CS 577 - Dynamic Programming Primer

Manolis Vlatakis

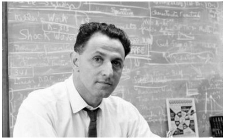
Department of Computer Sciences
University of Wisconsin – Madison

Fall 2024

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Dynamic Programming

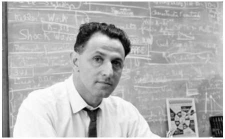
DYNAMIC PROGRAMMING



Richard Bellman

It is "programming" that is "dynamic"!

Dynamic Programming



Richard Bellman

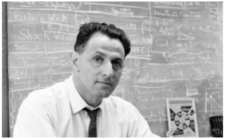
It is "programming" that is "dynamic"!

Why "Dynamic Programming"?

Reasons for the name:

- In the 1950s, "programming" was about "planning" rather than coding.
- "Dynamic" is exciting – Air Force director didn't like research and wanted pizzazz.
- "Dynamic" sounds better than "linear" (Re: rival Dantzig).

Dynamic Programming



Richard Bellman

It is "programming" that is "dynamic"!

What is it?

- Your new favourite algorithmic technique.
- Extreme Divide and Conquer
- Many sub-problems, but not quite brute-force.
- Dynamic in that it calculates a bunch of solutions from the "smallest" to the "largest".

Let's compute a recurrence

EXAMPLE: FACTORIAL FUNCTION

Example 1: Compute the factorial function $F(n) = n!$ for an arbitrary non-negative integer n. Since:

$$n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1, \quad 0! = 1$$

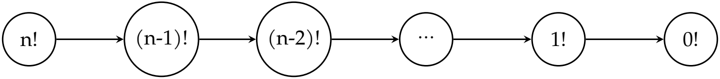
By definition, we can compute $Factorial(n) = Factorial(n-1) \cdot n$ using the following recursive algorithm.

- **Algorithm** *Factorial*(n) // Computes n! recursively // Input: A non-negative integer n // Output: The value of n!
  - if n == 0 return 1
  - else return Factorial(n-1) * n

🤔What is the size of the recurrence tree?

Factorial Recurrence Tree

$$\text{Factorial}(n) = \text{Factorial}(n-1)\cdot n, \quad \text{Factorial}(0) = 1$$



Factorial Recurrence Tree

$$\text{Factorial}(n) = \text{Factorial}(n-1) \cdot n, \quad \text{Factorial}(0) = 1$$

$$n! \longrightarrow (n\text{-}1)! \longrightarrow (n\text{-}2)! \longrightarrow \cdots \longrightarrow 1! \longrightarrow 0!$$

Interesting Observation:

- **Multiplications in Factorial Computation:**
  The number of multiplications $M(n)$ needed to compute $F(n) = n!$ must satisfy the recurrence:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0.$$

- **Explanation:** $M(n-1)$ multiplications are used to compute $F(n-1)$, and one more multiplication is required to multiply the result by n.

FACTORIAL RECURRENCE TREE

$$\text{Factorial}(n) = \text{Factorial}(n-1)\cdot n, \quad \text{Factorial}(0) = 1$$

$$n! \longrightarrow (n-1)! \longrightarrow (n-2)! \longrightarrow \cdots \longrightarrow 1! \longrightarrow 0!$$

Interesting Observation:

- **Multiplications in Factorial Computation:**
  The number of multiplications $M(n)$ needed to compute $F(n) = n!$ must satisfy the recurrence:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0.$$

- **Explanation:** $M(n-1)$ multiplications are used to compute $F(n-1)$, and one more multiplication is required to multiply the result by n.

How much is exactly $M(n)$?

Factorial Recurrence Tree

$$\text{Factorial}(n) = \text{Factorial}(n-1) \cdot n, \quad \text{Factorial}(0) = 1$$

$$n! \to (n\text{-}1)! \to (n\text{-}2)! \to \cdots \to 1! \to 0!$$

Interesting Observation:

- **Multiplications in Factorial Computation:**
  The nu… ompute
  $F(n) =$ …

Interesting!!!

We compute $\boxed{n!}$, which is a huge number, in only $\boxed{n}$ multiplications!

- **Explanation:** $M(n-1)$ multiplications are used to compute $F(n-1)$, and one more multiplication is required to multiply the result by n.

How much is exactly $M(n)$? $\boxed{M(n) = n}$

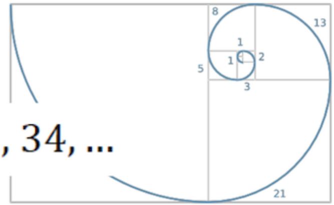
Fibonacci Sequence

Let’s see another sequence!!!

Fibonacci Sequence

Let's see another sequence!!!

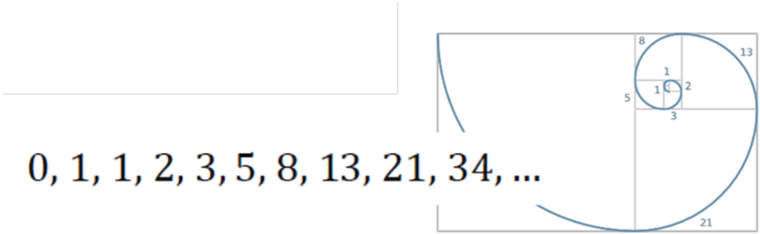
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...



Definition

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

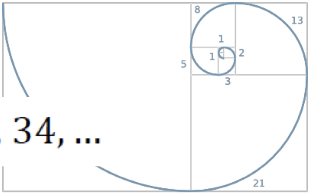
FIBONACCI SEQUENCE



Definition

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

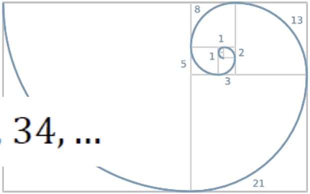
LET'S COMPUTE FIBONACCI SEQUENCE



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```
fib1(n)
  if n = 0 then return 0
  if n = 1 then return 1
  return fib1(n-1) + fib1(n-2)
```

Let's compute Fibonacci Sequence



0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```
fib1(n)
  if n = 0 then return 0
  if n = 1 then return 1
  return fib1(n-1) + fib1(n-2)
```

🤔 What is the main problem with this code???

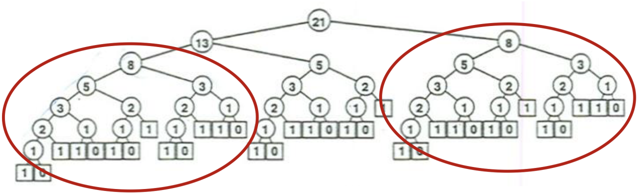
Let's compute Fibonacci Sequence

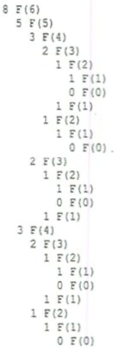
It is a big recurrence tree!!!

Let's compute Fibonacci Sequence

It is a big recurrence tree!!!

```
fib1(n)
  if n = 0 then return 0
  if n = 1 then return 1
  return fib1(n-1) + fib1(n-2)
```

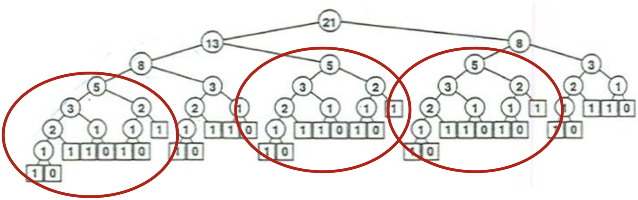


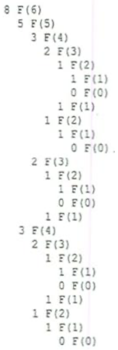


Let's compute Fibonacci Sequence

It is a very big recurrence tree!!!

```
fib1(n)
  if n = 0 then return 0
  if n = 1 then return 1
  return fib1(n-1) + fib1(n-2)
```



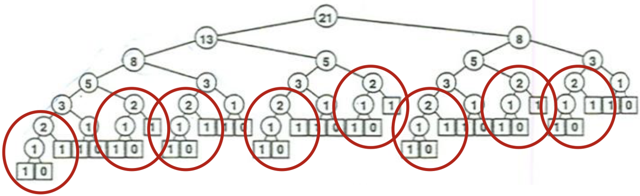


Let's compute Fibonacci Sequence

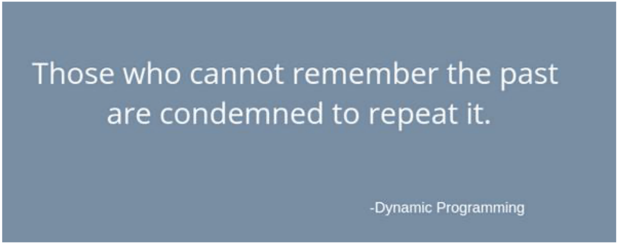
It is a very stupidly big recurrence tree!!!

```
fib1(n)
  if n = 0 then return 0
  if n = 1 then return 1
  return fib1(n-1) + fib1(n-2)
```

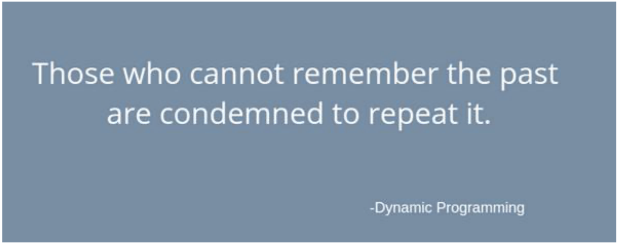
```
8 F(6)
 5 F(5)
  3 F(4)
   2 F(3)
    1 F(2)
     1 F(1)
     0 F(0)
    1 F(1)
   1 F(2)
    1 F(1)
    0 F(0)
  2 F(3)
   1 F(2)
    1 F(1)
    0 F(0)
   1 F(1)
 3 F(4)
  2 F(3)
   1 F(2)
    1 F(1)
    0 F(0)
   1 F(1)
  1 F(2)
   1 F(1)
   0 F(0)
```



Let's compute Fibonacci Sequence



Let's compute Fibonacci Sequence



What is the size of the tree (approximately - or - exactly?)

Computing the size of the tree

Introduction

The algorithm's basic operation is clearly addition, so let $A(n)$ be the number of additions performed by the algorithm in computing $F(n)$.

Computing the size of the tree

Introduction

The algorithm's basic operation is clearly addition, so let $A(n)$ be the number of additions performed by the algorithm in computing $F(n)$.

Recurrence Setup

Then, the numbers of additions needed for computing $F(n-1)$ and $F(n-2)$ are $A(n-1)$ and $A(n-2)$, respectively, and the algorithm needs one more addition to compute their sum. Thus, we get the following recurrence for $A(n)$:

- $A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1$
- $A(0) = 0, \quad A(1) = 0$

COMPUTING THE SIZE OF THE TREE

Introduction

The algorithm's basic operation is clearly addition, so let $A(n)$ be the number of additions performed by the algorithm in computing $F(n)$.

Recurrence Setup

Then, the numbers of additions needed for computing $F(n-1)$ and $F(n-2)$ are $A(n-1)$ and $A(n-2)$, respectively, and the algorithm needs one more addition to compute their sum. Thus, we get the following recurrence for $A(n)$:

- $A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1$
- $A(0) = 0, \quad A(1) = 0$

Professor!!!

But we don't know how to solve inhomogenius recurrences

Computing the size of the tree

To have an estimation you don't need really!!!

The algorithm's basic operation is clearly addition, so let $A(n)$

Can you give me now an estimation???

Observe: $\boxed{A(n-2) \leq A(n-1) \leq A(n)}$

Why???

Then, the numbers of additions needed for computing $F(n-1)$ and $F(n-2)$ are $A(n-1)$ and $A(n-2)$, respectively, and the algorithm needs one more addition to compute their sum.

Thus, we get the following recurrence for $A(n)$:

- $A(n) = A(n-1) + A(n-2) + 1 \quad$ for $n > 1$
- $A(0) = 0, \quad A(1) = 0$

Professor!!!

But we don't know how to solve inhomogenius recurrences

Computing the size of the tree

The size of tree

- $A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1$
- $A(0) = 0, \quad A(1) = 0$

Computing the size of the tree

The size of tree

- $A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1$
- $A(0) = 0, \quad A(1) = 0$

- Upper bound:
  - $A(n) \leq A(n-1) + A(n-1) + 1 \Rightarrow A(n) = 2A(n-1) + 1$

COMPUTING THE SIZE OF THE TREE

The size of tree

- $A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1$
- $A(0) = 0, \quad A(1) = 0$

- Upper bound:
  - $A(n) \leq A(n-1) + A(n-1) + 1 \Rightarrow A(n) = 2A(n-1) + 1$
  - $\Rightarrow A(n) = 2(2A(n-2) + 1) + 1 = \cdots = 2^n A(0) + n$

COMPUTING THE SIZE OF THE TREE

The size of tree

- $A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1$
- $A(0) = 0, \quad A(1) = 0$

- Upper bound:
  - $A(n) \leq A(n-1) + A(n-1) + 1 \Rightarrow A(n) = 2A(n-1) + 1$
  - $\Rightarrow A(n) = 2(2A(n-2) + 1) + 1 = \cdots = 2^n A(0) + n$
- Lower bound:
  - $A(n) \leq A(n-2) + A(n-2) + 1 \Rightarrow A(n) = 2A(n-2) + 1$

Computing the size of the tree

The size of tree

- $A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1$
- $A(0) = 0, \quad A(1) = 0$

- Upper bound:
  - $A(n) \leq A(n-1) + A(n-1) + 1 \Rightarrow A(n) = 2A(n-1) + 1$
  - $\Rightarrow A(n) = 2(2A(n-2) + 1) + 1 = \cdots = 2^n A(0) + n$
- Lower bound:
  - $A(n) \leq A(n-2) + A(n-2) + 1 \Rightarrow A(n) = 2A(n-2) + 1$
  - $\Rightarrow A(n) = 2(2A(n-2) + 1) + 1 = \cdots = 2^{n/2} A(0) + n/2$

Computing the size of the tree

The size of tree

- $A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1$
- $A(0) = 0, \quad A(1) = 0$

- Upper bound:
  - $A(n) \leq A(n-1) + A(n-1) + 1 \Rightarrow A(n) = 2A(n-1) + 1$
  - $\Rightarrow A(n) = 2(2A(n-2) + 1) + 1 = \cdots = 2^n A(0) + n$
- Lower bound:
  - $A(n) \leq A(n-2) + A(n-2) + 1 \Rightarrow A(n) = 2A(n-2) + 1$
  - $\Rightarrow A(n) = 2(2A(n-2) + 1) + 1 = \cdots = 2^{n/2} A(0) + n/2$

Thus:

$$O(2^{n/2}) \leq A(n) \leq O(2^n)$$

Computing the size of the tree

Exact Computation!!!

Inhomogeneous Recurrence

The recurrence $A(n) - A(n-1) - A(n-2) = 1$ is quite similar to the recurrence for $F(n) - F(n-1) - F(n-2) = 0$!!!

Homogenization Trick

We can reduce our inhomogeneous recurrence to a homogeneous one by rewriting it as:

$$[A(n)+1] - [A(n-1)+1] - [A(n-2)+1] = 0$$

and substituting $B(n) = A(n) + 1$.

Computing the size of the tree

Exact Computation!!!

Inhomogeneous Recurrence

The recurrence $A(n) - A(n-1) - A(n-2) = 1$ is quite similar to the recurrence for $F(n) - F(n-1) - F(n-2) = 0$!!!

Homogenization Trick

We can reduce our inhomogeneous recurrence to a homogeneous one by rewriting it as:

$$[A(n) + 1] - [A(n-1) + 1] - [A(n-2) + 1] = 0$$

and substituting $B(n) = A(n) + 1$.

$$[B(n)] - [B(n-1)] - [B(n-2)] = 0$$


WHAT?

Homogenization Trick

- What is $B(n) = A(n) + 1$??

Homogenization Trick

- What is $B(n) = A(n) + 1$?? The size of the tree!!
- And what kind of equation $B(n)$ satisfies??

Homogenization Trick

- What is $B(n) = A(n) + 1$?? The size of the tree!!
- And what kind of equation $B(n)$ satisfies??

$$[B(n)] - [B(n-1)] - [B(n-2)] = 0$$

B(n) is a Fibonacci!!!


WHAT?

An interesting paradox

Thus, in the recursion:

- If we need to compute $n!$, we need n operations.
- If we need to compute $Fib(n)$, we need $Fib(n) = \Omega(2^{n/2})$, which is greater than $2^n + n$.

But... $n!$ is a larger number. Huh?

AN INTERESTING PARADOX

Thus, in the recursion:

- If we need to compute $n!$, we need n operations.
- If we need to compute $Fib(n)$, we need $Fib(n) = \Omega(2^{n/2})$, which is greater than $2^n + n$.

But... $n!$ is a larger number. Huh?

🤔 What is the difference between them?

An interesting paradox

Thus, in the recursion:

- If we need to compute $n!$, we need n operations.
- If we need to compute $Fib(n)$, we need $Fib(n) = \Omega(2^{n/2})$, which is greater than $2^n + n$.

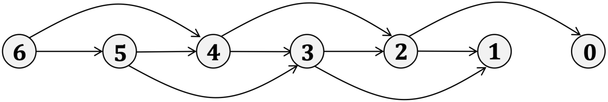
But... $n!$ is a larger number. Huh?

What is the difference between them?

It is <u>not</u> the type of equation. ☺
In $Factorial(n)$, it is not necessary to remember $Factorial(n-1), Factorial(n-2), \ldots$ multiple times!

FIBONACCI SEQUENCE

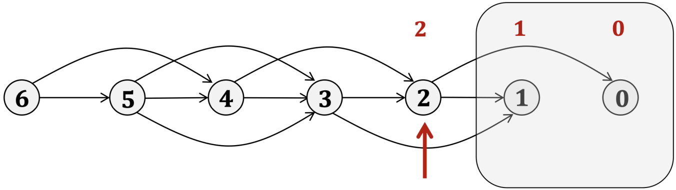
WHICH MEMORY IS IMPORTANT?




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

Fibonacci Sequence

Which Memory is important?

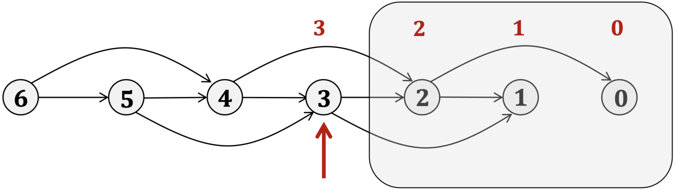


fib(2) = fib(1) + fib(0)

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

Fibonacci Sequence

Which Memory is important?

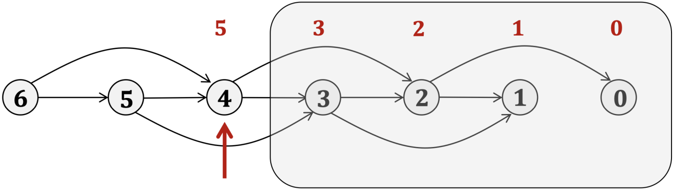


`fib(3) = fib(2) + fib(1)`

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

FIBONACCI SEQUENCE

WHICH MEMORY IS IMPORTANT?

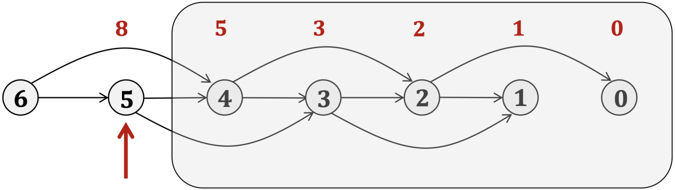


fib(4) = fib(3) + fib(2)

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

Fibonacci Sequence

Which Memory is important?

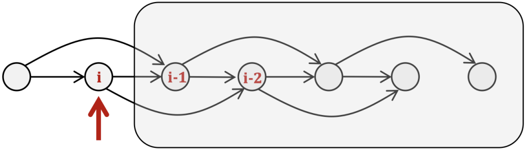


`fib(5) = fib(4) + fib(3)`

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

Fibonacci Sequence

Which Memory is important?




```
fib2(n)

1. if n = 0 then return 0
2. if n = 1 then return 1
3. F(0) = 0, F(1) = 1
4. for i = 2, 3, …, n
        F(i) = F(i-1) + F(i-2)
5. return F(n)
```

Moment of Truth

Of course...we could have only two variables ☺

MOMENT OF TRUTH

OF COURSE...WE COULD HAVE ONLY TWO VARIABLES ☺

Aesop's Moral

In both problems, finding the solution was straightforward!

Moment of Truth

Of course...we could have only two variables ☺

Aesop's Moral

In both problems, finding the solution was straightforward!

Aesop's Moral

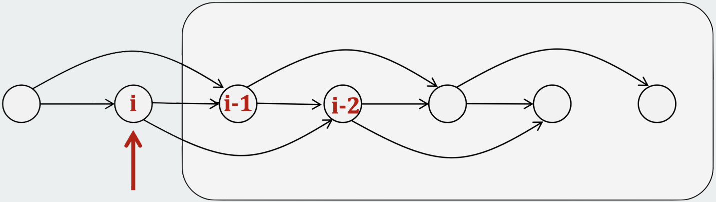
However, finding the solution is equivalent to store the correct subproblems

Mo

Of c

Solving Subproblems

I can solve a problem easily
if I define and keep the solutions
to the smaller subproblems !!!




```
fib(i) = fib(i-1) + fib(i-2)
```

The Crow and the Pitcher
The Dog and his Shadow
The Fox and the Stork
The Fox and the Grapes
The Lion and the Mouse
The Miller, His Son, and the Donkey
The Tortoise and the Hare

g the
vard!

ution
is equivalent to store the correct
subproblems

Mo

Of c

Solving Subproblems

I can solve a problem easily
if I define and keep the solutions
to the smaller subproblems !!!

g the
ward!

ution
correct

Can you do better?

Prove the equality

$$\begin{bmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \quad \text{for } n \geq 1.$$

🤔How can you do it in $O(\log n)$?

Shortest Path in DAGs

- A DAG is a Directed Acyclic Graph.
- The shortest path problem involves finding the minimum distance from a source node to all other nodes.

SHORTEST PATH IN DAGS

- A DAG is a Directed Acyclic Graph.
- The shortest path problem involves finding the minimum distance from a source node to all other nodes.

What is the special characteristic of a DAG
except the fact that is acyclic ☺?

SHORTEST PATH IN DAGS

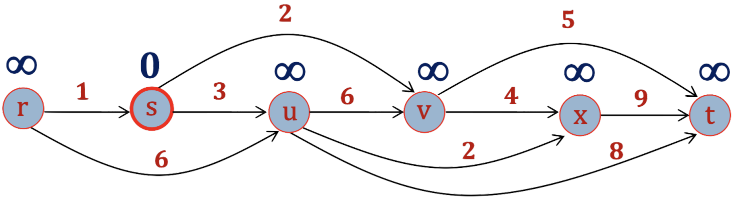
- A DAG is a Directed Acyclic Graph.
- The shortest path problem involves finding the minimum distance from a source node to all other nodes.

What is the special characteristic of a DAG
except the fact that is acyclic ☺?

Every DAG has a topological order!!! – Why is this useful?

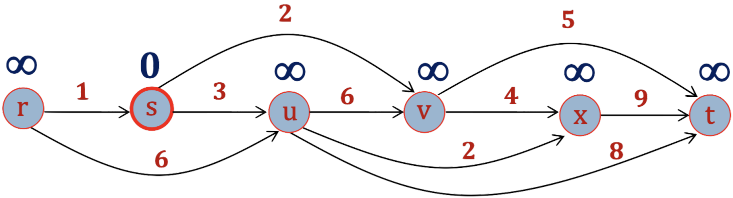
SHORTEST PATHS IN DAGS

Characteristic of a DAG: Topological Sorting



Shortest Paths in DAGs

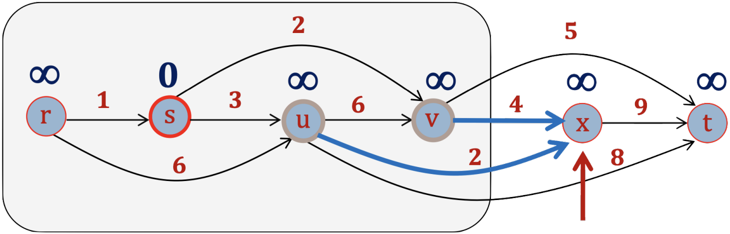
Characteristic of a DAG: Topological Sorting



Why does this characteristic help in computing shortest paths from a node, say *s*?

Shortest Paths in DAGs

Characteristic of a DAG: Topological Sorting

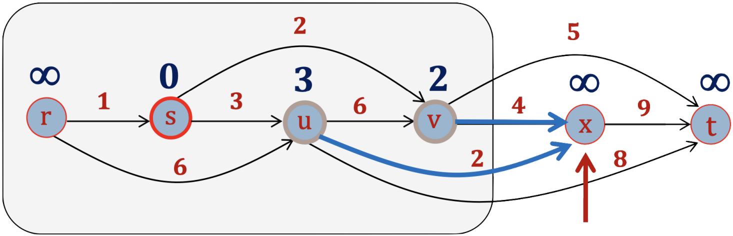


Let's focus on a node, say x.

The only way to reach x is through its predecessors: v or u.

Shortest Paths in DAGs

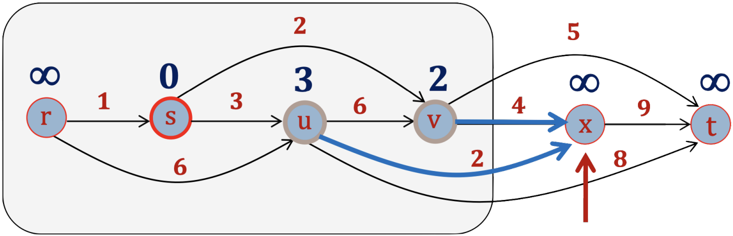
Characteristic of a DAG: Topological Sorting



How many optimal candidate paths do exist from *s* to *x*?

Shortest Paths in DAGs

Characteristic of a DAG: Topological Sorting

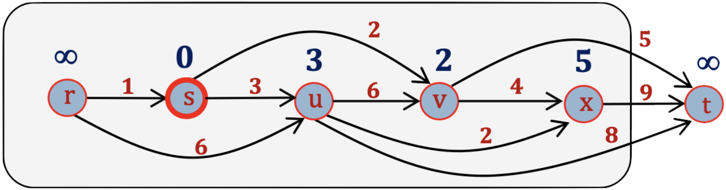


How many optimal candidate paths do exist from s to x?

Thus, to find the shortest distance from s to x, it is enough to compare the two paths:

$$d(x) = \min\{d(v) + 4, d(u) + 2\}$$

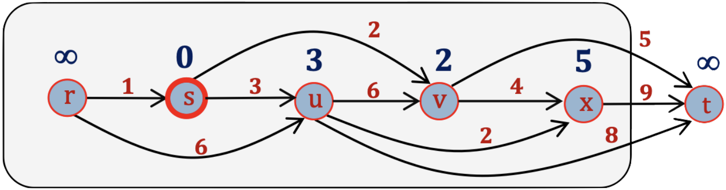
Shortest Paths in DAGs



- A similar relationship holds for every node! For example, for node t:

$$d(t) = \min\{d(x) + 9, d(v) + 5, d(u) + 8\}$$

Shortest Paths in DAGs

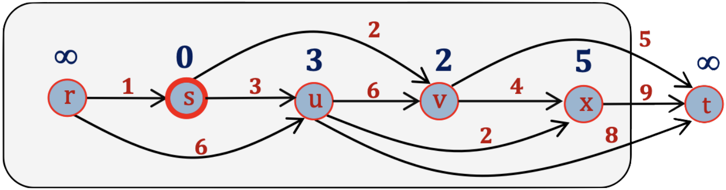


- A similar relationship holds for every node! For example, for node t:

$$d(t) = \min\{d(x) + 9, d(v) + 5, d(u) + 8\}$$

- If we compute the values $d(.)$ following the topological order, then:

SHORTEST PATHS IN DAGS



- A similar relationship holds for every node! For example, for node t:

$$d(t) = \min\{d(x)+9, d(v)+5, d(u)+8\}$$

- If we compute the values $d(.)$ following the topological order, then:

By the time we reach node x, we will have all the information needed to compute $d(x)$.

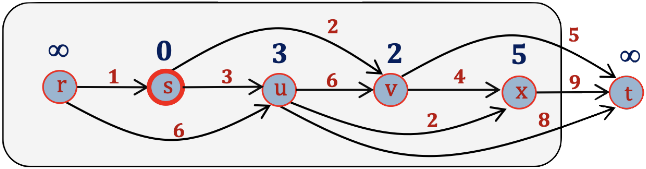
Shortest Paths in DAGs

Algorithm SP-DAG

1. `Initialize`(G, s)
2. `Topological-Sorting`(G)
3. For each node $x \in V - \{s\}$ in $\boxed{\underline{\text{topological order}}}$:

$$\boxed{d(x) = \min_{(u,x)\in E}\{d(u) + w(u,x)\}}$$

4. Return $d(.)$



Keep these two elements of the algorithm in mind!!!

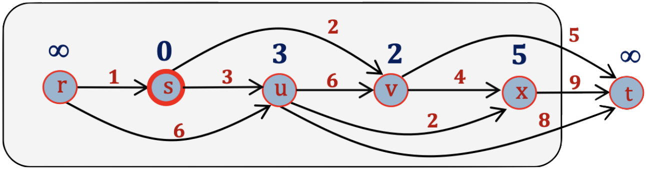
Shortest Paths in DAGs

Observations #1

- The SP-DAG algorithm solves subproblems of the form:

$$\{d(x) \mid x \in V - \{s\}\}$$

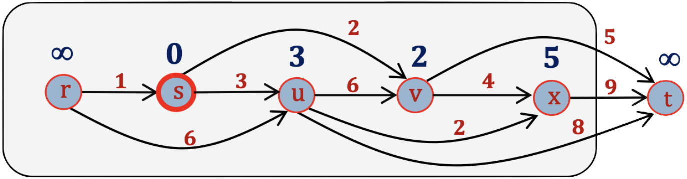
- It starts from the **smallest** subproblem and moves on to solve **larger** subproblems!!!
- A subproblem is considered **large** if we need to solve many other subproblems before arriving at it!!!



Shortest Paths in DAGs

Observations #2

- At each node x, the SP-DAG algorithm computes a function of the distances $d(.)$ from the predecessors of node x.
- Here, the function is a **minimum sum** of distances!
- The function could just as easily be:
  - **Maximum**: In which case we would compute the maximum paths, or
  - **Minimum product**: Where we calculate the path with the smallest product of edges.



Shortest Paths in DAGs

Observations #3

Find the differences in the implementation

Case #1:

```
// DP Shortest Path from source 's' to all nodes using a for loop
function ShortestPathFromS(DAG, s):
    // Step 1: Initialize distances
    for each node v in DAG:
        dist[v] = ∞
    dist[s] = 0

    // Step 2: Topologically sort all nodes in DAG
    topoOrder = TopologicalSort(DAG)

    // Step 3: Process nodes in topological order
    for each node u in topoOrder:
        for each edge (u, v) in DAG:
            if dist[v] > dist[u] + weight(u, v):
                dist[v] = dist[u] + weight(u, v)

    // Return the shortest path distances from s to all nodes
    return dist
```

Case #2:

```
// Step 1: Initialize memoization table
memo = {}

// Step 2: Recursive function with memoization
function DP(u):
    if u == t:
        return 0
    if u in memo:
        return memo[u]

    // Initialize the shortest path to 't' from 'u' as infinity
    shortest = ∞

    // Process each neighbor v of u
    for each edge (u, v) in DAG:
        shortest = min(shortest, DP(v) + weight(u, v))

    // Memoize and return the result
    memo[u] = shortest
    return memo[u]

// Step 3: Compute shortest path from source 's' to any node
for each node u in DAG:
    DP(u)

// Return the shortest path distance from source node s to target t
return DP(s)
```

Shortest Paths in DAGs

Observations #3

Case #1:

```
// DP Shortest Path from source 's' to all nodes using a for loop
function ShortestPathFromS(DAG, s):
    // Step 1: Initialize distances
    for each node v in DAG:
        dist[v] = ∞
    dist[s] = 0

    // Step 2: Topologically sort all nodes in DAG
    topoOrder = TopologicalSort(DAG)

    // Step 3: Process nodes in topological order
    for each node u in topoOrder:
        for each edge (u, v) in DAG:
            if dist[v] > dist[u] + weight(u, v):
                dist[v] = dist[u] + weight(u, v)

    // Return the shortest path distances from s to all nodes
    return dist
```

Shortest Paths in DAGs

Observations #3

Case #2:

```
// Step 1: Initialize memoization table
memo = {}

// Step 2: Recursive function with memoization
function DP(u):
    if u == t:
        return 0
    if u in memo:
        return memo[u]

    // Initialize the shortest path to 't' from 'u' as infinity
    shortest = ∞

    // Process each neighbor v of u
    for each edge (u, v) in DAG:
        shortest = min(shortest, DP(v) + weight(u, v))

    // Memoize and return the result
    memo[u] = shortest
    return memo[u]

// Step 3: Compute shortest path from source 's' to any node
for each node u in DAG:
    DP(u)

// Return the shortest path distance from source node s to target t
return DP(s)
```

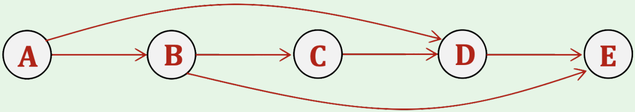
One Million Dollar Question for DP

🤔Why is the assumption of DAG crucial for the previous DP?

One Million Dollar Question for DP

🤔Why is the assumption of DAG crucial for the previous DP?

DAG explains the order of the subproblems that we have to solve to compose the final solution!!!



Comparison of Methods

Divide & Conquer,
Greedy Algo,
Dynamic Programming...
What should I choose?


Divide & Conquer, Greedy Algo, Dynamic Programming... What should I choose?
The optimal one ☺

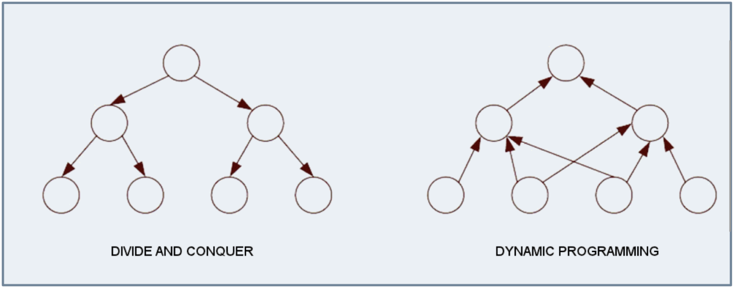
What are finally their differences?


What are finally their differences?


Let's check together!

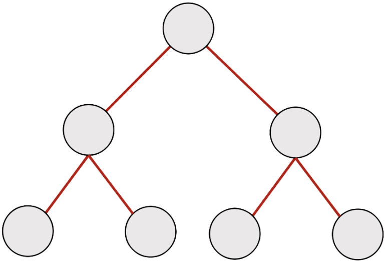
Divide & Conquer VS Dynamic Programming

What is the difference between the subproblems & the computation tree in DC & DP?



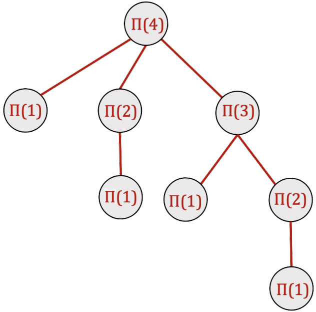
Dynamic Programming Characteristics

- **Divide-and-Conquer vs Dynamic Programming!!!**
  - ✓ In the Divide-and-Conquer technique, a problem of size n is expressed as subproblems $\Pi(1), \Pi(2), \ldots, \Pi(k)$ that are significantly smaller (for example $n/2$) and do not overlap!
  - ✓ Due to the sharp decrease in the size of Π, the recursion tree has:
    - Depth: $O(\log n)$
    - Number of nodes: $O(n)$



Dynamic Programming Characteristics

- **Divide-and-Conquer vs Dynamic Programming!!!**
  - ✓ In the Dynamic Programming technique, the subproblems $\Pi(1), \Pi(2), \ldots, \Pi(k)$ are slightly smaller. For example, $\Pi(i)$ depends on $\Pi(i-1)$, and the subproblems overlap!
  - ✓ Here, the recursion tree typically has:
    - Depth: $O(n)$
    - Number of nodes: $O(c^n)$, where $c > 1$
  - Exponential number of nodes!!!



Differences between Algorithmic Techniques

- **Greedy vs Dynamic Programming!!!**
  - ✓ In the Dynamic Programming technique, the subproblems $\Pi(1), \Pi(2), \ldots, \Pi(k)$ are slightly smaller. For example, $\Pi(i)$ depends on $\Pi(i-1)$, and the subproblems overlap!
  - ✓ In the Greedy Algorithms technique, the subproblems $\Pi(1), \Pi(2), \ldots, \Pi(k)$ are slightly smaller. For example, $\Pi(i)$ depends on $\Pi(i-1)$, and the subproblems overlap!
  - ✓ Greedy algorithms are like Factorial Problem:
    - If you sort correctly (greedy criterion) your data, every optimal solution depends only on the optimal solution of the previous step and a local choice!!!
    - Depth: $O(n)$
    - Number of nodes: $O(n)$, where $c > 1$
  - Linear number of nodes!!!

Differences between Algorithmic Techniques

- **Greedy vs Dynamic Programming!!!**

$\boxed{\text{Opt(n) = Greedy(n-1)} \oplus \text{Local-Choice(n)}}$

$\downarrow$

$\boxed{\text{Opt(n-1) = Greedy(n-2)} \oplus \text{Local-Choice(n-1)}}$

$\downarrow$

$\boxed{\text{Opt(n-2) = Greedy(n-1)} \oplus \text{Local-Choice(n-2)}}$

$\downarrow$

$\boxed{\cdots}$

$\downarrow$

$\boxed{\text{Opt(1) = Greedy(0)} \oplus \text{Local-Choice(1)}}$

$\downarrow$

$\boxed{\text{Opt(0) = Greedy-Choice}}$

DYNAMIC PROGRAMMING CHARACTERISTICS

The Basic Idea!!!... Solving problem Π with DP:

DP Design Steps

1. **Compute** a set of subproblems of Π
2. **Devise** a relation for solving one subproblem in terms of the others
3. **Solve** the subproblems, starting from the **smallest**, **store** their solutions, and move to the **larger** subproblems, **using the stored solutions of smaller ones**!!!
4. **Retrieve** the solution of the original problem Π, by solving all subproblems in a **determined order**!!!

Dynamic Programming Characteristics

Attention!!!

- ✓ In DP, the "technique model" can be considered as a computational graph G, which is a DAG!!!
- ✓ The nodes of G correspond to subproblems, and the edges represent dependencies between them:

$$\boxed{\boxed{A} \rightarrow \boxed{B}}$$

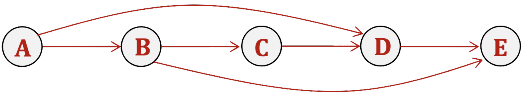
- A is considered a "smaller" subproblem than B, or
- To solve B, we need the solution for A!!!

Dynamic Programming Characteristics

Fundamental Property of DP!!!

There exists an ordering of the subproblems and a relation that shows how a subproblem can be solved using the solutions of "smaller" subproblems, i.e., subproblems that appear earlier in the order!

- **Order:**



- **Relation:**

$$\text{Large Subproblem} = f\,(\text{Smaller Subproblems})$$

Basic DP Outline

Algorithm Template

- Preprocessing of data
- Populate the matrix:
  - Iterate over the cells in the correct order.
  - Understand the work done per cell.

Basic DP Outline

Algorithm Template

- Preprocessing of data
- Populate the matrix:
  - Iterate over the cells in the correct order.
  - Understand the work done per cell.

Algorithm Guidelines

BASIC DP OUTLINE

Algorithm Template

- Preprocessing of data
- Populate the matrix:
  - Iterate over the cells in the correct order.
  - Understand the work done per cell.

Algorithm Guidelines

1. There are only a polynomial number of subproblems.

Basic DP Outline

Algorithm Template

- Preprocessing of data
- Populate the matrix:
  - Iterate over the cells in the correct order.
  - Understand the work done per cell.

Algorithm Guidelines

1. There are only a polynomial number of subproblems.
2. The solution to the larger problem can be efficiently calculated from the subproblems.

BASIC DP OUTLINE

Algorithm Template

- Preprocessing of data
- Populate the matrix:
  - Iterate over the cells in the correct order.
  - Understand the work done per cell.

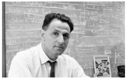
Algorithm Guidelines

1. There are only a polynomial number of subproblems.
2. The solution to the larger problem can be efficiently calculated from the subproblems.
3. Natural ordering of the subproblems from “smallest” to “largest”.

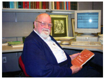
Appendix

References

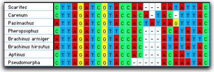
Image Sources I



https://medium.com/neurosapiens/2-dynamic-programming-9177012dcdd



https://angelberh7.wordpress.com/2014/10/08/biografia-de-lester-randolph-ford-jr/



http://www.sequence-alignment.com/

https://medium.com/koderunners/genetic-algorithm-part-3-knapsack-problem-b59035

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON
https://brand.wisc.edu/web/logos/

Image Sources II

https://www.pngfind.com/mpng/mTJmbx_
spongebob-squarepants-png-image-spongebob-cartoo

https://www.pngfind.com/mpng/xhJRmT_
cheshire-cat-vintage-drawing-alice-in-wonderland