WEEK 08

1. Preparation for Assignment

If, and *only if* you can truthfully assert the truthfulness of each statement below are you ready to start the exercises.

1.1. Reading Comprehension Self-Check.

- I know why it is **false** to say that **dynamic programming** is better than static programming (such as is taught in CS124 (or equivalent)).
- I know why it is **false** to say that applicability of dynamic programming to an optimization problem requires the problem to satisfy the *principle of suboptimality*: a suboptimal solution to any of its instances must be made up of suboptimal solutions to its subinstances.
- I know how to discover that one of the earliest applications of dynamic programming is an algorithm that solves the traveling salesman problem in time $\mathcal{O}(n^2 2^n)$.
- I know why it is **false** to say that solving a **continuous** knapsack problem by a dynamic programming algorithm exemplifies an application of this technique to difficult problems of combinatorial optimization.
- I know how *memory functions* such as this Fibonacci C++ code constitute a space-grabbing but time-saving technique.

```cpp
#include <ctime>
#include <iostream>
#include <iomanip>
#include <string>
#include <sys/time.h>
using namespace std;

/*
 * The famous fibonacci function, recursive version.
 */
long fibonacci(long n)
{
    if (n == 0 || n == 1)
    {
        return n;
    }
    else
```

Date: November 2, 2018.

```
    {
        return (fibonacci(n - 1) + fibonacci(n - 2));
    }
}

void testFibonacci()
{
    clock_t start = clock();
    long fib42 = fibonacci(42);
    clock_t finish = clock();
    cout << "\nTo compute fibonacci(42) = " << fib42
         << "\nrecursively took\n\n"
         << setprecision(5) << setw(4) << (finish - start) / (double) CLOCKS_PER_SEC
         << " seconds.\n\n";
    cout << "----------------------------------------\n" << endl;
}

long fibmem[100] = {0};

/*
 * A "memory-function" implementation of the fibonacci function.
 */
long memFuncFibonacci(long n)
{
    if (fibmem[n] == 0)
    {
        if (n == 0 || n == 1)
        {
            fibmem[n] = n;
        }
        else
        {
            fibmem[n] = (memFuncFibonacci(n - 1) + memFuncFibonacci(n - 2));
        }
    }
    return fibmem[n];
}

void testMemFuncFibonacci()
{
    long double usec1;
    long double usec2;
```

```
  long double elapsed;
  timeval start;
  timeval finish;
  long fib42;

  gettimeofday(&start, NULL);
  fib42 = memFuncFibonacci(42);
  gettimeofday(&finish, NULL);
  usec1 = (long double) (start.tv_sec) +
     (long double) (start.tv_usec / 1000000.0f);
  usec2 = (long double) (finish.tv_sec) +
     (long double) (finish.tv_usec / 1000000.0f);
  elapsed = usec2 - usec1;

  cout.setf(ios::fixed);
  cout << "To compute fibonacci(42) = " << fib42
       << "\nwith a memory function took\n\n"
       << setprecision(11) << setw(13) << elapsed
       << " seconds.\n\n";
  cout << "----------------------------------------\n" << endl;
}

/*
 * Run tests.
 */
int main()
{
  testFibonacci();
  testMemFuncFibonacci();
}
```

- I know how *memory functions* such as this Elisp Knapsack code constitute a space-grabbing but time-saving technique.

```
#+BEGIN_SRC emacs-lisp
  (require 'cl)

  (setq Values [0 12 10 20 15] Weights [0 2 1 3 2] F (make-vector 5 nil))

  (loop for i from 0 to 4
        do (setf (aref F i) (make-vector 6 0)))

  (loop for i from 1 to 4
        do (loop for j from 1 to 5
           do (setf (aref (aref F i) j) -1)))
```

```
(defun MFKnapsack (i j)
  (let ((value 0))
    (if (< (aref (aref F i) j) 0)
        (progn
          (setq value
                (if (< j (aref Weights i))
                    (MFKnapsack (1- i) j)
                  (max (MFKnapsack (1- i) j)
                       (+ (aref Values i) (MFKnapsack (1- i) (- j (aref Weights i)))))))
          (setf (aref (aref F i) j) value)))
    (aref (aref F i) j)))
#+END_SRC

#+BEGIN_SRC emacs-lisp
  (MFKnapsack 4 5)
#+END_SRC
```

- I know how many *distinct* binary search trees can be constructed for a set of 4 orderable keys: A, B, C and D.
- I know how to draw all optimal BSTs for a set of 4 orderable keys.
- I know how to label each tree I drew with its unique level-order traversal (like ABCD).
- I know how to draw the optimal BST given 0.1, 0.2, 0.3, and 0.4 as the probabilities of the four keys A, B, C, and D.
- I know how to compute the average search cost for an optimal BST.

1.2. **Memory Self-Check.** Compare and contrast **dynamic programming** and **divide and conquer** by putting an X in a table cell if the property is true:

Property	Dynamic Programming	Divide and Conquer
Works bottom up		
Works top down		
Divides problems into subproblems		
Subproblems may be overlapping		

(*Bottom up* means starting at smallest or simplest subproblem, *then* combining subproblem solutions of increasing size until the solution of the original problem is reached.)

2. Week 08 Exercises

2.1. **Exercise 4 on page 290.**

2.2. **Exercise 1 on page 296.**

2.3. **Exercise 2 on page 303.**

2.4. **Exercise 5 on page 303.**

2.5. **Exercise 2 on page 311.**

3. Week 08 Problems

3.1. **Not in the Book.** In a language of your choice, implement the search elements of an optimal binary search tree for a set of n keys. Test the time efficiency of your code for the following cases:

(a) Search for the largest element.
(b) Search for the smallest element.
(c) Search for an element not present in the tree.