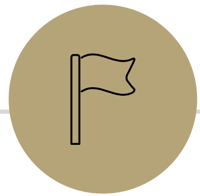
Even More Dynamic Programming

CSE 417 24Wi
Lecture 12



Modifying Problems

Updating the Problem

$OPT(i,j,f)$ is the maximum amount of eggs Baby Yoda can collect on a legal path from (i,j) to $(0,0)$ using the force f times to knock over rocks.

For simplicity, assume there are no rocks at the starting location (r-1,c-1)

Here was the old rule without the force – how do we update?

$$OPT(i,j,f) = \begin{cases} -\infty & \text{if } i < 0 \text{ or } j < 0 \text{ or } f < 0 \\ eggs(0,0) & \text{if } i = 0 \text{ and } j = 0 \text{ and } f \geq 0 \\ \max\{OPT(i-1,j,f-rocks(i-1,j)), OPT(i,j-1,f-rocks(i,j-1))\} + eggs(i,j) & \text{otherwise} \end{cases}$$

Casting Boolean as an integer (subtract 1 if you would need to knock over rocks)

Updating the Problem

$OPT(i,j,f)$ is the maximum amount of eggs Baby Yoda can collect on a legal path from (i,j) to $(0,0)$ using the force f times to knock over rocks.

For simplicity, assume there are no rocks at the starting location (r-1,c-1)

Here was the old rule without the force – how do we update?

$$OPT(i,j,f) = \begin{cases} -\infty & \text{if } i < 0 \text{ or } j < 0 \text{ or } f < 0 \\ eggs(0,0) & \text{if } i = 0 \text{ and } j = 0 \text{ and } f \geq 0 \\ \max\{OPT(i-1,j,f-rocks(i-1,j)), OPT(i,j-1,f-rocks(i,j-1))\} + eggs(i,j) & \text{otherwise} \end{cases}$$

`rocks(i,j)` doesn't guarantee $-\infty$ anymore. Only if you were out of force uses before trying to jump onto that location.

Baby Yoda Searching

(c-1,r-1)

r-1

Y-coordinate

0

X-coordinate

0

c-1

What can we fill in?

a/b

a is for (x,y,0)

b is for (x,y,1)

Baby Yoda Searching

(c-1,r-1)

r-1

Y-coordinate

0

$0/?$	$1/?$	$2/?$	$2/?$	$2/?$	$2/?$	$2/?$	$2/?$

X-coordinate

0 c-1

What can we fill in?

a/b

a is for (x,y,0)

b is for (x,y,1)

Baby Yoda Searching

(c-1, r-1)

r-1	$1/?$	$1/?$	$1/?$	$1/?$	$3/?$	$4/?$	$4/?$	$4/?$
	$1/?$	$1/?$	$1/?$	$1/?$	$3/?$	$4/?$	$4/?$	$4/?$
	$0/?$	$0/?$	$1/?$	$1/?$	$3/?$	$3/?$	$3/?$	$3/?$
	$0/?$	$1/?$	$-\infty/?$	$2/?$	$3/?$	$3/?$	$3/?$	$3/?$
	$0/?$	$1/?$	$2/?$	$2/?$	$2/?$	$2/?$	$2/?$	$2/?$
0	$0/?$	$1/?$	$2/?$	$2/?$	$2/?$	$2/?$	$2/?$	$2/?$

Y-coordinate

X-coordinate

0 c-1

What can we fill in?
Everything with $f = 0$ in the same order as before.

Entries are slightly different – we're handling rocks differently.

Baby Yoda Searching

(c-1, r-1)

r-1	1/?	1/?	1/?	1/?	3/?	4/?	4/?	4/?
	1/?	1/?	1/?	1/?	3/?	4/?	4/?	4/?
	0/?	0/?	1/?	1/?	3/?	3/?	3/?	3/?
	0/?	1/?	$-\infty$/?	2/?	3/?	3/?	3/?	3/?
	0/?	1/?	2/?	2/?	2/?	2/?	2/?	2/?
0	0/?	1/?	2/?	2/?	2/?	2/?	2/?	2/?

Y-coordinate

X-coordinate

0 c-1

What can we fill in?

Again from left to right, bottom to top, now filling in

Baby Yoda Searching

(c-1, r-1)

r-1	1/1	1/1	1/4	1/4	3/4	4/5	4/5	4/5
	1/1	1/1	1/4	1/4	3/4	4/5	4/5	4/5
	0/0	0/0	1/4	1/4	3/4	3/4	3/4	3/4
	0/0	1/1	$-\infty/3$	2/3	3/3	3/3	3/3	3/3
	0/0	1/1	2/2	2/2	2/2	2/2	2/2	2/2
0	0/0	1/1	2/2	2/2	2/2	2/2	2/2	2/2

Y-coordinate

X-coordinate

0 c-1

What can we fill in?
Again from left to right, bottom to top, now filling in

Dynamic Programming Process

1. Define the object you’re looking for

2. Write a recurrence to say how to find it

3. Design a memoization structure

4. Write an iterative algorithm

How we decide on the memo structure?

Ask yourself "what are all the possible valid combinations of parameters?" and then "what would store those?"

For Baby Yoda with the force:

Current row could be 0 to $r - 1$

Current column could be 0 to $c - 1$

Current force uses remaining could be 0 or 1

Each call returns an `int`, so

So `int[r][c][2]` works (or two `int[][]`, etc.)

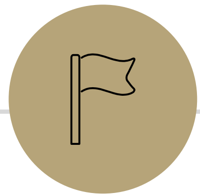
But how does it work?

The key to DP is having a recursive call for every (plausible) choice you could make.

We could go left or down (we could use the force here or save it for later).

The magic of recursion checks all the possibilities (if it’s better to “save” using the force for later, you check the option that ‘goes around’ by making a different decision earlier).

It’s not your job to figure out a rule (“we should go to the left when there aren’t rocks in the way and the number of eggs in the row is at least 3 or there is a set of rock directly below you…”) recursion builds up the best options automatically by trying all options.



Bells,Whistles, and optimiziation

Baby Yoda Searching

(c-1, r-1)

Y-coordinate								
r-1	1/1	1/1	1/4	1/4	3/4	4/5	4/5	4/5
	1/1	1/1	1/4	1/4	3/4	4/5	4/5	4/5
	0/0	0/0	1/4	1/4	3/4	3/4	3/4	3/4
	0/0	1/1	$-\infty/3$	2/3	3/3	3/3	3/3	3/3
	0/0	1/1	2/2	2/2	2/2	2/2	2/2	2/2
0	0/0	1/1	2/2	2/2	2/2	2/2	2/2	2/2

X-coordinate

0 c-1

So should Baby Yoda go left or down?

Which Way to Go

When you're taking the `max` in the recursive case, you can also record which option gave you the max.

That's the way to go.

We'll ask you to do that once…but for the most part we'll just have you find the number.

Optimizing

Do we need all that memory?

Let's go back to the simple version (no using the Force)

Recurrence Form

$$OPT(i,j) = \begin{cases} -\infty & \text{if } rocks(i,j) \text{ is true} \\ -\infty & \text{if } i < 0 \text{ or } j < 0 \\ eggs(0,0) & \text{if } i = 0 \text{ and } j = 0 \\ \max\{OPT(i-1,j), OPT(i,j-1)\} + eggs(i,j) & \text{otherwise} \end{cases}$$

What values do we need to keep around?

Baby Yoda Searching

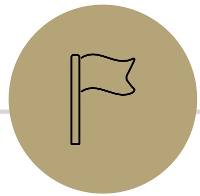
(c-1, r-1)

Y-coordinate								
r-1	1	1	1	2	3	4	4	4
	1	1	1	2	3	4	4	4
	0	0	1	2	3	**3**	3	3
	0	$-\infty$	$-\infty$	$-\infty$	3	3	3	3
	0	1	$-\infty$	2	2	2	2	2
0	0	1	2	2	2	2	2	2
	0							c-1

X-coordinate

Need one spot left and one down.

Keep one full row, and a partially full row around. $\Theta(c)$ memory.



More Problems

Maximum Contiguous Subarray Sum

We saw an $O(n \log n)$ divide and conquer algorithm.

Can we do better with DP?

Given: Array $A[]$

Output: i, j such that $A[i] + A[i+1] + \cdots + A[j]$ is maximized.

Dynamic Programming Process

1. Define the object you're looking for

2. Write a recurrence to say how to find it

3. Design a memoization structure

4. Write an iterative algorithm

Maximum Contiguous Subarray Sum

We saw an $O(n \log n)$ divide and conquer algorithm.

Can we do better with DP?

Given: Array $A[]$

Output: i, j such that $A[i] + A[i+1] + \cdots + A[j]$ is maximized.

For today: just output the value$A[i] + A[i+1] + \cdots + A[j]$.

Is it enough to know `OPT(i)`?

Approaching the problem recursively

For DP-style recursion, we’re usually looking for an array that’s one element smaller.

For today we’ll look at arrays with indicies 0,1,2,…k.

(where $k < n$)

Define $OPT(k)$ to be the sum of the maximum-sum-subarray for the subarray of indicies $0, \dots, k$. (I.e. answering the problem pretending the array from $k+1, \dots, n-1$ doesn’t exist).

Trying to Recurse

0	1	2	3	4	5	6	7
5	-6	3	4	-5	2	2	4

$OPT(3)$ would give i =2, $j = 3$

$OPT(4)$ would give $i = 2, j = 3$ too

$OPT(7)$ would give $i = 2, j = 7$ – we need to suddenly backfill with a bunch of elements that weren't optimal…

How do we make a decision on index 7? What information do we need?

What do we need for recursion?

If index i IS going to be included

We need the best subarray **that includes index $\boldsymbol{i-1}$**

If we include anything to the left, we'll definitely include index $i - 1$ (because of the contiguous requirement)

If index i isn't included

We need the best subarray up to $i - 1$, regardless of whether $i - 1$ is included.

Two Values

Pollev.com/robbie

Need two recursive values:

$INCLUDE(i)$: sum of the maximum sum subarray among elements from 0 to i **that includes index $\boldsymbol{i}$** in the sum

$OPT(i)$: sum of the maximum sum subarray among elements 0 to i (that might or might not include i)

How can you calculate these values? Try to write recurrence(s), then think about memoization and running time.

Recurrences

$$INCLUDE(i) = \begin{cases} \max\{A[i], A[i] + INLCUDE(i-1)\} & \text{if } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$OPT(i) = \begin{cases} \max\{INCLUDE(i), OPT(i-1)\} & \text{if } i \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

If we include i, the subarray must be either just i or also include $i-1$.

Overall, we might or might not include i. If we don't include i, we only have access to elements $i-1$ and before. If we do, we want $INCLUDE(i)$ by definition.

Example

A

0	1	2	3	4	5	6	7
5	-6	3	4	-5	2	2	4

$OPT(i)$

0	1	2	3	4	5	6	7
5							

$INCLUDE(i)$

0	1	2	3	4	5	6	7
5							

Example

A

0	1	2	3	4	5	6	7
5	-6	3	4	-5	2	2	4

$OPT(i)$

0	1	2	3	4	5	6	7
5	5						

$INCLUDE(i)$

0	1	2	3	4	5	6	7
5	-1						

Example

A

0	1	2	3	4	5	6	7
5	-6	3	4	-5	2	2	4

$OPT(i)$

0	1	2	3	4	5	6	7
5	5	5	7	7	7	7	10

$INCLUDE(i)$

0	1	2	3	4	5	6	7
5	-1	3	7	2	4	6	10

Pseudocode

```
int maxSubarraySum(int[] A)
  int n=A.length
  int[] OPT = new int[n]
  int[] Inc = new int[n]
  inc[0]=A[0]; OPT[0] = max{A[0],0}
  for(int i=0;i<n;i++)
    inc[i]=max{A[i], A[i]+inc[i-1]}
    OPT[i]=max{inc[i], opt[i-1]}
  endFor
return OPT[n-1]
```

Recursive Thinking In General

As before, the hardest part is designing the recurrence.

It sometimes helps to think from multiple different angles.

Top-down: What's the first step to take?

Baby Yoda will first go left or down. Use recursion to find out which of left or down is better.

The element at index i is either included or excluded. Try both options.

Recursive Thinking In General

Bottom-Up: What information could a recursive call give me that would help?

How does a path through most of the map help Baby Yoda?

Well we just need to know the values one left and one down.

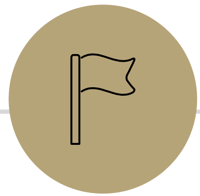
The maximum subarray of which parts of the array would help us solve the full array

Well if we know the best subarray without $i - 1$, and the best subarray without $i - 1$ we can figure out whether to add i or not.

Recursive Thinking In General

Some people refer to the “Optimal Substructure Property”

From the optimum (most eggs, largest sum) for a slightly smaller problem (Baby Yoda starting closer to the end, slightly smaller array), we need to be able to build up the optimum for the full problem.



Edit Distance

Edit Distance

More formally:

The edit distance between two strings is:

The minimum number of **deletions, insertions,** and **substitutions** to transform string x into string y.

Deletion: removing one character

Insertion: inserting one character (at any point in the string)

Substitution: replacing one character with one other.

Example

What's the distance between `babyyodas` and `tastysoda`?

B	A	B		Y	Y	O	D	A	S
sub		sub	ins		sub				del
T	A	S	T	Y	S	O	D	A	

Distance: 5, one point for each colored box

Quick Checks – can you explain these?
If x has length n and y has length m, the edit distance is at most $\max(x, y)$

The distance from x to y is the same as from y to x (i.e. transforming x to y and y to x are the same)

Finding a recurrence

What information would let us simplify the problem?

What would let us "take one step" toward the solution?

"Handling" one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the "distance" by 1

OR realizing the characters are the same and matching for free.

$OPT(i, j)$ is the edit distance of the strings $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$.
(we're indexing strings from 1, it'll make things a little prettier).

The recurrence

"Handling" one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the "distance" by 1

OR realizing the characters are the same and matching for free.

What does delete look like? $OPT(i-1,j)$ (delete character from x match the rest)

Insert $OPT(i,j-1)$ Substitution: $OPT(i-1,j-1)$

Matching charcters? Also $OPT(i-1,j-1)$ but only if $x_i = y_j$

The recurrence (v1, we'll improve soon)

"Handling" one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the "distance" by 1

OR realizing the characters are the same and matching for free.

Delete | Insert | Substitution | TODO: Just Match

$$OPT(i,j) = \begin{cases} \min\{ 1 + OPT(i-1,j), 1 + OPT(i,j-1),\ 1 + OPT(i-1,j-1), \ldots \} & \\ j & \text{if } i = 0 \\ i & \text{if } j = 0 \end{cases}$$

The recurrence (v1, we'll improve soon)

"Handling" one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the "distance" by 1

OR realizing the characters are the same and matching for free.

Delete | Insert | Substitution | Just Match

$$OPT(i,j) = \begin{cases} \min\{1 + OPT(i-1,j), 1 + OPT(i,j-1),\ 1 + OPT(i-1,j-1), OPT(i-1,j-1) + \infty \cdot \mathbb{I}\{x_i \neq y_j\}\} & \\ j & \text{if } i = 0 \\ i & \text{if } j = 0 \end{cases}$$

Idea: only allow "just match" when you can just match.
Otherwise make it ∞ (will never be the min).
In code: if/else branch, probably. This is a math notation trick.

"Indicator" – math for "cast bool to int"

The recurrence

"Handling" one character of x or y

i.e. choosing one of insert, delete, or substitution and increasing the "distance" by 1

OR realizing the characters are the same and matching for free.

Delete | Insert | Sub and matching

$$OPT(i,j) = \begin{cases} \min\{1 + OPT(i-1,j), 1 + OPT(i,j-1), \mathbb{I}[x_i \neq y_j] + OPT(i-1,j-1)\} & \\ j & \text{if } i = 0 \\ i & \text{if } j = 0 \end{cases}$$

"Indicator" – math for "cast bool to int"

When we could match, we will never substitute; matching will always give us a better score! Still have to check delete, insert (those could be better).

Recurrence to Code

Just like with Baby Yoda, if you write the recursive code for this "normally" you'll have very slow code. Starting from $OPT(m, n)$, $OPT(m-1, n-1)$ can be reached by:

Delete then insert

Insert then delete

Matching or substitution

Even worse than (left, down) or (down, left).

Just like before, memoize the results.

Not a typo! It's memoize, not memorize.

Edit Distance

$\mathrm{OPT}(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0										
T 1										
A 2										
S 3										
T 4										
Y 5										
S 6										
O 7										
D 8										
A 9										

Edit Distance

B	A	B	Y
T	A	S	

Current subproblem: edit dist between BABY and TAS

$OPT(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1									
A 2	2									
S 3	3									
T 4	4									
Y 5	5									
S 6	6									
O 7	7									
D 8	8									
A 9	9									

Edit Distance

B	A	B	Y
T	A	S	

Delete: get rid of Y (cost 1)
BAB
TAS

$\mathrm{OPT}(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1									
A 2	2									
S 3	3									
T 4	4									
Y 5	5									
S 6	6									
O 7	7									
D 8	8									
A 9	9									

Edit Distance

B	A	B	Y
T	A	S	

Insert: insert S (cost 1)
BABY
TA

$OPT(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1									
A 2	2									
S 3	3									
T 4	4									
Y 5	5									
S 6	6									
O 7	7									
D 8	8									
A 9	9									

Edit Distance

B	A	B	Y
T	A	S	

Sub: Transform Y to S (cost 1)
BAB
TA

$OPT(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1									
A 2	2									
S 3	3									
T 4	4									
Y 5	5									
S 6	6									
O 7	7									
D 8	8									
A 9	9									

Edit Distance

B	A	B	Y
T	A	S	

Gold entry will be min of:

$1 +$ delete

$1 +$ insert

$1 +$ sub

$\mathrm{OPT}(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1									
A 2	2									
S 3	3									
T 4	4									
Y 5	5									
S 6	6									
O 7	7									
D 8	8									
A 9	9									

Edit Distance

$1 + 2$
$1 + 3$
$1 + 2$
Min: 3

Gold entry will be min of:
$1 + \text{delete}$
$1 + \text{insert}$
$1 + \text{sub}$

$OPT(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1	1	2	3	4	5	6	7	8	9
A 2	2	2	1	2	3	4	5	6	7	8
S 3	3	3	2	2	3	4	5	6	7	7
T 4	4	4	3	3	3	4	5	6	7	8
Y 5	5	5	4	4	3					
S 6										
O 7										
D 8										
A 9										

Edit Distance

Fill in the next two entries. Be careful with the sub/match distinction!

$\mathrm{OPT}(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1	1	2	3	4	5	6	7	8	9
A 2	2	2	1	2	3	4	5	6	7	8
S 3	3	3	2	2	3	4	5	6	7	7
T 4	4	4	3	3	3	4	5	6	7	8
Y 5	5	5	4	4	3					
S 6										
O 7										
D 8										
A 9										

Edit Distance

Fill in the next two entries. Be careful with the sub/match distinction!

$\text{OPT}(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1	1	2	3	4	5	6	7	8	9
A 2	2	2	1	2	3	4	5	6	7	8
S 3	3	3	2	2	3	4	5	6	7	7
T 4	4	4	3	3	3	4	5	6	7	8
Y 5	5	5	4	4	3	**3**	**4**			
S 6										
O 7										
D 8										
A 9										

Y's match, so sub is free!

Edit Distance

$\text{OPT}(i, j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1	1	2	3	4	5	6	7	8	9
A 2	2	2	1	2	3	4	5	6	7	8
S 3	3	3	2	2	3	4	5	6	7	7
T 4	4	4	3	3	3	4	5	6	7	8
Y 5	5	5	4	4	3	3	4	5	6	7
S 6	6	6	5	5	4	4	4	5	6	6
O 7	7	7	6	6	5	5	4	5	6	7
D 8	8	8	7	7	6	6	5	4	5	6
A 9	9	9	8	8	7	7	6	6	4	5

What if we want the list if inserts,delete,subs?

Or with Baby Yoda the actual path he has to go?

You can always find it. Just ask “well which recursive call was the one I used?” (which one was the minimum, in this problem)

If $OPT(i-1, j)$ was the minimum, then that means you should delete!

You can pretty much always find “the object” this way.

On a future homework, a problem asks you to write the bookkeeping code. For lecture/most problems we’re going to just find the number.

Edit Distance

$\mathrm{OPT}(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1	1	2	3	4	5	6	7	8	9
A 2	2	2	1	2	3	4	5	6	7	8
S 3	3	3	2	2	3	4	5	6	7	7
T 4	4	4	3	3	3	4	5	6	7	8
Y 5	5	5	4	4	3	3	4	5	6	7
S 6	6	6	5	5	4	4	4	5	6	6
O 7	7	7	6	6	5	5	4	5	6	7
D 8	8	8	7	7	6	6	5	4	5	6
A 9	9	9	8	8	7	7	6	6	4	5

Edit Distance

$OPT(i,j)$	0	B, 1	A, 2	B, 3	Y, 4	Y, 5	O, 6	D, 7	A, 8	S, 9
0	0	1	2	3	4	5	6	7	8	9
T 1	1	1	2	3	4	5	6	7	8	9
A 2	2	2	1	2	3	4	5	6	7	8
S 3	3	3	2	2	3	4	5	6	7	7
T 4	4	4	3	3	3	4	5	6	7	8
Y 5	5	5	4	4	3	3	4	5	6	7
S 6	6	6	5	5	4	4	4	5	6	6
O 7	7	7	6	6	5	5	4	5	6	7
D 8	8	8	7	7	6	6	5	4	5	6
A 9	9	9	8	8	7	7	6	6	4	5

Dynamic Programming Process

1. Define the object you're looking for

    OPT(i,j) is the minimum number of insertions, deletions, and substitutions required to transform $x_1 \ldots x_i$ to $y_1 \ldots y_j$

2. Write a recurrence to say how to find it

3. Design a memoization structure

    $m \times n$ Array

4. Write an iterative algorithm

    Outer loop: increasing i (i.e., row-by-row starting from 1)
    Inner loop: increasing j (i.s., column-by-column starting from 1)

Longest Increasing Subsequence

0	1	2	3	4	5	6	7
5	-6	3	6	-5	2	8	10

Longest set of (not necessarily consecutive) elements that are increasing

5 is optimal for the array above

(indices $1,2,3,6,7$; elements $-6,3,6,8,10$)

For simplicity – assume all array elements are distinct.