Dynamic Programming: When your big problem is just too hard to solve

Daniel Epstein, 4/24/14
depstein@cs.washington.edu
https://github.com/depstein/programming-competitions

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).

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).

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).

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).

Don’t repeat work

Optimal Substructure

- By combining the optimal solutions to subproblems, you can solve the overall problem

Optimal Substructure

- By combining the optimal solutions to subproblems, you can solve the overall problem

4	1	8	6	5	2	3	7

Optimal Substructure

- By combining the optimal solutions to subproblems, you can solve the overall problem

Optimal Substructure

- By combining the optimal solutions to subproblems, you can solve the overall problem

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

Optimal Substructure

- By combining the optimal solutions to subproblems, you can solve the overall problem

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

Optimal Substructure

- By combining the optimal solutions to subproblems, you can solve the overall problem

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

Optimal Substructure

- By combining the optimal solutions to subproblems, you can solve the overall problem

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

Overlapping Subproblems

- To find the optimal solution, the subproblem must be referred to multiple times

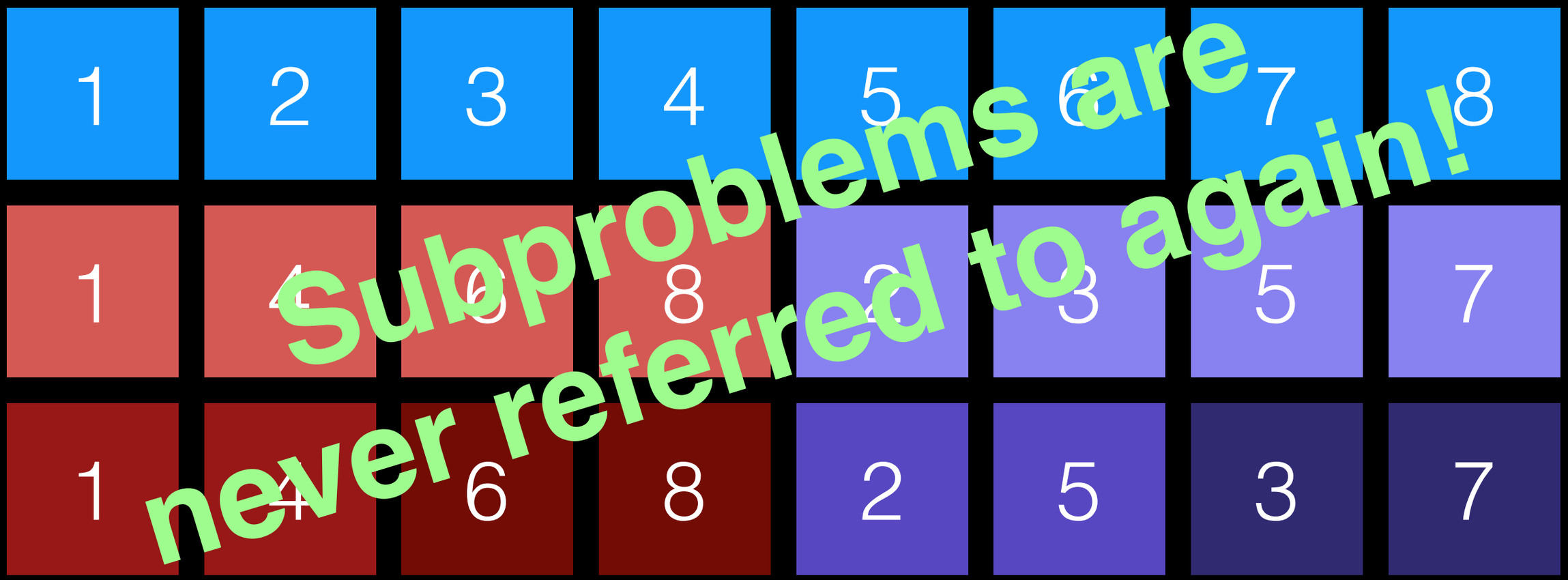
Overlapping Subproblems

- To find the optimal solution, the subproblem must be referred to multiple times

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

Overlapping Subproblems

- To find the optimal solution, the subproblem must be referred to multiple times



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

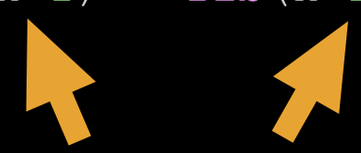
```java
import java.io.*;
import java.util.*;

public class fibonacci {
  public static void main(String[] args) {
    for(int i=0;i<50;i++) {
      System.out.printf("The %dth Fibonacci number is %d.\n", i, fib(i));
    }
  }

  public static long fib(int n) {
    if(n==0 || n==1) {
      return 1;
    }
    return fib(n-1) + fib(n-2);
  }
}
```

```
                    .
                    .
                    .
The 30th Fibonacci number is 1346269.
The 31th Fibonacci number is 2178309.
The 32th Fibonacci number is 3524578.
The 33th Fibonacci number is 5702887.
The 34th Fibonacci number is 9227465.
The 35th Fibonacci number is 14930352.
The 36th Fibonacci number is 24157817.
The 37th Fibonacci number is 39088169.
The 38th Fibonacci number is 63245986.
The 39th Fibonacci number is 102334155.
The 40th Fibonacci number is 165580141.
The 41th Fibonacci number is 267914296.
The 42th Fibonacci number is 433494437.
```

```
                    .
                    .
                    .
The 30th Fibonacci number is 1346269.
The 31th Fibonacci number is 2178309.
The 32th Fibonacci number is 3524578.
The 33th Fibonacci number is 5702887.
The 34th Fibonacci number is 9227465.
The 35th Fibonacci number is 14930352.
The 36th Fibonacci number is 24157817.
The 37th Fibonacci number is 39088169.
The 38th Fibonacci number is 63245986.
The 39th Fibonacci number is 102334155.
The 40th Fibonacci number is 165580141.
The 41th Fibonacci number is 267914296.
The 42th Fibonacci number is 433494437.
```

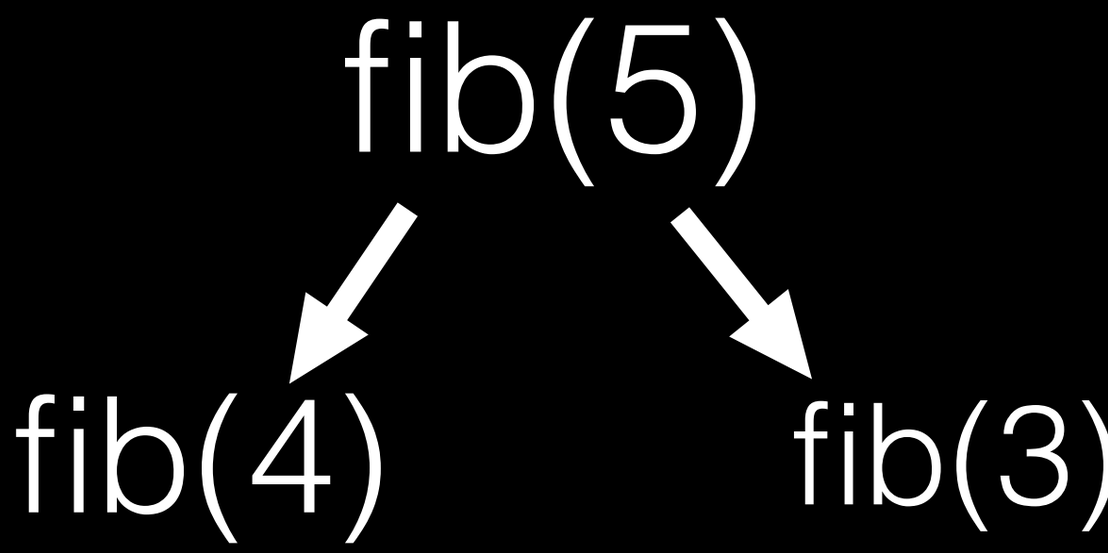
Slows… waaaaay… doooooooooooooooown

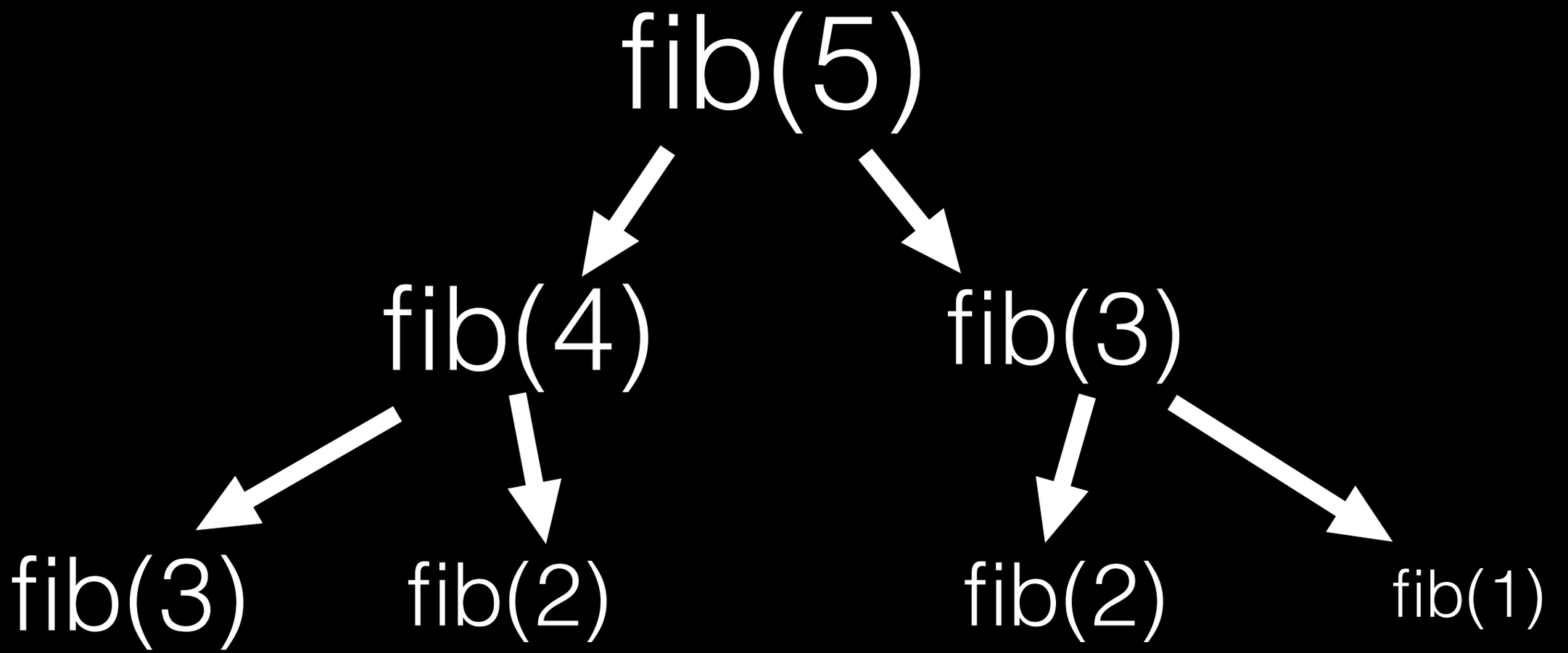
```java
import java.io.*;
import java.util.*;

public class fibonacci {
  public static void main(String[] args) {
    for(int i=0;i<50;i++) {
      System.out.printf("The %dth Fibonacci number is %d.\n", i, fib(i));
    }
  }

  public static long fib(int n) {
    if(n==0 || n==1) {
      return 1;
    }
    return fib(n-1) + fib(n-2);
  }
}
```

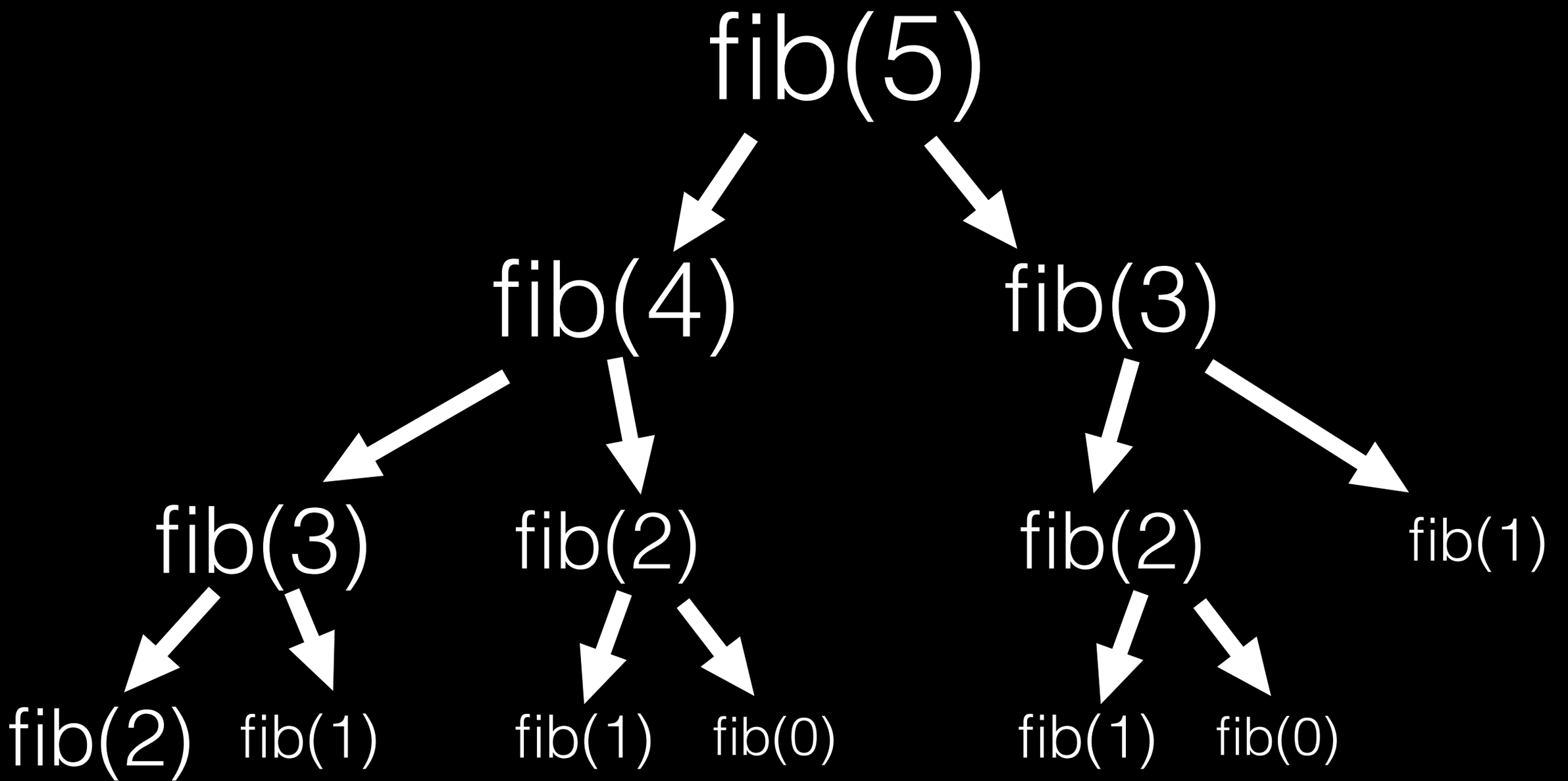


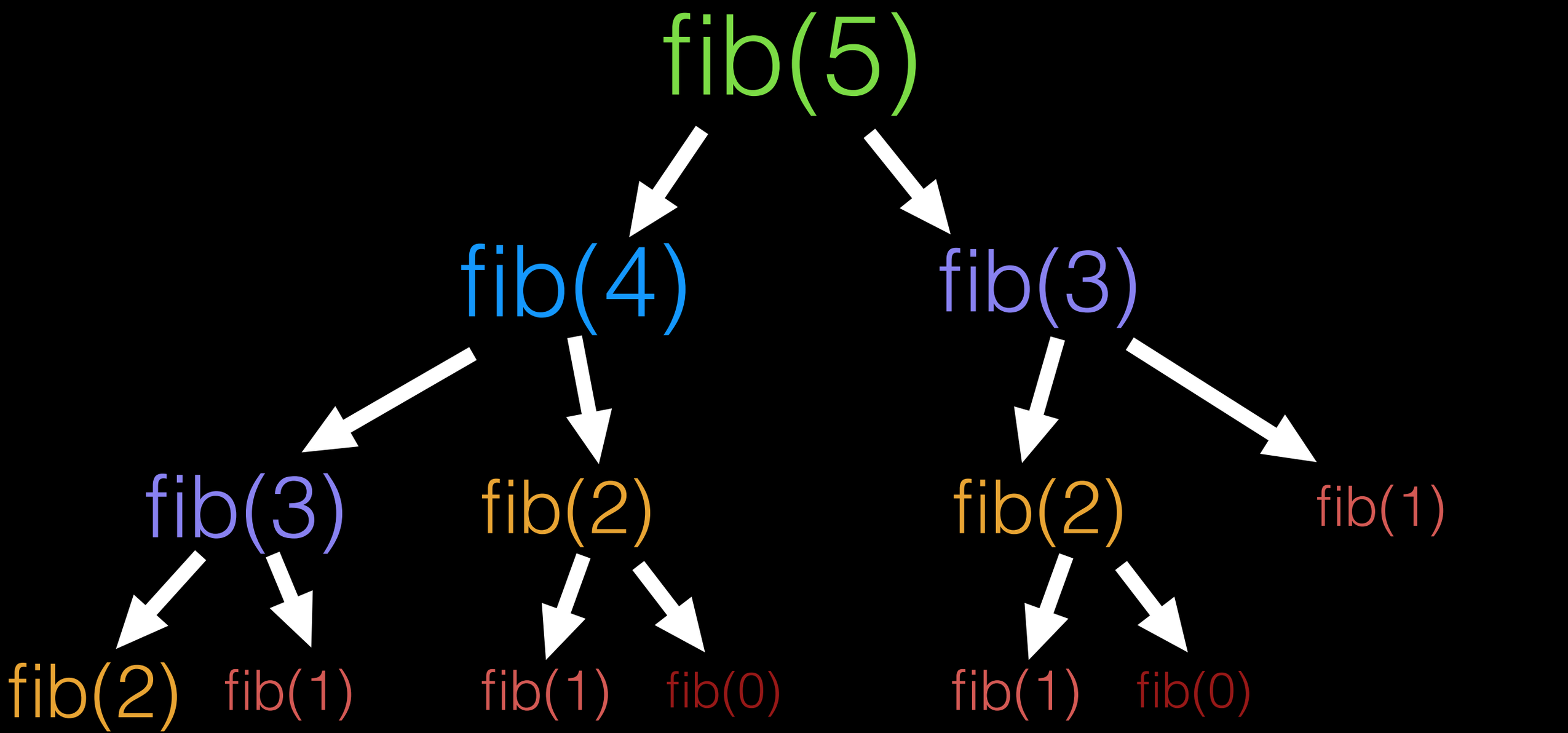
Optimal Substructure!

fib(5)


fib(5)
fib(4)
fib(3)


fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)


fib(5)
fib(4)
fib(3)
fib(3)
fib(2)
fib(2)
fib(1)
fib(2)
fib(1)
fib(1)
fib(0)
fib(1)
fib(0)



Overlapping subproblems!

Memoization

In computing, **memoization** is an optimization technique used primarily to speed up computer programs by keeping the results of expensive function calls and returning the cached result when the same inputs occur again.

In computing, **memoization** is an optimization technique used primarily to speed up computer programs by keeping the results of expensive function calls and returning the cached result when the same inputs occur again.

```java
import java.io.*;
import java.util.*;

public class fibonacci {
  public static long[] memoize = new long[50];

  public static void main(String[] args) {
    memoize[0] = 1;
    memoize[1] = 1;

    for(int i=0;i<50;i++) {
      System.out.printf("The %dth Fibonacci number is %d.\n", i, fib_dp(i));
    }
  }

  public static long fib_dp(int n) {
    if(memoize[n] != 0) { //0 is the default value for the array elements
      return memoize[n];
    }

    memoize[n] = fib_dp(n-1) + fib_dp(n-2);
    return memoize[n];
  }
}
```

How about a harder problem?

Longest Common Subsequence

TATGT

CATAG

Longest Common Subsequence

TATGT

CATAG

```java
import java.io.*;
import java.util.*;

public class lcs {
  public static void main(String[] args) {
    String s1 = "TATGT";
    String s2 = "CATAG";
    System.out.printf("The longest common subsequence is of length %d.\n", lcs(s1, s2, 0,
0));
  }

  public static int lcs(String s1, String s2, int s1index, int s2index) {
    if(s1index == s1.length() || s2index == s2.length()) {
      return 0;
    }
    if(s1.charAt(s1index) == s2.charAt(s2index)) {
      return 1 + lcs(s1, s2, s1index+1, s2index+1);
    }
    return Math.max(lcs(s1, s2, s1index+1, s2index), lcs(s1, s2, s1index, s2index+1));
  }
}
```

```java
import java.io.*;
import java.util.*;

public class lcs {
  public static void main(String[] args) {
    String s1 = "TATGT";
    String s2 = "CATAG";
    System.out.printf("The longest common subsequence is of length %d.\n", lcs(s1, s2, 0,
0));
  }

  public static int lcs(String s1, String s2, int s1index, int s2index) {
    if(s1index == s1.length() || s2index == s2.length()) {
      return 0;
    }
    if(s1.charAt(s1index) == s2.charAt(s2index)) {
      return 1 + lcs(s1, s2, s1index+1, s2index+1);
    }
    return Math.max(lcs(s1, s2, s1index+1, s2index), lcs(s1, s2, s1index, s2index+1));
  }
}
```

Optimal Substructure!

TATGT

CATAG

TATGT
CATAG

TATGT
CATAG

TATGT
CATAG

TATGT
CATAG

TATGT
CATAG

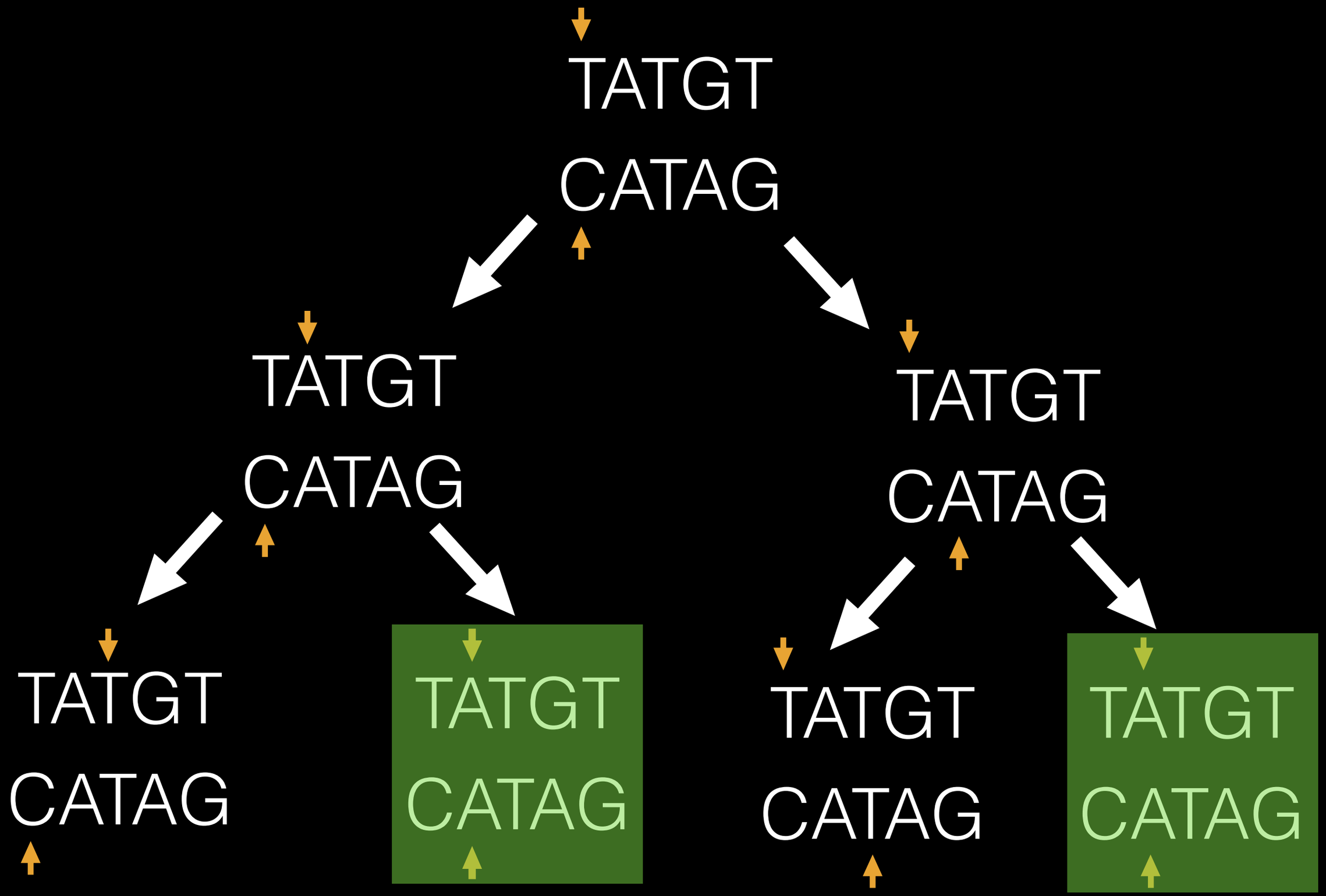
TATGT
CATAG

TATGT
CATAG

TATGT
CATAG

TATGT
CATAG

TATGT
CATAG



Overlapping subproblems!

	/	C	A	T	A	G
/						
T						
A						
T						
G						
T						

	/	C	A	T	A	G
/	0	0	0	0	0	0
T	0					
A	0					
T	0					
G	0					
T	0					

	/	C	A	T	A	G
/	0	0	0	0	0	0
T	0	0				
A	0					
T	0					
G	0					
T	0					

	/	C	A	T	A	G
/	0	0	0	0	0	0
T	0	0	0			
A	0					
T	0					
G	0					
T	0					

	/	C	A	T	A	G
/	0	0	0	0	0	0
T	0	0	0	1		
A	0					
T	0					
G	0					
T	0					

	/	C	A	T	A	G
/	0	0	0	0	0	0
T	0	0	0	1	1	1
A	0					
T	0					
G	0					
T	0					

	/	C	A	T	A	G
/	0	0	0	0	0	0
T	0	0	0	1	1	1
A	0	0	1	1	2	2
T	0					
G	0					
T	0					

	/	C	A	T	A	G
/	0	0	0	0	0	0
T	0	0	0	1	1	1
A	0	0	1	1	2	2
T	0	0	1	2	2	2
G	0					
T	0					

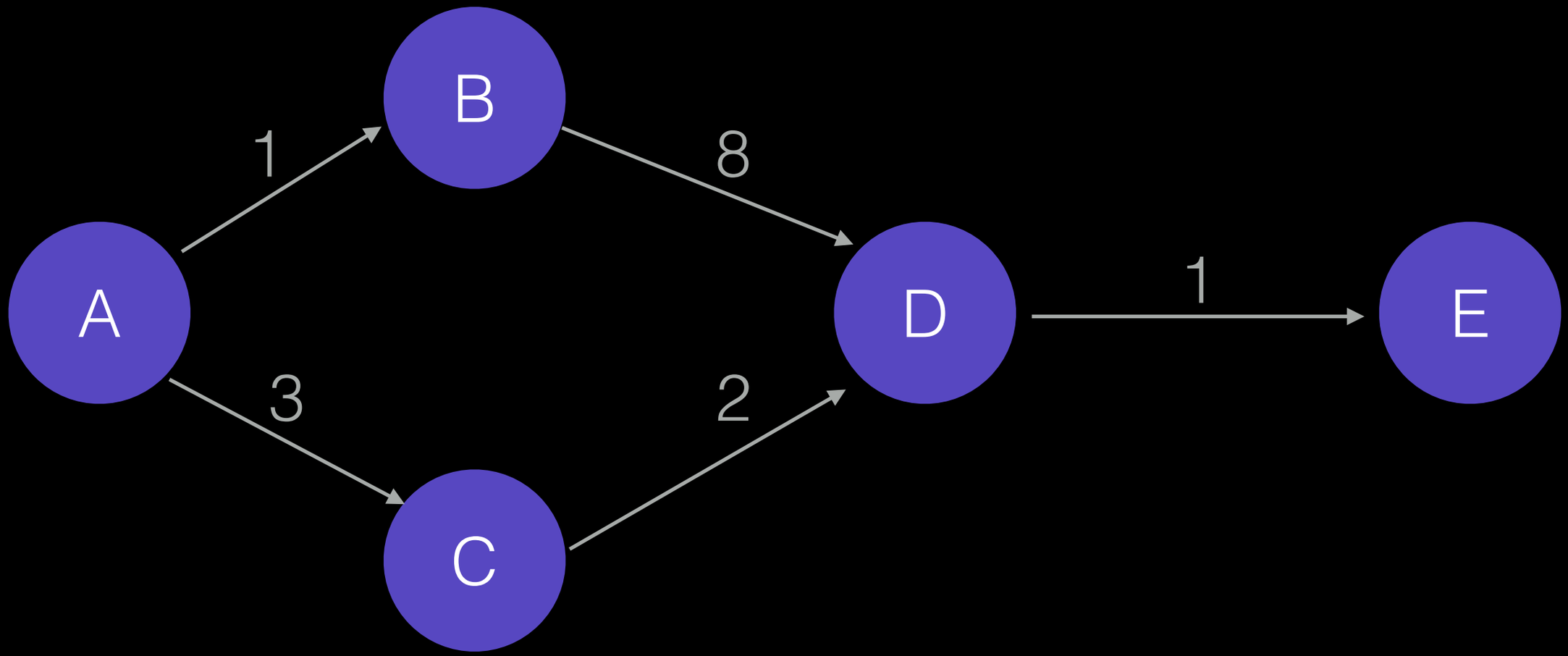
	/	C	A	T	A	G
/	0	0	0	0	0	0
T	0	0	0	1	1	1
A	0	0	1	1	2	2
T	0	0	1	2	2	2
G	0	0	1	2	2	3
T	0					

	/	C	A	T	A	G
/	0	0	0	0	0	0
T	0	0	0	1	1	1
A	0	0	1	1	2	2
T	0	0	1	2	2	2
G	0	0	1	2	2	3
T	0	0	1	2	2	3

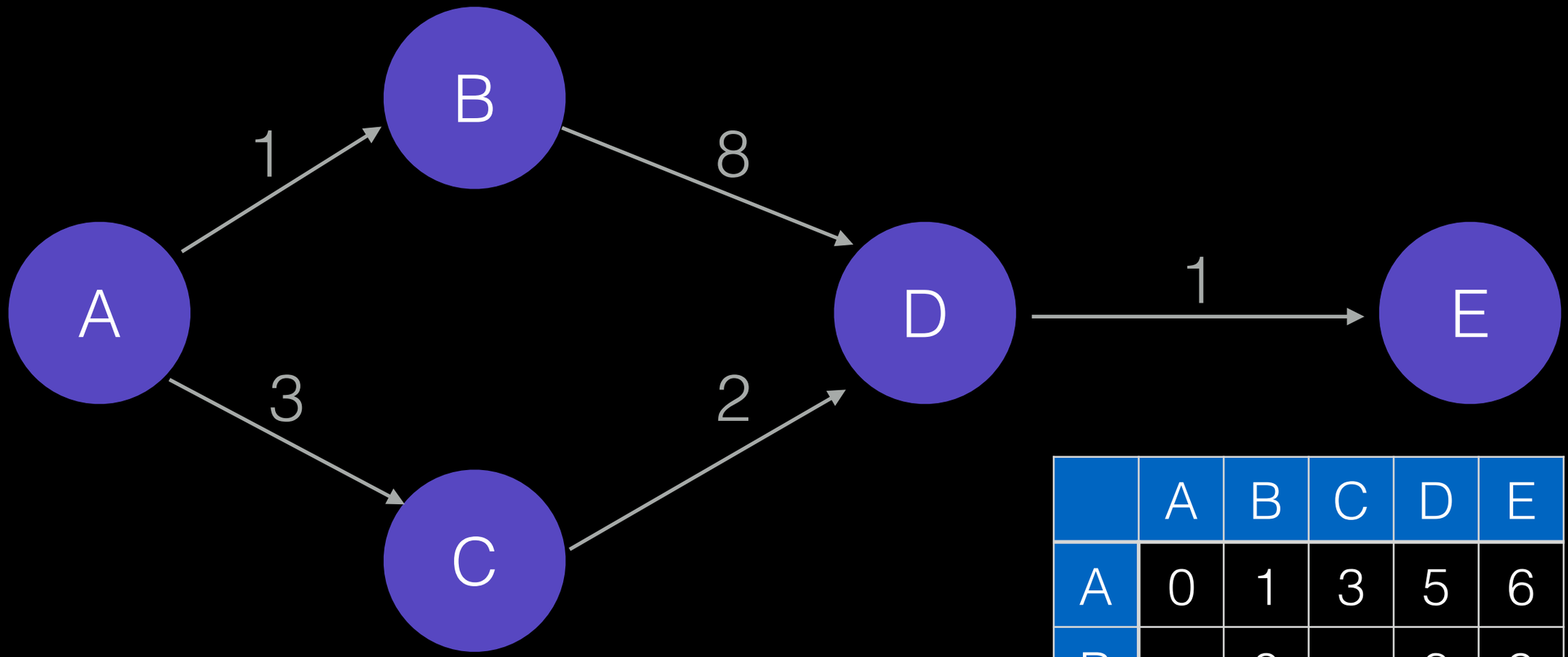
```java
import java.io.*;
import java.util.*;

public class lcs {
  public static int[][] memoize = new int[6][6];

  public static void main(String[] args) {
    for(int i=1;i<6;i++) {
      for(int j=1;j<6;j++) {
        memoize[i][j] = -1;
      }
    }

    String s1 = "TATGT";
    String s2 = "CATAG";
    System.out.printf("The longest common subsequence is of length %d.\n", lcs_dp(s1, s2, 5, 5));
  }

  public static int lcs_dp(String s1, String s2, int s1index, int s2index) {
    if(memoize[s1index][s2index] == -1) {
      if(s1.charAt(s1index-1) == s2.charAt(s2index-1)) { // the first row & column are all 0's, so charAt
is off-by-1
        memoize[s1index][s2index] = 1 + lcs_dp(s1, s2, s1index-1, s2index-1);
      } else {
        memoize[s1index][s2index] = Math.max(lcs_dp(s1, s2, s1index-1, s2index), lcs_dp(s1, s2, s1index,
s2index-1));
      }
    }
    return memoize[s1index][s2index];
  }
}
```

How about a harder problem?

All-Pairs Shortest Path



All-Pairs Shortest Path



	A	B	C	D	E
A	0	1	3	5	6
B	∞	0	∞	8	9
C	∞	∞	0	2	3
D	∞	∞	∞	0	1
E	∞	∞	∞	∞	0

All-Pairs Shortest Path

- Consider only the first *k* vertices
- What's the shortest path from *i* to *j* that can only go through the first *k* vertices?

All-Pairs Shortest Path

- Consider only the first *k* vertices
- What's the shortest path from *i* to *j* that can only go through the first *k* vertices?

```
shortestPath(i,j,k+1) = min(shortestPath(i,j,k),
shortestPath(i,k+1,k)+shortestPath(k+1,j,k)
```

```java
import java.io.*;
import java.util.*;

public class allpairs {
  public static int[][] dist = new int[5][5];

  public static void main(String[] args) {
    for(int i=0;i<5;i++)
      for(int j=0;j<5;j++) {
        if(i != j)
          dist[i][j] = 1000; // Not using Integer.MAX_VALUE to avoid integer overflowing
      }
    //Initialize graph as described
    dist[0][1] = 1;
    dist[0][2] = 3;
    dist[1][3] = 8;
    dist[2][3] = 2;
    dist[3][4] = 1;

    for(int k=0;k<5;k++)
      for(int i=0;i<5;i++)
        for(int j=0;j<5;j++) {
          dist[i][j] = Math.min(dist[i][j], dist[i][k] + dist[k][j]);
        }

    for(int i=0;i<5;i++)
      for(int j=0;j<5;j++)
        System.out.printf("The shortest path from %c to %c is %s.\n", (char)('A' + i), (char)
('A' + j), (dist[i][j] == 1000) ? "Infinity": (""+dist[i][j]));
  }
}
```

Dynamic Programming: When your big problem is just too hard to solve

Daniel Epstein, 4/24/14
depstein@cs.washington.edu
https://github.com/depstein/programming-competitions