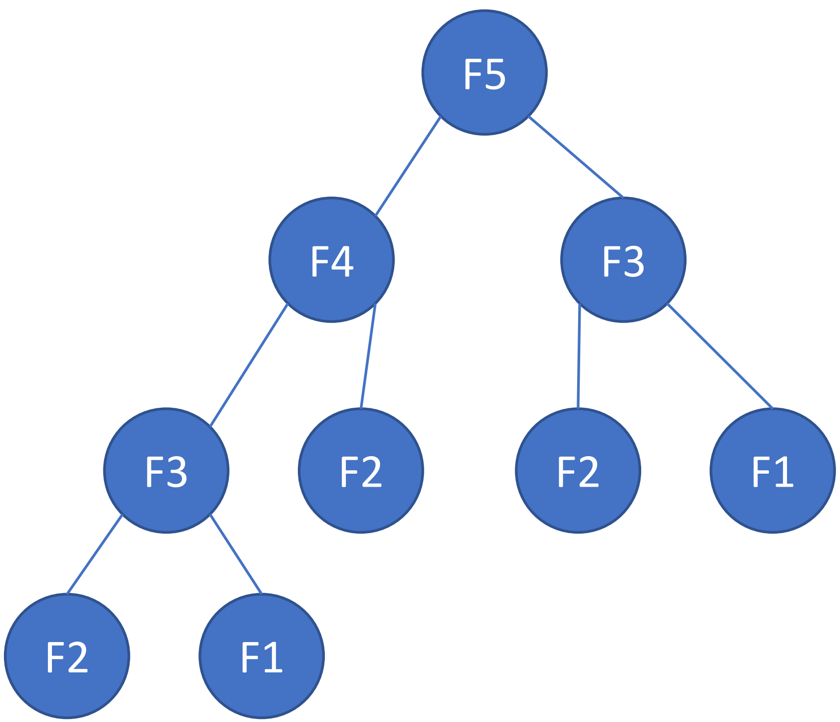
What's the time complexity?

Example: calculate Fibonacci numbers.

$F_1 = F_2 = 1, F_n = F_{n-1} + F_{n-2}$. How to calculate F_n?

```python
def fib_1(n):
    return fib_1(n-1) + fib_1(n-2) if n > 2 else 1
```

What's the time complexity?

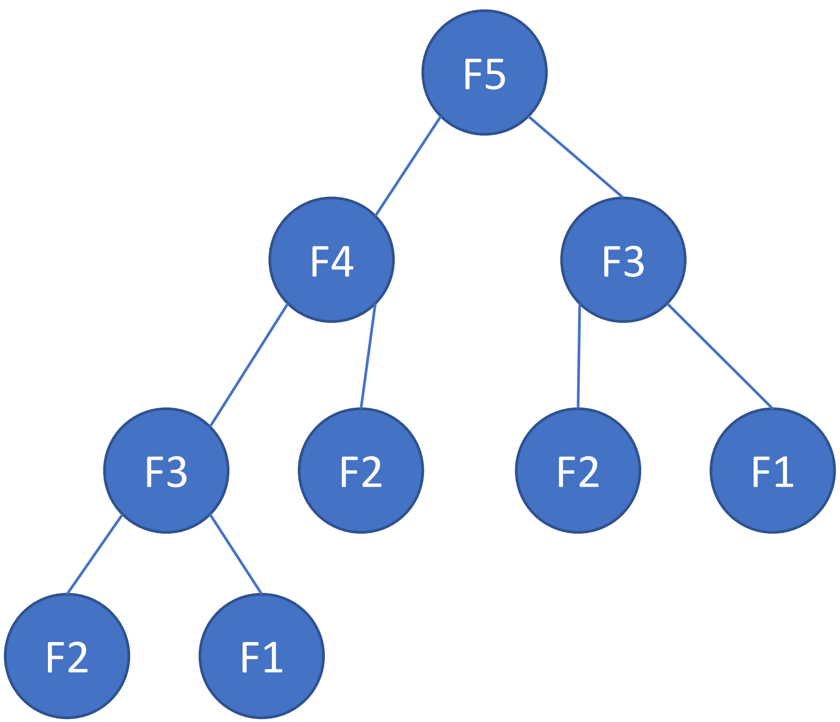


Example: calculate Fibonacci numbers.

$F_1 = F_2 = 1, F_n = F_{n-1} + F_{n-2}$. How to calculate F_n?

```
def fib_1(n):
    return fib_1(n-1) + fib_1(n-2) if n > 2 else 1
```

What's the time complexity?



Consider the right-most path

Height: [(n-1)/2]

Thus, # vertices $> 2^{[(n-1)/2]}$

Time complexity: $O(2^n)$

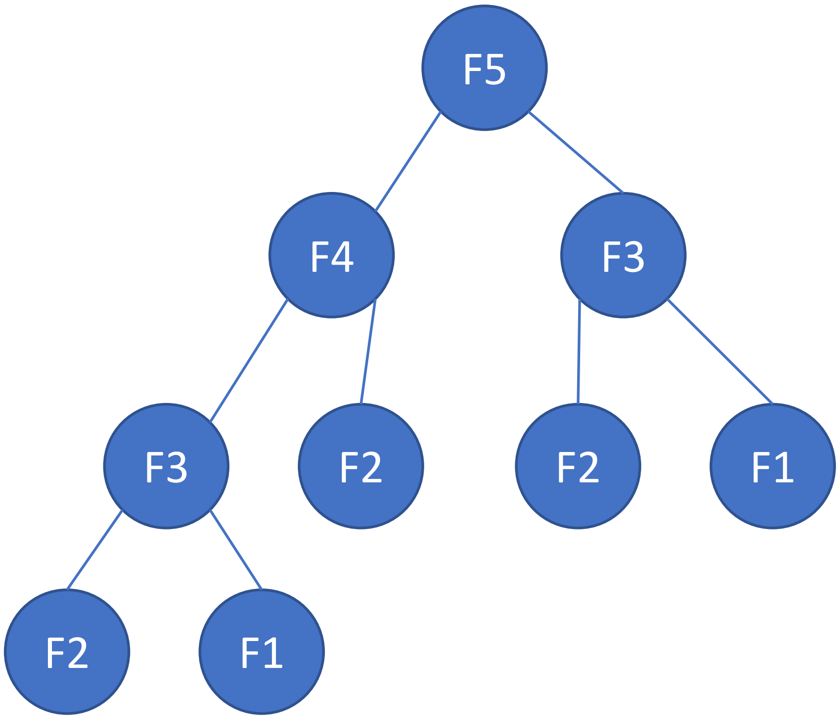
Exponential, very bad.

Can we do better?

Of course! We have only calculated n functions, not 2^n!

Idea to improve Fibonacci: Note F3 calculated twice.

Can we calculate once and remember it?



Memorization

Time complexity: O(n).

Using a dict

```
memo = {}
def fib_2(n):
    if n not in memo:
        memo[n] = fib_2(n-1) + fib_2(n-2) if n > 2 else 1
    return memo[n]
```

Using built-in cache

```
from functools import lru_cache
@lru_cache(maxsize=None)
def fib_3(n):
    return fib_3(n-1) + fib_3(n-2) if n > 2 else 1
```

```
print(fib_3.cache_info()) # check cache efficiency
```

BTW: Mathematica:

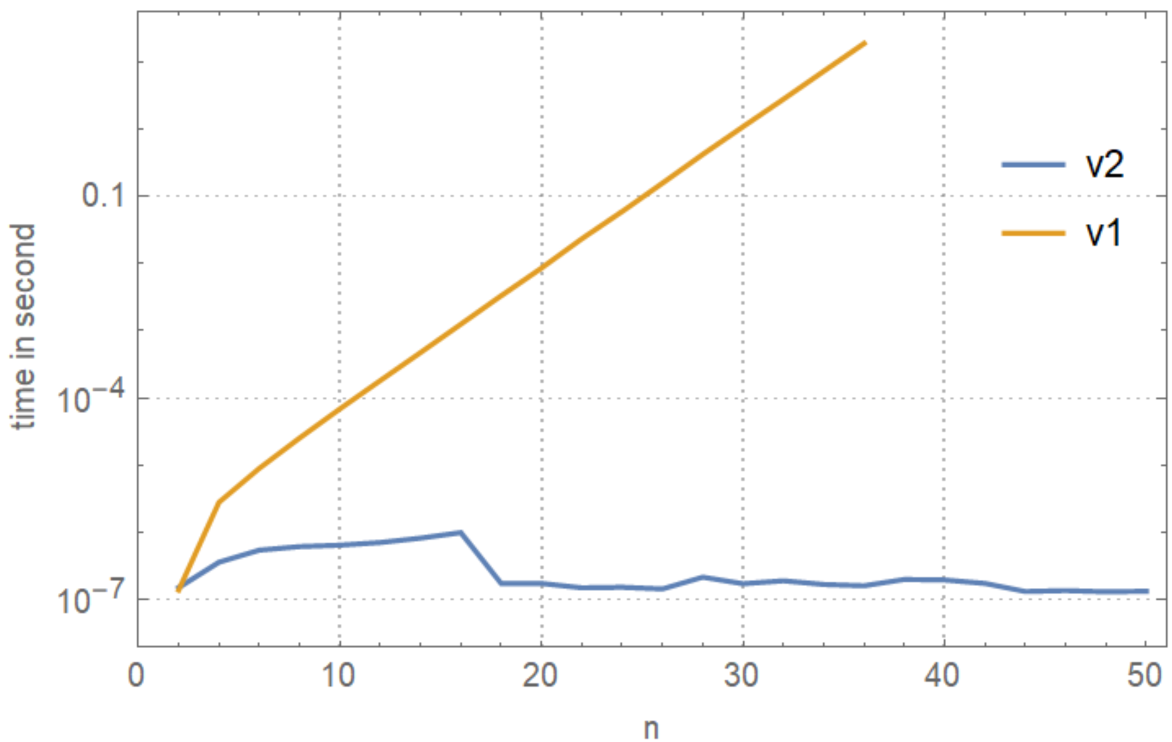
```
fib[1] := 1
fib[2] := 1
fib[n_] := fib[n] = fib[n - 1] + fib[n - 2]
```

Version 1

```
fib[1] = fib[2] = 1;
fib[n_] := fib[n - 1] + fib[n - 2]
```

Version 2

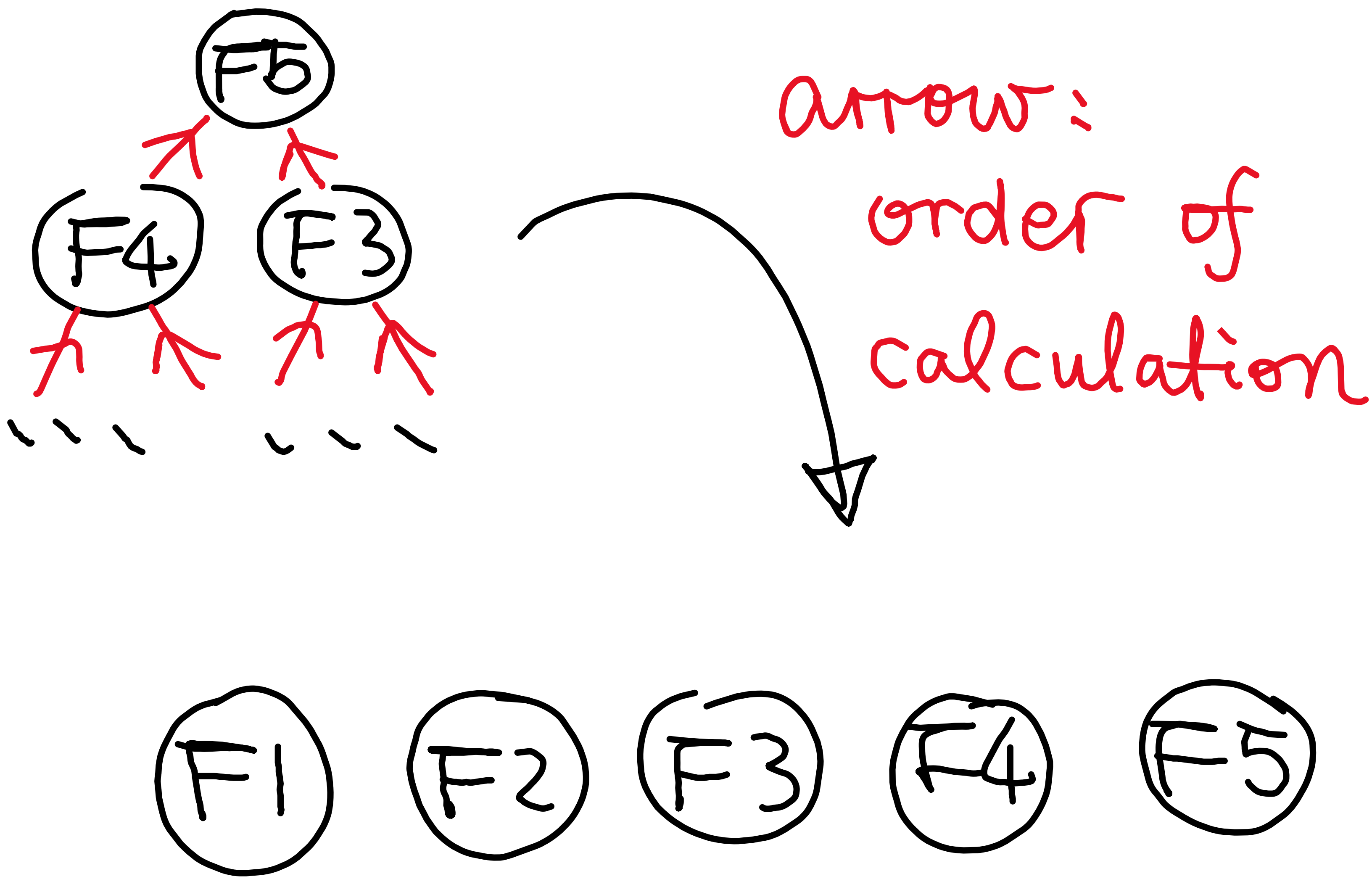
```
fib[1] = fib[2] = 1;
fib[n_] := fib[n] = fib[n - 1] + fib[n - 2]
```

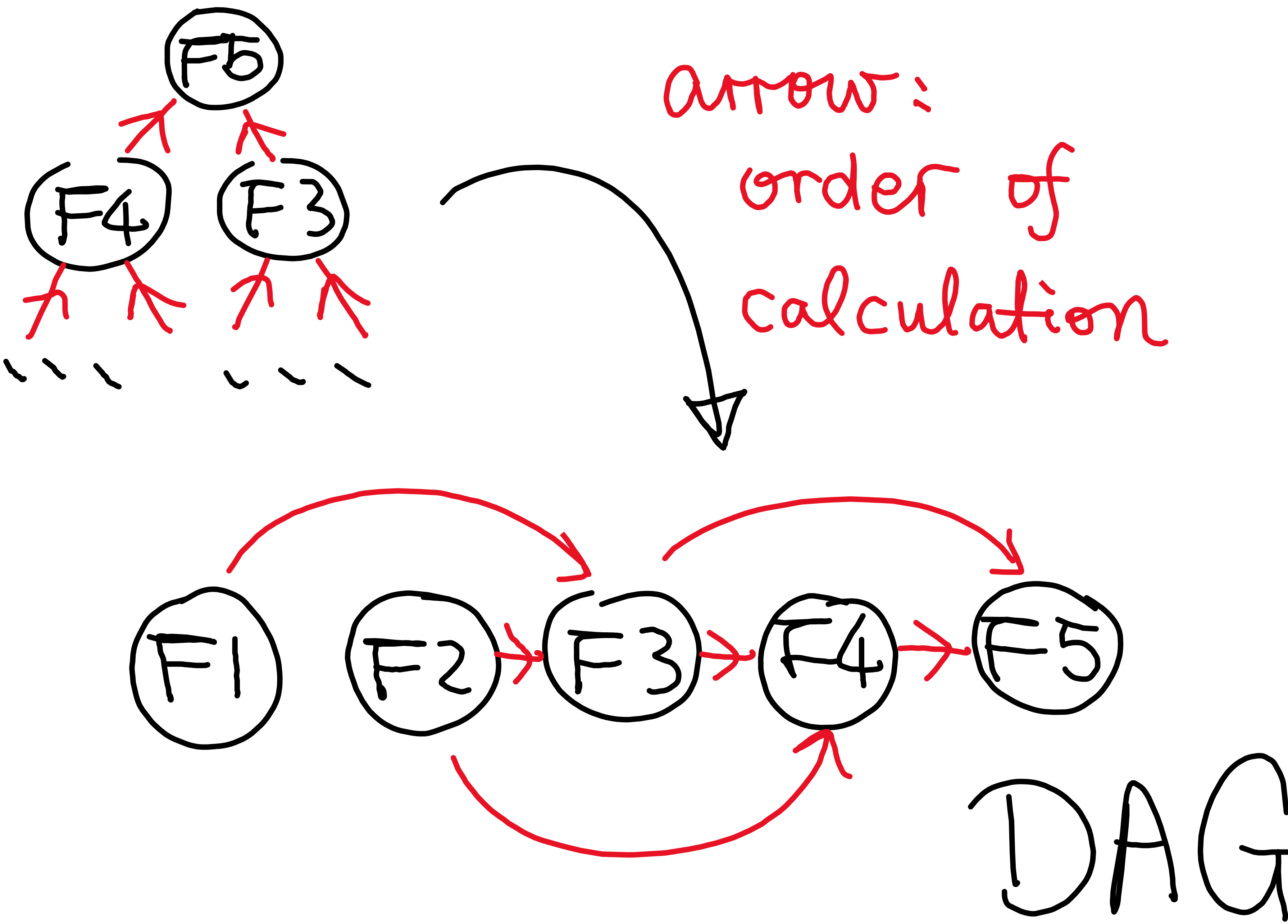


```python
memo = {}
def fib_2(n):
    if n not in memo:
        memo[n] = fib_2(n-1) + fib_2(n-2) if n > 2 else 1
    return memo[n]

memo = {1:1, 2:1}
def fib_2p(n):
    if n not in memo:
        memo[n] = fib_2p(n-1) + fib_2p(n-2)
    return memo[n]
```

Eliminate recursion?


F5
F4
F3
arrow: order of calculation
F1
F2
F3
F4
F5


F5
F4
F3
arrow: order of calculation
F1
F2
F3
F4
F5
DAG

Eliminate recursion:

Calculate the vertices in topological order.

Needed: fib(1) → fib(2) → fib(3) … → fib(n)

```python
def fib_4(n):
    fib = {1:1, 2:1}
    for i in range(3, n+1):
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

```python
def fib_5(n):
    fib = [1 for i in range(n+1)]
    for i in range(3, n+1):
        fib[i] = fib[i-1] + fib[i-2]
    return fib[n]
```

So what’s dynamic programming?

Recursive version:

1. Reduce to smaller problems
2. Remember result of called functions

Iterative version:

1. Construct “dependency” graph
2. Compute answers in topological order

In fib(n): we know for sure

how to reduce to smaller problems

fib(n) = fib(n-1) + fib(n-2)

More complicated problems: need to use

if, for to try possible solutions.

Let’s see two examples with if statements:

Longest common subsequence

Knapsack problem

Longest common subsequence problem

DC Readout Experiment in Enhanced LIGO

Provisional: Tobin Fricke,[1,*] Nicolás Smith,[2] Rich Abbott,[3] Rana Adhikari,[3] Kate Dooley,[4] Matthew Evans,[2] Peter Fritschel,[2] Valera Frolov,[5] Keita Kawabe,[6] Sam Waldman[2]

[1] Department of Physics and Astronomy, Louisiana State University, Baton Rouge, LA 70803-4001
[2] LIGO Laboratory, Massachusetts Institute of Technology, Cambridge, MA 02139
[3] LIGO Laboratory, California Institute of Technology, MS 100-36, Pasadena, CA 91125
[4] Department of Physics, University of Florida, Gainesville, FL 32611-8440
[5] LIGO Livingston Observatory, PO Box 940, Livingston, LA 70754-0940
[6] LIGO Hanford Observatory, PO Box 159, Richland, WA 99352-0159
[*] Corresponding author: tfricke@ligo.caltech.edu

Compiled June 14, 2011

We characterize the DC readout systems implemented on the 4 km LIGO laser- interferometer gravitational wave detectors in Livingston, LA and Hanford, WA.

DC readout is a single-port homodyne detection scheme in which the local oscillator is produced by introducing a microscopic offset in the differential arm length.

We (do/do not) observe a commensurate improvement in shot-noise limited sensitivity due to effect1, effect2, effect3. The readout is compatible with high input power operation, and provides a path forward to Advanced LIGO.

Target journal: Something open-access, like PRX, Optics Express, New Journal of Physics??

ROUGH DRAFT

© 2011 Optical Society of America

OCIS codes: 000.0000, 999.9999.

The ~~community of gravitational wave observatories operates a collection of laser-interferometer gravitational-wave detectors scattered around the globe. These detectors operate modified Michelson interferometers in an effort to detect the tiny oscillatory optical path length modulations induced by gravitational waves of astrophysical origin. The LIGO project operates two such detectors, in Hanford, WA and Livingston, LA, which are described in...~~

~~These detectors have historically~~ initial generation of laser interferometer gravitational wave detectors used a heterodyne detection scheme in an effort to evade baseband laser noises in the 10-1000 Hz ~~region~~range [1,2]. ~~With subsequent~~ Subsequent improvement in laser stability ~~it has become possible/attractive to now implement homodyne detection. During 2008-2010,~~ has made homodyne detection an achievable and attractive option. In 2008 the LIGO detectors [3] were successfully modified to operate using the DC readout form of homodyne detection~~and completed a data-taking science run using the new configuration.~~.

DC readout uses the existing optical infrastructure to produce the homodyne local oscillator, and this local oscillator field is significantly filtered by the interferometer before reaching the detection port, mitigating two of the usual issues in implementing homodyne detection.

The new homodyne detection scheme is limited by photon quantum (shot) noise above 200 Hz and provides a path towards higher power interferometer operation and squeezed light injection.

The implementation of DC readout was part of the Enhanced LIGO program [4,5] of detector improvements, which culminated with LIGO's sixth science run, between July 2009 and October 2010.

DC readout has been implemented previously at the Caltech 40 meter prototype [7,8] and the GEO 600 detector [9,10]. The current configuration of Virgo incorporates an output mode cleaner but uses RF heterodyne readout [11]. Earlier prototype experiments were also conducted at the big LIGO interferometers; an RF OMC was tried at Hanford [12,13], and DC readout using a spare (input) pre-mode cleaner cavity as an OMC was tried at Livingston. These experiments demonstrated the

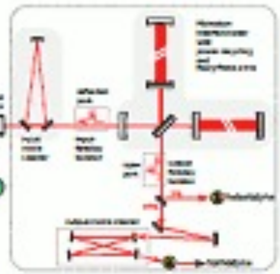


Fig. 1. Experimental arrangement. The dotted line represents the vacuum enclosure.

1

need to place the OMC in vacuum to avoid acoustic noise, and informed the choice of cavity topology and g-factor.

Principle of operation

The interferometer consists of six core suspended optics, forming the Michelson interferometer, the arm cavities, and the power-recycling cavity, as depicted in figure 1. The gravitational wave signal appears in the difference of the lengths of the two resonant arm cavities, the differential arm (DARM) degree of freedom. It is this length that we control using DC readout.

At its simplest, DC readout consists of ~~simply offsetting~~ introducing a small offset in the DARM degree of freedom~~(to do: define the degrees of freedom), creating a linear power (displacement relationship)~~, moving the system slightly off of the dark fringe. Small perturbations around this point will now linearly produce power fluctuations at the output port, which can be sensed directly by a photodiode with no further demodulation.

In the frequency domain, the DARM offset is seen to introduce a carrier-mode local oscillator at the output port ~~(to do: define do ports)~~ without otherwise modifying the response function of the machine[1].

~~This form of single-port homodyne detection is called~~ "DC readout"~~, where "DC" indicates its baseband nature (in contrast to "RF" readout / heterodyne detection) and "direct conversion"~~." specifically denotes the homodyne configuration where the local oscillator arises from a displacement from the dark fringe, as opposed to the alternative scheme where a local oscillator is independently delivered to the readout and combined with the interferometer output via a beamsplitter.

DC readout is best implemented in conjunction with an output mode cleaner. An output mode cleaner also has benefits for RF readout.

Motivation

Initial LIGO experienced a number of deleterious effects that were a result of interactions between the heterodyne readout system and the production of higher-order spatial modes in the marginally stable recycling cavity.

Poor overlap of the signal beam and the sideband beam reduced the optical gain, elevating the shot-noise-limited noise floor. One measurement in 2003 found only half the expected optical gain [15]. This was largely alleviated through the use of ~~TCS~~, thermal compensation system (TCS), which projected CO_2 laser light onto optics to adjust their effective radii of curvature. By the start of S5, no elevation in the shot noise level was observed [16].

However, junk light still caused headaches by producing a large signal in the uncontrolled quadrature of the heterodyne readout (AS_I); left uncontrolled, this signal would saturate the photodiode electronics. An electronic servo was introduced to cancel this signal ~~, but~~ in the photodiode head. While the electronic AS SUBSCRIPT NB servo did eliminate the saturations, it was found to introduce noise.

To cope with this excess power in junk light, initial LIGO split the light at the detection port onto four detection photodiodes. Scaling the interferometer input power would require a commensurate increase in the number of photodiodes at the output port to handle the power, a situation that was seen to quickly become unwieldy.

A prototype RF output mode cleaner was tried. The RF OMC successfully reduced the AS_I signal, but, not isolated by a vacuum, the (acoustic) noise was too high to be used in production.

Plans were made to introduce an OMC for either RF or DC readout. DC readout was chosen due to several additional benefits.

- In addition to mitigating technical difficulties of RF detection, homodyne detection confers a fundamental improvement in SNR by up to a factor of ??. The extra noise in heterodyne detection can be considered either a result of time dependence in the average power leading to correlations in the shot noise [17], or the simple fact that demodulation ~~brings~~ introduces noise from around $2f_{mod}$, giving an extra dose of shot noise. A more sophisticated analysis ascribes this noise to the two heterodyne demodulation quadratures acting as non-commuting quantum operators [18].
- The electronic mixers used to demodulate the heterodyne error signal are typically used in a fully-saturated mode, effectively mixing the photodiode signal with a square wave rather than the (optimal) sinusoid [19]. Noise at harmonics of the (de)modulation frequency is downconverted to baseband. Homodyne detection skirts this issue by avoiding the need for any demodulation.

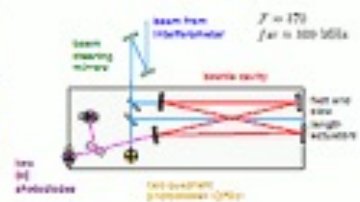
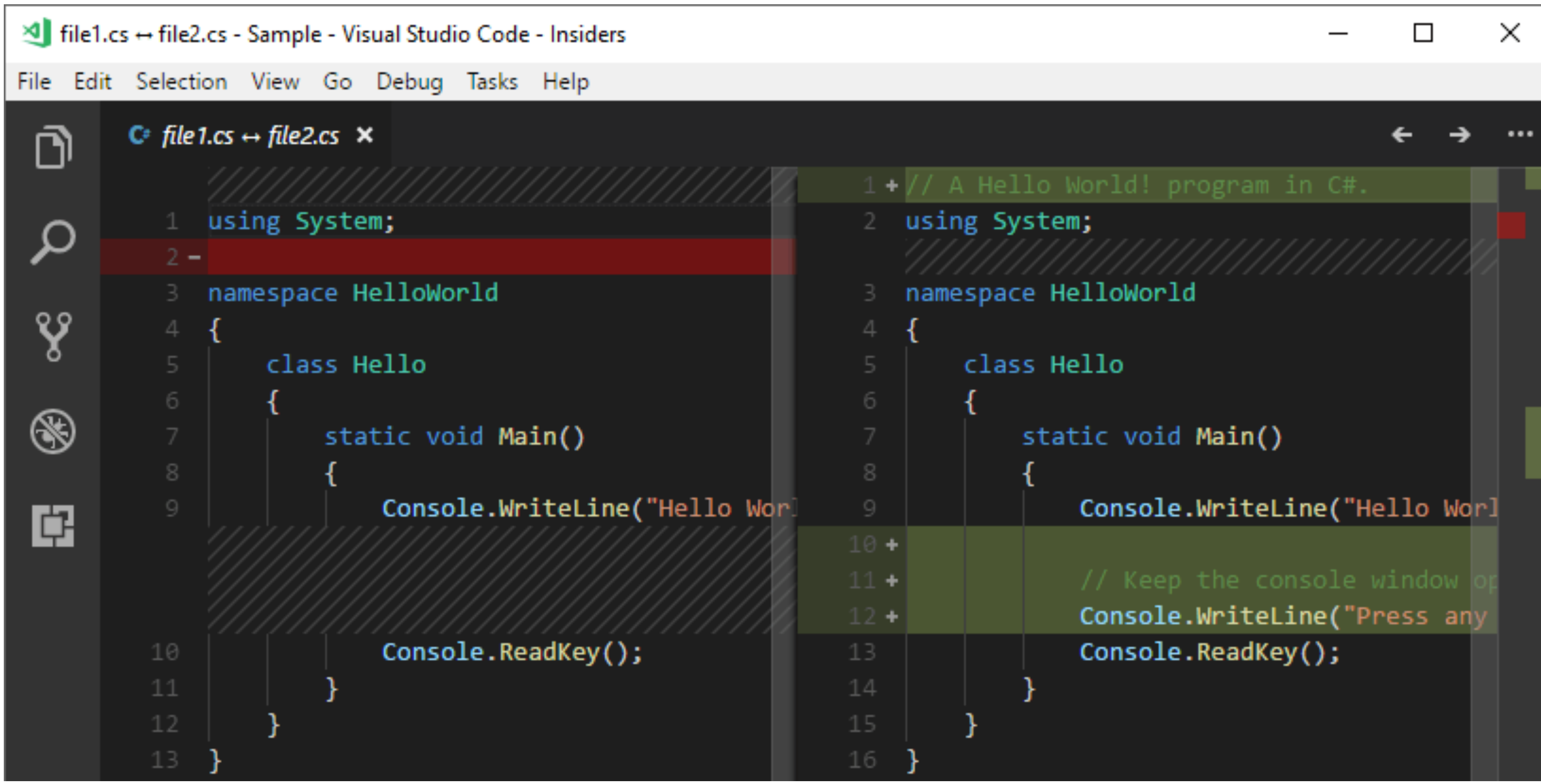


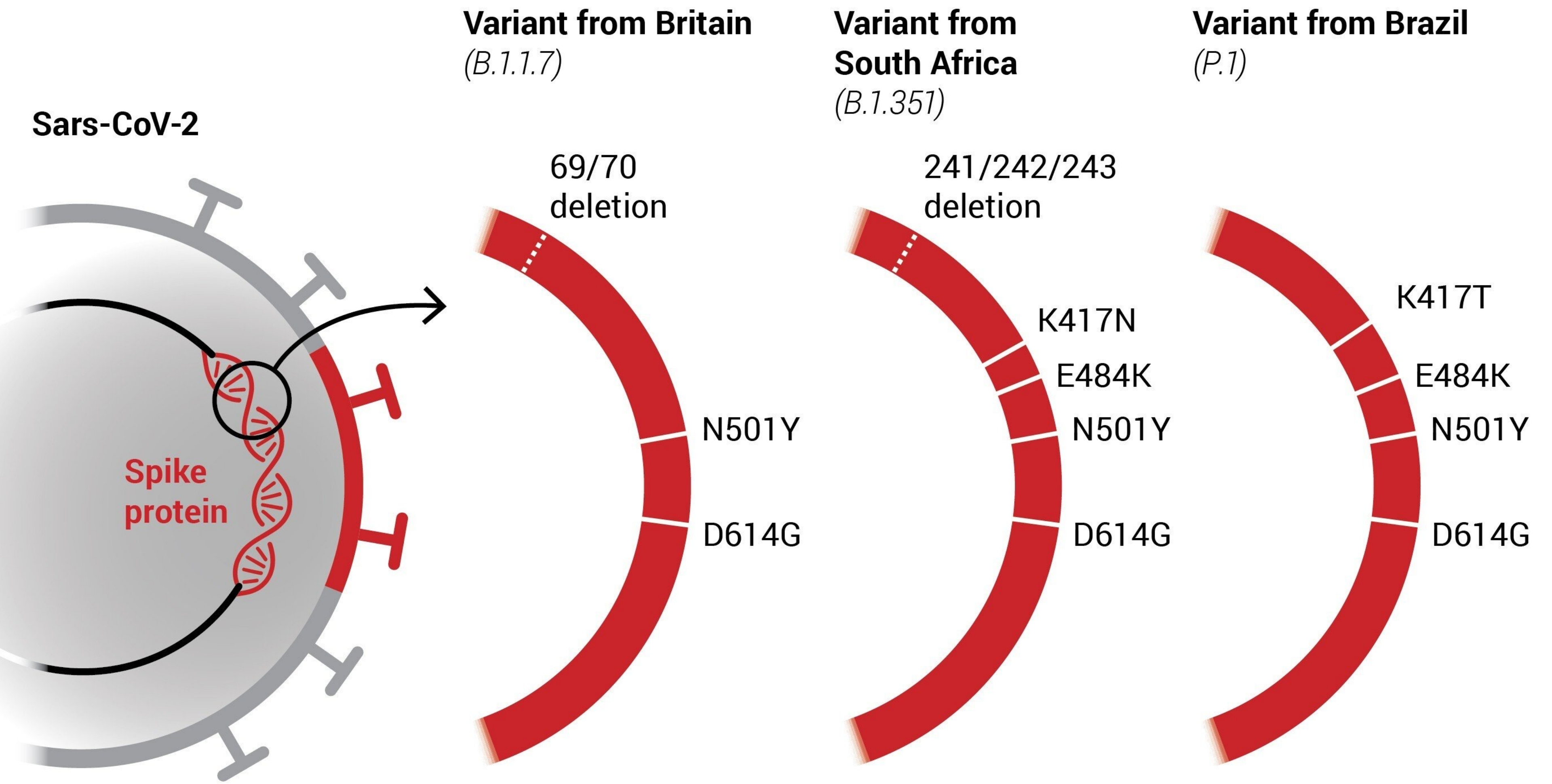
Fig. 2. Schematic of the OMC layout.

[1] Because the Fabry-Perot arm cavities are now operated slightly off resonance, an optical spring is created, with a modified optomechanical transfer function. However, the effect within the LIGO detection band is negligible.

2


file1.cs ↔ file2.cs - Sample - Visual Studio Code - Insiders
File
Edit
Selection
View
Go
Debug
Tasks
Help
file1.cs ↔ file2.cs
using System;
namespace HelloWorld
{
class Hello
{
static void Main()
{
Console.WriteLine("Hello Wor
Console.ReadKey();
}
}
}
// A Hello World! program in C#.
using System;
namespace HelloWorld
{
class Hello
{
static void Main()
{
Console.WriteLine("Hello Worl
// Keep the console window op
Console.WriteLine("Press any
Console.ReadKey();
}
}
}

Key mutations in genetic codes in variants of concern



SCMP

Problem: Given strings S and T

Find the longest common subsequence that appear left-to-right

(but not necessarily contiguous).

For example:

S = `“SDL TQL WSL”`

T = `“SQL server on Windows Subsystem for Linux”`

`Expected output: “SQL WSL”`

How to do it? Ideas?

Idea: use LCS(n, m) to denote

- 0, if $n<0$ or $m<0$ (empty substring has no LCS with other strings)
- Otherwise: the LCS of S[:n] and T[:m]

If S[n] == T[m], then LCS(n, m) = LCS(n-1, m-1) + S[n]

Otherwise: LCS(n, m) = the_longer_of(LCS(n-1, m), LCS(n, m-1))

How to realize this?

```python
def LCS_1(S, T, n, m):
    if m<0 or n<0: return ""
    if S[n] == T[m]:
        return LCS_1(S, T, n-1, m-1) + S[n]
    elif len(LCS_1(S, T, n-1, m)) > len(LCS_1(S, T, n, m-1)):
        return LCS_1(S, T, n-1, m)
    else:
        return LCS_1(S, T, n, m-1)
```

```python
def LCS_1(S, T, n, m):
    if m<0 or n<0: return ""
    if S[n] == T[m]:
        return LCS_1(S, T, n-1, m-1) + S[n]
    elif len(LCS_1(S, T, n-1, m)) > len(LCS_1(S, T, n, m-1)):
        return LCS_1(S, T, n-1, m)
    else:
        return LCS_1(S, T, n, m-1)
```

Time complexity?

```python
from functools import lru_cache
@lru_cache(maxsize=None)
def LCS_2(S, T, n, m):
    if m<0 or n<0: return ""
    if S[n] == T[m]:
        return LCS_2(S, T, n-1, m-1) + S[n]
    elif len(LCS_2(S, T, n-1, m)) > len(LCS_2(S, T, n, m-1)):
        return LCS_2(S, T, n-1, m)
    else:
        return LCS_2(S, T, n, m-1)
```

```python
memo = {}
def LCS_3(S, T, n, m):
    if m<0 or n<0: return ""
    if (n, m) in memo: return memo[(n, m)]
    if S[n] == T[m]:
        result =  LCS_3(S, T, n-1, m-1) + S[n]
    elif len(LCS_3(S, T, n-1, m)) > len(LCS_3(S, T, n, m-1)):
        result =  LCS_3(S, T, n-1, m)
    else:
        result =  LCS_3(S, T, n, m-1)
    memo[(n, m)] = result
    return result
```

usually a “DP table”

Iteration version: build up calculation in topological order

Two loops: over substrings of S and substrings of T

e.g. LCS(“SDLDL”,”DLL”)

Try recursion and compare performance

```python
def LCS_4(S, T, n, m):
    memo = {}
    for i in range(-1, len(S)):
        for j in range(-1, len(T)):
            if i == -1 or j == -1:
                memo[(i, j)] = ""
                continue
            if S[i] == T[j]:
                memo[(i, j)] = memo[(i-1, j-1)] + S[i]
            elif len(memo[(i-1, j)]) > len(memo[(i, j-1)]):
                memo[(i, j)] = memo[(i-1, j)]
            else:
                memo[(i, j)] = memo[(i, j-1)]
    return memo[len(S)-1, len(T)-1]
```

Exercise: LCS for 3 strings?

Exercise: Shortest Common Supersequence (SCS) Problem

Given strings X and Y

Find a shortest superstring containing both X and Y as subsequence

Example:

X = “SDLTQL”

Y = “DL666”

SCS = “SDLTQL666” (may not be unique though)

```python
@lru_cache(maxsize=None)
def SCS_1(X, Y, n, m):
    if m == -1: return X[:n+1]
    if n == -1: return Y[:m+1]
    if X[n] == Y[m]: return SCS_1(X, Y, n-1, m-1) + X[n]
    if len(SCS_1(X, Y, n-1, m)) < len(SCS_1(X, Y, n, m-1)):
        return SCS_1(X, Y, n-1, m) + X[n]
    else:
        return SCS_1(X, Y, n, m-1) + Y[m]
```

Assume weight is an integer, not too large. Say, 10,000 fine. 10^{10} or 1.234 not fine.

Knapsack problem:

Bag has capacity (e.g. weight no heavier than 15 kg)

Put in items, each item has feature weight and value

```
class item:
    def __init__(self, weight, value):
        self.weight = weight
        self.value = value
```

How to put items into bag with maximum total value?

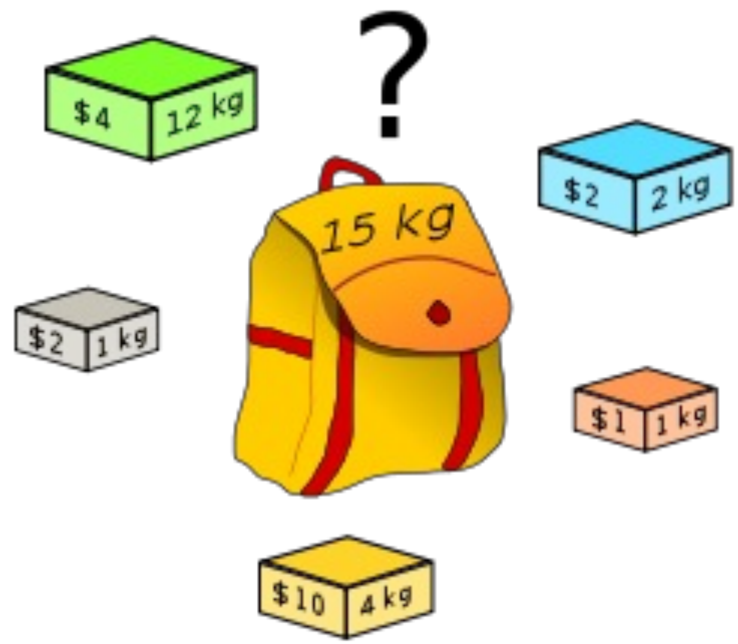


Image: Wikipedia

Idea: turn the problem into

a smaller bag and a smaller collection of items

Let S be the capacity of bag;

Let k be pointer to last item. If k = -1: no item.

knapsack(S, k) return maximal total value

Then:

```
if k = -1: knapsack(..., -1) = 0 (no item, no value)
elif S – item[k].weight < 0: knapsack(S, k) = knapsack(S, k-1)  # 要不起
else: knapsack(S, k) = max(
    knapsack(S, k-1),
    knapsack(S – item[k].weight, k-1) + item[k].value
)
```

```python
from functools import lru_cache
def knapsack_1(S, item_array):
    items = [item(i[0], i[1]) for i in item_array]

    @lru_cache(maxsize=None)
    def DP(S, k):
        if k == -1: return 0
        if S - items[k].weight < 0: return DP(S, k-1)
        return max(DP(S, k-1), DP(S-items[k].weight, k-1) + items[k].value)
    print_solution(S, items, DP)
```

```python
def knapsack_2(S, item_array):
    memo = {}
    items = [item(i[0], i[1]) for i in item_array]
    def DP(S, k):
        if k == -1: return 0
        if S - items[k].weight < 0:
            memo[(S, k)] = DP(S, k-1)
            return memo[(S, k)]
        if (S, k) not in memo:
            memo[(S, k)] = max(DP(S, k-1), DP(S-items[k].weight, k-1) + items[k].value)
        return memo[(S, k)]
    print_solution(S, items, DP)
```

```python
def knapsack_3(S, item_array):
    memo = {}
    items = [item(i[0], i[1]) for i in item_array]
    k = len(items)
    for ls in range(S+1):
        for lk in range(-1, k):
            if lk == -1:
                memo[(ls, lk)] = 0
                continue
            if ls - items[lk].weight < 0:
                memo[(ls, lk)] = memo[(ls, lk-1)]
                continue
            memo[(ls, lk)] = max(memo[(ls, lk-1)], memo[(ls-items[lk].weight, lk-1)] + items[lk].value)
    print_solution(S, items, lambda ls, lk: memo[(ls, lk)])
```

Now we get a matrix of DP(S, k). How to know which item to pick?

For example: knapsack(8, [[1, 15], [5, 10], [3, 9], [4, 5]]), we get the DP table:

[[0, 0, 0, 0, 0], # S = 0, k = -1, 0, 1, 2, 3

[0, 15, 15, 15, 15], #

Value the same: k = 1 is NOT picked. Check k=0 at same S

[0, 15, 15, 15, 15],

[0, 15, 15, 15, 15],

[0, 15, 15, 24, 24],

Value increased: k = 2 is picked. Jump to S = 4 – 3 = 1

[0, 15, 15, 24, 24],

[0, 15, 25, 25, 25],

[0, 15, 25, 25, 25],

Value increased: k = 3 is picked. Jump to S = 8 – 4 = 4

[0, 15, 25, 25, 29] # S = 8, k = -1, 0, 1, 2, 3]

```
def print_solution(S, items, DP):
    print("Total value = ", DP(S, len(items)-1))
    remaining = S
    picked = []
    for k in reversed(range(len(items))):
        if DP(remaining, k) != DP(remaining, k-1):
            picked.append(k)
            remaining -= items[k].weight
    print(picked)
```

Now we get a matrix of DP(S, k). How to know which item to pick?

For example: knapsack(8, [[1, 15], [5, 10], [3, 9], [4, 5]]), we get the DP table:

[[0, 0, 0, 0, 0], # S = 0, k = -1, 0, 1, 2, 3
[0, 15, 15, 15, 15],
[0, 15, 15, 15, 15],
[0, 15, 15, 15, 15],
[0, 15, 15, 24, 24],
[0, 15, 15, 24, 24],
[0, 15, 25, 25, 25],
[0, 15, 25, 25, 25],
[0, 15, 25, 25, 29] # S = 8, k = -1, 0, 1, 2, 3]

Value the same: k = 1 is NOT picked. Check k=0 at same S

Value increased: k = 0 is picked. Jump to S = 1 – 1 = 0

Value increased: k = 2 is picked. Jump to S = 4 – 3 = 1

Value increased: k = 3 is picked. Jump to S = 8 – 4 = 4

```
def print_solution(S, items, DP):
    print("Total value = ", DP(S, len(items)-1))
    remaining = S
    picked = []
    for k in reversed(range(len(items))):
        if DP(remaining, k) != DP(remaining, k-1):
            picked.append(k)
            remaining -= items[k].weight
    print(picked)
```

Comment about knapsack problem:

For general S, the problem is NP-complete!

Because:

input bit $\propto$ number of digits of S

Time complexity $O(S \times |item\_array|)$ is considered exponential.

In fib(n): we know for sure

how to reduce to smaller problems

fib(n) = fib(n-1) + fib(n-2)

In longest common subsequence, knapsack:

use if statement but still definite.

Sometimes, we need blind (brute force) search

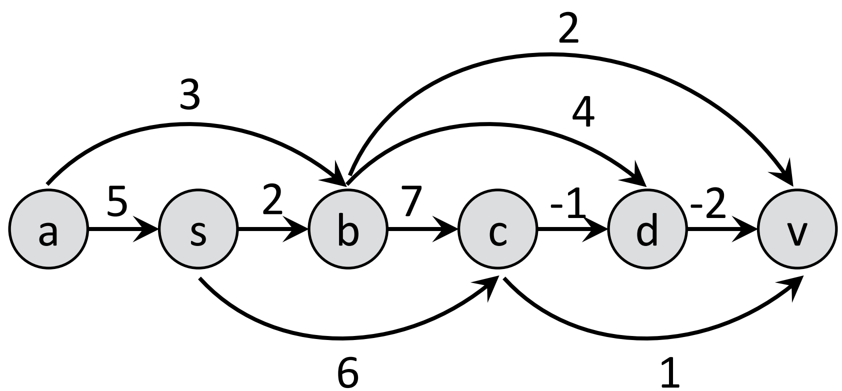
for all possibilities.

Example: shortest path problems

Shortest path of DAG revisited

Dynamic programming example:

Shortest path from s on a DAG.



Previous method:

(1) Topological sort

(2) Relax each right vertex

Thinking in the recursion way: to find $\delta(s, v)$:

```
def delta(s, v):
    return min([delta(s, u) + w(u, v) for u in in_degree(v)])
```

Time complexity? Exponential.

How to improve it?

Thinking in the recursion way: to find $\delta(s, v)$:

Time complexity? Exponential.

How to improve it?

```
from functools import lru_cache
@lru_cache(maxsize=None)
def delta(s, v):
    return min([delta(s, u) + w(u, v) for u in in_degree(v)])
```

O(V+E)

Thinking in the recursion way: to find $\delta(s, v)$:

Time complexity? Exponential.

How to improve it?

```
from functools import lru_cache
@lru_cache(maxsize=None)
def delta(s, v):
    return min([delta(s, u) + w(u, v) for u in in_degree(v)])
```

O(V+E)

Too opaque? DIY

```python
from functools import lru_cache
@lru_cache(maxsize=None)
def delta(s, v):
    return min([delta(s, u) + w(u, v) for u in in_degree(v)])
```

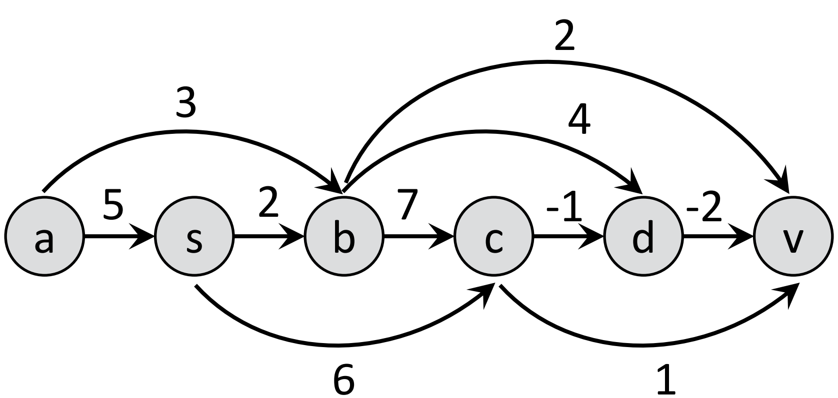
Too opaque? DIY

```python
memo = {}
def delta(s, v):
    attempts = []
    for u in in_degree(v):
        delta_s_u = memo[u] if u in memo else delta(s, u)
        attempts.append(delta_s_u + w(u, v))
    delta_s_v = min(attempts)
    memo[v] = delta_s_v
    return delta_s_v
```

To write a non-recursive version?

1. Find out what needed – topological sort
2. Start from s, calculate $\delta(s, v)$ for each v to the right of s

The same as the previous method ☺



Previous method:

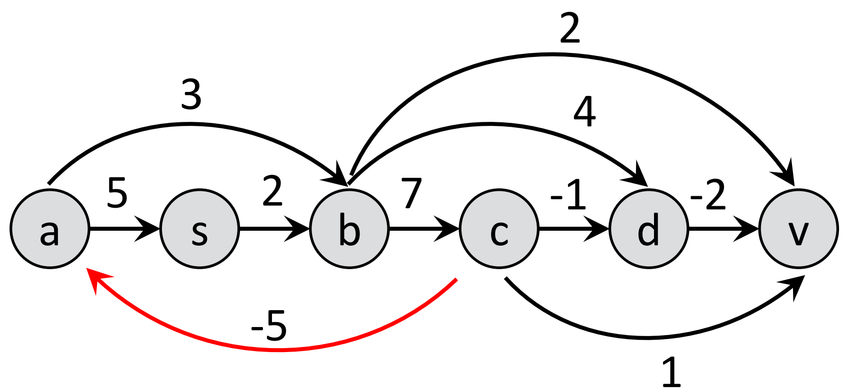
(1) Topological sort

(2) Relax each right vertex

Previously rely on smart ideas. Now: systematic.

General single-source shortest path problem revisited

Can we directly use algorithm for DAG?



Does DAG algorithm still work?

Recursion version:

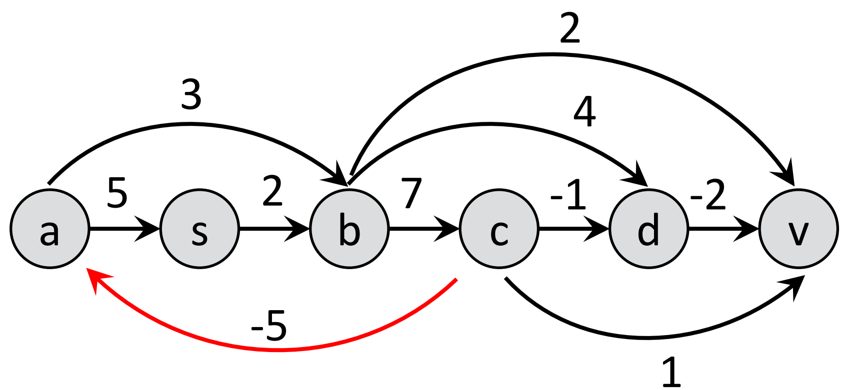
Memorize and use recursion → Infinite loop

Iteration version:

(1) Topological sort

(2) Relax each right vertex → No topological order

Can we directly use algorithm for DAG?



Way out?

Not to visit a vertex visited before?

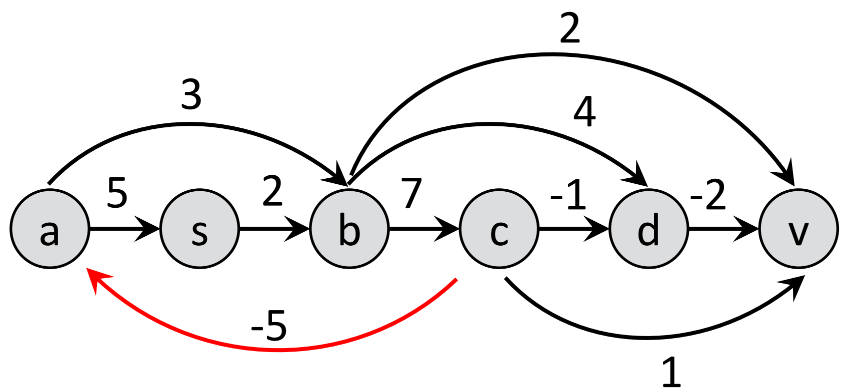
Does not work. E.g. vertex c.

The first time of visit: edge -1 is used.

The second time of visit: edge -5 is used.

If not visiting c, -5 is neglected and $\delta(s, a)$ is wrong.

Can we directly use algorithm for DAG?

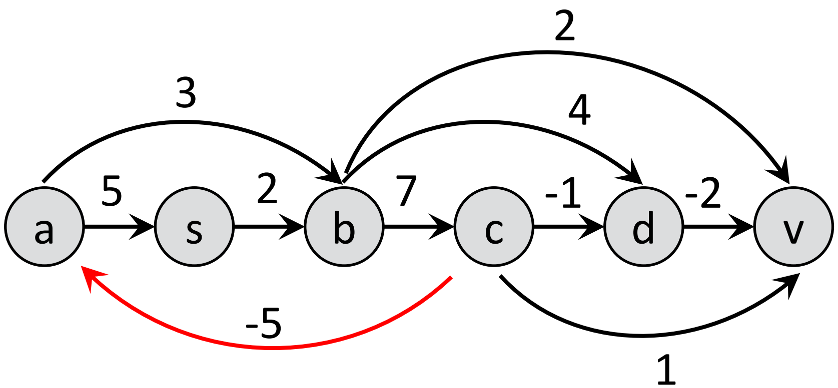


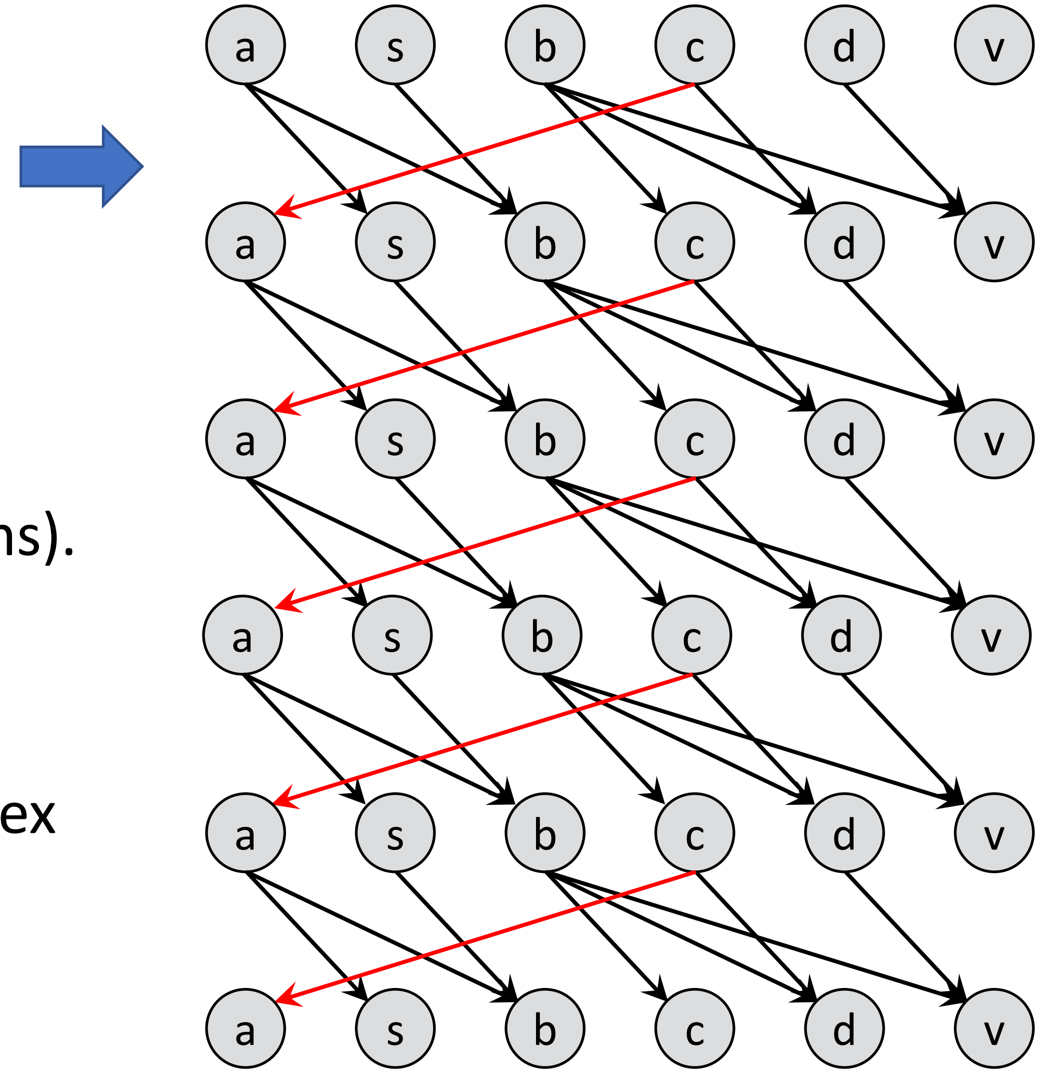
Way out?

Turn a space diagram into a spacetime diagram

And time never has backward edges ☺

Can we directly use algorithm for DAG?





Problem converted to DAG

with V x V vertices (subproblems).

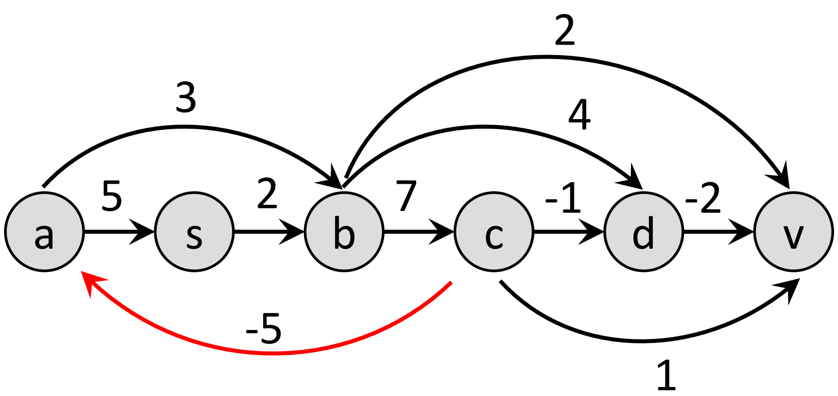
Time for each subproblem

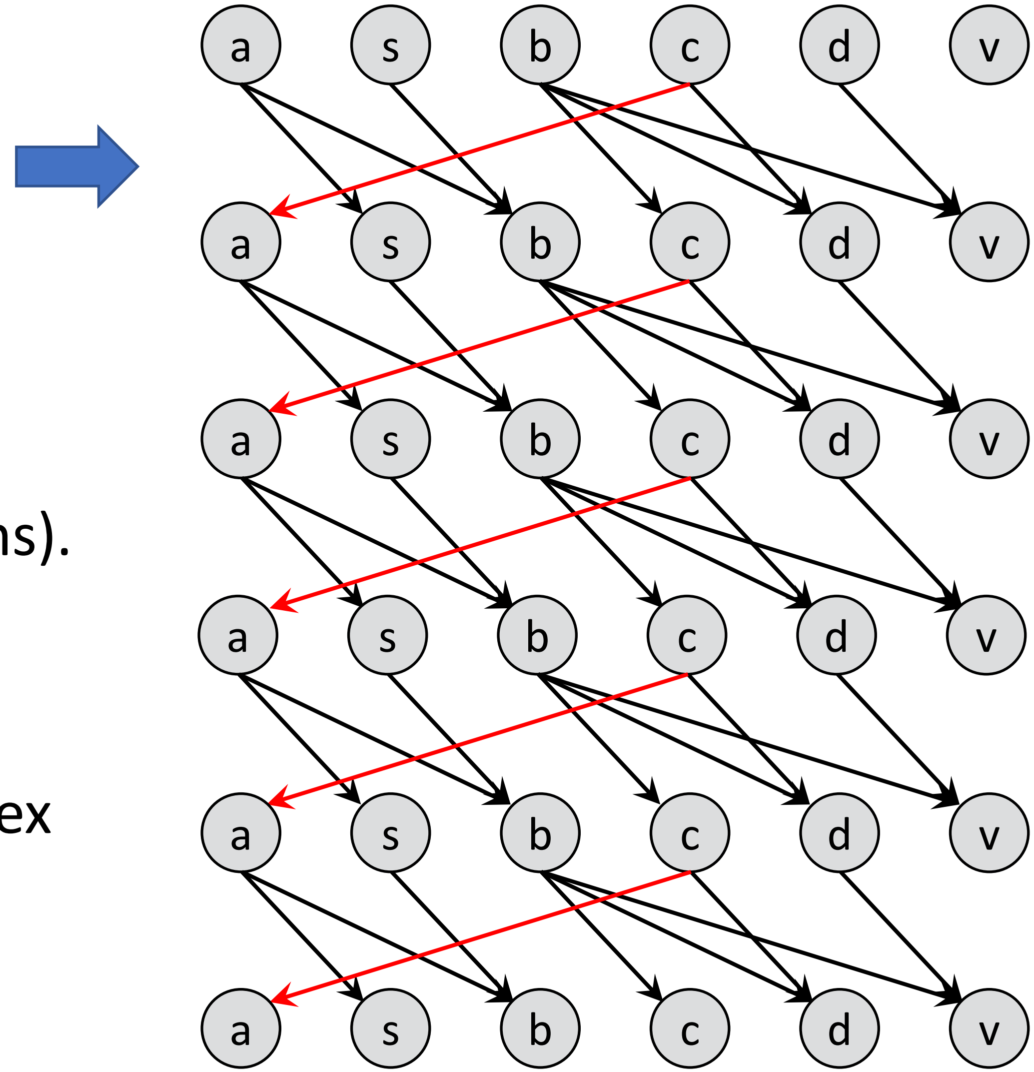
= # incoming edges of that vertex

Total time complexity: O(VE)

Does this look familiar?

Can we directly use algorithm for DAG?





Problem converted to DAG

with V x V vertices (subproblems).

Time for each subproblem

= # incoming edges of that vertex

Total time complexity: O(VE)

This is in fact identical to Bellman-Ford

Exercise:

Text justification (word wrap) problem

Given a string, and a line-width:

(Cost of a line) = (Number of extra spaces in a line)

(Total cost) = (Sum of costs of all lines)

How to minimize total cost for word wrap?

Summary of dynamic programming?

Recursive version:

1. Reduce to smaller problems
   - Two direct recursive calls (Fibonacci)
   - Using if statements to try (LCS, Knapsack)
   - Using for statements to try (shortest path, word wrap)
2. Remember result of called functions

Iterative version:

1. Construct “dependency” graph
2. Compute answers in topological order