Synthesis Of Dynamic Programming Algorithms

Yewen (Evan) Pu
Rastislav Bodik
Saurabh Srivastava

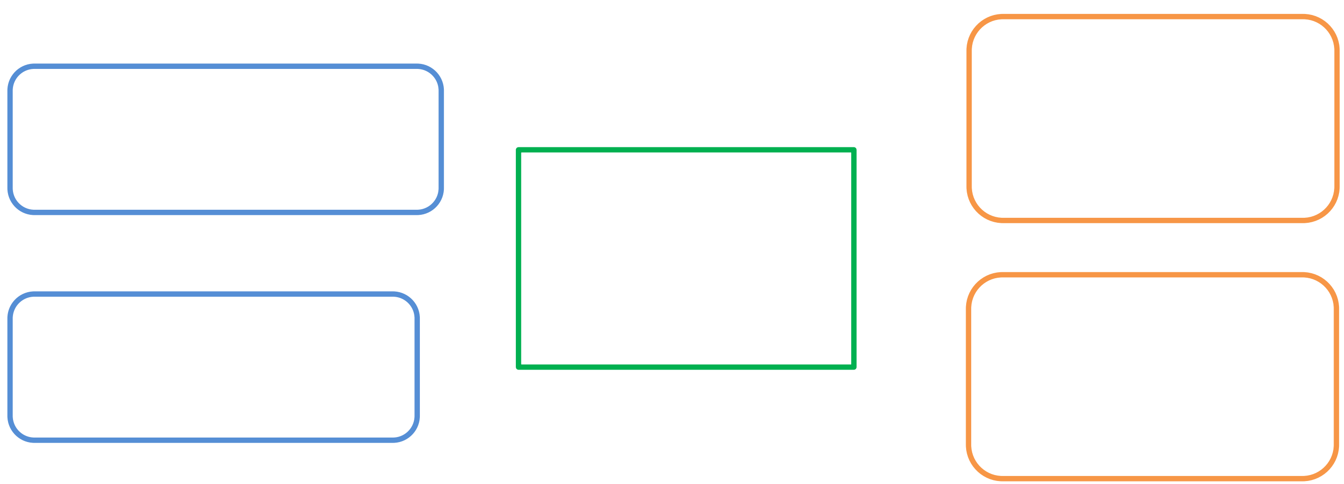
Synthesis of DP: Why?

- Dynamic Programming is ParLab pattern #10
- Dynamic Programming is prevalent:
    - AI: variable elimination, value iteration
    - Biology: Gene matching
    - Database: Query optimization
- Dynamic Programming is difficult
- Certain Dynamic Programming Algorithm can be parallelized

Synthesis of DP: Goal

Synthesizer for a subset of DP

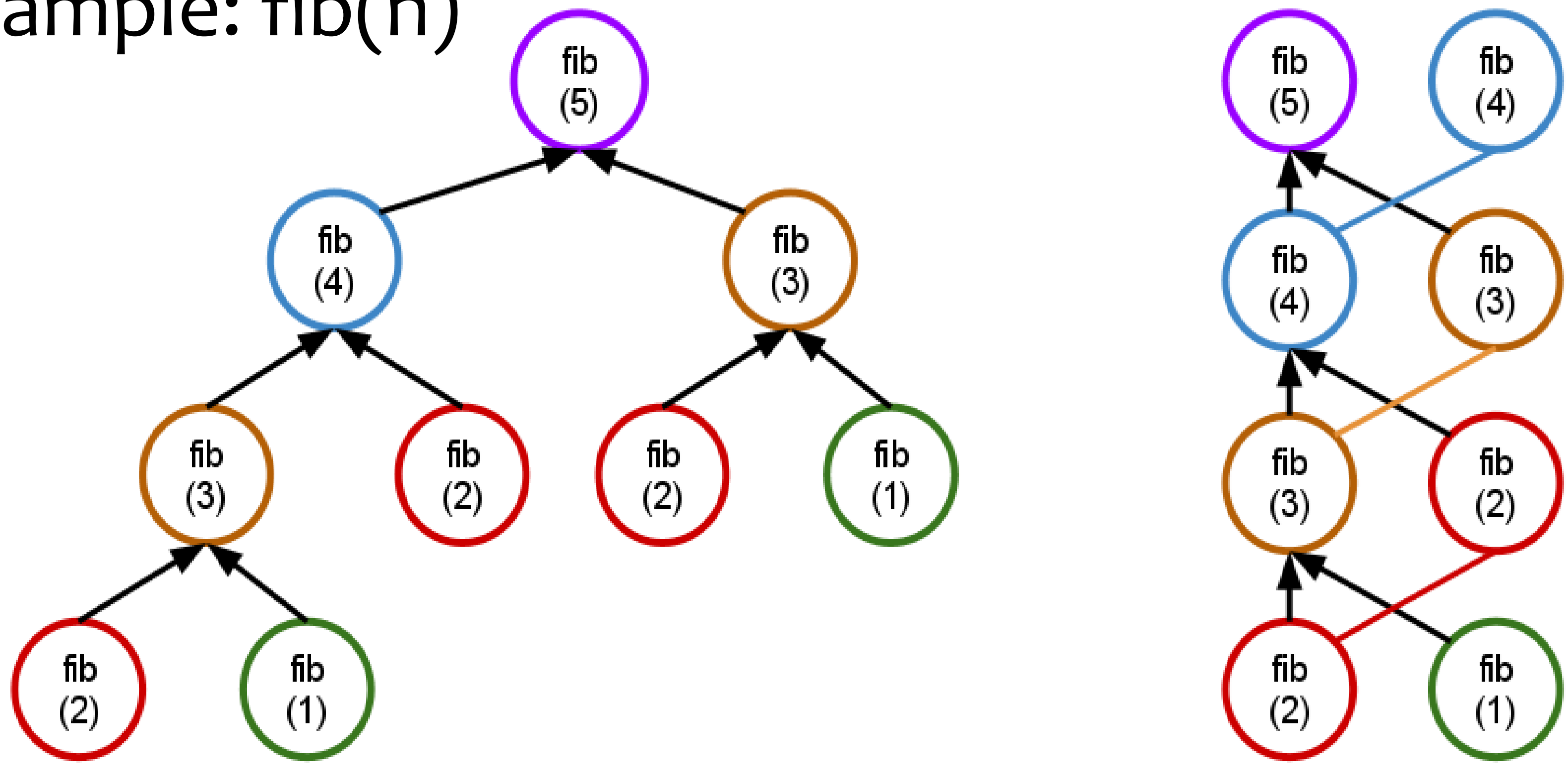
- First-order recurrence: Captures O(n) DP
- A domain-specific parameterizable compiler
- Input: Specifications, Output: Algorithms
- Building block for harder DP algorithms



Dynamic Programming

Speed up search algorithm that is exponential run-time by combining common sub-problems

Example: fib(n)



Challenges in DP algorithm design

Invent sub-problems: *Decompose original problem*

Sub-problems may not be explicitly stated in the original problem.

We may need to invent different sub-problems.

Recurrence: *Solve problem from its sub-problems*

Formulate recurrences over the new sub-problems that puts them back together

Maximal Segment Sum

Given an array of positive and negative integers, find the greatest sum of a consecutive substring.

Maximal Segment Sum

Given an array of positive and negative integers, find the greatest sum of a consecutive substring.

```
naive_mss(array):
  best = 0
    for i from 0 to n-1:
      for j from i to n-1:
        v = sum(array[i,j])
        best = max(best, v)
  return best
```

```
linear_mss(array):
  best_suffix = array()
  best_sofar =  array()
  best_suffix[0] = 0
  best_sofar[0]  = 0
  for i from 1 to n:
    best_suffix[i] =
       max(best_suffix[i-1]+array[i-1],0)
    best_sofar[i] =
       max(best_suffix[i-1], best_sofar)
  return best_sofar[n]
```

Maximal Segment Sum

Given an array of positive and negative integers, find the greatest sum of a consecutive substring.

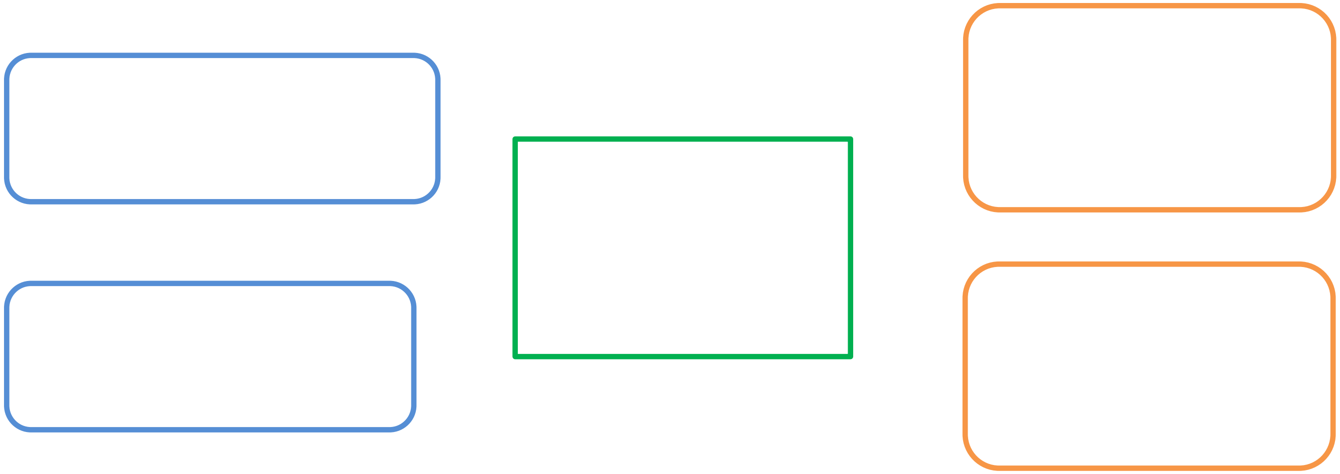
```
naive_mss(array):
  best = 0
    for i from 0 to n-1:
      for j from i to n-1:
        v = sum(array[i,j])
        best = max(best, v)
  return best
```

```
linear_mss(array):

  best_suffix[0] = 0
  best_sofar[0]  = 0
  for i from 1 to n:

  return best_sofar[n]
```

Synthesizer Work-flow

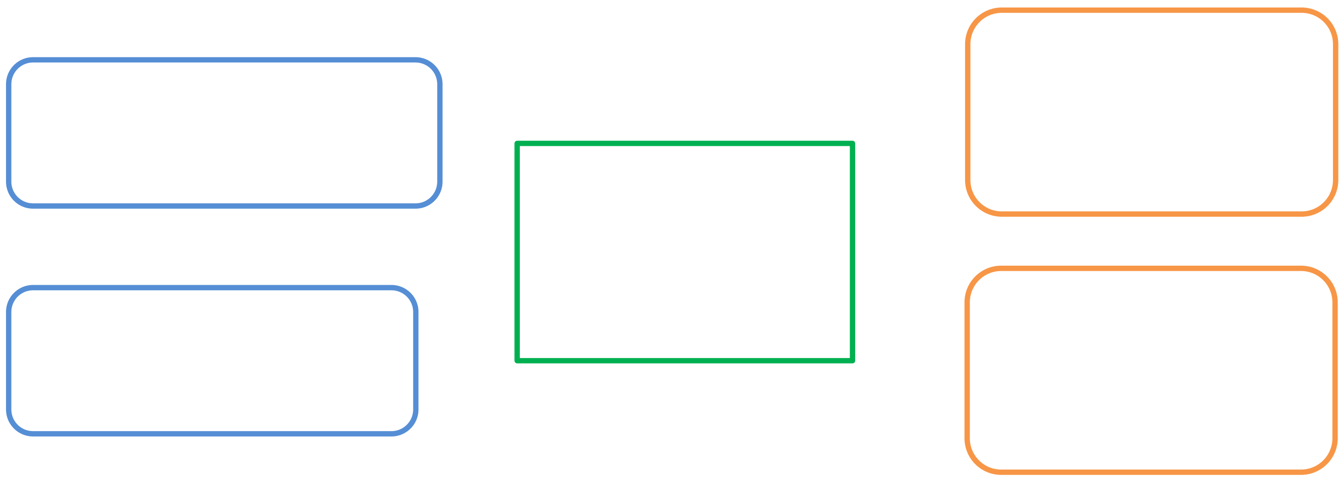


Maximal Independent Sum (MIS)

Input: Array of positive integers

Output: Maximal sum of a non-consecutive selections of its elements.

Synthesizer Work-flow

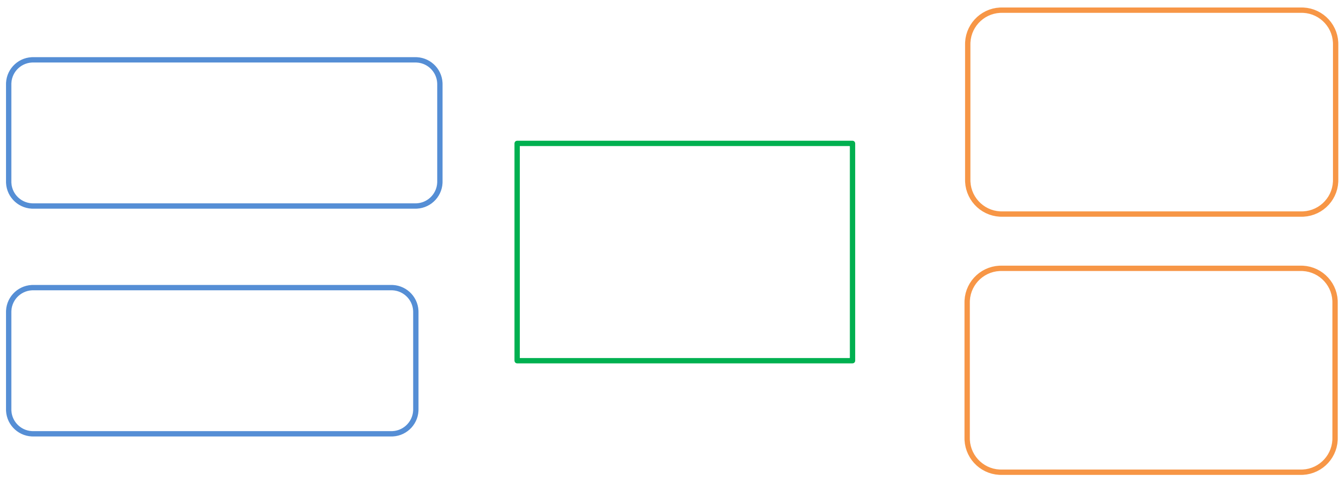


Exponential Specification

The user can define a specification as an exponential algorithm for MIS, it is:

```
mis(A):
   best = 0
   forall selections:
      if legal(selection):
         best = max(best, value(A[selection]))
   return best
```

Synthesizer Work-flow

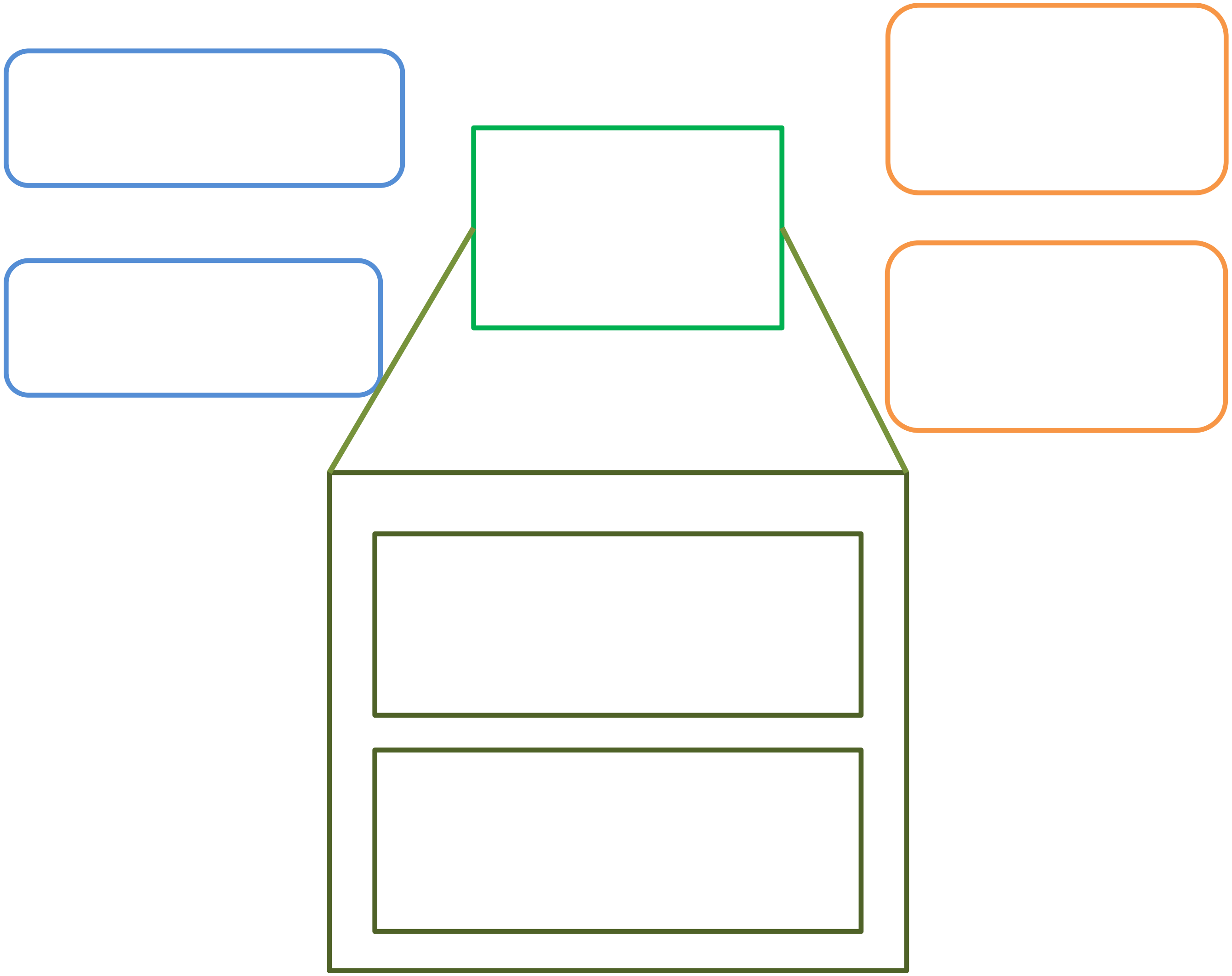


Parameters

- Comes from the user
- For simple problems, extract from specification

For MIS:

Synthesizer Work-flow



Skeleton: Shape of F.O.R

```
linear_mis(A):
  tmp1 = array()
  tmp2 = array()
  tmp1[0] = initialize1()
  tmp2[0] = initialize2()
  for i from 1 to n:
   tmp1[i] = update1(tmp1[i-1],tmp2[i-1],A[i-1])
   tmp2[i] = update2(tmp1[i-1],tmp2[i-1],A[i-1])
  return term(tmp1[n],tmp2[n])
```

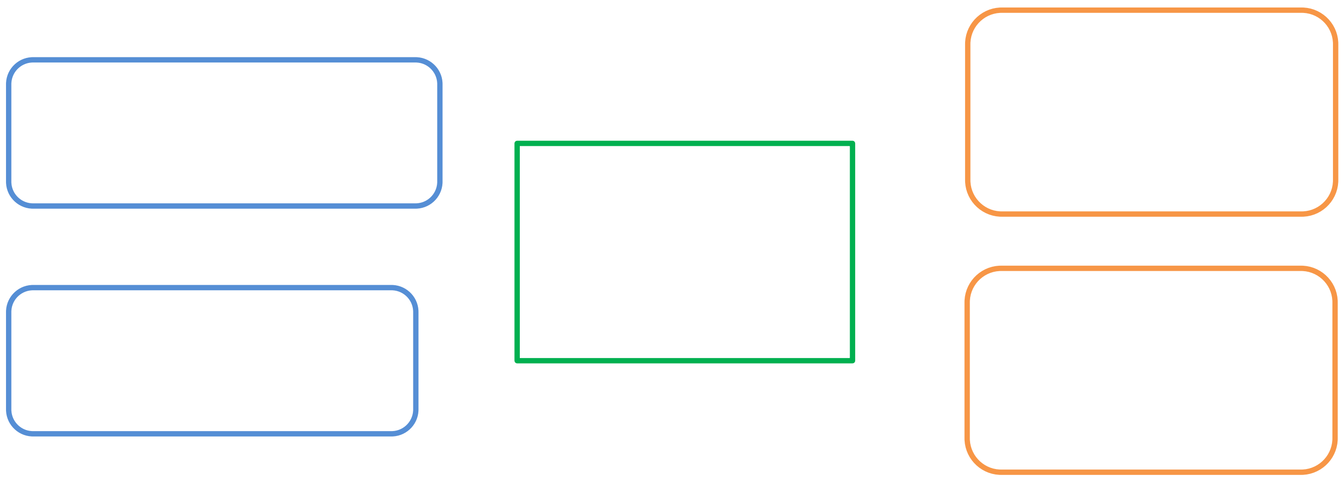
Update: Propagating Forward

- Constructed from user's parameters
- Enumerates all compositions of operations
- Selects the correct program

```
update1(x,y,z) =
    choose_from(
      {0,x,y,z,…x+y,…,max(x,y)+z,…})
```

- For m sub-problems and n operators:
  Total of $O((m^m n^m)^m)$ possible programs

Synthesizer Work-flow



MIS: The solution algorithm

```
linear_mis(A):
   tmp1 = array()
   tmp2 = array()
   tmp1[0] = 0
   tmp2[0] = 0
   for i from 1 to n:
    tmp1[i] = tmp2[i-1]+A[i-1]
    tmp2[i] = max(tmp1[i-1],tmp2[i-1])
   return max(tmp1[n],tmp2[n])
```

A Guy walks into an interview…

The Problem:

Given an array of integers: $A = [a_1, a_2, \ldots, a_n]$,
return: $B = [b_1, b_2, \ldots , b_n]$
such that: $b_i = a_1 + \ldots + a_n - a_i$

Do it in O(n) and cannot use subtraction?

Composition of Skeletons

```
puzzle(A):
  B = skeleton1(A)
  C = skeleton2(A,B)
  D = skeleton3(A,B,C)
  return D
```

Solution

```
puzzle(A):
  B = skeleton1(A)
  C = skeleton2(A,B)
  D = skeleton3(A,B,C)
  return D
```

```
skeleton1(A):
  tmp1 = array()
  tmp1[0] = 0
  for i from 1 to n-1:
    tmp1[i] = tmp[i-1]+A[n-1]
  return tmp1
```

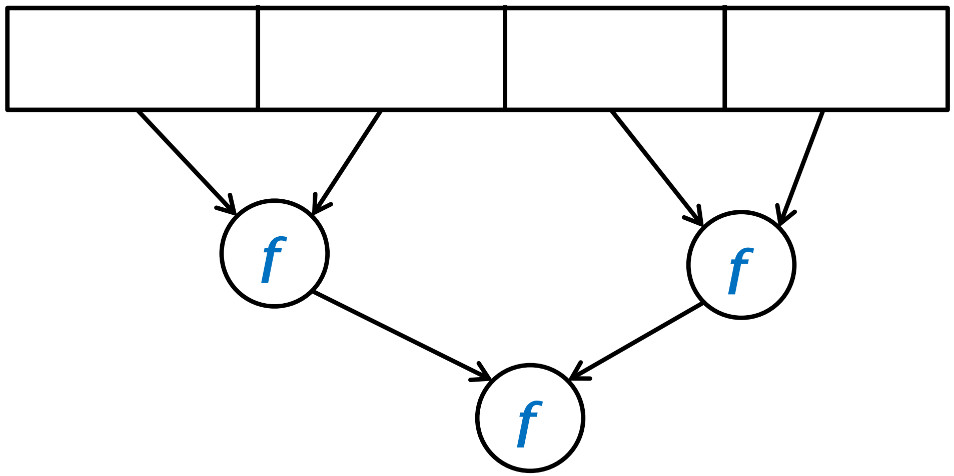
```
skeleton2(A,B):
  tmp2 = array()
  tmp2[n-1] = 0
  for i from 1 to n-1:
    tmp2[n-i-1]
          = tmp2[n-i]+A[n-i]
```

```
skeleton3(A,B,C):
  tmp3 = array()
  for i from 0 to n-1:
    tmp3[i] = B[i] + C[i]
  return tmp3
```

Synthesis of Parallelization: Prefix Sum

Compute a F.O.R. out of order

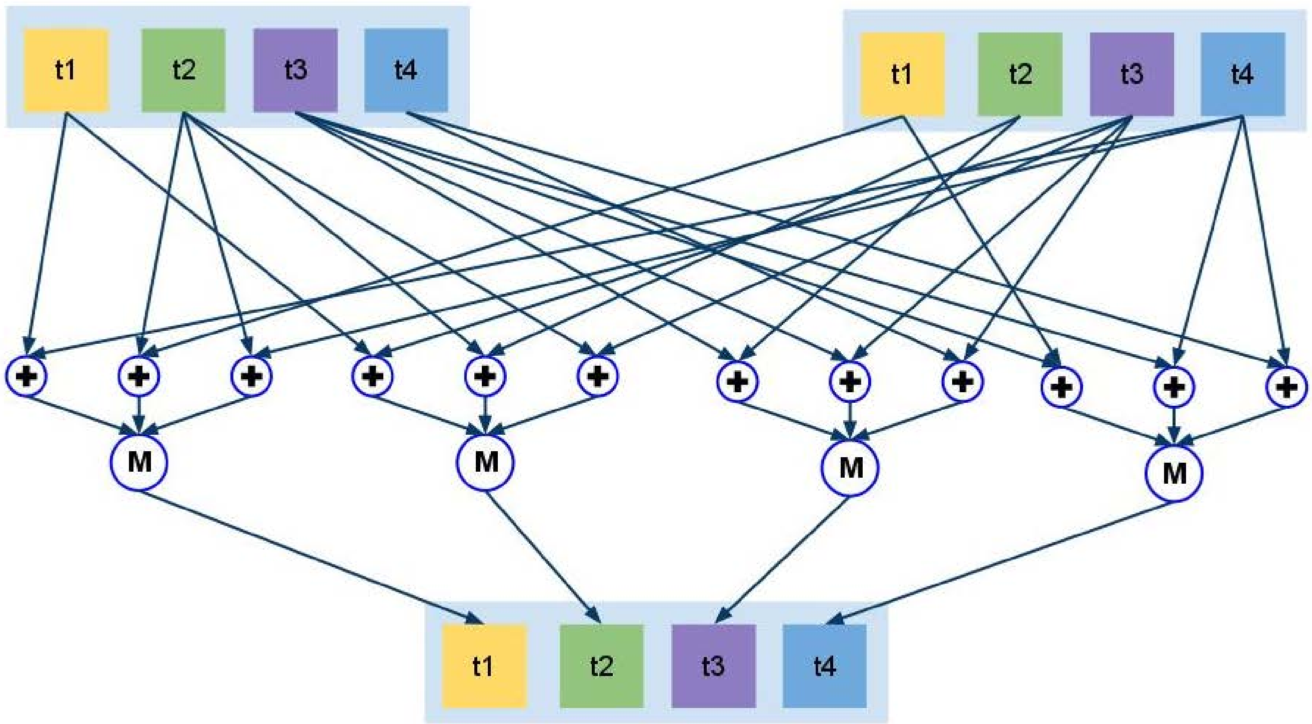
Goal: synthesize an *associative* function that allows solving the problem in parallel, as a prefix sum.



The Approach: Exactly the same. The Skeleton is now a tree, the update needs to be associative.

```
result = update(update(A[0],A[1]),
                update(A[2],A[3]))
```

Synthesized associative operator for MIS



This operator requires invention of 4 sub-problems

Scalabilities of Synthesizer

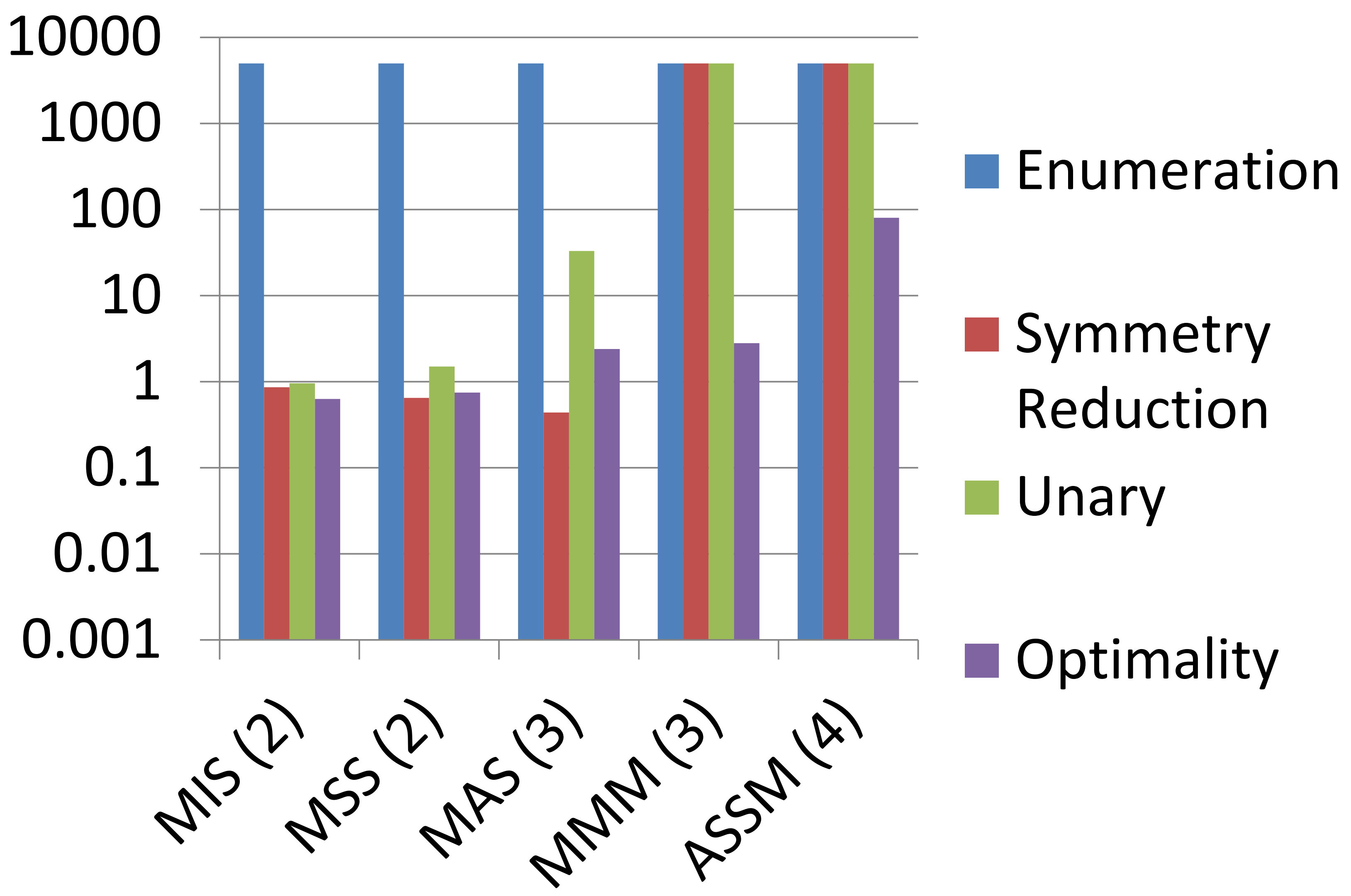
```
update1(x,y,z) =
    choose_from(
      {0,x,y,z,…x+y,…,max(x,y)+z,…})
```

- For m sub-problems and n operators:
  Total of $O((m^m n^m)^m)$ possible programs,
  many of them are redundant.

Reduce the search space by:

- Symmetry reduction of commutative binary operators
- Apply unary operators at the leaves
- Encode DP optimality structure

Scalabilities of Synthesizer



Comparison to Other Approaches

Suppose the user wants to write a DP algorithm…

Future Works

- Synthesis of Real World Problems
- Synthesis of more prefix sum (on these problems)
- Other DP Problems (that are not F.O.R)
- Further scalability tricks
- Complete the pipe (implementation on GPU)

The End: Questions?