Lecture #3: Dynamic Programming and Memoization

COSC 3020: Algorithms and Data Structures

Lars Kotthoff[1]
`larsko@uwyo.edu`

[1]with material from various sources

Outline

- Motivating Example: Fibonacci
- Dynamic Programming
- Memoization
- Tail Recursion
- Dynamic Programming Examples

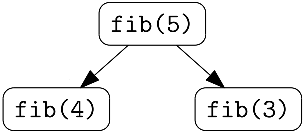
Learning Goals

- Be able to solve recursive problems more efficiently.
- Know the difference between memoization and dynamic programming.
- Know when to apply these techniques and when to let the compiler take care of it.

Fibonacci

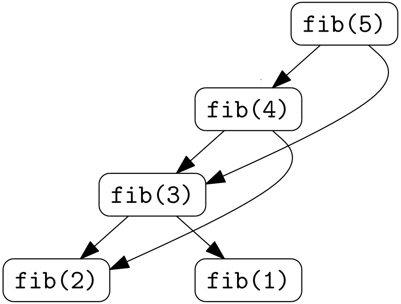
```
function fib(n) {
    if (n <= 2) return 1;
    else        return fib(n-1) + fib(n-2);
}
```

Finish the recursion tree for `fib(5)`...



Fixing Fib with Iteration

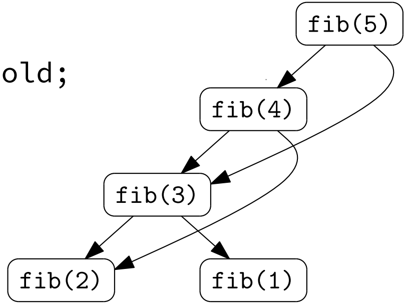
What we really want is to “share” nodes in the recursion tree:



Fixing Fib with Iteration and Dynamic Programming

Here's one fix that "walks down" the left of the tree:

```
function fib_dp(n) {
  var fib_old = 1;
  var fib = 1;
  var fib_new;
  while(n > 2) {
    fib_new = fib + fib_old;
    fib_old = fib;
    fib = fib_new;
    --n;
  }
  return fib;
}
```



Fixing Fib with Recursion and Memoization

Here's another fix that stores solutions it has calculated before:

```
var fib_solns = [];

fib_solns[1] = 1;
fib_solns[2] = 1;

function fib_memo(n) {
  // If we don't know the answer, compute it.
  if(fib_solns[n] === undefined)
    fib_solns[n] = fib_memo(n-1) +
                   fib_memo(n-2);
  return fib_solns[n];
}
```

Dynamic Programming vs. Memoization

Both ways of avoiding repeatedly computing the same thing and can make exponential computations linear.

- ▷ Memoization can be directly applied to recursion by adding memory. Can be considered subset of dynamic programming.
- ▷ Dynamic programming is not necessarily recursion.

Recursion/Divide and conquer $+$ memoization $=$ top-down dynamic programming

starting with the "smallest" thing and building up $=$ bottom-up dynamic programming

Recursion vs. Iteration

Which one is more efficient? Recursion or iteration?

Recursion vs. Iteration

Which one is more efficient? Recursion or iteration?

It's probably easier to shoot yourself in the foot without noticing when you use recursion, and the call stack may carry around more memory than you really need to store, but otherwise...

Neither is more efficient.

Managing the Call Stack: Tail Recursion

```
function endlesslyGreet() {
  console.log("Hello world!");
  endlesslyGreet();
}
```

This is clearly infinite recursion. The call stack will get as deep as it can get and then bomb, right?

Managing the Call Stack: Tail Recursion

```
function endlesslyGreet() {
  console.log("Hello world!");
  endlesslyGreet();
}
```

This is clearly infinite recursion. The call stack will get as deep as it can get and then bomb, right?

But…why have a call stack? There's no (need to) return to the caller.

Not all JavaScript runtimes implement this, but run it on Safari or a webkit-based browser and it won't give a stack overflow!

Tail Recursion

A function is “tail recursive” if for any recursive call in the function, that call is the last thing the function needs to do before returning.

In that case, we don’t need to have a recursive call. Replace the current call with the recursive one.

That’s what most compilers will do (and some JavaScript runtimes).

Tail Recursive?

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

Tail Recursive?

```
function fact(n) {
   if (n == 0) return 1;
   else        return n * fact(n - 1);
}
```

Tail Recursive?

```
function fact(n) { return fact_acc(n, 1); }

function fact_acc(b, acc) {
  if (b == 0) return acc;
  else        return fact_acc(b - 1, acc * b);
}
```

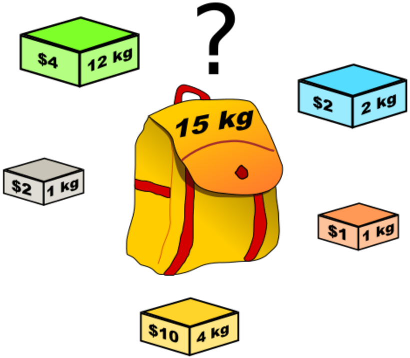
Examples

All Pairs Shortest Path – Floyd-Warshall Algorithm

- initialize a $|V| \times |V|$ matrix `dist` to ∞
- for each vertex $v \in V$, `dist[v][v] = 0`
- for each edge $(u, v) = e \in E$, `dist[u][v] = weight((u,v))`
- for each vertex $k \in V$:
  - for each vertex $i \in V$:
    - for each vertex $j \in V$:
      - ```
        if dist[i][j] > dist[i][k] + dist[k][j]:
            dist[i][j] = dist[i][k] + dist[k][j]
        ```
        ```

Knapsack Problem

We are given a set of n items, where each item i is specified by a size s_i and a value v_i. The size of the knapsack is S. The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack).



Knapsack Problem: Recursive Solution

```
function ks(items, cap) {
  if(items.length == 0) return 0;

  var result = 0;
  for(var i = 0; i < items.length; i++) {
    var it = items[i],
        newItems = items.filter(i => i !== it);
    result = Math.max(result, ks(newItems, cap));
    if(it.size <= cap)
      result = Math.max(result, it.value +
          ks(newItems, cap - it.size));
  }
  return result;
}
```

Knapsack Problem: Recursive Solution + Memoization

```
var cache = [];
function ks_memo(items, cap) {
  if(items.length == 0) return 0;
  var key = JSON.stringify(items);
  if(cache[key] === undefined) cache[key] = [];
  if(cache[key][cap] !== undefined)
    return cache[key][cap];

  var result = 0;
  for(var i = 0; i < items.length; i++) {
    var it = items[i],
        newItems = items.filter(i => i !== it);
    result = Math.max(result, ks_memo(newItems, cap));
    if(it.size <= cap)
      result = Math.max(result, it.value +
          ks_memo(newItems, cap - it.size));
  }
  cache[key][cap] = result;
  return result;
}
```

Knapsack Problem: Bottom-up Dynamic Programming

```
function ks_dp(items, cap) {
  var cache = [];

  for(var i = 0; i <= items.length; i++) {
    cache[i] = [];
    for(var s = 0; s <= cap; s++) {
        if(i == 0 || s == 0) cache[i][s] = 0;
        else if(items[i-1].size <= s)
          cache[i][s] = Math.max(cache[i-1][s],
              items[i-1].value +
              cache[i-1][s-items[i-1].size]);
        else
              cache[i][s] = cache[i-1][s];
    }
  }
  return cache[items.length][cap];
}
```

Knapsack Problem: Greedy Solution (NOT Dynamic Programming)

```
function ks_greedy(items, cap) {
  items.sort((a, b) => b.value - a.value);
  var result = 0;
  items.forEach(function(it) {
      if(it.size <= cap) {
          result += it.value;
          cap -= it.size;
      }
  });
  return result;
}
```

Do try this at home

Floyd-Warshall
`https://www.cs.usfca.edu/~galles/visualization/Floyd.html`

Knapsack `https://algorithm-visualizer.org/dynamic-programming/knapsack-problem`