1

Dynamic Programming

Updated Feb 2002

2

Outline

Dynamic Programming is a technique that can be used to reduce the amount of time needed to solve some problems.

Classic time-space tradeoff - we store intermediate results.

It is not always applicable, but very helpful when it is.

- Idea - store solutions to subproblems
- Sample problems
- Top down and Bottom up
- When does Dynamic Programming fail to work?
- More problems where Dynamic Programming helps

3

Motivation

Computing the Fibonacci numbers recursively provides a good example of a bad algorithm   1, 1, 2, 3, 5, 8, 13, 21, ...

f(0) = 1

f(1) = 1

f(n) = f(n-1) + f(n-2)   for n > 1.

```
int fib(int n) {
   if (n < 0)     // Check input parameter
      return 0;
   if (n < 2)     // Deal with base case
      return 1;
   return f(n-1) + f(n-2);    // Recursion
}
```

4

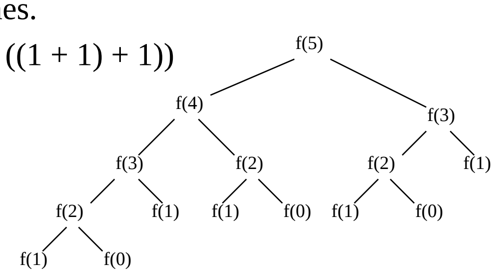
Why is this so bad?

To compute f(5), we compute f(4) once, f(3) twice, f(2) three times, and f(1) five times.

We call f a number of times: each call either returns 1 or adds two numbers. Thus we need to return 1 f(n) times, and to add them up f(n) - 1 times.

((((1 + 1) + 1) + (1 + 1)) + ((1 + 1) + 1))

Very wasteful



5

Dynamic Programming

We create a structure to hold previous computations (array **knownValues**)
When we call the routine, check to see if we have already found a solution.
  If so, we use it.
  If not, we compute the value as before and store it in our array

```
static int fib2(int n, int known[]) {
    …
    if (UNKNOWN != known[n])  // If we already know it
        return known[n];      //return pre-computed value
    …
    known[n] = result;                // Save our work for future
    ...
}
```

6

Dynamic Programming

```
static int fib2(int n, int known[]) {
    if (n < 0)                // Check input parameter
        return 0;
    if (UNKNOWN != known[n])  // If we already know it
        return known[n];      //return pre-computed value

    int result = 1;           // Common result
    if (n > 1)                // Make recursive call
        result = fib2(n-1, known) + fib2(n-2, known);

    known[n] = result;        // Save our work for future
    return result;
}
```

7

Initializing array knownValues

```
public class Fibonacci  {
    static final int UNKNOWN = -1;
    public static void main(String[] args)  {
        if (args.length < 1) {                    // Check usage.
            System.err.println("Usage: Fibonacci val"); return;
        }
        Integer N = new Integer(args[0]);
        int   val = N.intValue();
        // Create array to hold results and initialize it
        int knownValues[] = new int[val+1];
        for (int i = 0; i <= val; i++)
            knownValues[i] = UNKNOWN;
```

8

Dynamic Programming

There are two ways to fill in the results: bottom up, or top down.
In top down, we make an initial call, and only compute the values we need.
In bottom up, we start with small values and work up

Our previous function works in either direction.

Top Down: Start computation by calling fib2(100): still works

Bottom up: Precompute with a loop

```
for (int j = 0; j < val; j++)
    fi2(j, knownValues)
```

Here is a third version of the Fibonacci function that works from bottom up each time. It uses the insight we have gained to reduce the problem to bare essentials. This is quite fast, and requires no additional storage

9

Iterative Version

```
static int fib3(int n) {
    if (n < 0) return 0;
    if (n < 2) return 1;

    int first  = 1, second = 1, third  = 0;

    while (n-- > 1) {
        third  = first + second;
        first  = second;
        second = third;
    }
    return third;
}
```

10

Puzzle

Problem: given this array, pick a path that goes from top to bottom, and maximizes the values hit

Path must descend with every step: cannot meander around.

2	5	1	6	7	3	2
3	2	6	8	2	9	3
1	7	6	8	5	3	8
8	6	8	3	4	2	1
2	6	3	8	2	3	4
6	7	5	6	8	4	2
6	3	4	6	8	3	6

Puzzle path

Here is a sample path.

The value is

7 + 8 + 8 + 4 + 3 + 8 + 3

It is clear that this isn't the best we can do. It is not even a local best. (Tweak the tail of the path to select 8 rather than 3 - what other changes do you see?)

But how can we be sure that we always find the best?

2	5	1	6	7	3	2
3	2	6	8	2	9	3
1	7	6	8	5	3	8
8	6	8	3	4	2	1
2	6	3	8	2	3	4
6	7	5	6	8	4	2
6	3	4	6	8	3	6

Algorithm

Look at the second row. It is easy to decide what the best path would be if the puzzle only had two levels

2	5	1	6	7	3	2
3	2	6	8	2	9	3
1	7	6	8	5	3	8

For each new row

For each element of the row

Look at the three (or fewer) choices: pick the best of them

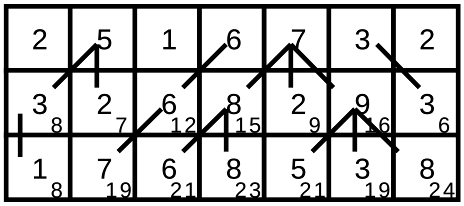
Store the running total for following round

For each square, remember which

spot the path came from (lines)

2	5	1	6	7	3	2
3 8	2 7	6 12	8 15	2 9	9 16	3 6
1	7	6	8	5	3	8

Iteration



At each stage, we build on the previous results.

- Note that some squares are never selected (the 1 and 2s in first row)
- Note that some paths are started, and then dropped (3 to 3)
  - These will never be used again

This algorithm is bottom-up rather than top-down.

Input to each new round: contents of current row, and the running totals from previous row. We don't care about prior path yet.

For solution: select the largest total in last row, and follow path back.

Integer Knapsack Problem

A thief with a knapsack finds a horde of jewels.

He has a limited amount of space in his knapsack

He knows the size (width of rectangle below) and value (area) of each jewel.

Problem - fill his knapsack with greatest value possible.

Sample problem:

{size, value} = {{3, 4}, {4, 5}, {7, 10}, {8, 11}, {9, 13}};

4/3 = 1.33333...,

5/4 = 1.25,

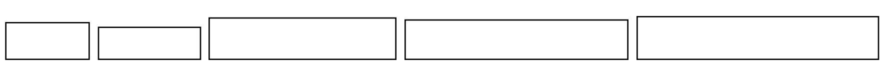
10/7 =1.42,

11/8 =1.375,

13/9 = 1.44444...

Many instances of each type of jewel: some of greater value/size (and thus more efficient)

In figure below, wish to maximize area within a fixed length



15

Application

Assume that a factory can make several different products.
  Some take longer than others.
    size == time
  Some have higher profit margin than others
    value == profit
All work must be done in one shift, to increase the quality.

Problem: find the collection of jobs that maximizes profit.

16

Sample results

{size, value} = {{3, 4}, {4, 5}, {7, 10}, {8, 11}, {9, 13}};
Cap 0 result 0
Cap 1 result 0
Cap 2 result 0
Cap 3 result 4 # Smallest jewel has length 3
Cap 4 result 5
Cap 5 result 5
Cap 6 result 8 {3,4} {4,5} {7, 10}
Cap 7 result 10 {3, 4} + {4, 5} vs..{7, 10}
Cap 8 result 11
{7, 10} {8, 11}
Note {8,11} has more area, but {7,10} is more efficient (higher)

Prepare the Jewels

```
class Jewel {
    private int size;
    private int value;
    Jewel(int s, int v) {
        size  = s;
        value = v;
    }
    int getValue() { return value; }
    int getSize() { return size; }
}
static Jewel prices[] = { new Jewel(3, 4), new Jewel(4, 5),
            new Jewel(7, 10), new Jewel(8, 11), new Jewel(9, 13) };
```


Sample Recursive Algorithm

```
// Our initial algorithms do not use dynamic programming
static int knap(int cap, Jewel items[]) {
    int max = 0;  // max == best we have seen yet
    if (0 == cap)
        return max;                 // Base case
    for (int i = 0; i < items.length; i++) {     // Try each jewel in turn
        int space = cap - items[i].getSize(); // What is left after this
        if (space >= 0) { // Use jewel and the best we can do with the rest
            int t = items[i].getValue() + knap(space, items);
            if (t > max)
                max = t; // Found something better than previous best
        }
    }
    return max;                     // Return the best we found
}
```

19

Main Program

```
// The main program.  This is where Java will start
public static void main(String args[])  {
    for ( int i = 0 ; i < prices.length ; i++ )   // Display the problem
        System.out.println(prices[i]);
    Tics time = new Tics();             // Start the global clock
    // Solve the problem for knapsacks of size [1..RUNS]
    for (int i = 0; i <= RUNS; i++) {
        Tics tic = new Tics();          // Start a clock for this problem
        int res = knap(i, prices);      // Get result
        System.out.println("Cap " + i + " result " + res + " took " + tic +
          " tics");
    }
    System.out.println("Took " + time + " tics to solve all of the cases");
}
```

20

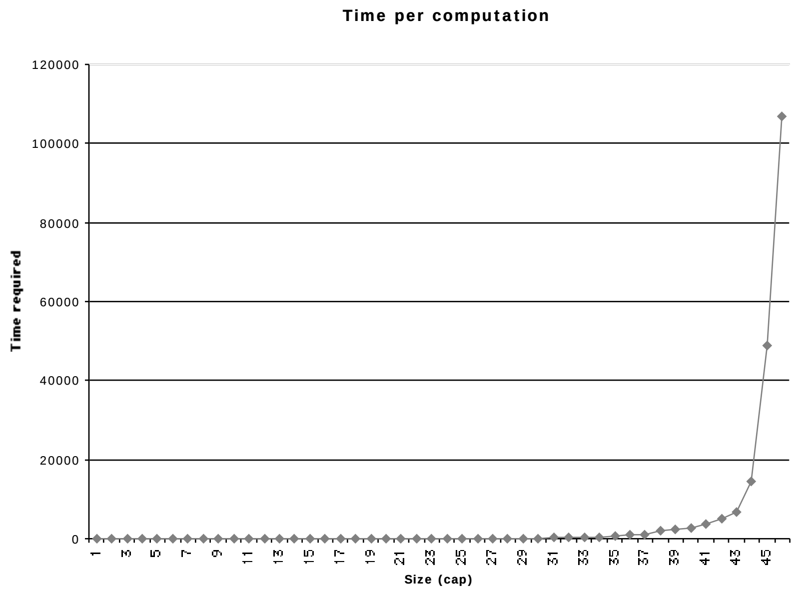
Some Results

Cap 12 result 17 took 2 tics
Cap 13 result 18 took 3 tics
….
Cap 16 result 23 took 5 tics
Cap 17 result 24 took 111 tics
Cap 18 result 26 took 8 tics
...
Cap 28 result 40 took 200 tics
...
Cap 39 result 56 took 11081 tics
Cap 40 result 57 took 17797 tics
Cap 41 result 59 took 12434 tics

Notice

- Changes are not smooth
- Later cases take a long time

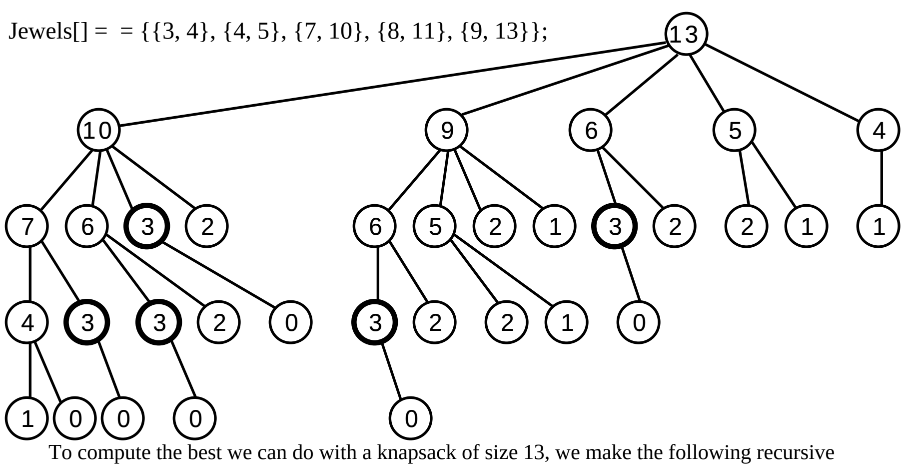
21



22

Why does it take so long?

Jewels[] = = {{3, 4}, {4, 5}, {7, 10}, {8, 11}, {9, 13}};



To compute the best we can do with a knapsack of size 13, we make the following recursive calls. (13 - 3 = 10, 13 - 4 = 9, 13 - 7 = 6, 13 - 8 = 5, 13 - 9 = 4.)

Some values are recomputed may times: see 3 above.

We don't need to compute 8, 11, or 12 to solve the problem for size 13. Can solve Top down.

Minor Improvements

1) Don't make call when there's no space - remove leaves with value 0
2) Since we store jewels in increasing size, break out when space < 0
3) Or store jewels in order of efficiency, so we try better values first.
4) Generate solutions in sorted order - see next slide
5) Branch and Bound

```
static int knap(int cap, Jewel items[]) {
    int max = 0;    // max == best we have seen yet
    if (0 == cap) return max;                       // Base case
    for (int i = 0; i < items.length; i++) {        // Try each jewel in turn
        int space = cap - items[i].getSize();       // What is left after this
        if (space >= 0) {
            int t = items[i].getValue() + knap(space, items);
            if (t > max) max = t;
        }
    return max;                     // Return the best we found
}
```


Ordering Sets

Results for first two improvements

311614 tics	Original
250712 tics	Don't make calls when space = 0
133122 tics	Break out when space < 0

Can do even better if we generate solutions in order by Jewel size

Algorithm looks at every combination of first choice, second choice, etc

But the sets {Opal, Diamond, Ruby}, {Ruby, Diamond, Opal}and {Ruby, Opal, Diamond} are not really different: just different order.

By generating the solutions in increasing or decreasing order of jewel size, we can look at far fewer possibilities (Don't look at all 3! cases above)

Implementation: Try the first jewel 0 or more times. For each attempt, try the next largest jewel 0 or more times, and then….

Branch and Bound

Branch and Bound is a way to prune off legal, but fruitless, searches

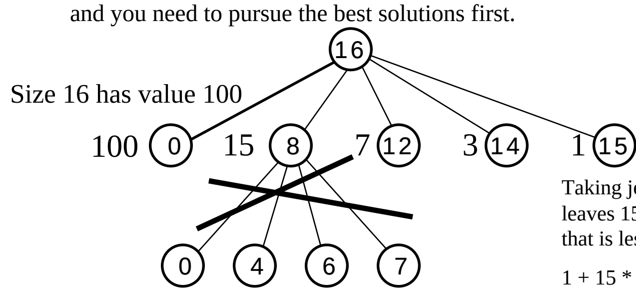
Jewels[] = {{1, 1}, {2, 3}, {4, 7}, {8, 15}, {16, 100}};

The best we can do is 100/16 per unit of space: most value w/ 16.

There is no way we can extend a value of 15 with only space 8 to beat a score of 100. Thus we do not need to look at any of the descendants of 8.

Similarly, there is no need to extend the other choices (4, 2, or 1).

For this to work well, you need to have jewels of very different efficiency, and you need to pursue the best solutions first.



Taking jewel of size 1 scores 1, leaves 15 left. Best we can do with that is less than 100

1 + 15 * 100/16 < 100

Dynamic Programming

However Dynamic Programming does much better

311614 tics	Original
250712 tics	Don't make calls when space = 0
133122 tics	Break out when space < 0
1604 tics	Dynamic Programming - two orders of magnitude better

```
// Initialize an array of solutions
static final int UNKNOWN = -1;
...
int maxKnown[] = new int[RUNS + 1];
for (int i = 0; i <= RUNS; i++)
    maxKnown[i] = UNKNOWN;
```

27

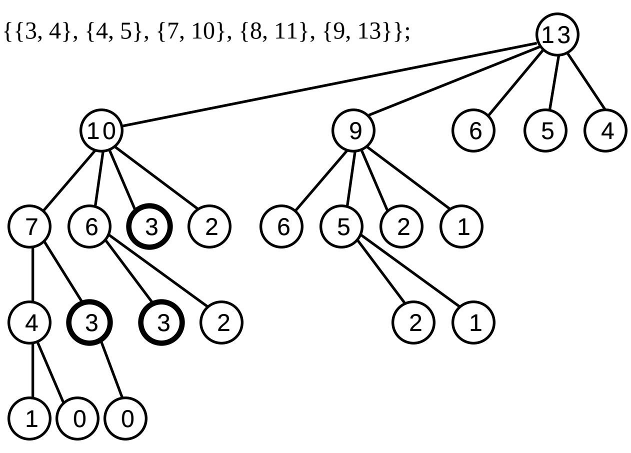
New Algorithm

```
static int knap(int cap, Jewel items[], int maxKnown[]) {
    if (maxKnown[cap] != UNKNOWN)
        return maxKnown[cap];      // Don't compute this twice
    int t, max = 0;                // best we have seen yet
    for (int i = 0; i < items.length; i++) {      // Try each jewel in turn
        if ((int space = cap - items[i].getSize()) >= 0) { // Room left?
            t = knap(space, items, maxKnown) + items[i].getValue;
            if (t> max)
                max = t; // Found something better than best
        }
    }
    maxKnown[cap] = max;           // Remember the best
    return max;                    // Return the best we found
}
```

28

Algorithm in action

Jewels[] = {{3, 4}, {4, 5}, {7, 10}, {8, 11}, {9, 13}};



This graph is smaller because we don't expand starting points we have seen before.

Thus we only need to expand 3, 4, 5, and 6 one time.

But all we know is value!

Thief now knows max value of his load - but what choices should he make?

We add a new array to hold the solution, illustrated below

- We save time and space if we do not store the whole solution
- For each value of cap, we store the size of the last jewel we add rather than all the jewels, and then lookup the rest of the solution
  - Recall our example {{3, 4}, {4, 5}, {7, 10}, {8, 11}, {9, 13}}
- The array below lets us replace the problem of filling a knapsack of size 15 with the problem of adding a jewel of size 3, and then fill a knapsack of size 12 (= 15-3). Solve this problem by looking up entry for 12.
- Solution for 15 is {3, 4} + {3, 4} + {9, 13}. Representation is not unique: we store as 3, 3, 9, but we could store as 9, 3, 3 or 3, 9, 3.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	3	4	4	3	7	8	9	3	3	3	3	7	3	7

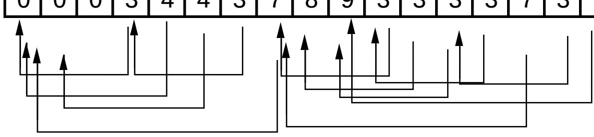
Translating Array into solution

In effect, the array holds links to earlier values

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	3	4	4	3	7	8	9	3	3	3	3	7	3	7

The array and some of the skips implied. To solve 15, skip to 12, 9, then 0

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	3	4	4	3	7	8	9	3	3	3	3	7	3	7



31

New Algorithm

```
static int knap(int cap, Jewel items[], int maxKnown[], int firstPick[]) {
	if (maxKnown[cap] != UNKNOWN)
		return maxKnown[cap];		// Don't compute this twice
	int t, max = 0;			// best we have seen yet
	for (int i = 0; i < items.length; i++) {	// Try each jewel in turn
		if ((int space = cap - items[i].getSize()) >= 0) { // Room left?
			t = knap(space, items, maxKnown) + items[i].getValue;
			if (t> max) {
				max = t;	// Found something better than best
				first = items[i].getSize();		// Remember size of the last jewel we added
			}
		}
	}
	maxKnown[cap] = max;	// Remember the best
	firstPick[cap] = first;			// Remember the jewel
	return max;				// Return the best we found
}
```

32

Limits to Dynamic Programming

Why can't we use Dynamic Programming all the time?

Dynamic Programming assumes that you can store enough previous results to reduce the amount of work.

What if there are too many different sub problems?

Consider the case that the sizes are not integers: say

{{10/3, 4}, {47/7, 5}, {34/5, 10}, {89/11, 11}, {120/13, 13}};

Rather than reducing the problem 13 to problems 10, 9, 6, 5, and 4, we face the problems 9 2/3, 44/7, etc. A simple array is not enough.

(In fact, we could express all the elements above as a fractions with denominator 3*5*7*11*13, and solve this case with an array of 3*5*7*11*13*13 items. This may not save us much time, and does not work if the values are irrational numbers like sqrt(2) and pi.)

More Limits: Dimension

Consider another case: the solution to the Queens problem. We could store non-attacking positions for the first three columns, and use these to decide which positions for the rest of the columns would be fruitful.

- Can we store the prior positions briefly?
- Can we search that storage faster than we could compute new values from scratch?

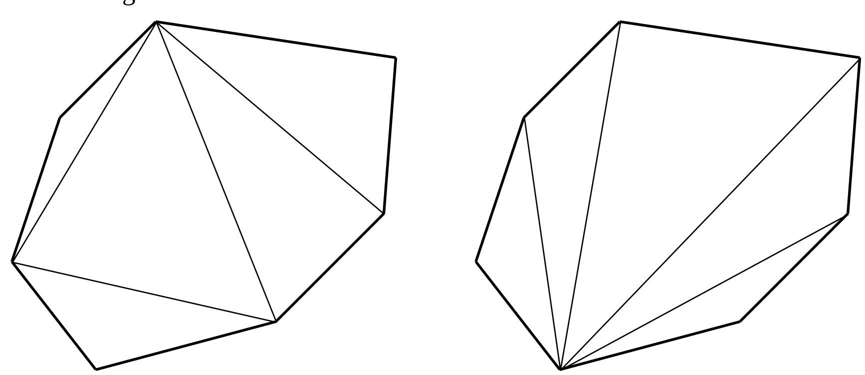
In general, Dynamic Programming depends upon being able to characterize previous problems succinctly.

The traveling salesman problem (given a collection of cities, find a path that starts and ends in the same city with the minimal path length) has too many sub-problems to store.

More good problems

Given a convex polygon, find the triangulation with the smallest total perimeter.

- Here is the a polygon with two triangulations. We wish to minimize the length of the added lines.

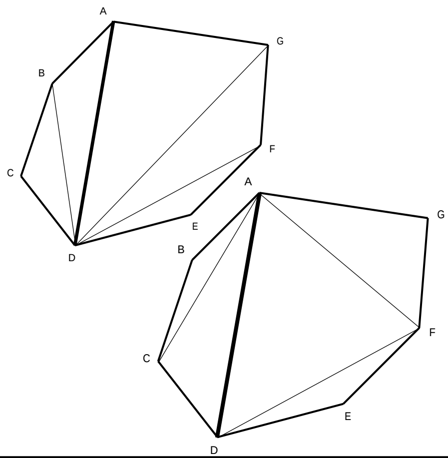


35

Insight

What are the subproblems that we need to solve and save?

Vertex A will be connected to vertex C, D, E, or F.



Assume that A is connected to D. We can use this to split our original problem in two.

Then we have two sub-problems: minimizing the triangulation of the polygon A, B, C, D and minimizing the triangulation of the polygon A, D, E, F, G

We then combine optimal solutions from two halves to get optimal solution to whole.

A 1-dimensional array of known solutions is not enough for this problem

36

References

Sedgewick's Algorithms has a good discussion of these problems.

Baase and Van Gelder, Computer Algorithms has a chapter on Dynamic Programming

Cormen, Leiserson, and Rivest, Introduction to Algorithms discuss the polygon triangulation problem

One of the Advanced Topics Lectures covers non-exact pattern matching (the problem and algorithm are described in Baase and Van Gelder)