WISE: Automated Test Generation for Worst-Case Complexity

Jacob Burnim
Sudeep Juvekar
Koushik Sen

Performance-Directed Testing

- Automated tested has focused on correctness bugs.
- **Goal:** Apply to software performance.
  - Find performance bottlenecks.
  - Security: Algorithmic denial-of-service.
- Today: Computational complexity testing.
  - How slow is an operation in the worst case?
  - Does a function meet its algorithmic complexity spec?

Performance-Directed Testing

- Example: Performance bug in `Jar`
  - Reported by Sun on May 15, 2009
  - `update` method $O(N^2)$ instead of $O(N)$
  - $O(N)$ look-up on every file, rather than $O(1)$
  - wasted 75% of run-time building `rt.jar`

Goal of WISE

- **W**orst-case **I**nputs from **S**ymbolic **E**xecution

Input

Size: N

```
// insertion sort
for(i = 0 .. N-1)
for(j = i .. 1)
  if (A[j] < A[j-1])
    swap(A[j], A[j-1])
  else
    break
```

Goal of WISE

- **W**orst-case **I**nputs from **S**ymbolic **E**xecution

Input

Size: N

```
// insertion sort
for(i = 0 .. N-1)
for(j = i .. 1)
  if (A[j] < A[j-1])
    swap(A[j], A[j-1])
  else
    break
```

WISE

Output

1: 1

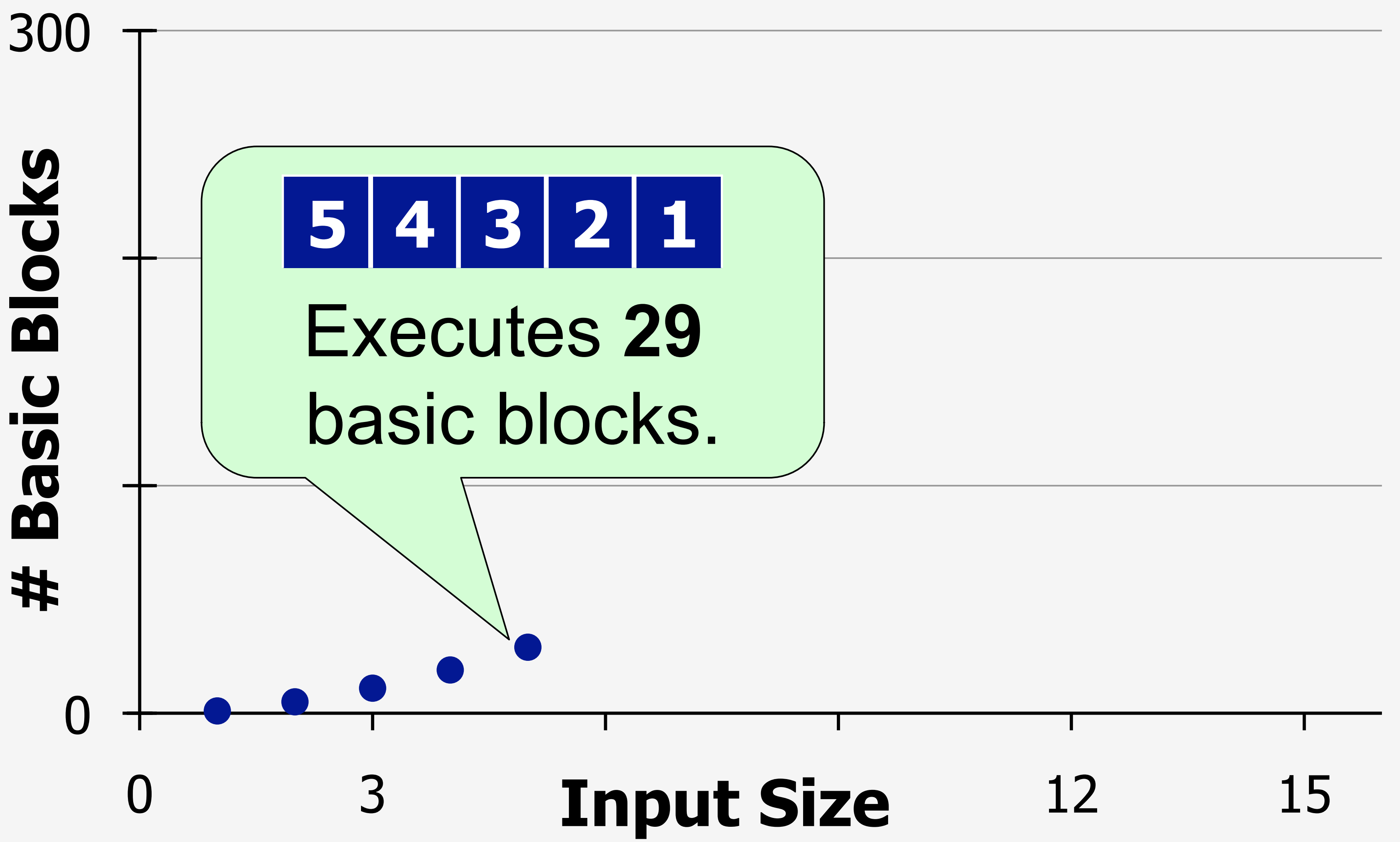
2: 2 1

3: 3 2 1

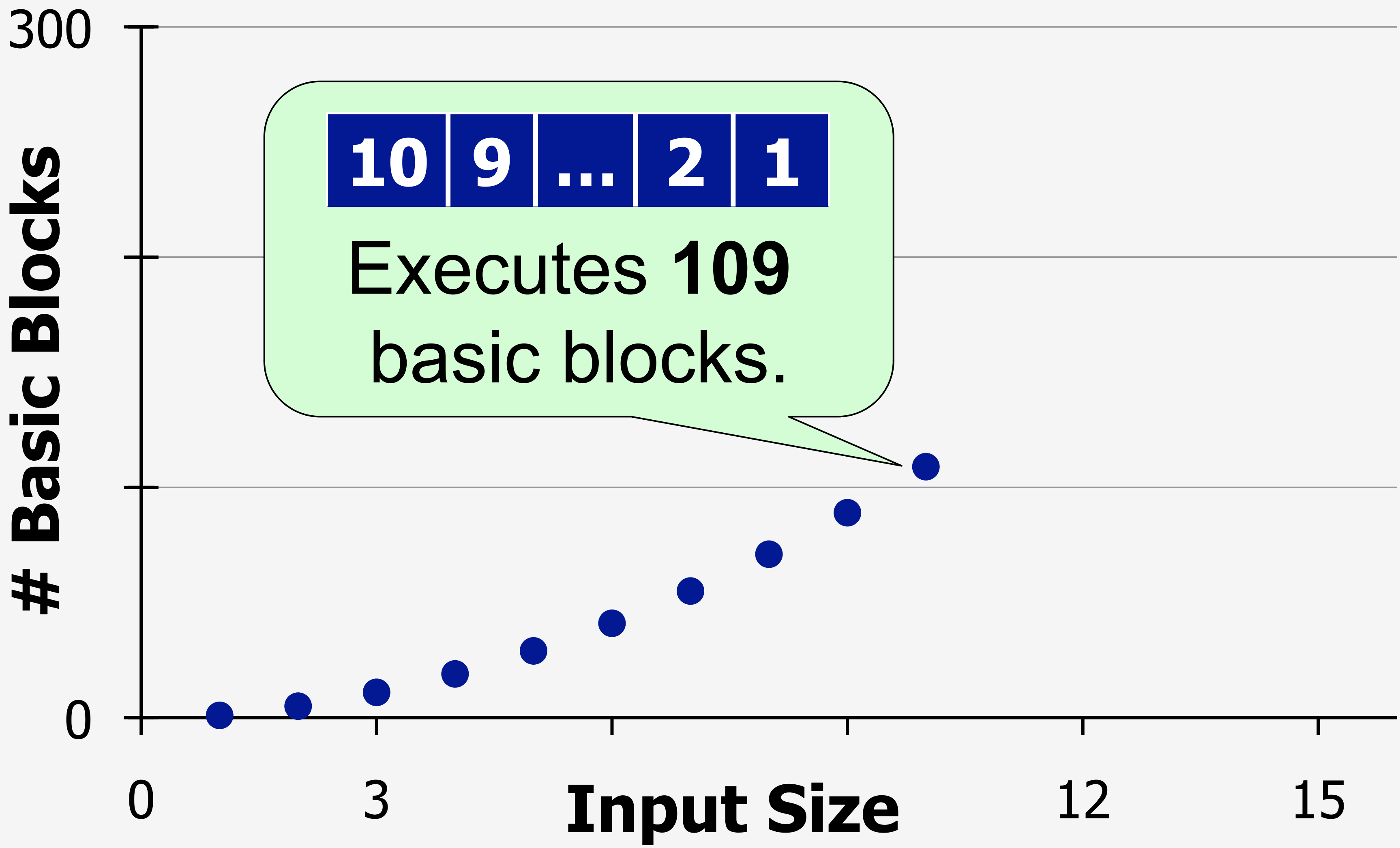
…

N: N … 2 1

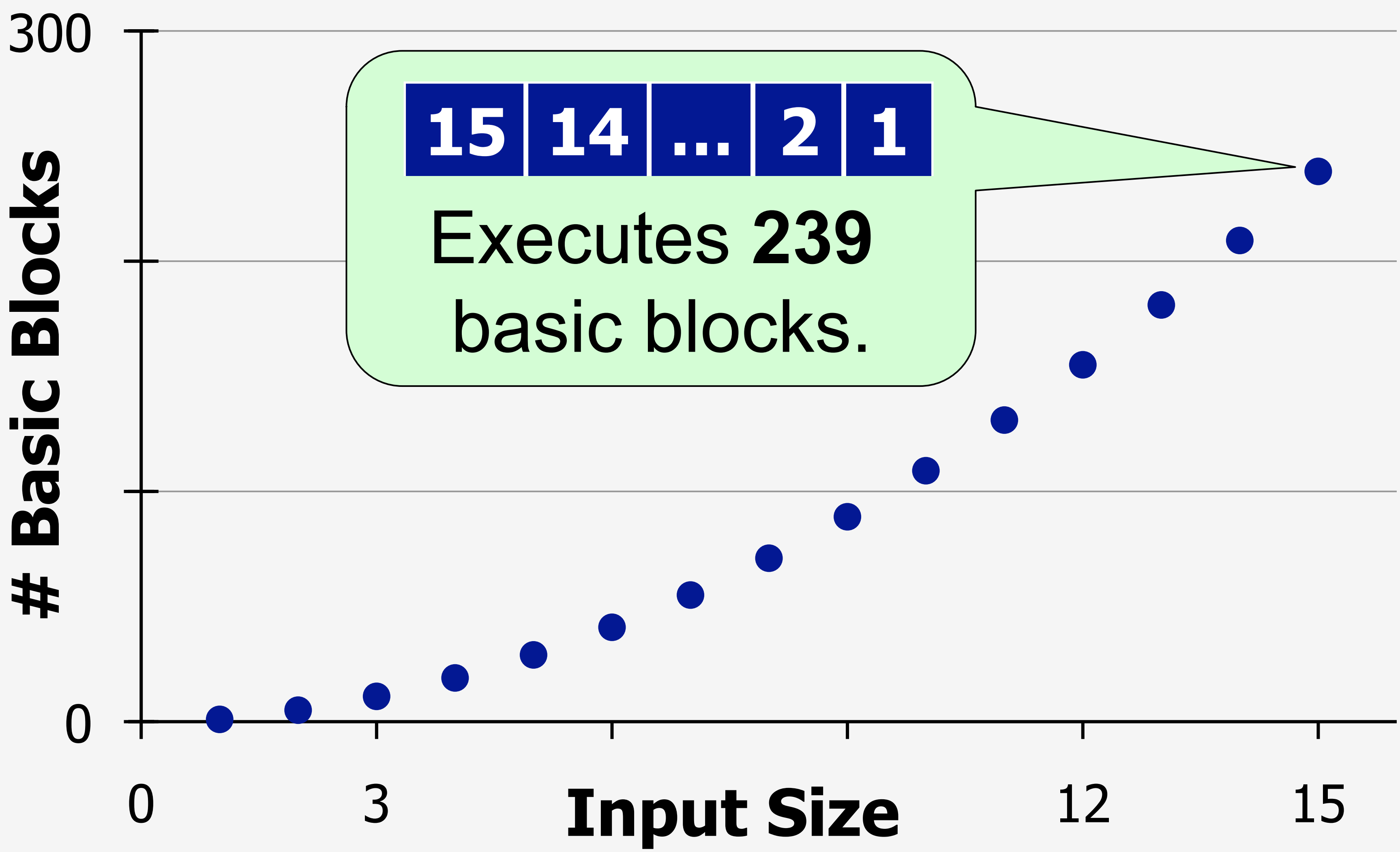
Worst-Cast Empirical Complexity



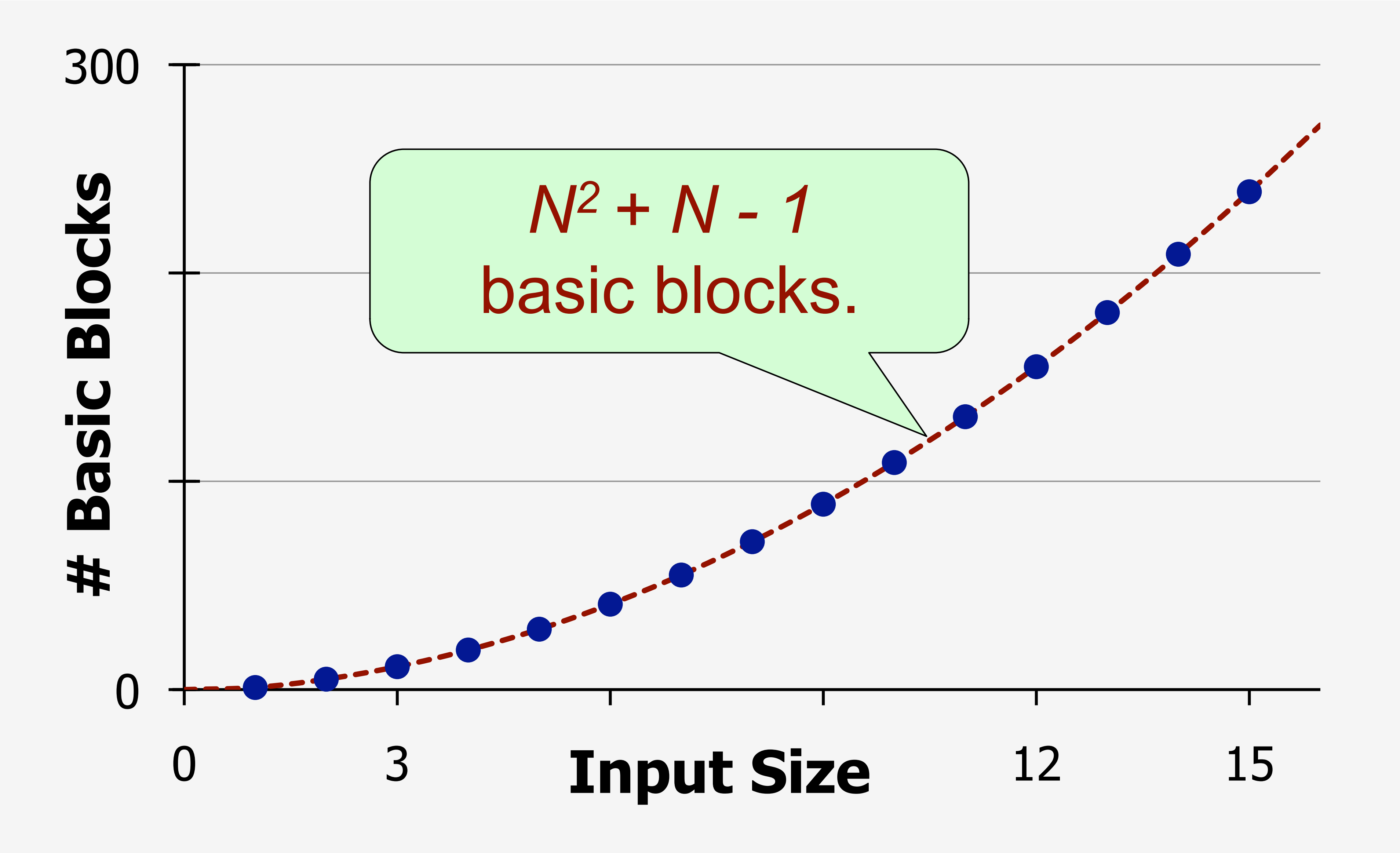
Worst-Cast Empirical Complexity



Worst-Cast Empirical Complexity



Worst-Cast Empirical Complexity



Overview of WISE

- Uses symbolic test generation to explore possible program executions.
  - Widely used in automated software testing. (DART, CUTE, SAGE, EXE, KLEE, JPF, …)

- **Key Idea:**
  Learn from executions on small inputs.
  - *In Quicksort, pivot should be smaller than all elements to which it's compared.*

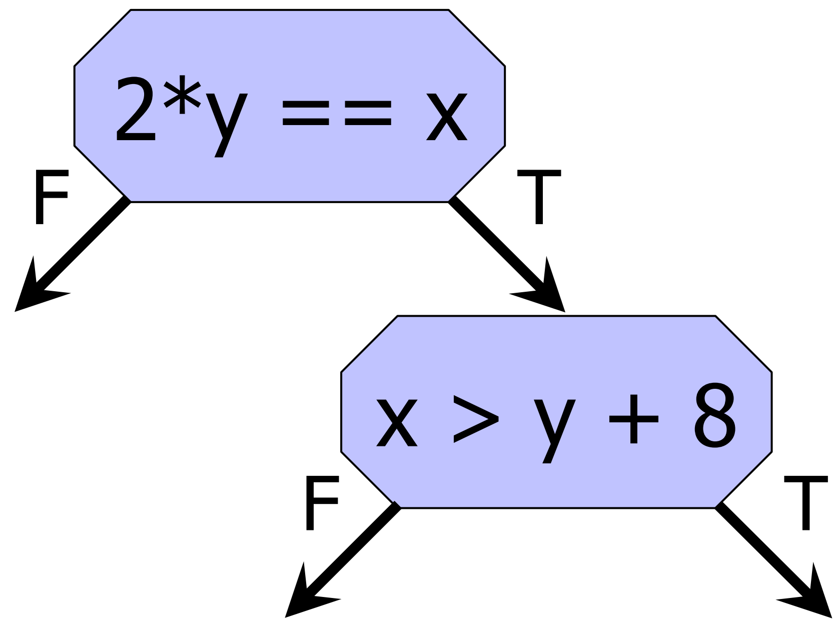
Outline

- Motivation + Goal of WISE
- **Background: Symbolic Test Generation**
- Naïve Algorithm for Finding Complexity
- WISE Algorithm
- Evaluation
- Conclusions + Future Work

Symbolic Test Generation

- **Goal:** A test input for every program path.

```
f(int x, int y)
{
  z = 2*x;
  if (z == x)
    if (x > y + 8)
      print("Hi")
}
```

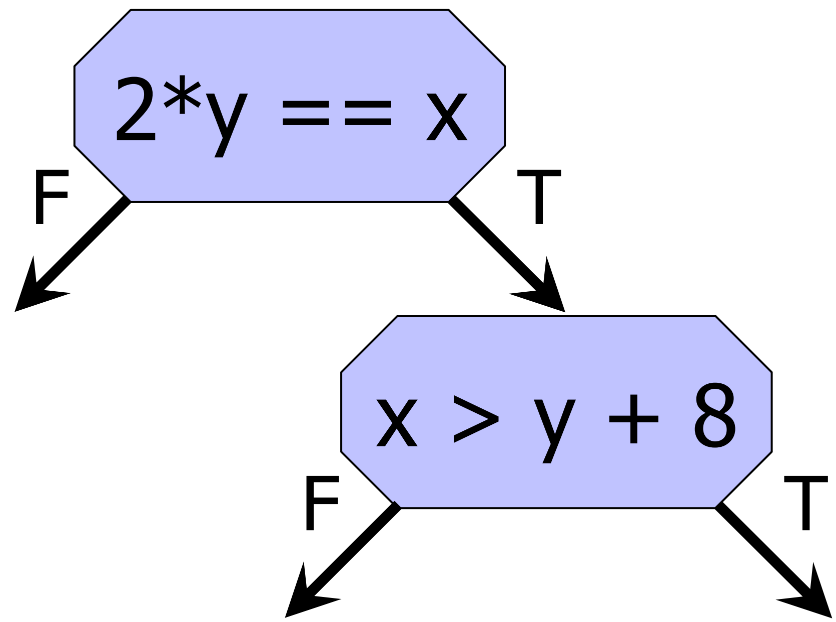


Computation Tree

Symbolic Test Generation

- Depth-first search of computation tree.

```
f(int x, int y)
{
  z = 2*x;
  if (z == x)
    if (x > y + 8)
      print("Hi")
}
```

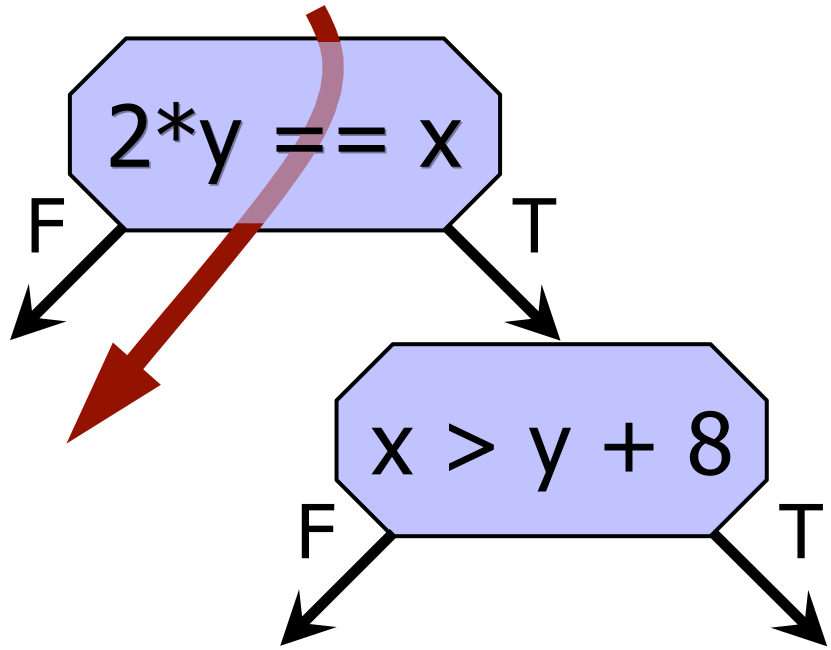


Computation Tree

Symbolic Test Generation

- Depth-first search of computation tree.

```
f(int x, int y)
{
  z = 2*x;
  if (z == x)
    if (x > y + 8)
      print("Hi")
}
```



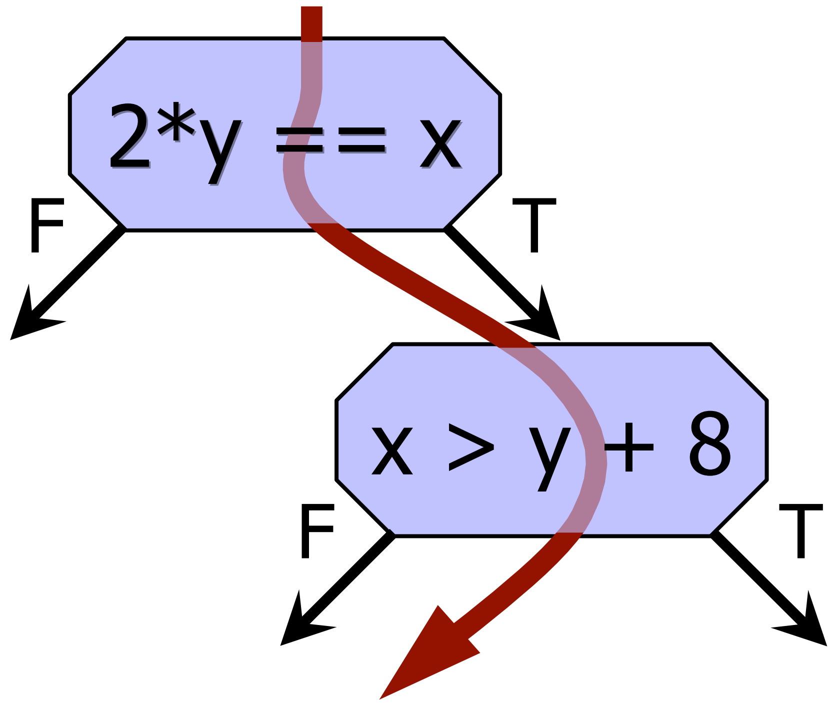
$\Phi(path)$: $2y \neq x$

Input: $x = 0,\ y = 1$

Symbolic Test Generation

- Depth-first search of computation tree.

```
f(int x, int y)
{
  z = 2*x;
  if (z == x)
    if (x > y + 8)
      print("Hi")
}
```



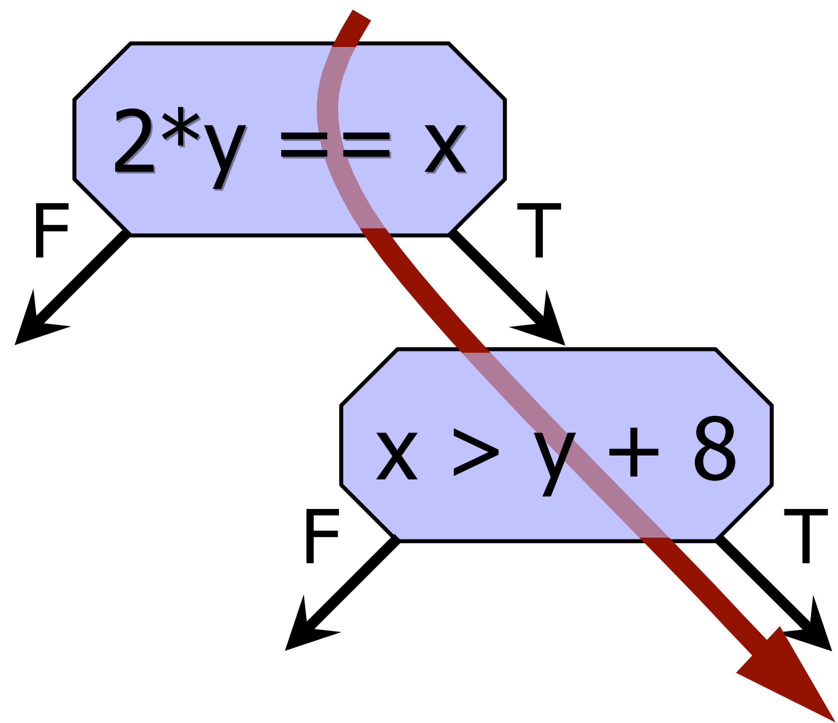
$\Phi(path)$: $2y = x \wedge x \leq y+8$

Input: $x = 1, y = 2$

Symbolic Test Generation

- Depth-first search of computation tree.

```
f(int x, int y)
{
  z = 2*x;
  if (z == x)
    if (x > y + 8)
      print("Hi")
}
```



Φ(*path*): $2y = x \wedge x > y+8$

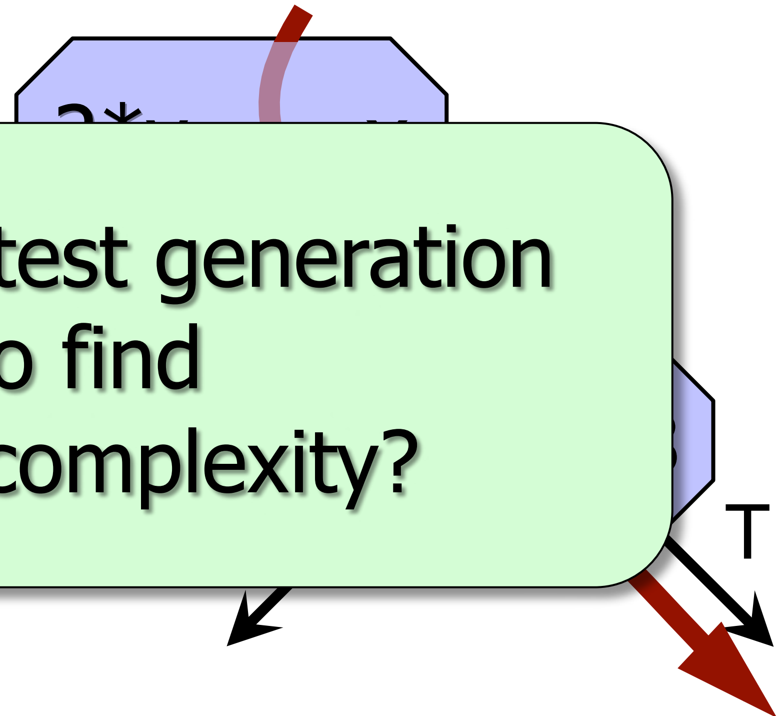
Input: $x = -10$, $y = -20$

Symbolic Test Generation

- Depth-first search of computation tree.

```
f(in
{
  z
  if
    if (x > y + 8)
      print("Hi")
}
```

How can symbolic test generation be used to find computational complexity?



$\Phi(path)$: $2y = x \wedge x > y+8$

Input: x = -10, y = -20

Outline

- Motivation + Goal of WISE
- Background: Symbolic Test Generation
- **Naïve Algorithm for Finding Complexity**
- WISE Algorithm
- Evaluation
- Conclusions + Future Work

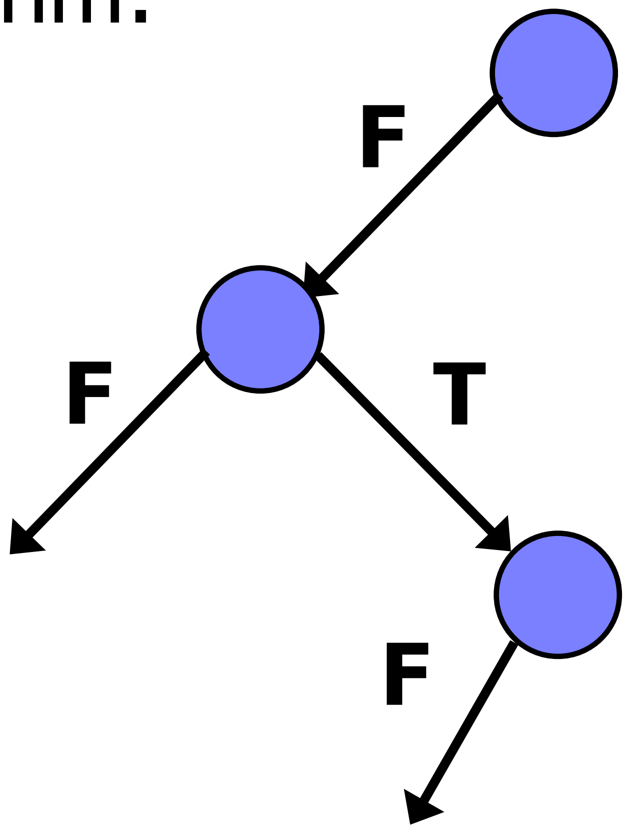
Symbolic Execution for Complexity

- Naïve Algorithm:
  - Generate every execution on N inputs.
  - Return input for longest execution.

Symbolic Execution for Complexity

- Naïve Algorithm:

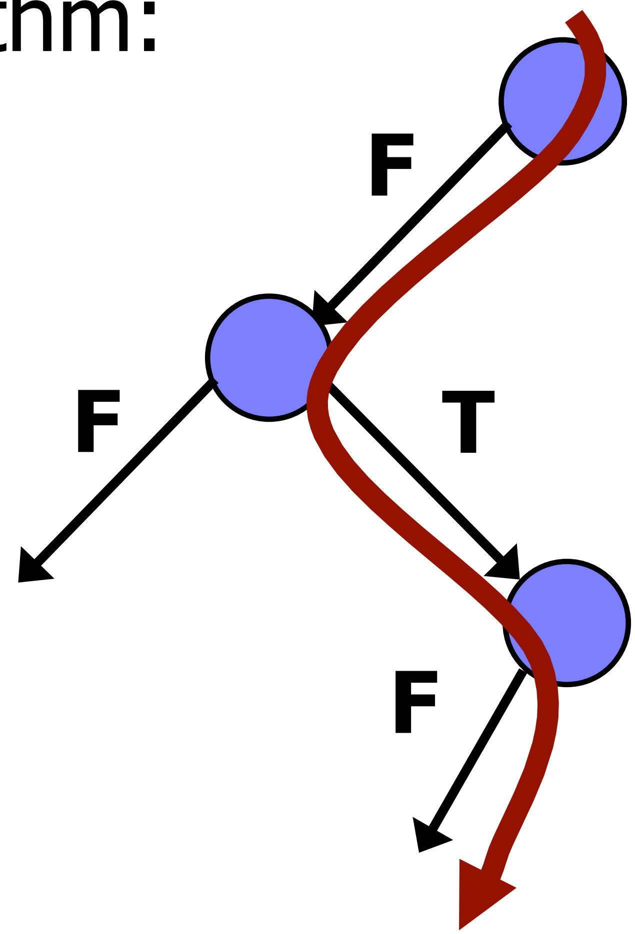
N=2:



Symbolic Execution for Complexity

- Naïve Algorithm:

N=2:

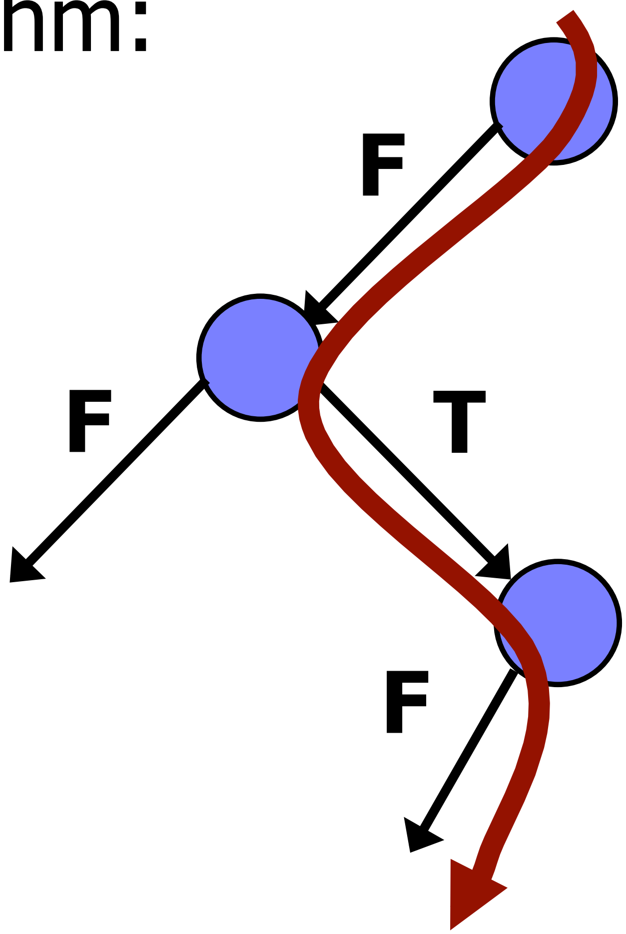


Longest Execution (4 basic blocks)

Symbolic Execution for Complexity

- Naïve Algorithm:

N=2:

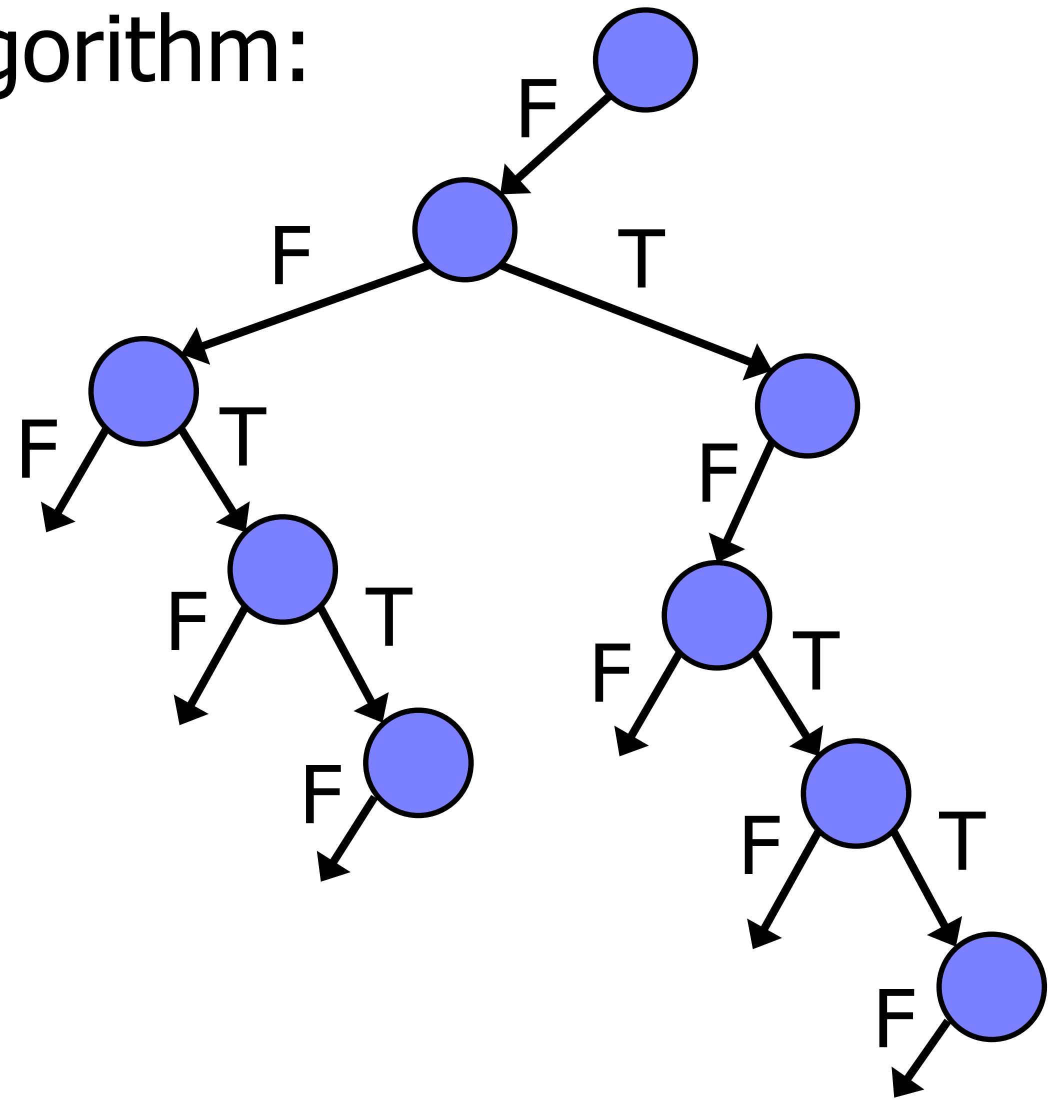


Worst-case Input: **2** **1**

Symbolic Execution for Complexity

- Naïve Algorithm:

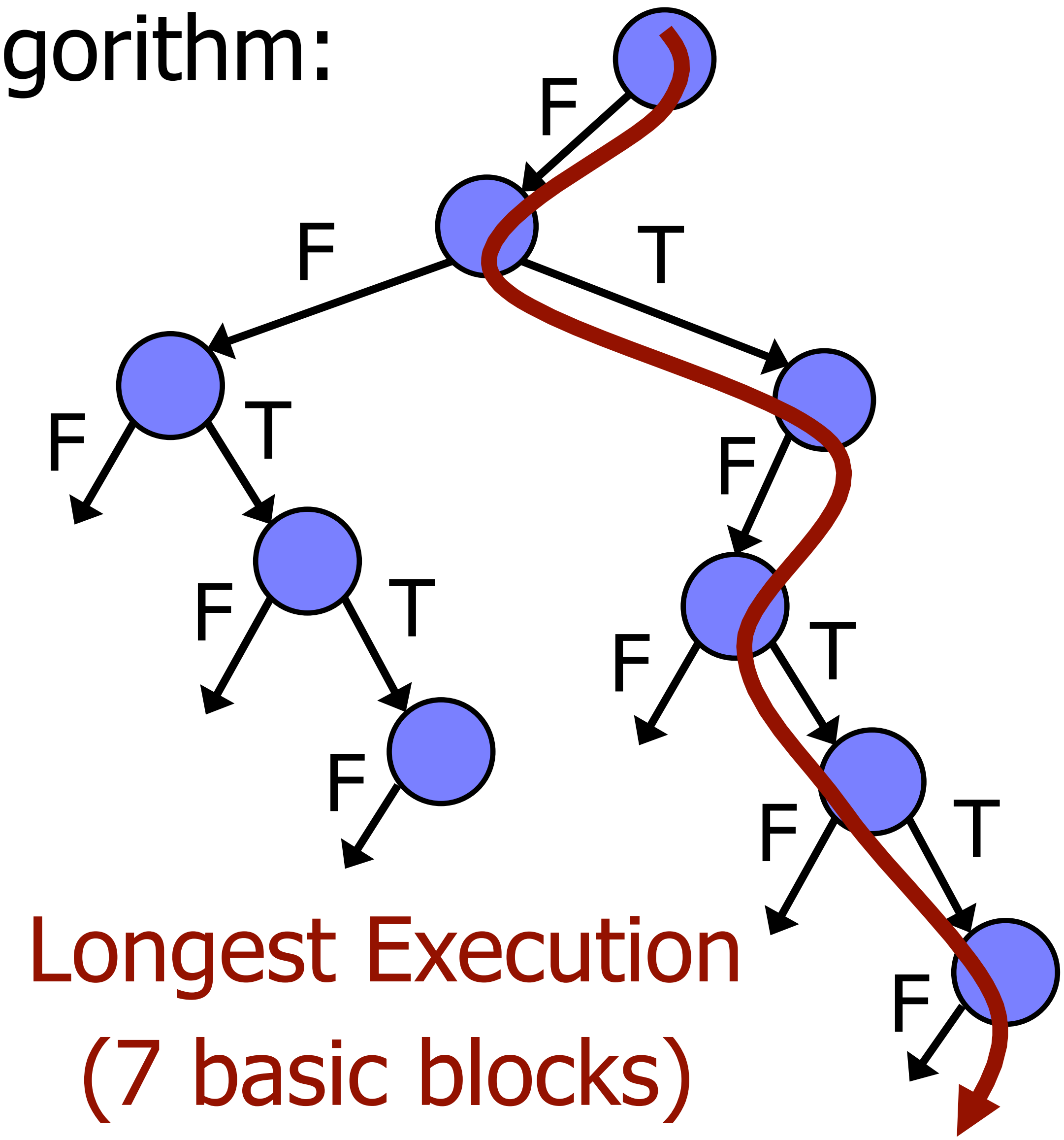
N=3:



Symbolic Execution for Complexity

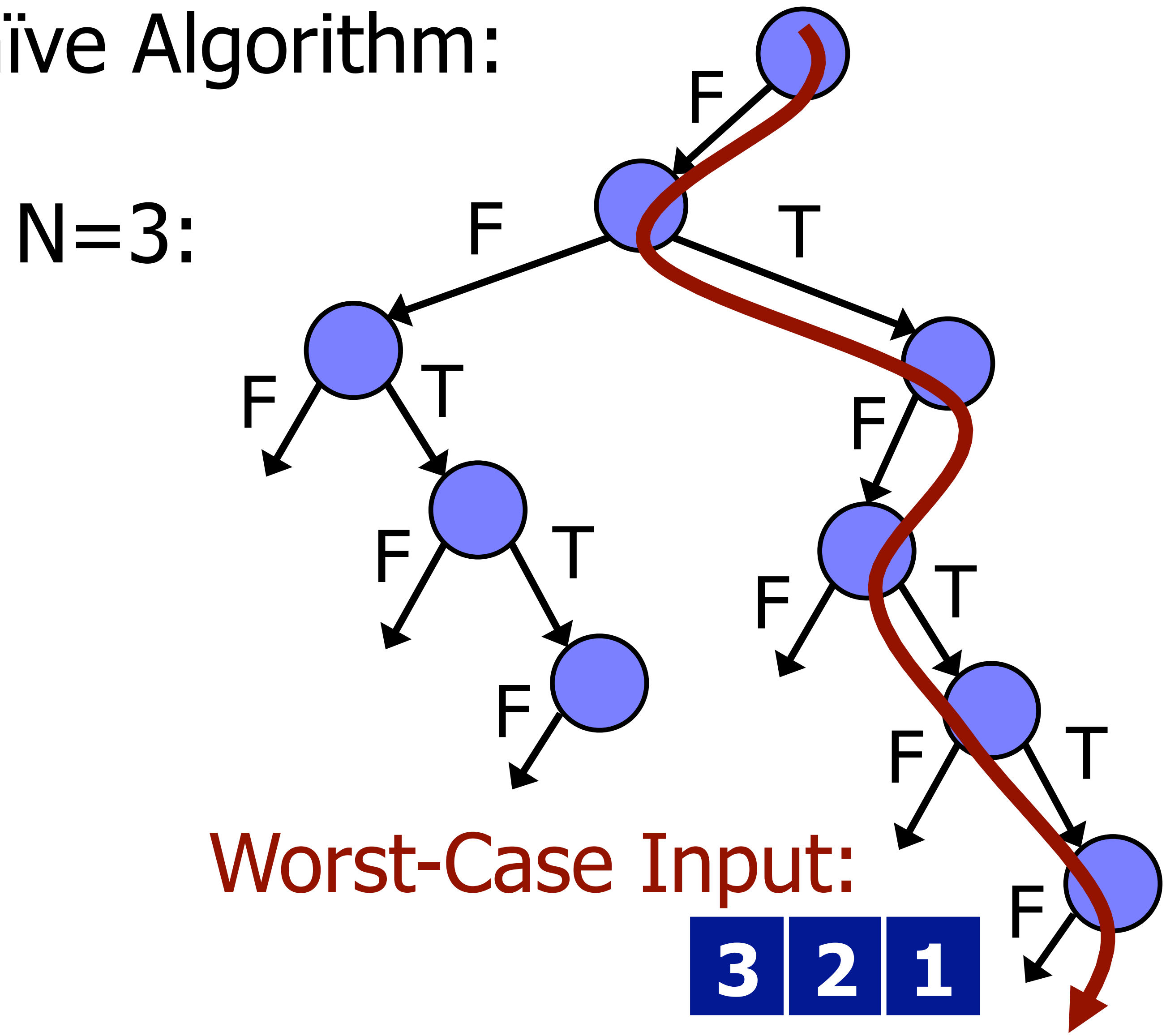
- Naïve Algorithm:

N=3:



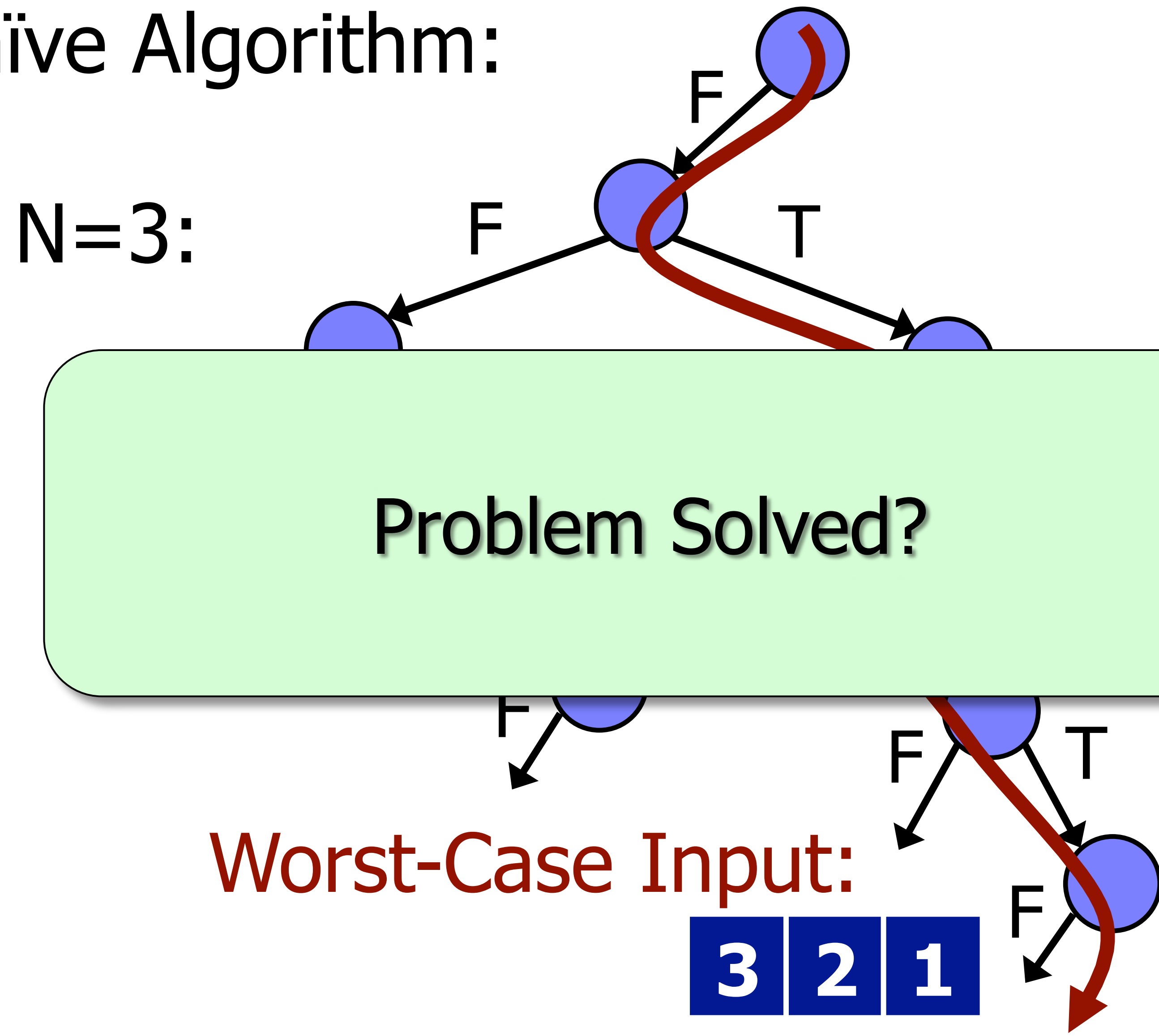
Symbolic Execution for Complexity

- Naïve Algorithm:



Symbolic Execution for Complexity

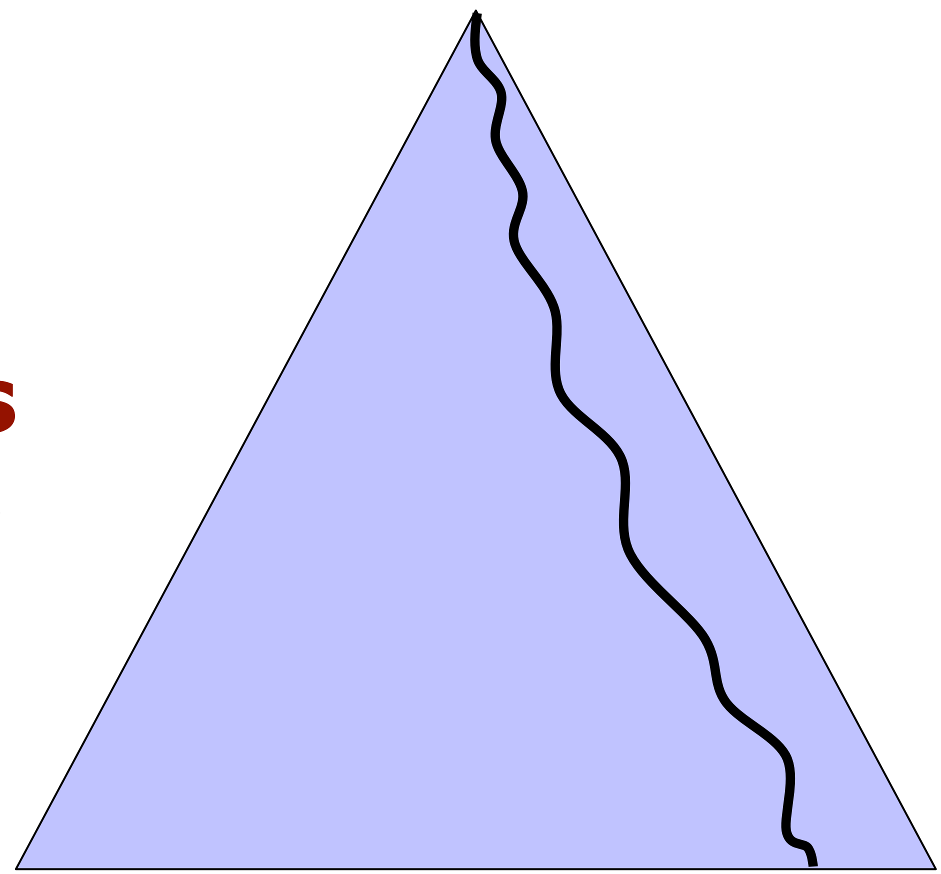
- Naïve Algorithm:



Path Space Explosion

- Naïve algorithm does not scale.

N=15:

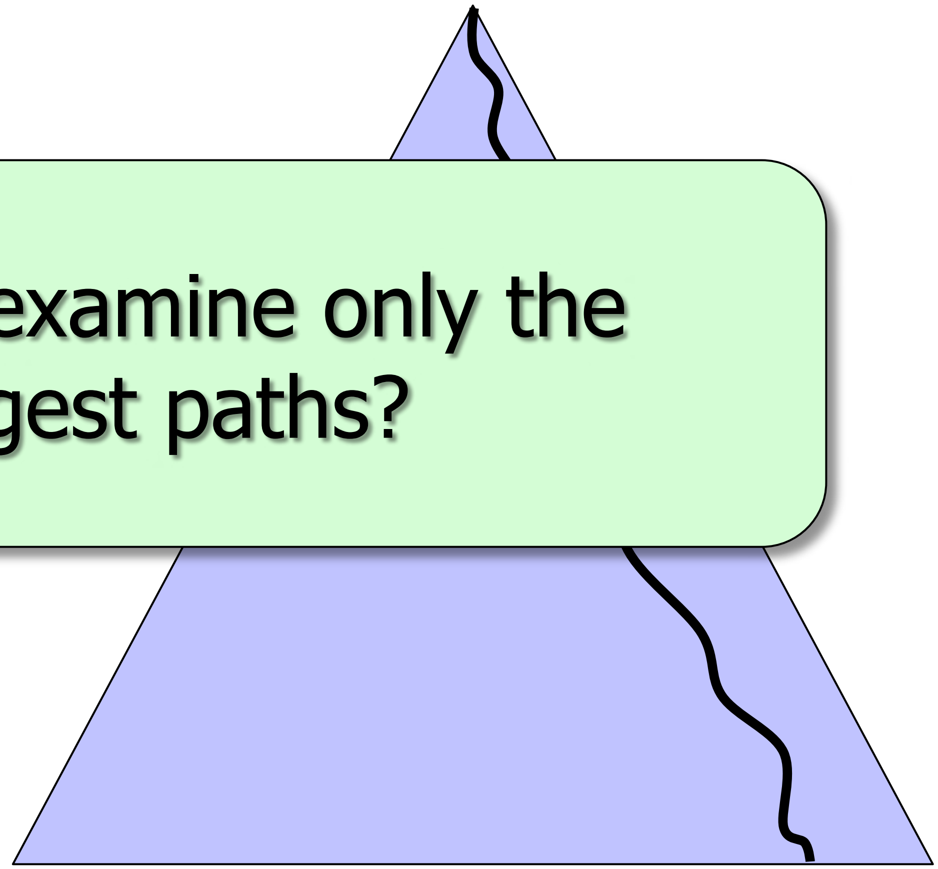


- **1.6×10^{25} paths**
- Longest has only 121 basic blocks

Path Space Explosion

- Naïve algorithm does not scale.

N=15:



Can we examine only the longest paths?

- **1.6**
- Long
  121 basic blocks

Outline

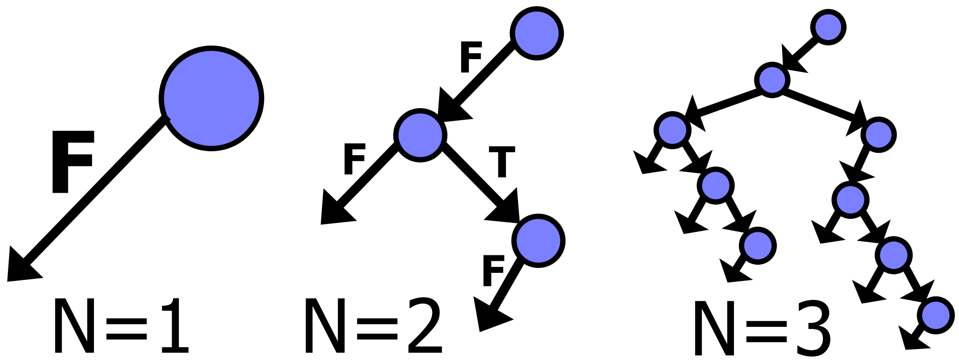
- Motivation + Goal of WISE
- Background: Symbolic Test Generation
- Naïve Algorithm for Finding Complexity
- **WISE Algorithm**
- Evaluation
- Conclusions + Future Work

Overview of WISE

- **Step 1**: From executions on small inputs, learn oracle for longest paths.

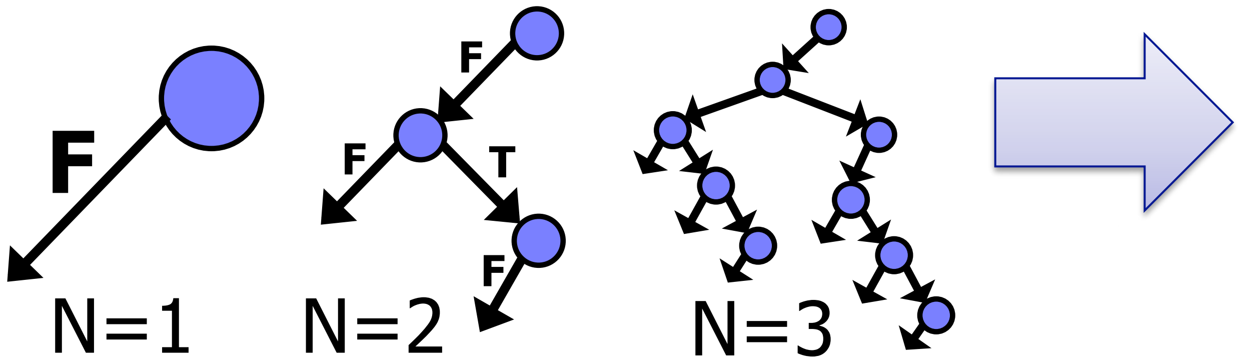
Overview of WISE

- **Step 1**: From executions on small inputs, learn oracle for longest paths.



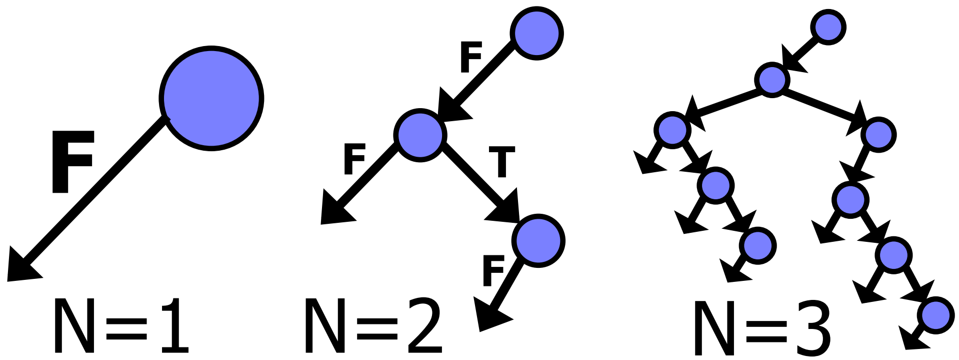
Overview of WISE

- **Step 1**: From executions on small inputs, learn oracle for longest paths.



Overview of WISE

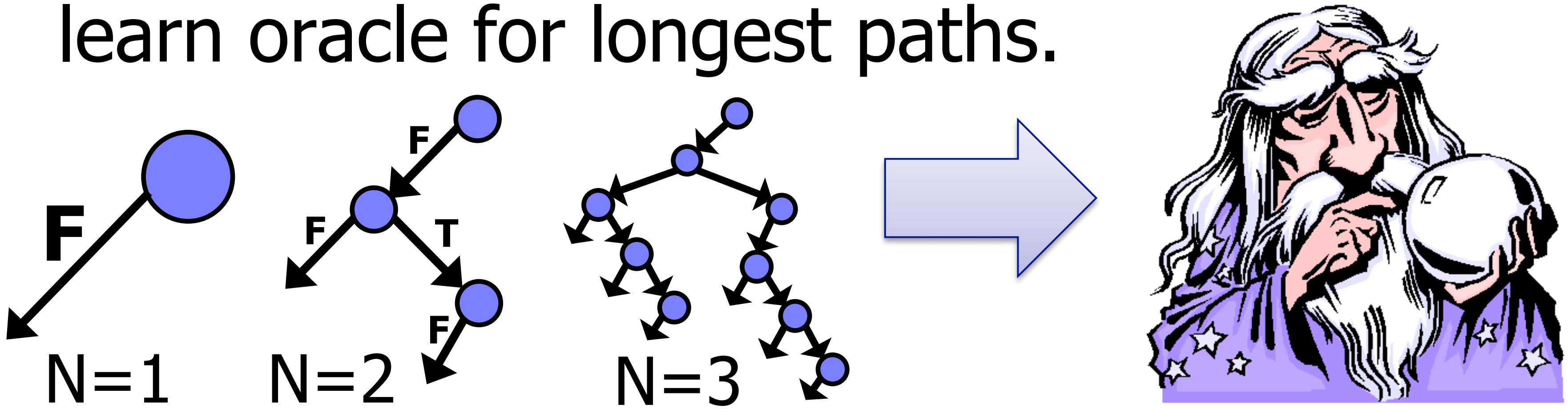
- **Step 1**: From executions on small inputs, learn oracle for longest paths.



- **Step 2**: For large inputs, only examine paths generated by oracle.

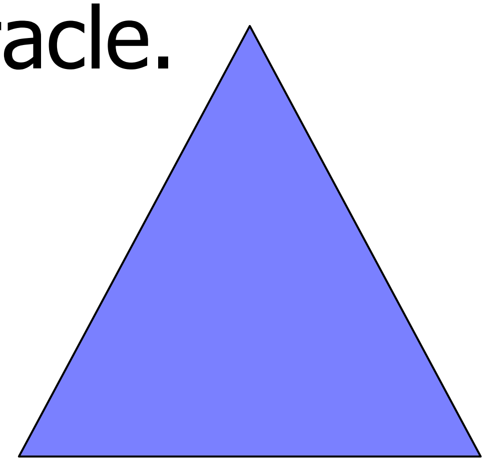
Overview of WISE

- **Step 1**: From executions on small inputs, learn oracle for longest paths.



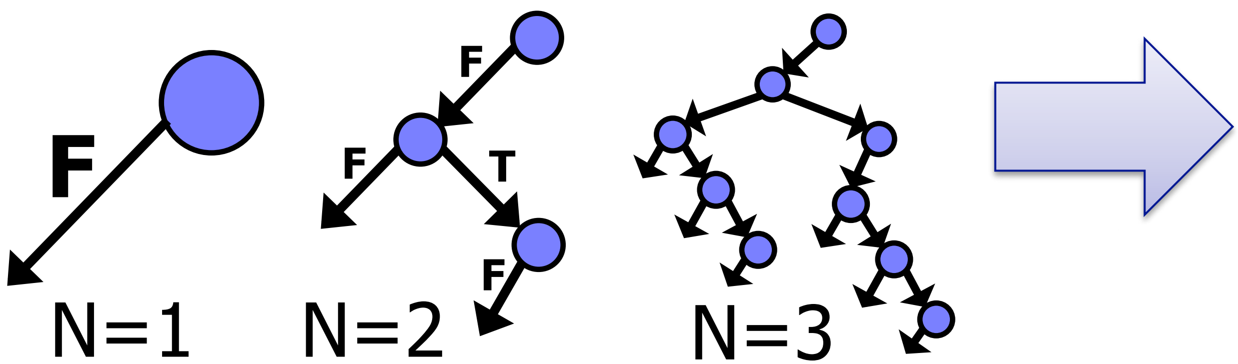
- **Step 2**: For large inputs, only examine paths generated by oracle.

N=15

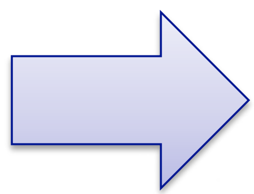


Overview of WISE

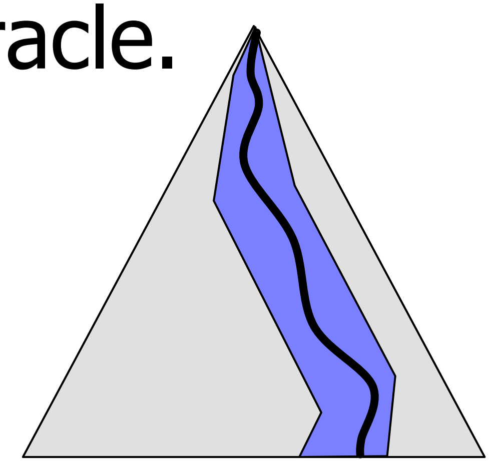
- **Step 1**: From executions on small inputs, learn oracle for longest paths.



- **Step 2**: For large inputs, only examine paths generated by oracle.

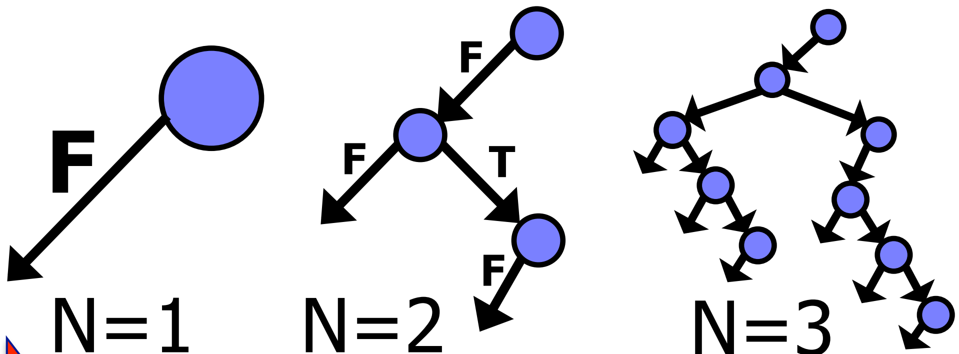


N=15



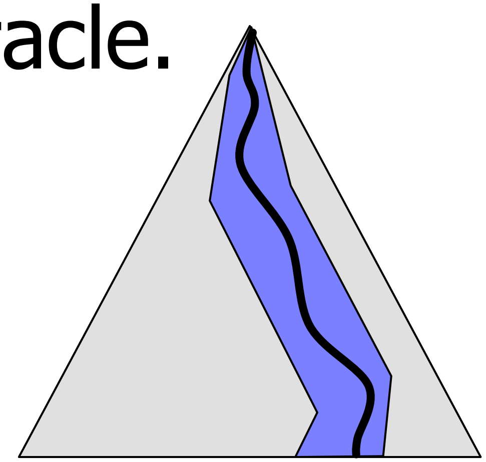
Overview of WISE

- **Step 1**: From executions on small inputs, learn oracle for longest paths.



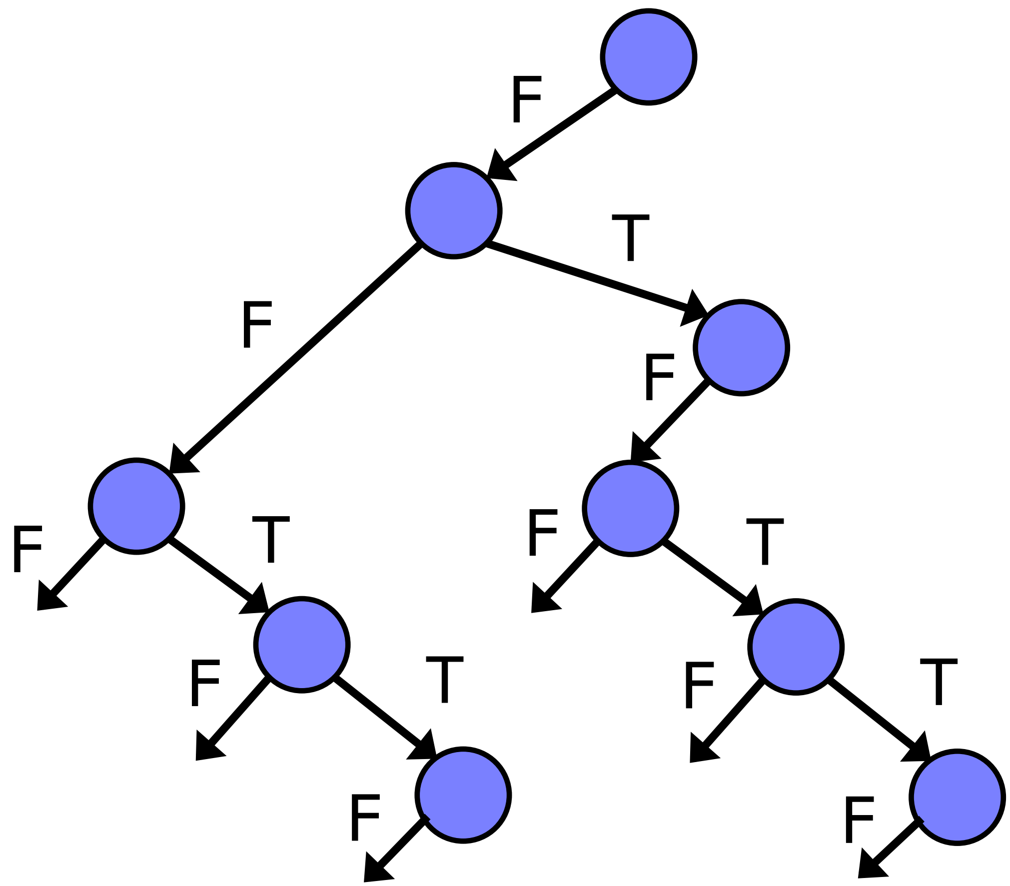
- **Step 2**: For large inputs, only examine paths generated by oracle.

N=15



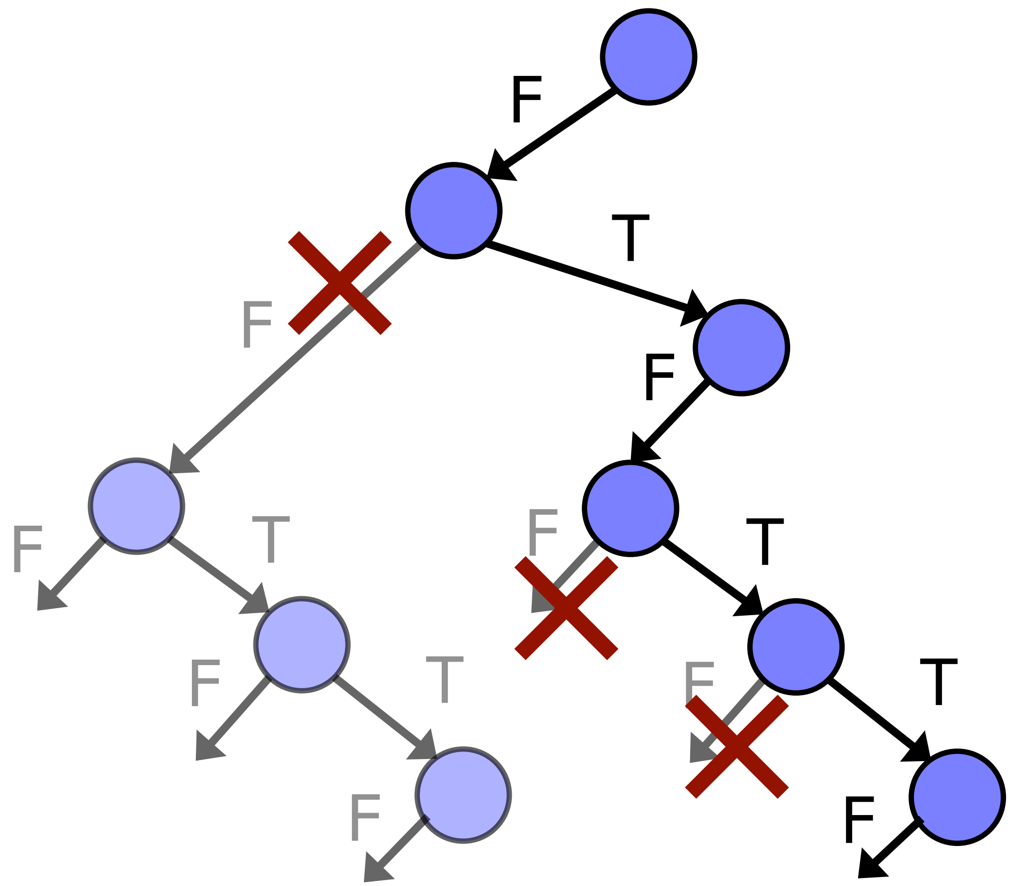
Oracles for Longest Paths

- **Goal**: Prune search of computation tree.



Oracles for Longest Paths

- **Goal**: Prune search of computation tree.



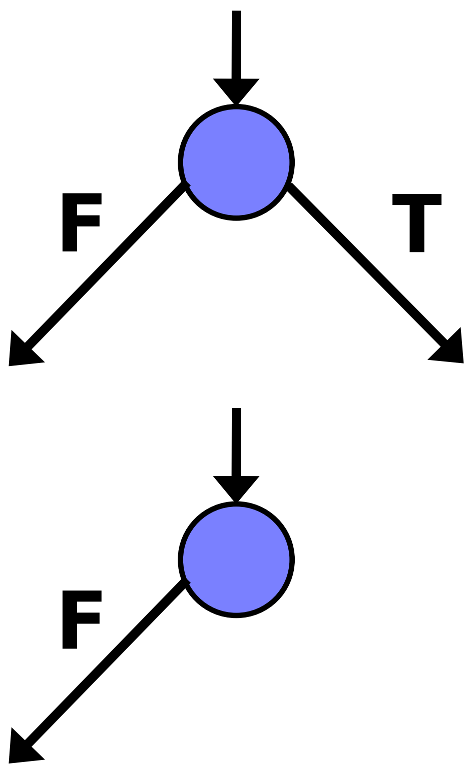
Branch Policy Oracles

- Classify each conditional in P:
  - **Free:** Must explore true or false branch.
  - **Biased:** When feasible, only explore true (resp. false) branch.

Branch Policy Oracles

- Each conditional in P classified as:

Free:

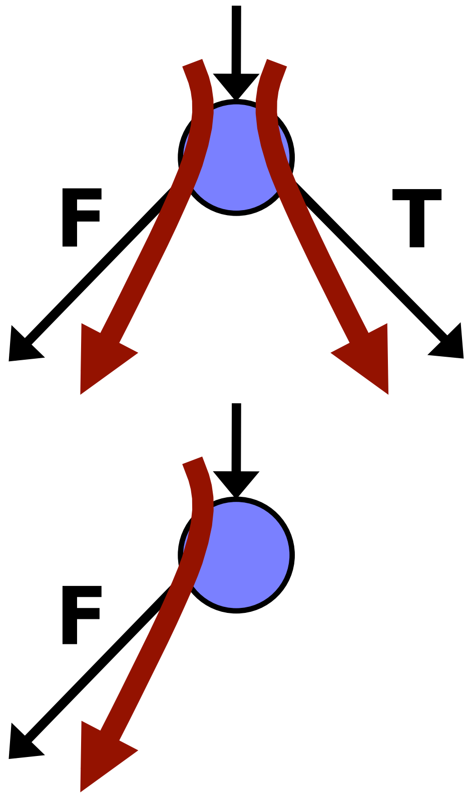


Biased:

Branch Policy Oracles

- Each conditional in P classified as:

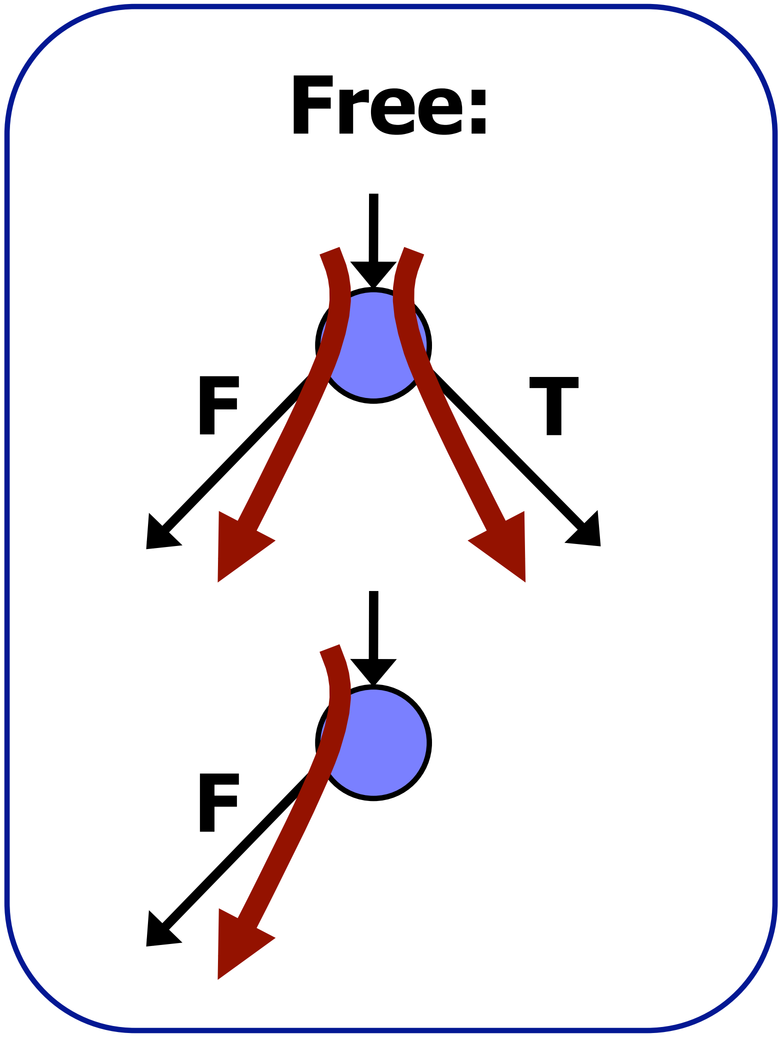
Free:

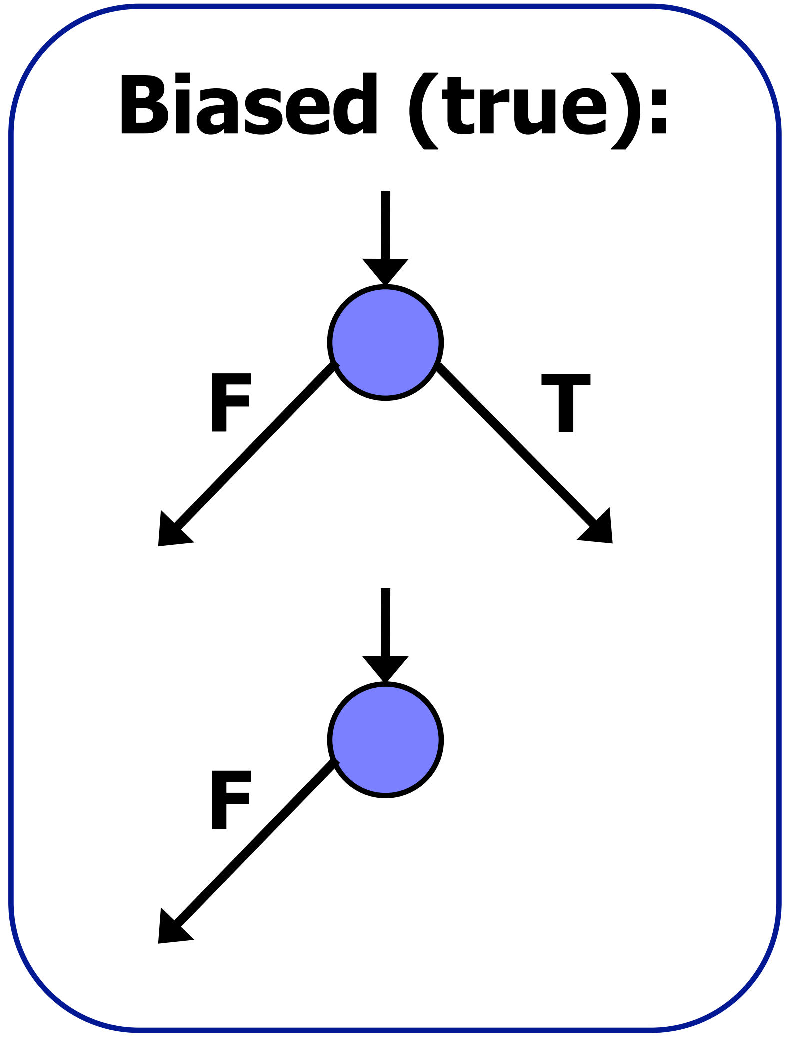


Biased:

Branch Policy Oracles

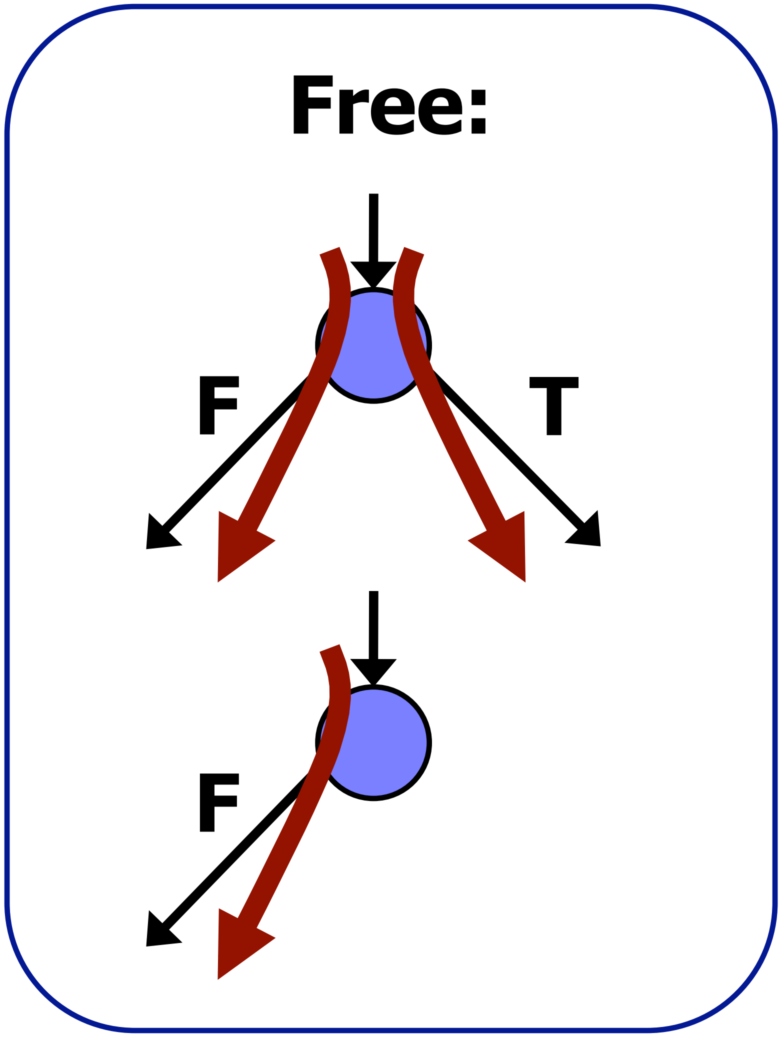
- Each conditional in P classified as:

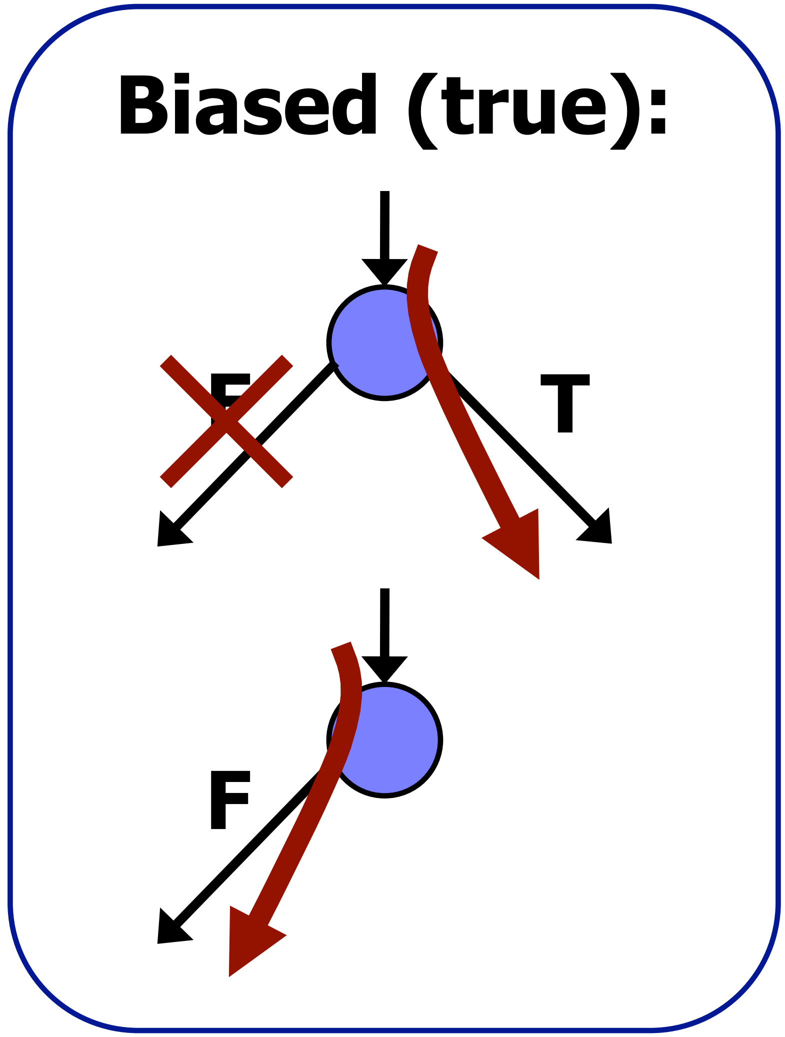




Branch Policy Oracles

- Each conditional in P classified as:





Example: Searching w/ Branch Policy

- N insertions into empty sorted list:

```
// list with sentinel INT_MAX
insert(list* p, int x) {
   while (x > p->data) {
      p = p->next;
   }
   p->next = new list(p->data,
                      p->next);
   p->data = x;
}
```

Example: Searching w/ Branch Policy

- N insertions into empty sorted list:

```
// list with sentinel INT_MAX
insert(list* p, int x) {
    while (x > p->data) {
        p = p->next;
    }
    p->next = new list(p->data,
                       p->next);
    p->data = x;
}
```

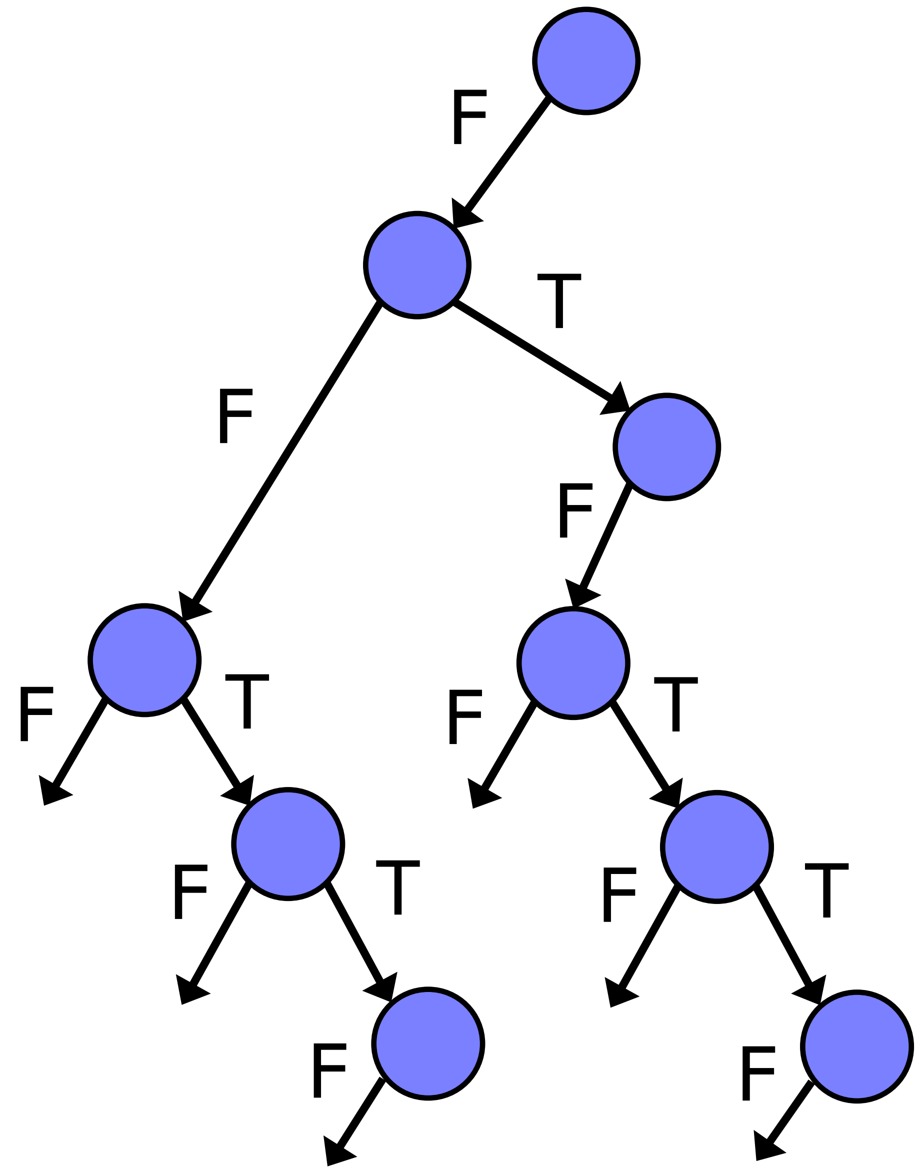
Biased to **true** branch.

Example: Searching w/ Branch Policy

sorted list:

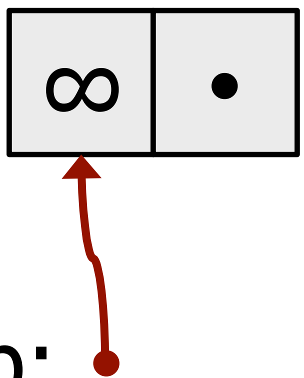
∞	•

```
insert(list, x₁);
insert(list, x₂);
insert(list, x₃);
```



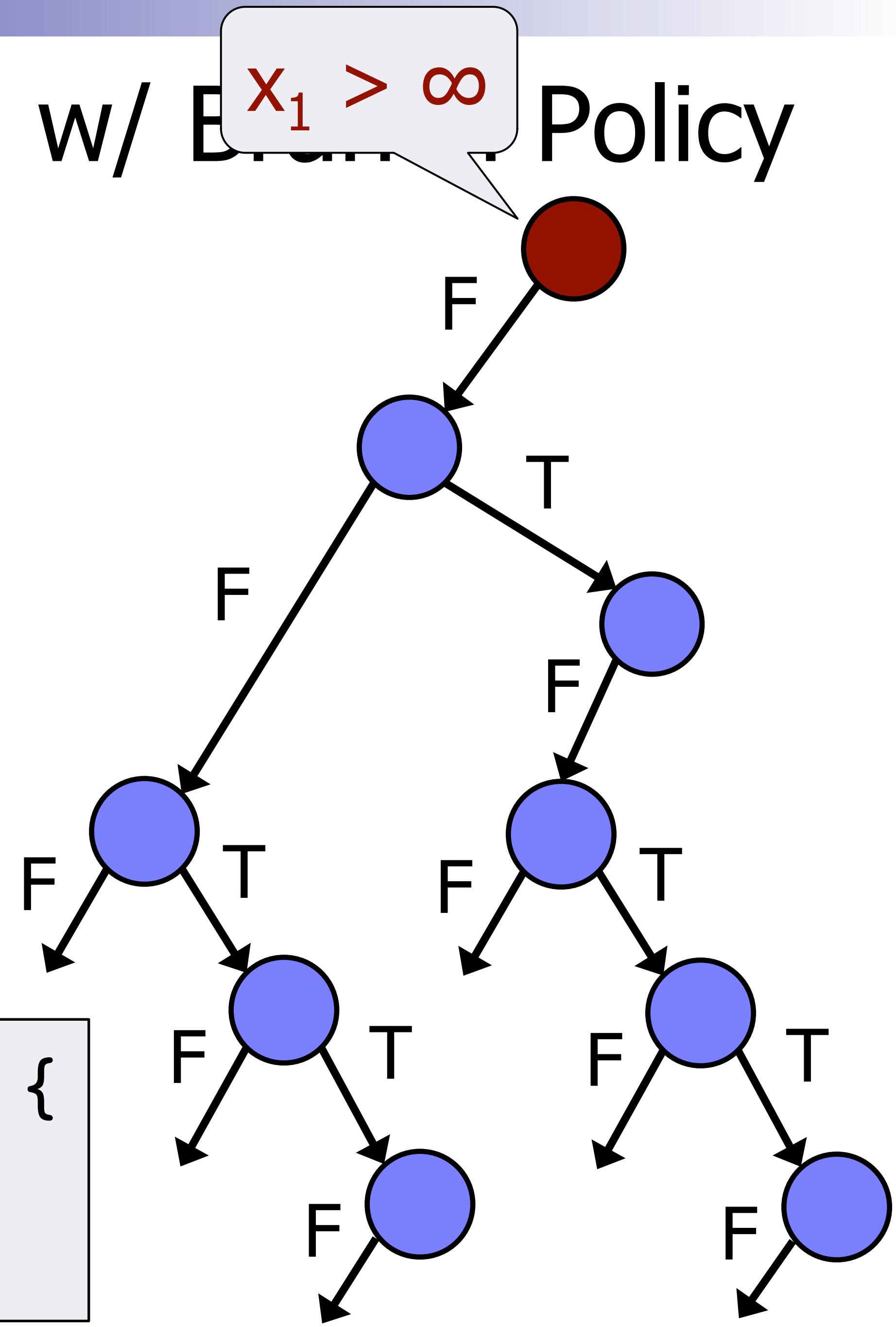
Example: Searching w/ B... Policy

sorted list:



p:

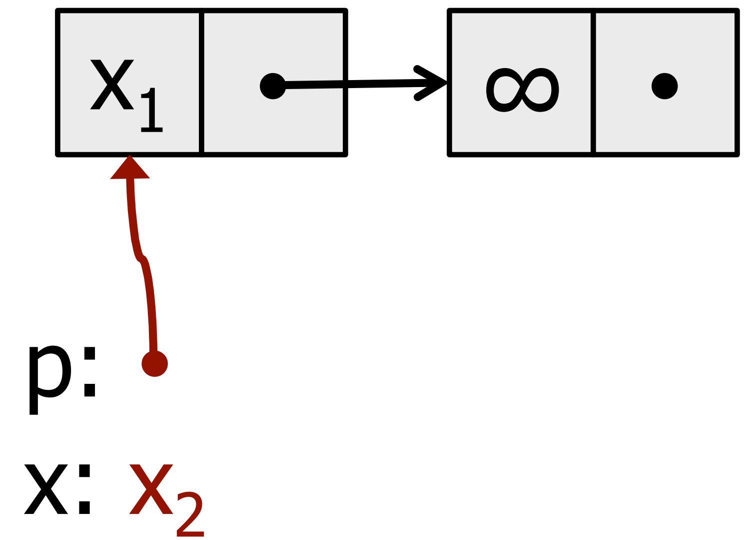
x: x_1



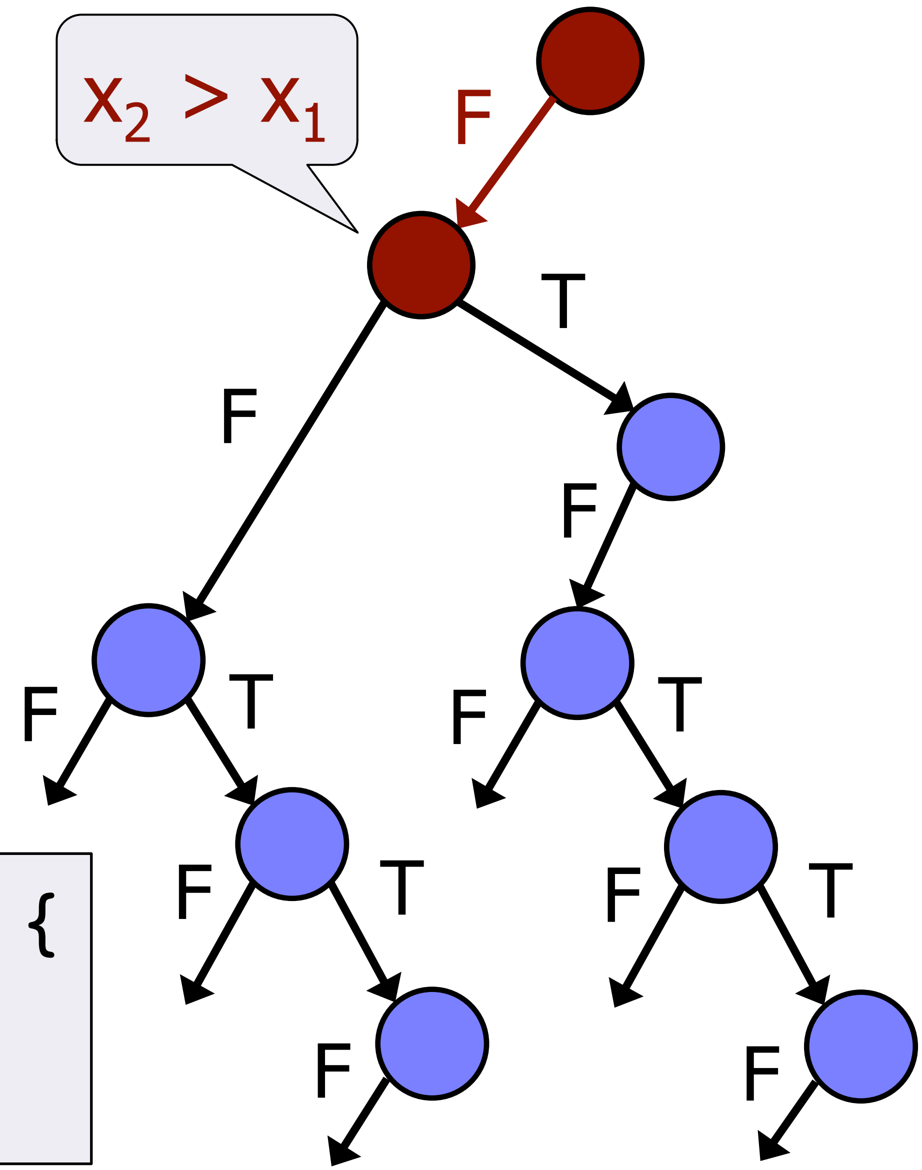

```
while (x > p->data) {
    p = p->next;
}
```

Example: Searching w/ Branch Policy

sorted list:

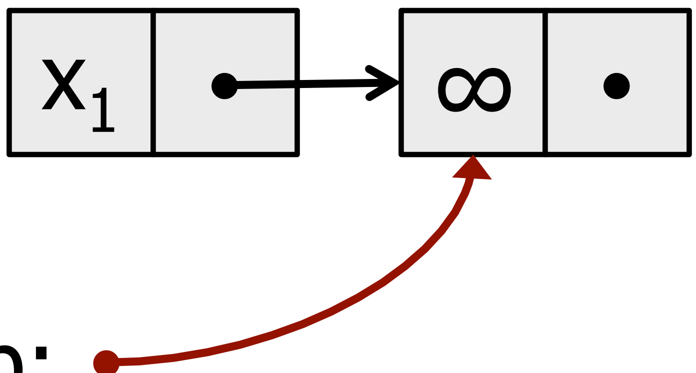



```
while (x > p->data) {
    p = p->next;
}
```



Example: Searching w/ Branch Policy

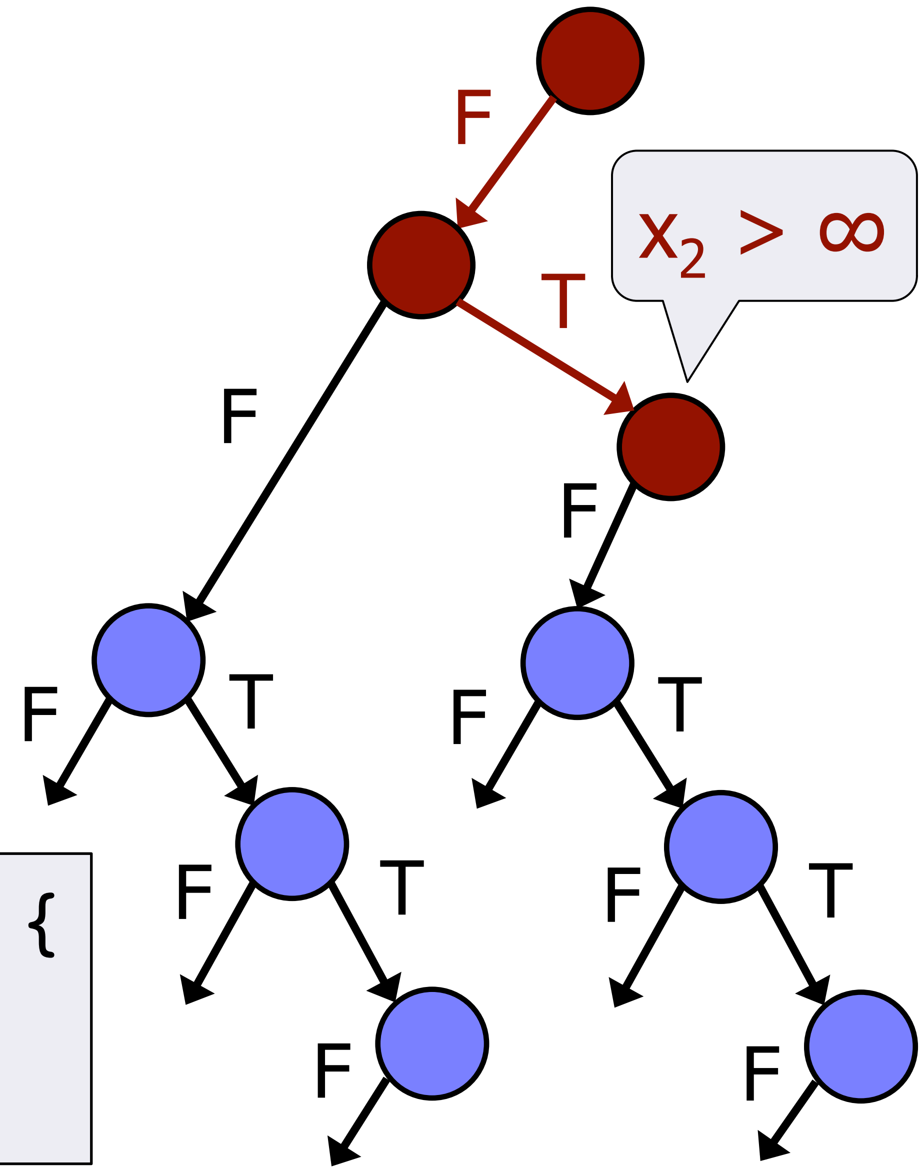
sorted list:



p:

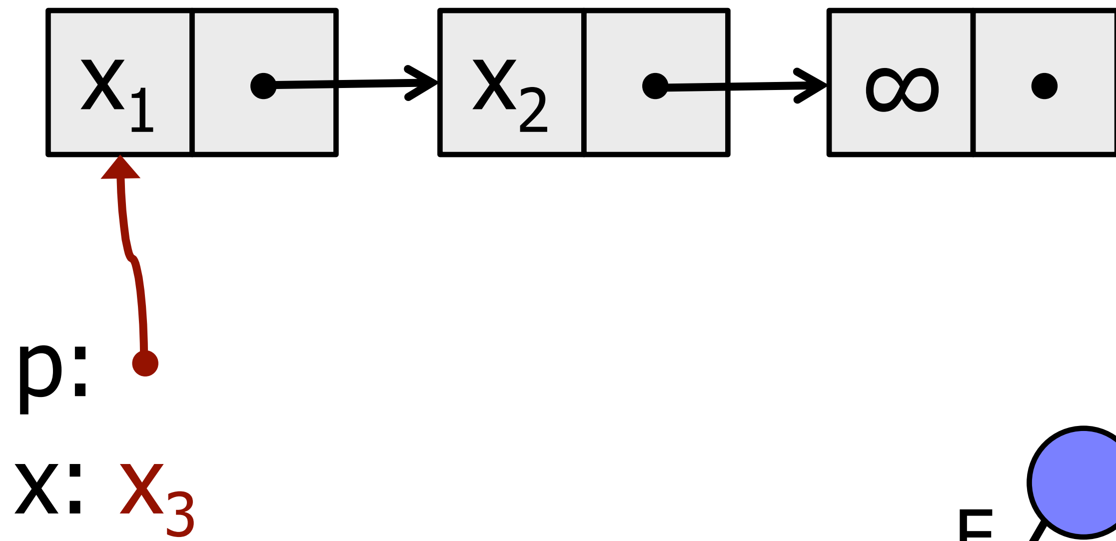
x: x_2

```
while (x > p->data) {
    p = p->next;
}
```

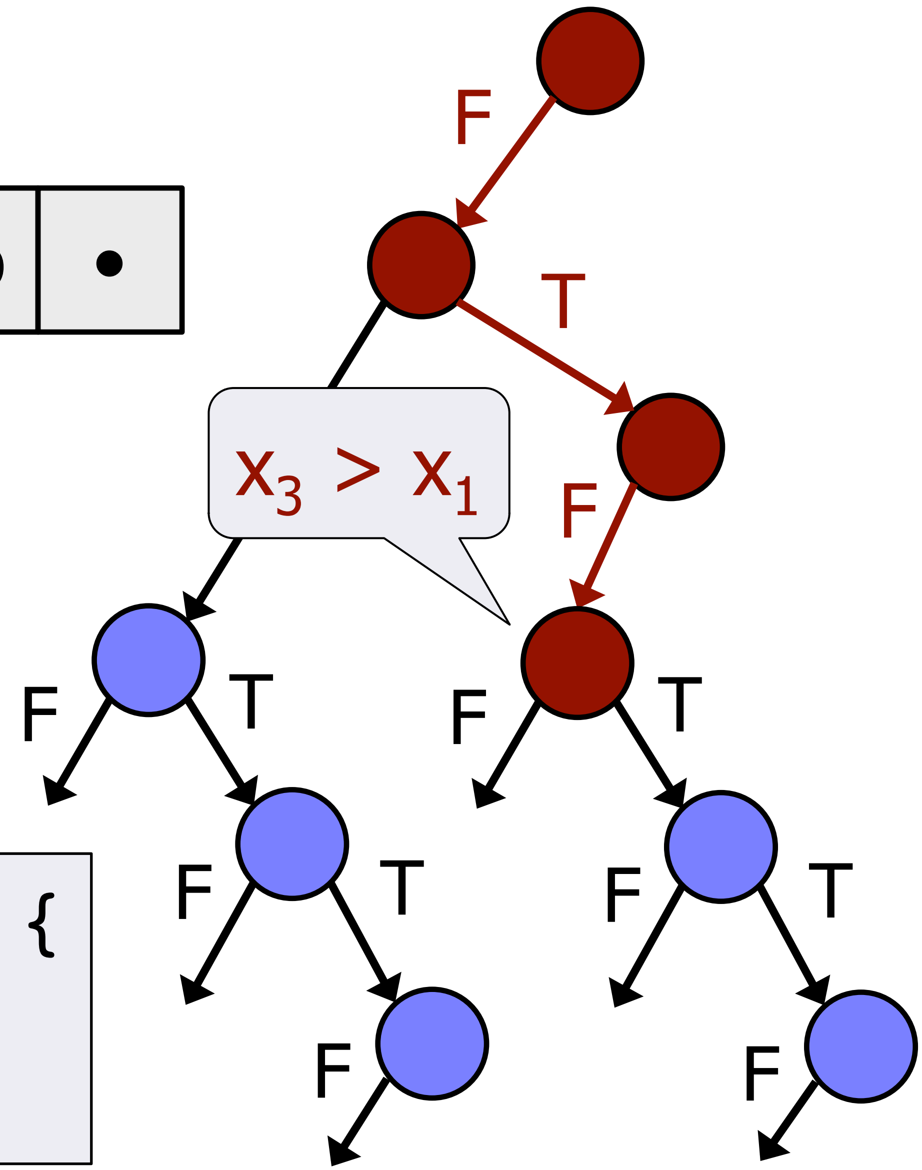


Example: Searching w/ Branch Policy

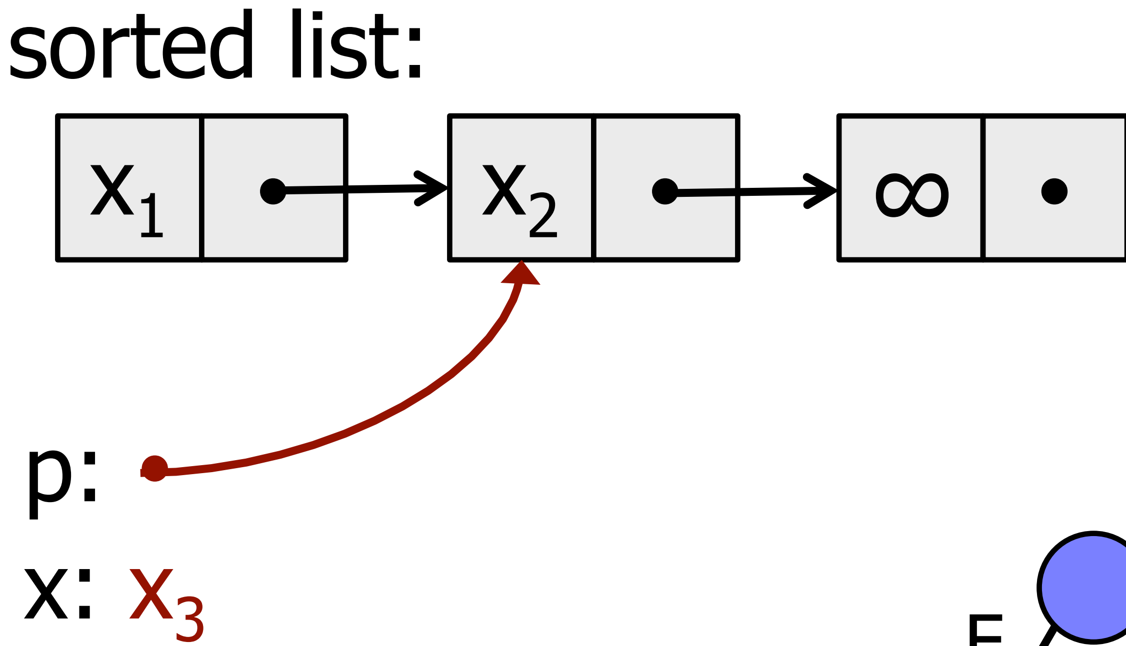
sorted list:



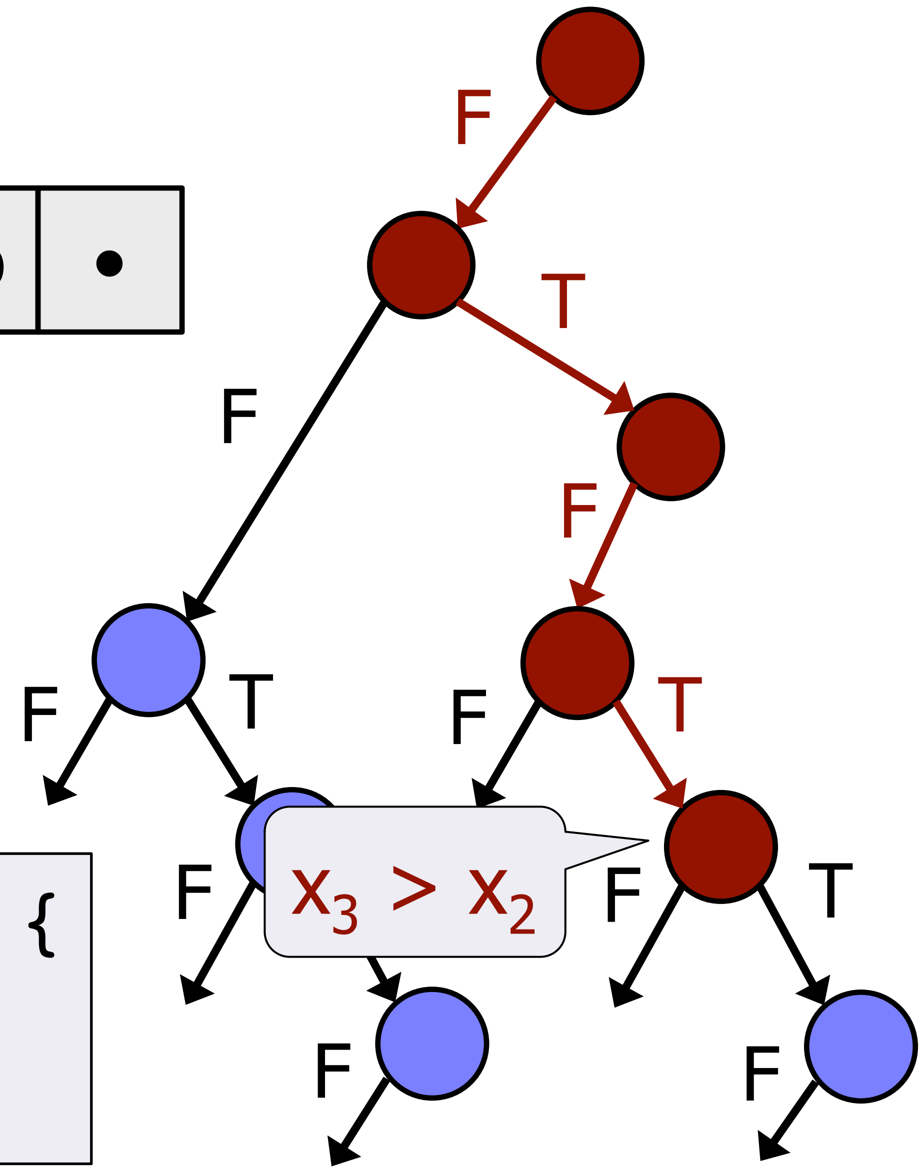

```
while (x > p->data) {
    p = p->next;
}
```



Example: Searching w/ Branch Policy

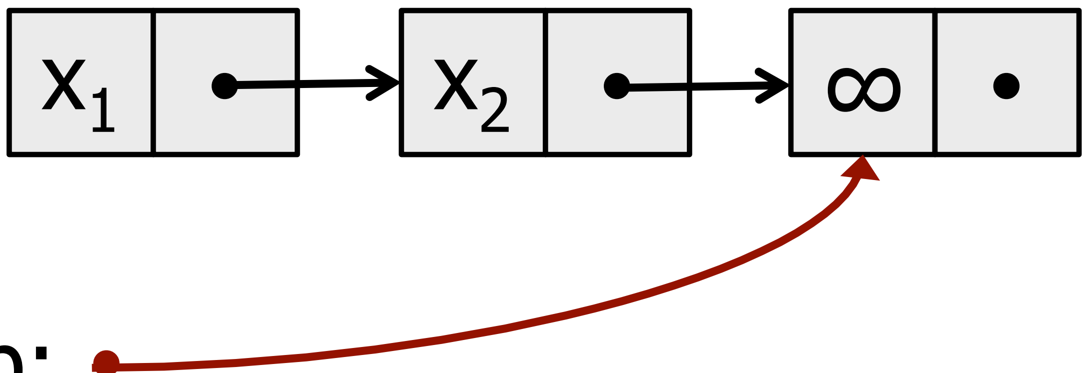



```
while (x > p->data) {
    p = p->next;
}
```



Example: Searching w/ Branch Policy

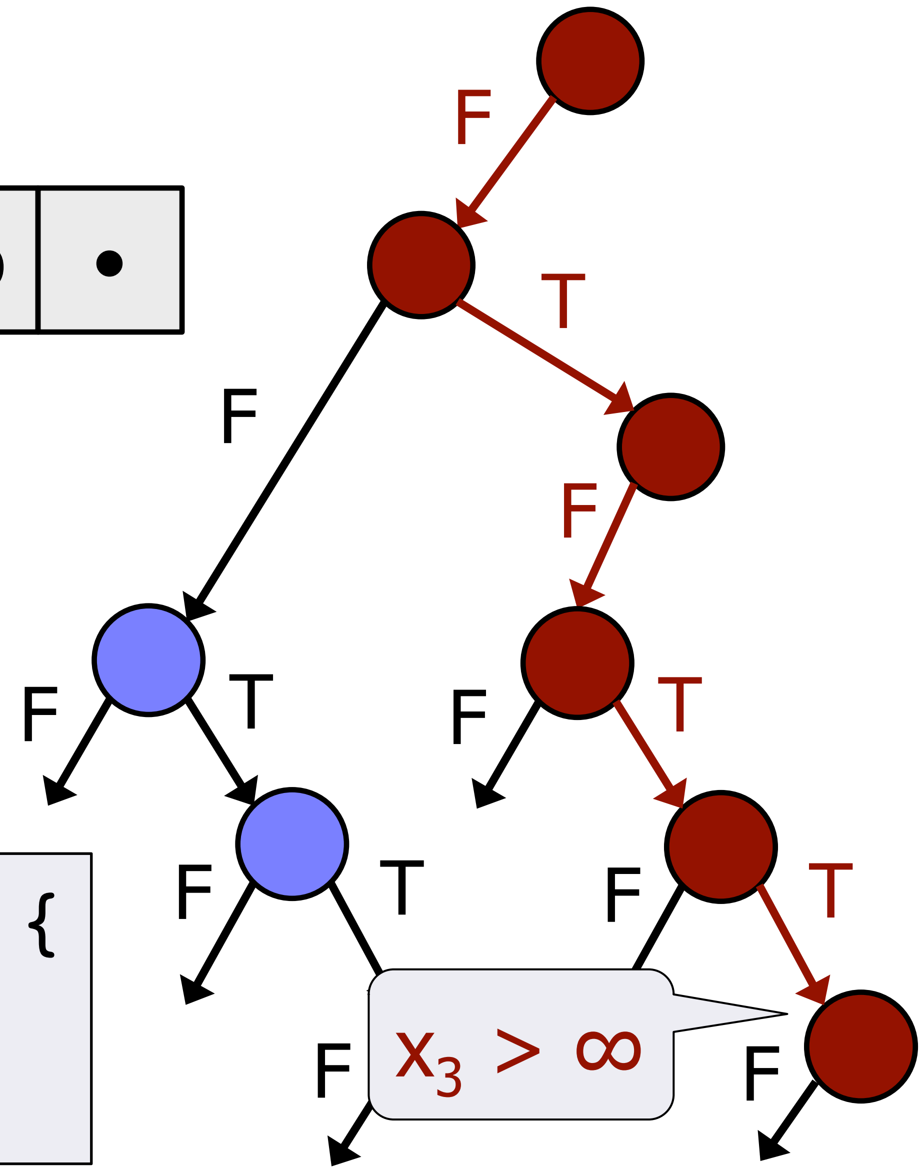
sorted list:



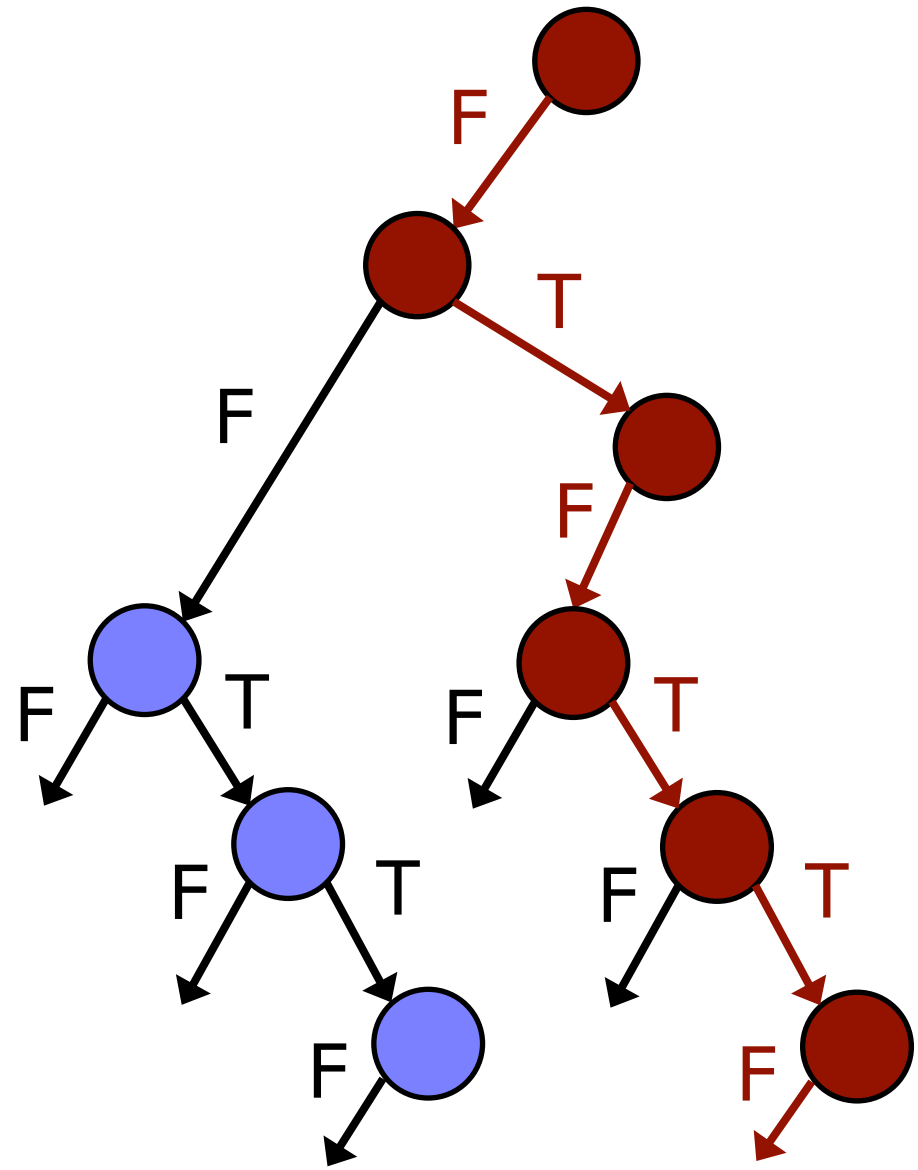
p:

x: x_3

```
while (x > p->data) {
    p = p->next;
}
```

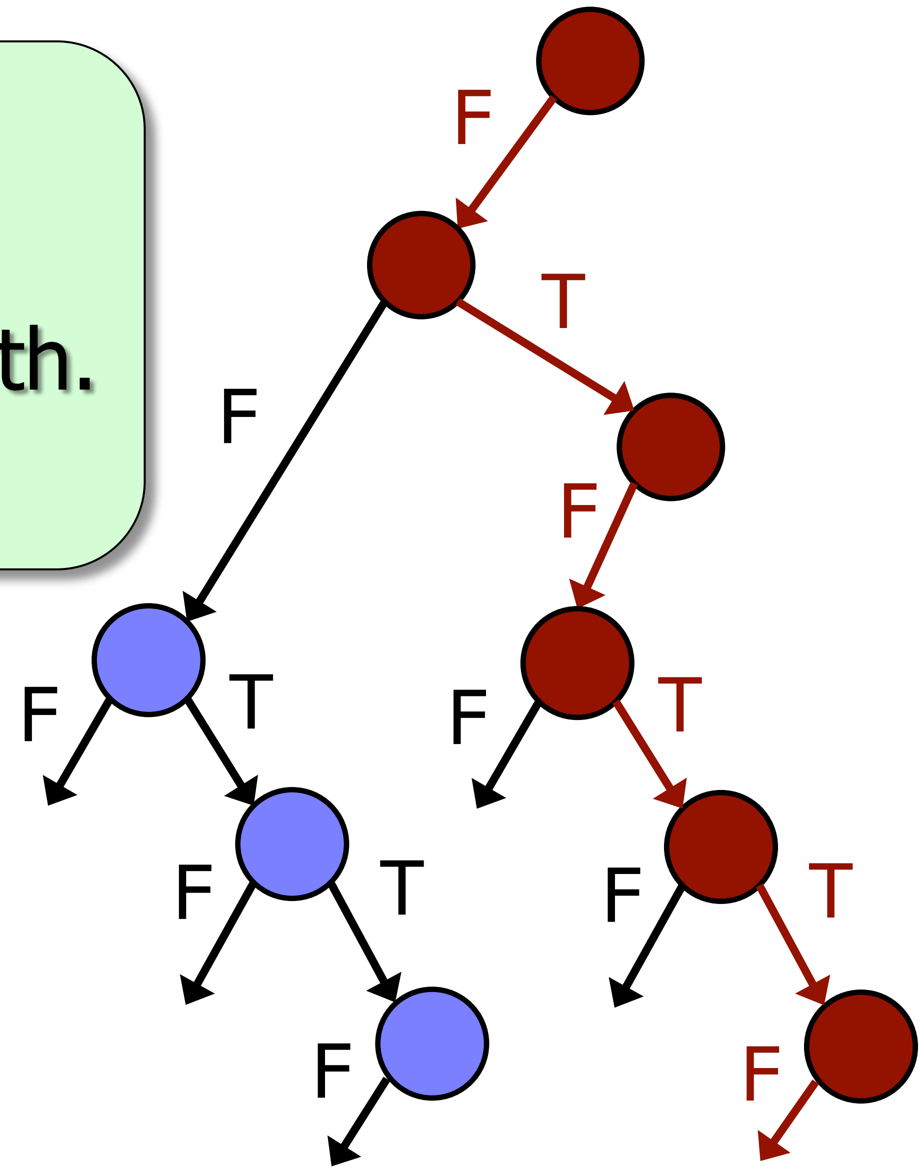


Example: Searching w/ Branch Policy



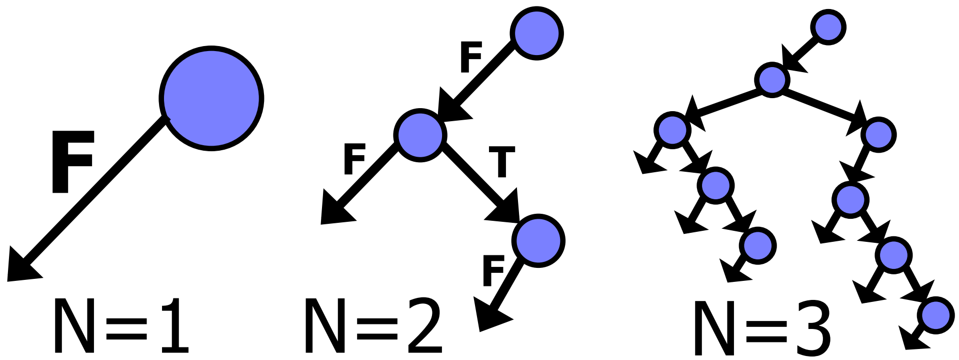
Example: Searching w/ Branch Policy

Search is directed precisely to longest path.



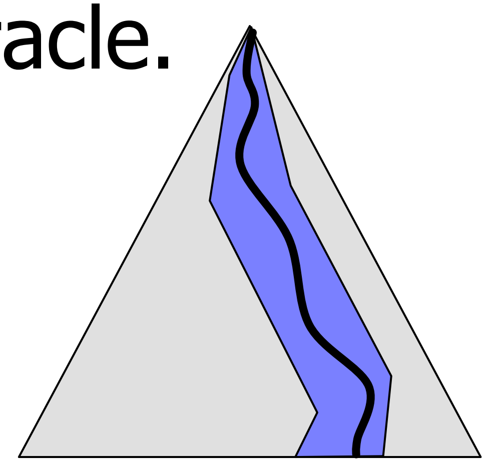
Overview of WISE

- **Step 1**: From executions on small inputs, learn oracle for longest paths.



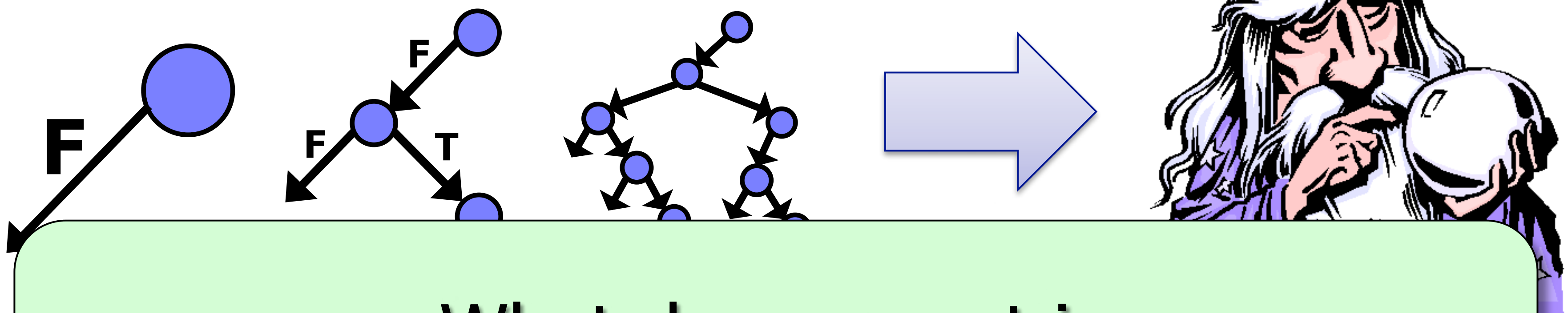
- **Step 2**: For large inputs, only examine paths generated by oracle.

N=15

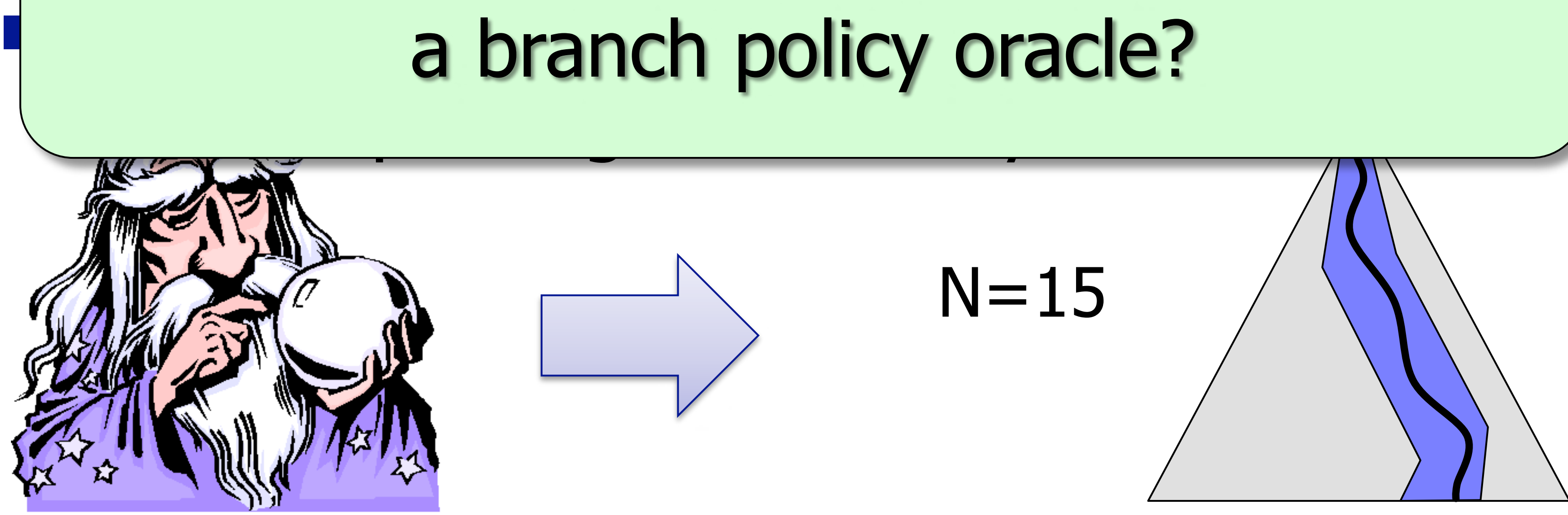


Overview of WISE

Step 1: From executions on small inputs, learn oracle for longest paths.



What do we want in
a branch policy oracle?



Selecting a Branch Policy

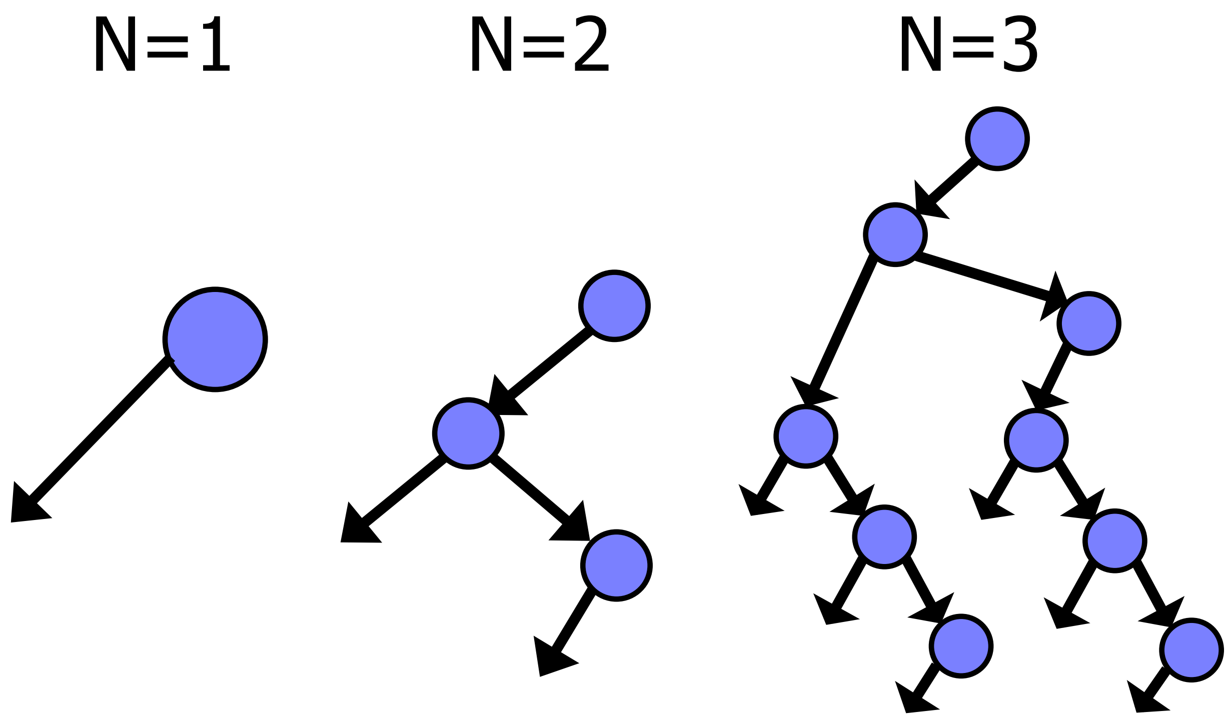
- Find all executions on size-1,…,T inputs.
- Pick branch policy B that:
  - gives a longest path for each 1,…,T
  - gives fewest # paths on 1,…,T

Selecting a Branch Policy

- Pick branch policy B that:
  - gives a longest path for each 1,...,T
  - gives fewest # paths on 1,...,T

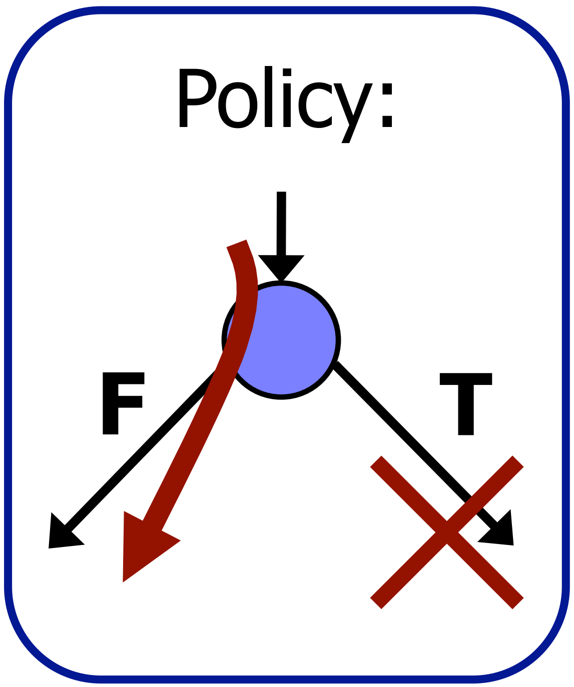
Policy:

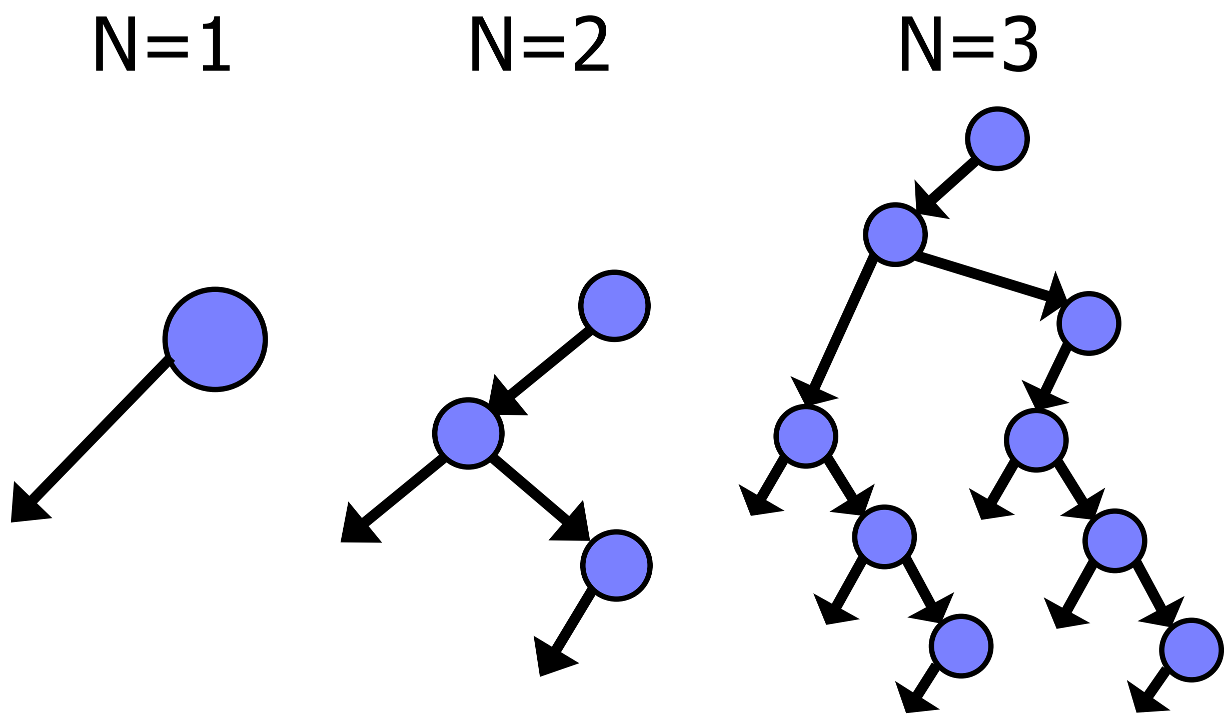
?



Selecting a Branch Policy

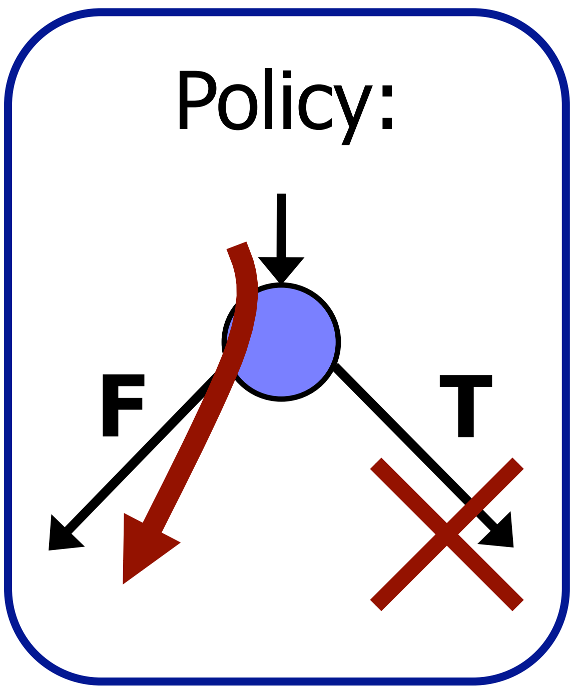
- Pick branch policy B that:
  - gives a longest path for each 1,…,T
  - gives fewest # paths on 1,…,T

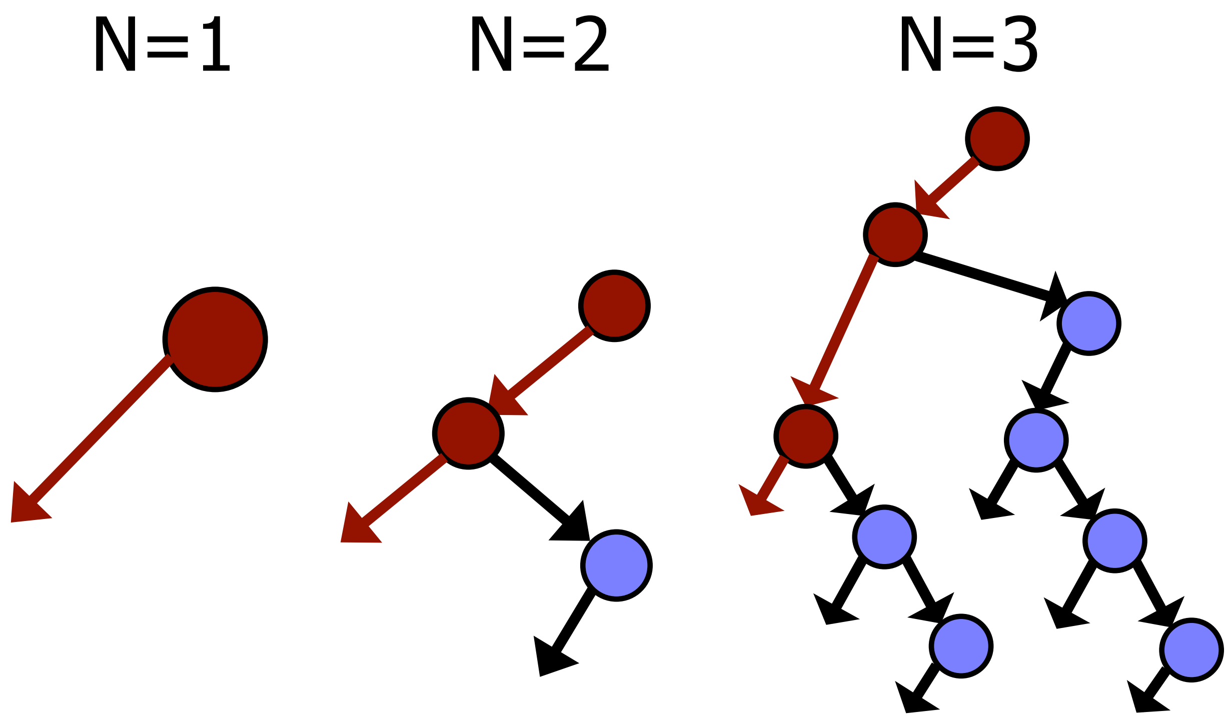




Selecting a Branch Policy

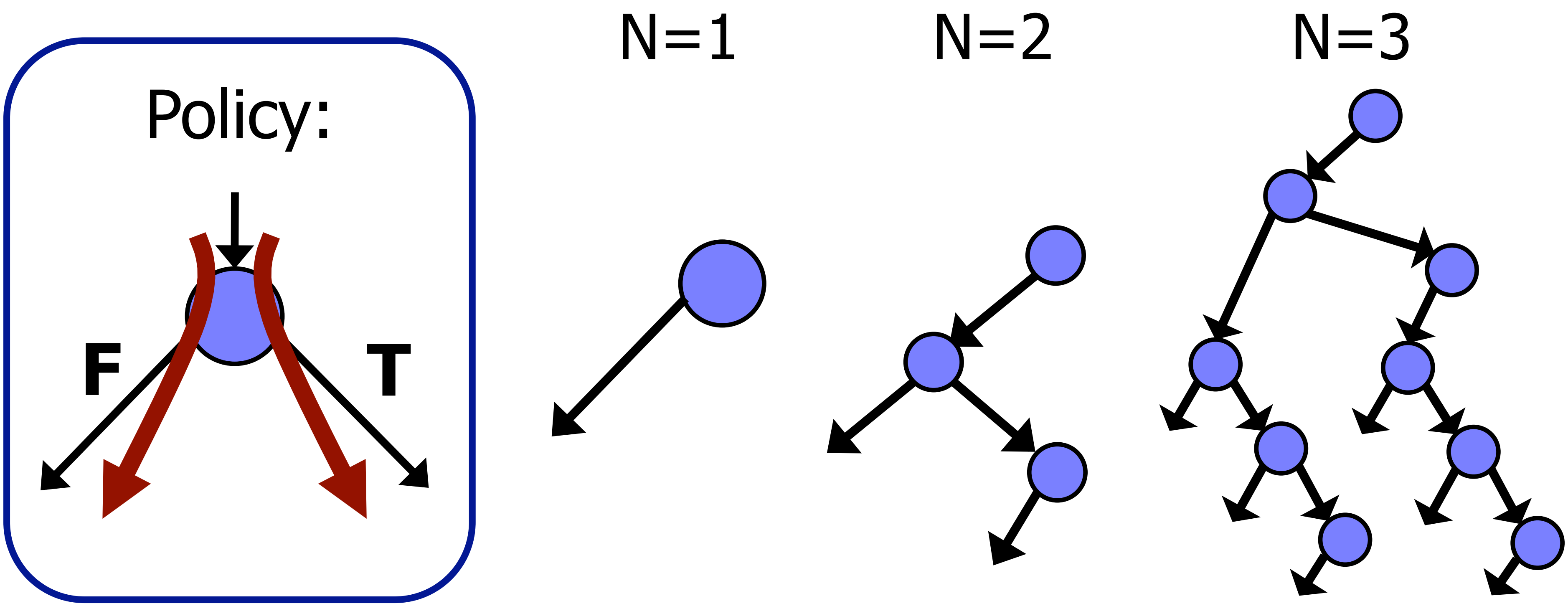
- Pick branch policy B that:
  - ~~gives a longest path for each 1,…,T~~
  - gives fewest # paths on 1,…,T





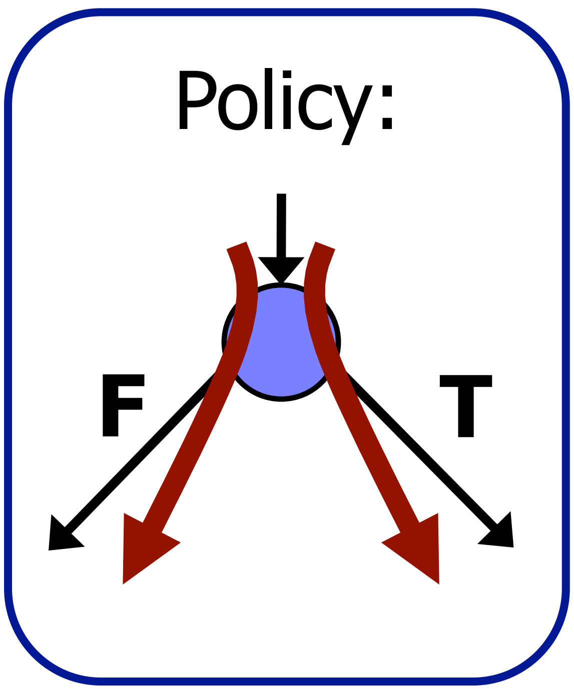
Selecting a Branch Policy

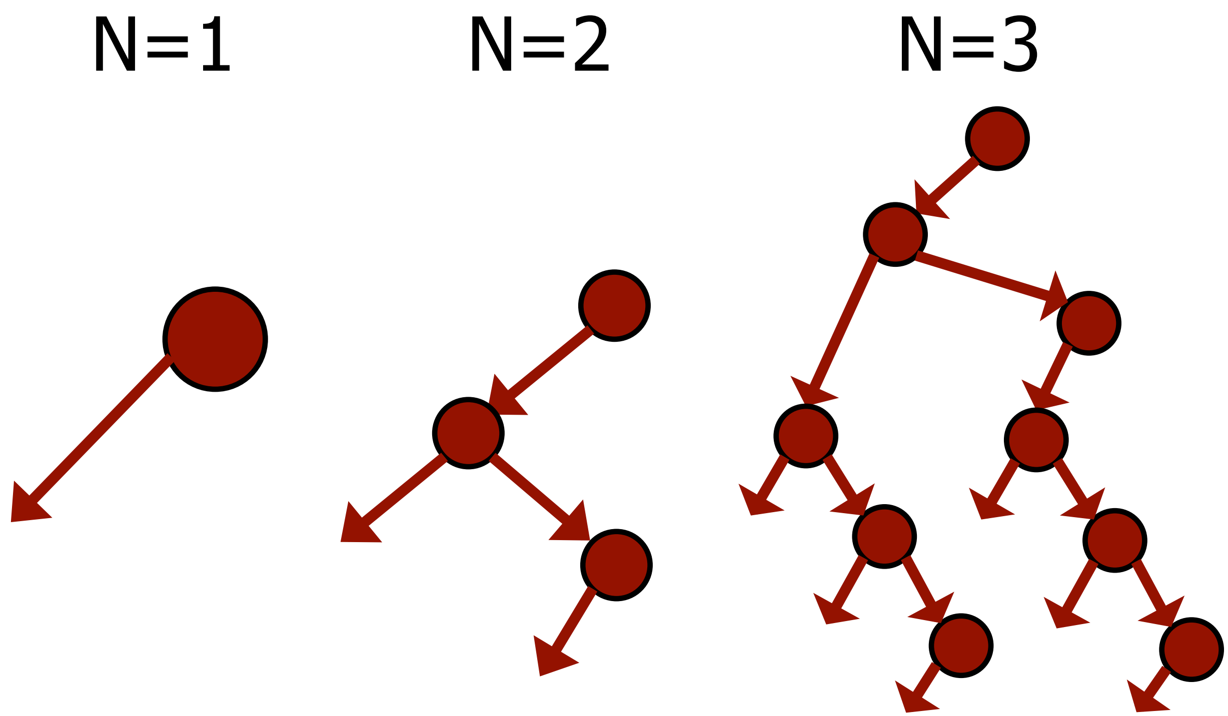
- Pick branch policy B that:
  - gives a longest path for each 1,...,T
  - gives fewest # paths on 1,...,T



Selecting a Branch Policy

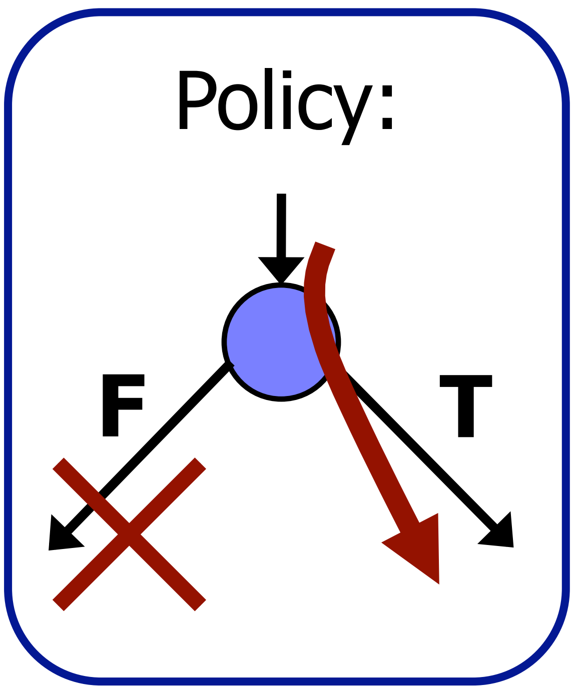
- Pick branch policy B that:
  - gives a longest path for each 1,…,T
  - ~~gives fewest # paths on 1,…,T~~

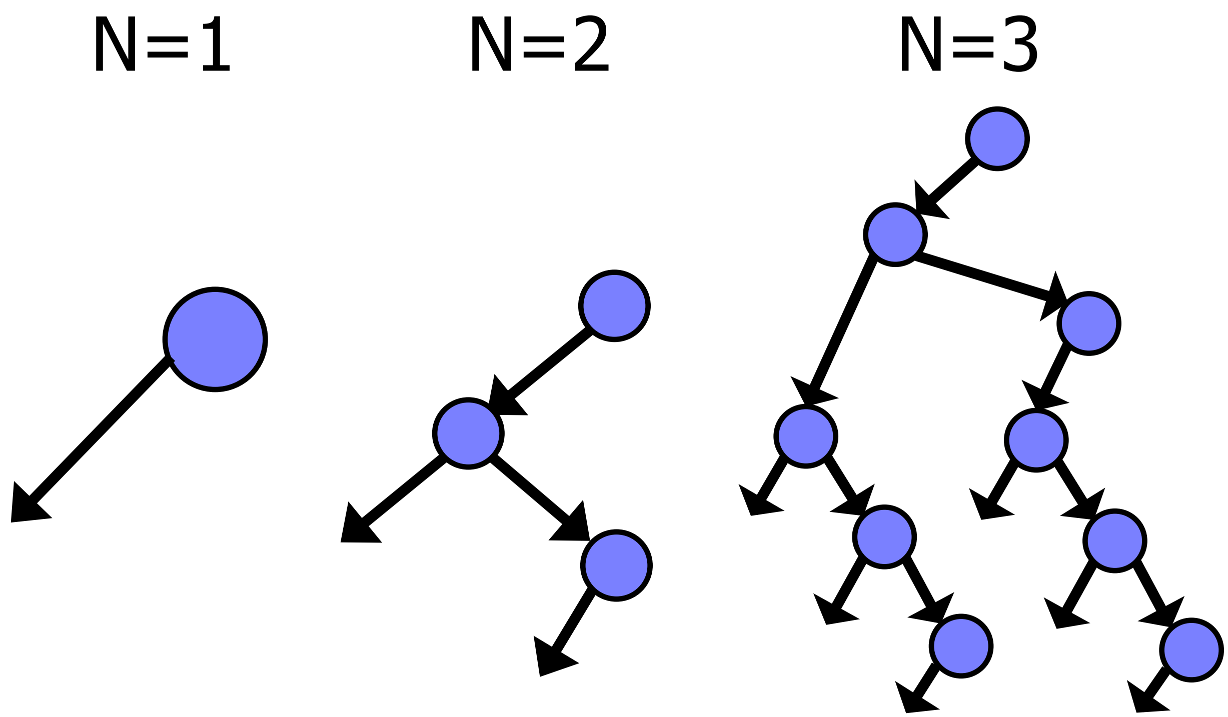




Selecting a Branch Policy

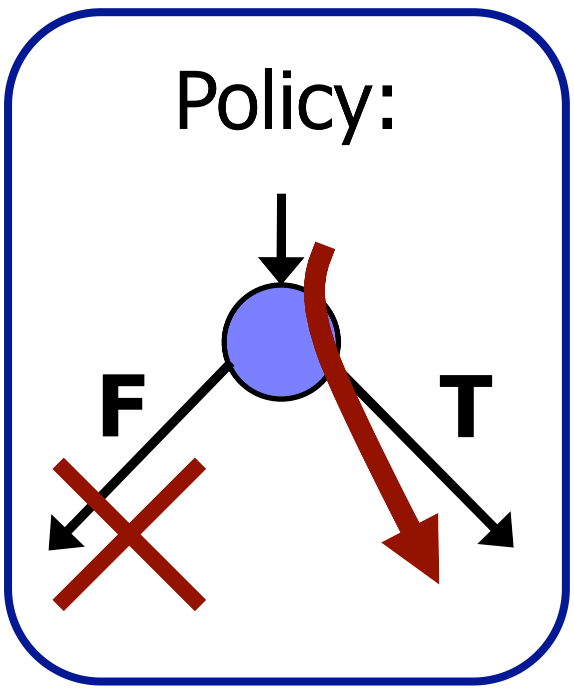
- Pick branch policy B that:
  - gives a longest path for each 1,...,T
  - gives fewest # paths on 1,...,T

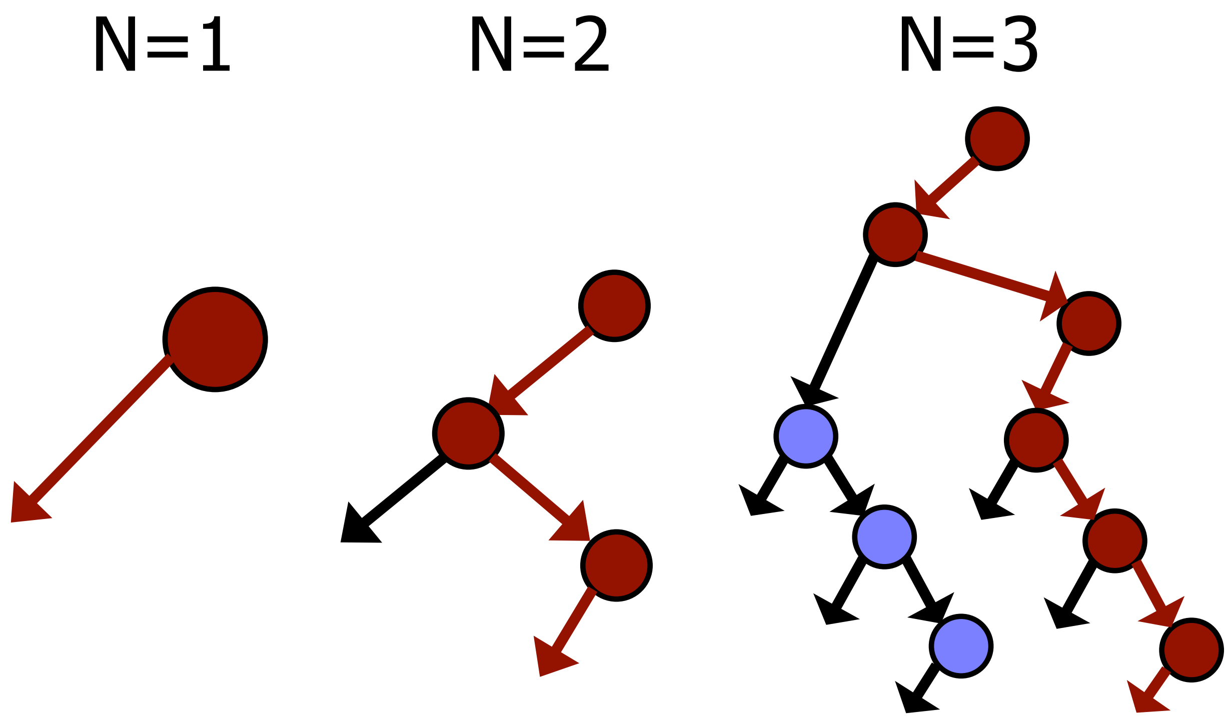




Selecting a Branch Policy

- Pick branch policy B that:
  - gives a longest path for each 1,…,T
  - gives fewest # paths on 1,…,T





Outline

- Motivation + Goal of WISE
- Background: Symbolic Test Generation
- Naïve Algorithm for Finding Complexity
- WISE Algorithm
- **Evaluation**
- Conclusions + Future Work

Evaluating the WISE Algorithm

- **Correctness**
  - **Does WISE find worst-case inputs?**
- Efficiency (Scalability)
  - For large inputs, how well does WISE prune the search?

Correctness of WISE

- Does WISE find worst-case inputs?

- Recall:
  - Find all executions on size-1,…,T inputs.
  - Pick branch policy B that:
    (1) gives a longest path for each 1,…,T
    (2) gives fewest # paths on 1,…,T

- Will B give longest paths for larger inputs?

Correctness of WISE: The Theory

- **Yes**, if T is large enough.

- **Proposition**: For any program P, there exists a T^* such that:
  - Branch policy B works for 1,..,T*
    $\Longrightarrow$ B works for all input sizes.

- How to find T^*? We don't know.
  - In benchmarks, $2 \leq T^* \leq 9$.

Evaluating the WISE Algorithm

- Correctness
  - Does WISE find worst-case inputs?
- **Efficiency (Scalability)**
  - **For large inputs, how well does WISE prune the search?**

Experiments: Data Structures

Benchmark	O(•)	# Paths	# Paths Searched	T*
Sorted List Insert	O(n)	n!	1	2
Heap Insert	O(log n)	~ (log n)!	1	2
Red-Black Tree Search	O(log n)	> n!	1	8
Binary Search Tree Insert	O(n)	> n!	1	3

Experiments: Data Structures

```
// binary search tree insert
void insert(tree** t, int x) {
  while (*t != NULL) {
      if (x <= (*t)->data) {
        t = &(*t)->left;
      } else {
        t = &(*t)->right;
      }
  }
  *t = new tree(x, NULL, NULL);
}
```

Experiments: Data Structures

```
// binary search tree insert
void insert(tree** t, int x) {
  while (*t != NULL) {
     if (x <= (*t)->data) {
        t = &(*t)->left;
     } else {
        t = &(*t)->right;
     }
  }
  *t = new tree(x, NULL, NULL);
}
```

Bias to **true** branch.

Experiments: Data Structures

- For sorted list, tree, and heap insert:
  - *At any conditional comparing a new element to an existing one, the new element should be smaller.*
- For red-black tree search:
  - *Search value should be smaller than all tree elements to which it's compared.*

Experiments: Algorithms

Benchmark	O(•)	# Paths	# Paths Searched	T*
Insertion Sort	$O(n^2)$	n!	1	3
Quicksort	$O(n^2)$	n!	1	8
Mergesort	$O(n \log n)$	n!	$\sim 2^n$	7
Bellman-Ford	$O(nm)$	$> (2^n)^n$	1	3
Dijsktra's	$O(n^2)$	$> 4^n$	1	3
TSP	$O(n!)$	huge	1	5

Experiments: Algorithms

```
quicksort(int A[], int l, int r) {
   …
   // partition
   for (i = l; i < r; i++) {
      if (A[i] <= pivot) {
         swap(A[i], A[mid++];
      }
   }
   …
}
```

Experiments: Algorithms

```
quicksort(int A[], int l, int r) {
  …
  // partition
  for (i = l; i < r; i++) {
    if (A[i] <= pivot) {
      swap(A[i], A[mid++]);
    }
  }
  …
}
```

Bias to **true** branch.

Experiments: Algorithms

- For Bellman-Ford and Dijkstra's:
  - *In each iteration, every edge should be relaxed when feasible.*
- For Traveling Salesman:
  - *The search should never be pruned by the heuristic bound.*

Experiments: Algorithms

Benchmark	O(•)	# Paths	# Paths Searched	T*
Insertion Sort	$O(n^2)$	n!	1	3
Quicksort	$O(n^2)$	n!	1	8
Mergesort	$O(n \log n)$	n!	**$\sim 2^n$**	7
Bellman-Ford	$O(nm)$	$> (2^n)^n$	1	3
Dijsktra's	$O(n^2)$	$> 4^n$	1	3
TSP	$O(n!)$	huge	1	5

Limitation: Mergesort

```
…
// merge
while (i <= lenL && j <= lenR) {
    if (left[i] <= right[j])
    {
        A[k++] = left[i++];
    } else {
        A[k++] = right[j++];
    }
}
// copy rest of left or right
```

Limitation: Mergesort

```
…
// merge
while (i <= lenL && j <= lenR) {
   if (left[i] <= right[j])
   {
      A[k++] = left[i++];
   } else {
      A[k++] = right[j++];
   }
}
// copy rest of left or right
```

Longest paths alternate.

Outline

- Motivation + Goal of WISE
- Background: Symbolic Test Generation
- Naïve Algorithm for Finding Complexity
- WISE Algorithm
- Evaluation
- **Conclusions + Future Work**

Related Work

- Worst-case Execution Time (WCET)
  - For real-time, embedded systems
  - Large body of work
- Profiling – e.g. gprof [Graham, et al., 1982]
- Empirical asymptotic complexity
  - [Goldsmith, Aiken, Wilkerson, FSE 07]
- Static loop bounds
  - Linear ranking functions [Colon, Sipma, TACAS 01]
  - [Gulavani, Gulwani, CAV 08]
  - SPEED [Gulwani, et a., POPL 08]

Conclusions + Future Work

- Automated testing typically for correctness
  - Have adapted for performance/complexity
- **W**orst-case **I**nputs from **S**ymbolic **E**xecution
  - Generalizes from runs on small inputs
  - For small functions/components
- Next: Algorithmic denial-of-service
  - E.g. regular expression matching
  - E.g. NIDS packet matching

QUESTIONS?