CS/COE 1501

www.cs.pitt.edu/~lipschultz/cs1501/

Greedy Algorithms and Dynamic Programming

Consider the change making problem

- What is the minimum number of coins needed to make up a given value k?
- If you were working as a cashier, what would your algorithm be to solve this problem?

This is a *greedy algorithm*

- At each step, the algorithm makes the choice that seems to be best at the moment
- Have we seen greedy algorithms already this term?

… But wait …

- Nearest neighbor doesn't solve travelling salesman
  - Does not produce an optimal result
- Does our change making algorithm solve the change making problem?
  - For US currency…
  - But what about a currency composed of pennies (1 cent), thrickels (3 cents), and fourters (4 cents)?
    - What denominations would it pick for k=6?

So what changed about the problem?

- For greedy algorithms to produce optimal results, problems must have two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - The greedy choice property
    - Globally optimal solutions can be assembled from locally optimal choices
- Why is optimal substructure not enough?

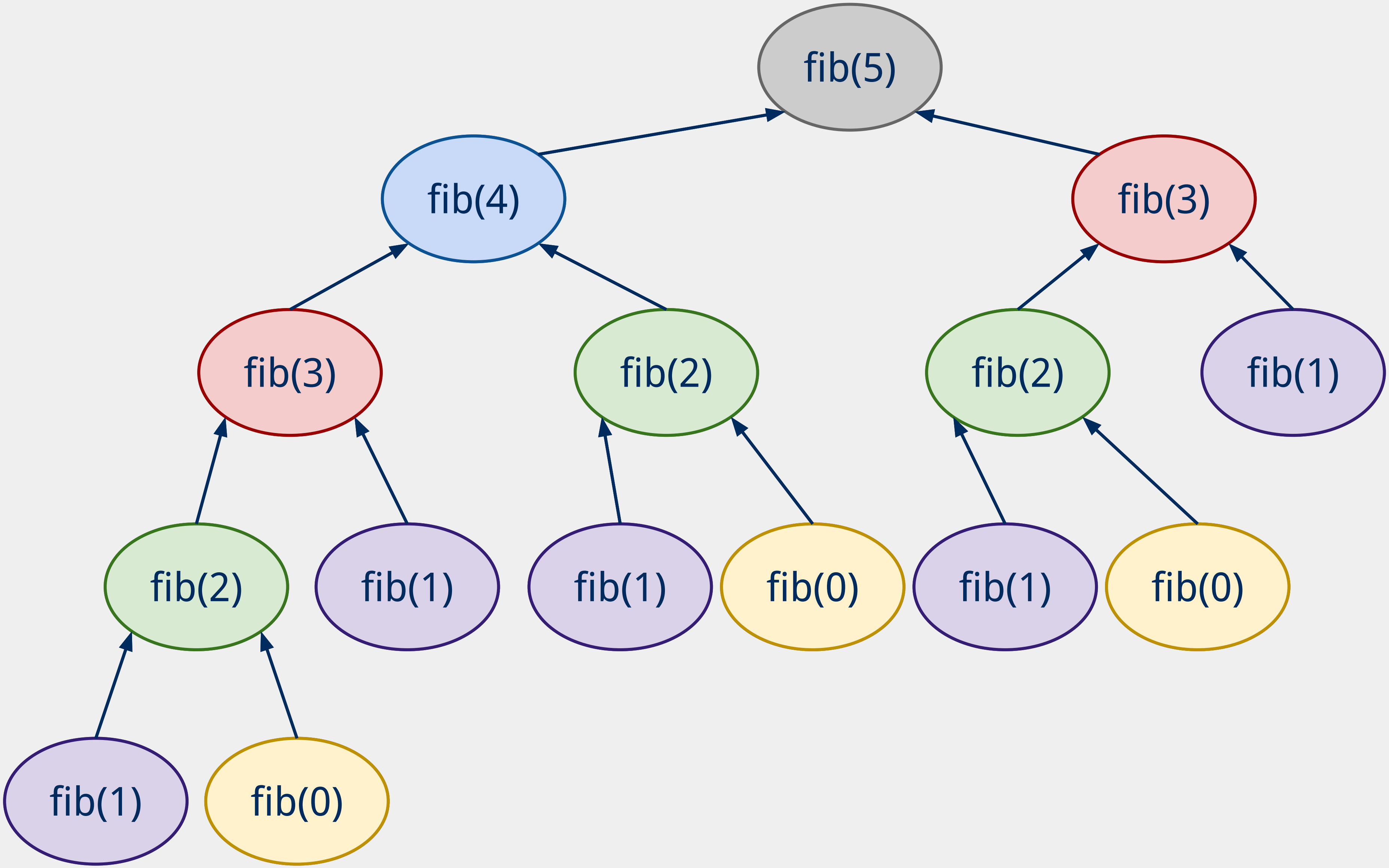
Finding all subproblems solutions can be inefficient

- Consider computing the Fibonacci sequence:

```
int fib(n) {
    if (n == 0) return 0;
    else if (n == 1) return 1;
    else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

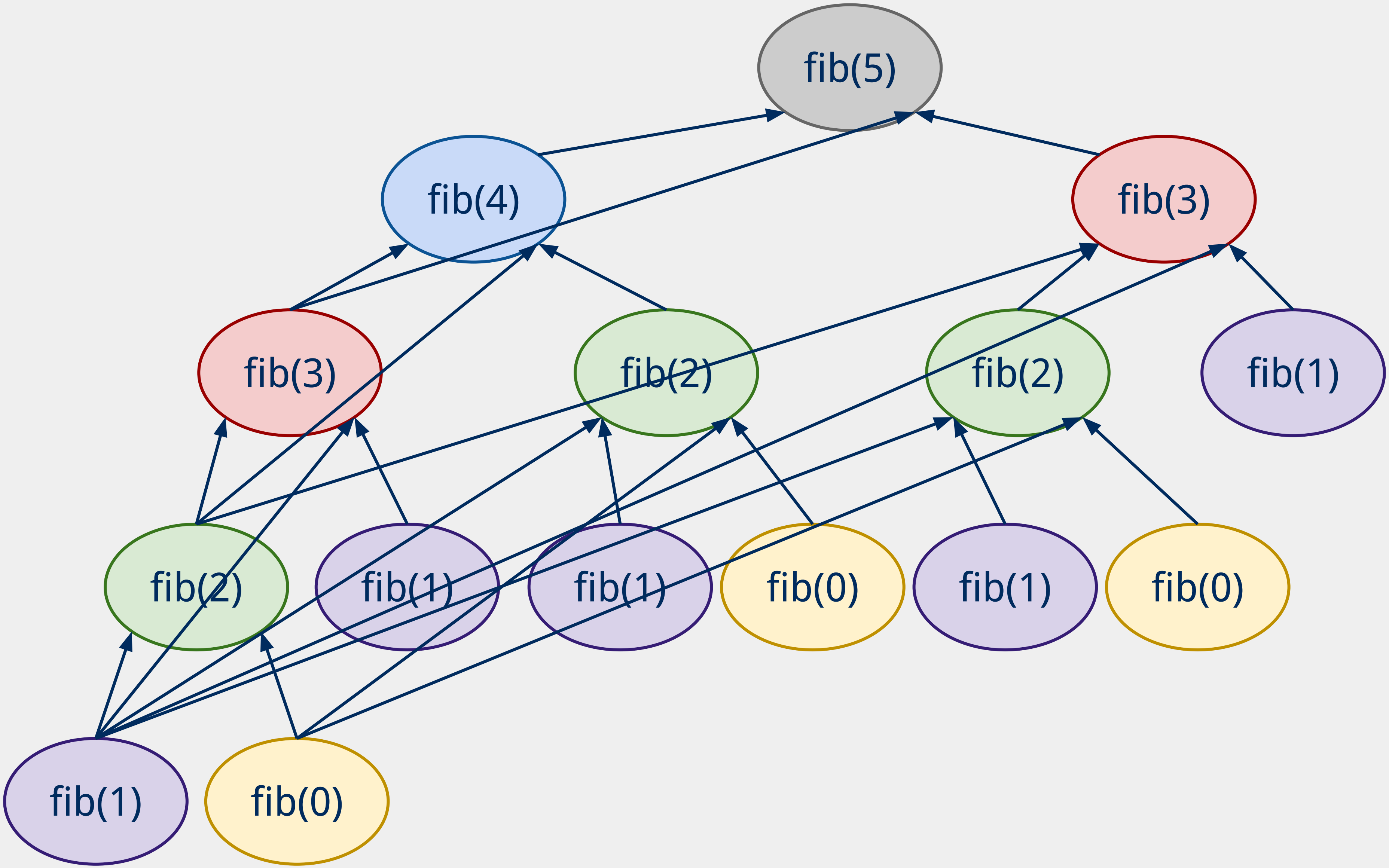
- What does the call tree for n = 5 look like?

fib(5)



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)
fib(1)
fib(0)

How do we improve?



Memoization

```
int[] F = new int[n+1];
F[0] = 0;
F[1] = 1;
for(int i = 2; i <= n; i++) F[i] = -1;

int dp_fib(x) {
    if (F[x] == -1)
        F[x] = dp_fib(x-1) + dp_fib(x-2);
    return F[x];
}
```

Note that we can also do this bottom-up

```
int bottomup_fib(n) {
    if (n == 0)
        return 0;
    int[] F = new int[n+1];
    F[0] = 0;
    F[1] = 1;
    for(int i = 2; i <= n; i++) {
        F[i] = F[i-1] + F[i-2];
    }
    return F[n];
}
```

Can we improve this bottom-up approach?

Where can we apply dynamic programming?

- Problems with two properties:
  - Optimal substructure
    - Optimal solution to a subproblem leads to an optimal solution to the overall problem
  - Overlapping subproblems
    - Naively, we would need to recompute the same subproblem multiple times
- How do these properties contrast with those for greedy algorithms?

The unbounded knapsack problem

- Given a knapsack that can hold a weight limit L, and a set of n types of items that each has a weight (w_i) and value (v_i), what is the maximum value we can fit in the knapsack if we assume we have unbounded copies of each item?

```
K[0] = 0
for (l = 1; l <= L; l++)
    K[l] = max (v_i + K[l - w_i])
         w_i<=l
```

The 0/1 knapsack problem

- What if we have a finite set of items, with each item having a weight and value?
  - Two choices for each item:
    - Goes in the knapsack
    - Left out of the knapsack

The 0/1 knapsack problem

```
int knapSack(int[] wt, int[] val, int L, int n) {
    if (n == 0 || L == 0):
        return 0;
    if (wt[n-1] > L):
        return knapSack(wt, val, L, n-1);
    else:
        return max( val[n-1] + knapSack(wt, val, L-wt[n-1], n-1),
                    knapSack(wt, val, L, n-1)
                    );
}
```

The 0/1 knapsack dynamic programming solution

```
int knapSack(int wt[], int val[], int L, int n) {
    int[][] K = new int[n+1][L+1];
    for (int i = 0; i <= n; i++) {
        for (int l = 0; l <= L; l++) {
            if (i==0 || l==0) K[i][l] = 0;
            else if (wt[i-1] > l) K[i][l] = K[i-1][l];
            else
                K[i][l] = max(val[i-1] + K[i-1][l-wt[i-1]],
                              K[i-1][l]);
        }
    }
    return K[n][L];
}
```

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2						
3						
4						

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3						
4						

The 0/1 knapsack dynamic programming example

```
wt  = [ 2, 3, 4, 5 ]
val = [ 3, 4, 5, 6 ]
```

i\l	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4						

The 0/1 knapsack dynamic programming solution

```
int knapSack(int wt[], int val[], int L, int n) {
    int[][] K = new int[n+1][L+1];
    for (int i = 0; i <= n; i++) {
        for (int l = 0; l <= L; l++) {
            if (i==0 || l==0) K[i][l] = 0;
            else if (wt[i-1] > l) K[i][l] = K[i-1][l];
            else
                K[i][l] = max(val[i-1] + K[i-1][l-wt[i-1]],
                              K[i-1][l]);
        }
    }
    return K[n][L];
}
```

How can we also return the items stored in the knapsack?

To review...

- Questions to ask in finding dynamic programming solutions:
    - Does the problem have optimal substructure?
        - Can solve the problem by splitting it into smaller problems?
        - Can you identify subproblems that build up to a solution?
    - Does the problem have overlapping subproblems?
        - Where would you find yourself recomputing values?
            - How can you save and reuse these values?

The change-making problem

- Consider a currency with n different denominations of coins d_1, d_2, …, d_n. What is the minimum number of coins needed to make up a given value k?