

Note: these are just my own notes that I used in class. The material is mostly drawn from the Dasgupta book, chapters 6.1, 6.2, 6.4, 6.6 –jfh

Last time: examined the knapsack problem.

Room is on fire. You have a knapsack that can hold 10 lbs.

item	value	weight
T&G	\$40	5
Abba	\$90	4
Ortiz bobb	\$30	3
Socks	\$3	1

Idea: solve subproblems!

Given capacity $w = 0, 1, \dots, 10$, and items $0, 1, 2, \dots, k$, which ones would you take? Fill in grid.

Idea: make a table saying for each capacity c and each possible selection of items ($0, 01, 012, 0123, \dots$) whether you would take the LAST item, given this capacity knapsack. Also note, for each such knapsack, what's the value you get by filling it in the best possible way.

To compute $V(c, i)$:

If we take item i , we've got $c-w(i)$ worth of capacity left, and value $v(i) + V(c-w(i), i-1)$ worth of value.

If we OMIT item i , we've got c worth of capacity left, and value $V(c, i-1)$. So we should compare:

```
if (w(i) <= c) :
    valueWith = v(i) + V(c-w(i), i-1);
else
    valueWith = -1; // so it'll never be chosen!
valueWithout = V(c, i-1);
if valueWith > valueWithout
    select(c, i) = 1
    V(c, i) = valueWith;
else
    select(c, i) = 0;
    V(c, i) = valueWithout;
```

Now all we have to do is compute this for $c = 0, 1, \dots$ up to the knapsack's capacity, and for $i=1, 2, \dots$ number of items.

Reading back the answer takes some chops. Have students write code to do it!

Another example: longest increasing subsequence
4 1 7 5 2 5 8 6

Solution: draw all permissible arrows. It's a DAG. Path-in-dag = increasing subsequence. So just find longest path in dag.

```
for j = 1..n :
  L(j) = 1 + max[L(i) : (i, j) in E]
return max_j L(j)
```

[for the empty set, max is zero!]

Again, there are SUBPROBLEMS, they're ORDERED, and a relation that says how to solve a subproblem, given answers to "smaller" or "previous" subproblems.

Final example: Recursive function like "Fibonacci". As long as we have a way to INDEX the subproblems, we can make a memo of the result that we got in solving the subproblem.

```
Proc fib(n)
If (n = 0 or 1) return 1
Else return fib(n-1) + fib(n-2)
```

```
Proc fib(n)
If (n = 0 or 1) return 1
Else
  A = fib(n-1)
  B = fib(n-2)
  Return A + B
```

```
Proc fib(n)
Set value[0..n] to Null
```

```
If (n = 0 or 1) return 1
Else
  A = fibH(n-1)
  B = fibH(n-2)
  Return A + B
```

```
Proc fibH(k)
```

```
If value(k) is null
    Value(k) = fib(k)
Return value(k)
```

This reduces $O(2^n)$ to $O(n)$.
Called "memoization"

Notice the difference:

```
KnapsackFast(w, v, n, cap)
// n items, values v(0..n-1), weights w(0, ..., n-1)
// knapsack of capacity cap.
// Find the value of the best possible filling of knapsack
select = n+1 x cap+1 array of Booleans, all false
value = n+1 x cap+1 array of ints, all None initially
```

```
value[0, <all>] = 0;
value[<all>, 0] = 0;
```

```
for c = 1 to cap
    for i = 1 to n
        if (c <= w(i) ) :
            valueWith = v(i) + value(c-w(i), i-1)
        else
            valueWith = -1;
            valueWithout = value(c, i-1);
            value(c, i) = max(valueWith, valueWithout);
return value(n, c)
```

```
KnapsackSlow(w, v, n, cap)
// n items, values v(0..n-1), weights w(0, ..., n-1)
// knapsack of capacity cap.
// Find the value of the best possible filling of knapsack
return value(n, c)
```

```
function value(c, i)
    if (c == 0) or (i == 0) : return 0
    else
        if (c <= w(i) ) :
            valueWith = v(i) + value(c-w(i), i-1)
        else
            valueWith = -1;
            valueWithout = value(c, i-1);
            return max(valueWith, valueWithout);
```

Difference: $O(n*c)$ vs $O(\text{HUGE})$

Another problem: longest increasing subsequence.

Another example: longest increasing subsequence
4 1 7 5 2 5 8 6

Solution: draw all permissible arrows. It's a DAG. Path-in-dag = increasing subsequence. So just find longest path in dag.

```
for j = 1..n :  
  L(j) = 1 + max[L(i) : (i, j) in E]  
return max_j L(j)
```

[for the empty set, max is zero!]

Again, there are SUBPROBLEMS, they're ORDERED, and a relation that says how to solve a subproblem, given answers to "smaller" or "previous" subproblems.