Algorithms TIN092

Yet another exercise session

What is a Dynamic Programming?

From lectures:
“...Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.”

Shorter:
Recurrences, recurrences and yet another time recurrences.

Q: Recurrence for what?
A: *Optimization* problem assumes there is a function that has to be minimized/maximized, recurrence is written for this function values on different subproblems.

Q: I don't know how to solve a recurrence! What should I do???
A: That's what you need algorithms for, crybaby

Q: Why “Dynamic Programming”, then?
A: Historical issue. Never mind.

How do I apply Dynamic Programming?

1. Write a **recurrence** for optimal value function.

2. Write an algorithm to calculate the optimal value of a function

    2.1 Top-down (memoization)
    2.2 Bottom-up (iterative)

3. Recover solution if needed

(Optimal value ≠ Solution that reaches optimal value)

Example 1: KT6.1

*Find an independent set (IS) of **maximum weight** on a path*
Path = Graph G(V,E) (undirected) s.t.
V=[1:n], E={(i,j)|i-j=1}={(1,2),(2,3),(3,4),...,(n,n-1)}

Independent Set = subset V' of V, s.t. no two vertices of V' are connected

(8 ISs. Can you find all of them?)

Weight: a function w, that assigns each node v, some numeric value v(e)

(Solution is {1,3})

How do I apply Dynamic Programming?

1. OPT(n) = (OPT(n-2)+w(v_n), OPT(n-1)),
   OPT(0) = 1, OPT(1) = w(v_1)

   OPT(n) – max weight independent set for subpath on [1:n-1] vertices

2.1 As usual: M[n] for storage

```
M[1]=w(v[1]);M[0]=0;
OPT(x) {
   if (M[x] is empty) then
     M[x]=max(OPT(x-2)+w(V[x]),OPT(x-1));
   fi
   return M[x];
}
return OPT(n);
```

How do I apply Dynamic Programming?

2.2. Iterative:

```
M[1]=w(v[1]);M[0]=0;

For i=2:n do
  M[i]=max(M[i-2]+w(v[i]),M[i-1]);
od

return M[n]
```

3. We need to output the vertices of our choice in top-bottom approach:

```
...
M[x]=max(OPT(x-2)+w(V[x]),OPT(x-1));
if (OPT(x-2)+w(V[x])>=OPT(x-1))
   output w(V[x])
fi
...
```

1 vertex might be lost. Which one? How to deal with it?

How do I apply Dynamic Programming?

Correctness argument:
Inductive,
Claim: recurrence is giving the correct value of optimal value function.
Proof: Base: correct for n=0,1 (obvious),
Step: correct fo n<k, n=k?
Split ISs into two classes,
one with vertex v[n] (I),
another one without v[n] (J).

Any IS from I **can be represented** as union of an IS for problem of size k-2 (subpath v[1],...,v[k-2]]) and v[k] and *vice versa*, thus *best* and **reachable** solution for this class is OPT(k-2)+w(v_k).

Any IS from J **can be represented** as an IS for problem of size k-1 (subpath v[1],...,v[k-1]) and *vice versa*, thus *best* and **reachable** solution for this class is OPT(k-1).

Conclusion: best and reachable solution for union of I and J is given by max(OPT(k-1),OPT(k-2)+w(v_k)).

That concludes the proof.

How do I apply Dynamic Programming?

Complexity:

Iterative: obvious (simple loop)

Recursive:
Arithmetical operations are assumed to take O(1) time (small numbers). Calls take O(1). Let us put assign all arithmetical operations to corresponding calls (still constant running time).

Number of recursive calls is no more than 2n (each time 2 recursive calls are made, one value in M array is filled).

Then, running time is no more than C (worst time of arithmetic operation+time for recursive call, 0(1)) * 2n, and 2nC belongs to O(2n) = O(n) class (in fact, theta).

Example 2: KT 6.8

Robots waves, r[n],
EMP charges, f[n].

Choose series of shooting days, d[n], s.t.
Sum[i=1:n](min(r[i],f[i-j])*d[i] is maximized (where j<i is the last shooting day before i)

Hint: solve with DP :-D

Hint: still several options to choose, but not with “the last vertex”, indeed, one should shoot on the last day.

Hint: last shooting day is definitely a parameter of recurrence. It suffices to have only it as a parameter.

Hint: nothing changes in between two shooting days, except EMP charge (no robots are killed).

Hint: complexity is “worse” than it was before. It is dependent on number of recursive calls after all. Try to do iterative version first – it will give an obvious bound.