

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.

```
1 #include <ctime>
2 #include <iostream>
3 #include <iomanip>
4 #include <string>
5 #include <sys/time.h>
6 using namespace std;
7
8 /*
9  * The famous fibonacci function, recursive version.
10 */
11 long fibonacci(long n)
12 {
13     if (n == 0 || n == 1)
14     {
15         return n;
16     }
17     else
```

Date: November 6, 2018.

```

18     {
19         return (fibonacci(n - 1) + fibonacci(n - 2));
20     }
21 }
22
23 void testFibonacci()
24 {
25     clock_t start = clock();
26     long fib42 = fibonacci(42);
27     clock_t finish = clock();
28     cout << "\nTo compute fibonacci(42) = " << fib42
29         << "\nrecursively took\n\n"
30         << setprecision(5) << setw(4) << (finish - start) / (
31             double) CLOCKS_PER_SEC
32         << " seconds.\n\n";
33     cout << "_____ \n" << endl;
34 }
35 long fibmem[100] = {0};
36
37 /*
38  * A "memory-function" implementation of the fibonacci function
39  *
40 long memFuncFibonacci(long n)
41 {
42     if (fibmem[n] == 0)
43     {
44         if (n == 0 || n == 1)
45         {
46             fibmem[n] = n;
47         }
48         else
49         {
50             fibmem[n] = (memFuncFibonacci(n - 1) +
51                 memFuncFibonacci(n - 2));
52         }
53     }
54     return fibmem[n];
55 }
56 void testMemFuncFibonacci()
57 {
58     long double usec1;
59     long double usec2;

```

```

60  long double elapsed;
61  timeval start;
62  timeval finish;
63  long fib42;
64
65  gettimeofday(&start , NULL);
66  fib42 = memFuncFibonacci(42);
67  gettimeofday(&finish , NULL);
68  usec1 = (long double) (start.tv_sec) +
69          (long double) (start.tv_usec / 1000000.0f);
70  usec2 = (long double) (finish.tv_sec) +
71          (long double) (finish.tv_usec / 1000000.0f);
72  elapsed = usec2 - usec1;
73
74  cout.setf(ios::fixed);
75  cout << "To compute fibonacci(42) = " << fib42
76        << "\nwith a memory function took\n\n"
77        << setprecision(11) << setw(13) << elapsed
78        << " seconds.\n\n";
79  cout << "—————\n" << endl;
80 }
81
82 /*
83  * Run tests.
84  */
85 int main()
86 {
87     testFibonacci();
88     testMemFuncFibonacci();
89 }

```

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

```

1  #+BEGIN_SRC emacs-lisp
2  (require 'cl)
3
4  (setq Values [0 12 10 20 15] Weights [0 2 1 3 2] F (make-
      vector 5 nil))
5
6  (loop for i from 0 to 4
7        do (setf (aref F i) (make-vector 6 0)))
8
9  (loop for i from 1 to 4
10       do (loop for j from 1 to 5
11                 do (setf (aref (aref F i) j) -1)))

```

```

12
13 (defun MFKnapsack (i j)
14   (let ((value 0))
15     (if (< (aref (aref F i) j) 0)
16       (progn
17         (setq value
18           (if (< j (aref Weights i))
19             (MFKnapsack (1- i) j)
20             (max (MFKnapsack (1- i) j)
21               (+ (aref Values i) (MFKnapsack (1- i)
22                 (- j (aref Weights i)))))))
23         (setf (aref (aref F i) j) value)))
24     (aref (aref F i) j)))
25
26 #+BEGIN_SRC emacs-lisp
27 (MFKnapsack 4 5)
28 #+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.