Dynaplex: Analyzing Program Complexity using Dynamically Inferred Recurrence Relations

DIDIER ISHIMWE, University of Nebraska-Lincoln, USA
KIMHAO NGUYEN, University of Nebraska-Lincoln, USA
THANHVU NGUYEN, George Mason University, USA

Being able to detect program runtime complexity is useful in many tasks (e.g., checking expected performance and identifying potential security vulnerabilities). In this work, we introduce a new dynamic approach for inferring the asymptotic complexity bounds of recursive programs. From program execution traces, we learn *recurrence relations* and solve them using pattern matching to obtain closed-form solutions representing the complexity bounds of the program. This approach allows us to efficiently infer simple recurrence relations that represent nontrivial, potentially nonlinear polynomial and non-polynomial, complexity bounds.

We present Dynaplex, a tool that implements these ideas to automatically generate recurrence relations from execution traces. Our preliminary results on popular and challenging recursive programs show that Dynaplex can learn precise relations capturing worst-case complexity bounds (e.g., $O(n \lg n)$ for mergesort, $O(2^n)$ for Tower of Hanoi and $O(n^{1.58})$ for Karatsuba's multiplication algorithm).

CCS Concepts: • **Software and its engineering → Software verification**; **Dynamic analysis**; **Formal software verification**; • **Theory of computation → Invariants**; **Program analysis**.

Additional Key Words and Phrases: complexity analysis, dynamic invariant generation, recurrence relations

ACM Reference Format:
Didier Ishimwe, KimHao Nguyen, and ThanhVu Nguyen. 2021. Dynaplex: Analyzing Program Complexity using Dynamically Inferred Recurrence Relations. *Proc. ACM Program. Lang.* 37, 4, Article 111 (August 2021), 21 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Program invariants describe properties that always hold at a program location. Examples of invariants include pre/post-conditions, loop invariants, and assertions. Generated invariants are useful in many programming tasks, including verification, documentation, testing, debugging, code generation, and more [Ball and Rajamani 2001; Das et al. 2002; De Moura and Bjørner 2008; Ernst 2000; Henzinger et al. 2002; Leroy 2006].

The Syminfer work in [Nguyen et al. 2017b] uses dynamic analysis to infer nonlinear invariants (e.g., $x = qy + r$) among numerical program variables from program execution traces. A rather surprising use of SymInfer's invariants is that they can help characterize program runtime complexity. This works by instrumenting a counter for the number of blocks executed and inferring an equality relationship involving that counter and the program's input variables at the end of the program. For example, it can be shown this way that a program runs in $O(n^2)$, where n is a program input.

Authors' addresses: Didier Ishimwe, ishimwed@huskers.unl.edu, University of Nebraska-Lincoln, USA; KimHao Nguyen, kdnguyen@huskers.unl.edu, University of Nebraska-Lincoln, USA; ThanhVu Nguyen, nguyenthanhvuh@gmail.com, George Mason University, USA.

2475-1421/2021/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

Indeed, runtime complexity is an interesting class of program properties, and the ability to automatically discover it has many benefits. For example, we can determine if an implementation of an existing algorithm exceeds the theoretical worst-case complexity and thus has a performance bug [Song and Lu 2014]. We can also discover the complexity of a new implementation and attempt to optimize it. Moreover, complexity analysis can help detect several important security vulnerabilities, e.g., exhausting the system's resources or leaking side-channel information [Burnim et al. 2009; Petsios et al. 2017].

While being interesting, SymInfer's polynomial equalities are just too strict to capture the complexity of most programs. For example, unless the value of the counter variable is exactly equal to n^2, Syminfer cannot compute the necessary equality to conclude that the program has a quadratic complexity. Moreover, polynomial invariants cannot represent complexity bounds that involve non-polynomial, e.g., logarithmic and exponential, terms that are common in asymptotic complexity. Nonetheless, this work shows the effectiveness of dynamic invariant inference and the potential uses of the inferred invariants. Indeed, another recent work applies dynamically inferred numerical invariants to compute ranking functions and termination sets and use them to analyze program termination and non-termination behaviors [Le et al. 2020].

Inspired by dynamic invariant generation, we propose Dynaplex, a new dynamic inference technique and tool for learning recurrence relations (or simply *recurrences*) to capture the asymptotic complexity bounds of *recursive programs*. Briefly, a recurrence defines the complexity to solve a problem in terms of the work to solve its subproblems [Cormen et al. 2009]. Moreover, we can solve the recurrence to obtain a closed-form solution that describes the asymptotic complexity.

At a high level, Dynaplex is a dynamic invariant generation tool that learns recurrences from program traces. The tool supports two forms of recurrences that often appear in recursive programs: *divide-and-conquer* recurrences such as $T(n) = 2T(\frac{n}{2}) + n$ for merge sort and *linear* recurrences such as $T(n) = T(n-2) + T(n-1) + 1$ for Fibonacci. Dynaplex analyzes program traces to infer these recurrences and then applies the Master theorem and other pattern matching techniques to map recurrences to different classes of complexity bounds.

Dynaplex differs from other complexity analyses (e.g., [Breck et al. 2020; Chatterjee et al. 2019; Gulwani 2009]) in the use of a dynamic, instead of static or hybrid, analysis to discover asymptotic complexity. By using dynamic analysis, Dynaplex is language agnostic and works with complex programs (e.g., using non-trivial modulo and exponential operations) that might be difficult for static analyses. By considering only program runs over a small number of randomly generated inputs, Dynaplex efficiently learns divide-and-conquer and linear recurrences. Finally, by using pattern matching techniques optimized for common recurrences, Dynaplex quickly identifies correct worst-case complexity bounds for challenging programs.

We evaluated Dynaplex on 37 recursive programs in C++, Python, and OCaml with a wide range of worst-case complexity bounds. We found that Dynaplex's recurrences can capture nontrivial, potentially nonlinear and even non-polynomial, complexity bounds, e.g., $O(n^{\lg_2 7})$ for the `Strassen` matrix multiplication algorithm. Even when analyzing traces obtained from running the programs on randomly generated inputs, Dynaplex is efficient and effective in learning correct recurrences and solving them for precise complexity bounds. Moreover, in a few cases Dynaplex was not able to obtain the correct recurrences from traces but Dynaplex still can "recover" from these inaccuracies and derive correct worst-case complexity results.

Contributions. In summary, this paper makes the following contributions:

- We introduce a new dynamic analysis technique for learning divide-and-conquer and linear recurrences that appear in many recursive programs.

```
def mergesort(L, id):
  #id is a list, e.g., [1]
  trace(len(L), id)

  t = 0 #ctr variable
  n = len(L)
  if n == 0 or n == 1:
    return copy(L)

  mid = n // 2

  id_ = id + [++t] # [1,1]
  A = mergesort(L[0:mid], id_)

  id_ = id + [++t] # [1,2]
  B = mergesort(L[mid:n], id_)

  C = merge(A, B, n)
  return C
```

```
def merge(A, B, n):
  t = 0 #ctr variable
  a = 0; b = 0; C = []
  while (a < len(A)
       and b < len(B)):
    t++
    if A[a] <= B[b]:
      C += [A[a]]; a++
    else:
      C += [B[b]]; b++

  while a < len(A):
    t++; C += [A[b]]

  while b < len(B):
    t++; C += [B[b]]

  trace(t, n)
  return C
```

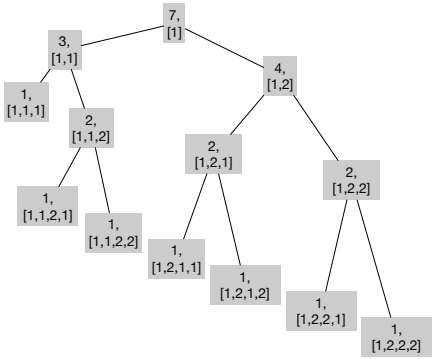

Fig. 2. A mergesort program and its execution tree.

- We develop a formal complexity bound analysis that directly maps recurrences to various classes of program complexity.
- We implement these ideas in the Dynaplex tool and evaluate it on a set of 37 benchmark programs written in multiple languages and having a wide range of complexity bounds. We also compare Dynaplex with other 2 static analyses tools.

Dynaplex and all benchmark data are available at https://github.com/caecus-commits/dynacom.

2 OVERVIEW

Figure 1 gives an overview of Dynaplex, which uses dynamic analysis to infer recurrences to capture the runtime complexity of recursive programs. Dynaplex takes as input program execution traces, obtained by running an instrumented recursive program on a set of inputs, and returns the asymptotic complexity of that program.

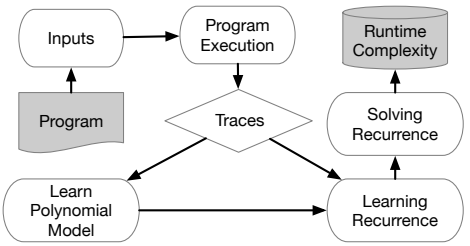

Fig. 1. Dynaplex overview

First, Dynaplex analyzes the program traces to compute a recurrence that defines the recursive execution and a polynomial relation that defines the non-recursive imperative execution. Next, Dynaplex combines the obtained relations into a recurrence defining the full program execution. Finally, to solve the recurrence, Dynaplex applies a pattern matching technique that maps the recurrence to a closed-form solution representing the program complexity.

2.1 Example

Figure 2 shows an implementation of the classical merge sort algorithm, which sorts a list of elements by recursively sorting half of the input list and merging the results together. The runtime complexity of this program depends on the complexity of both the recursive function `mergesort` and the non-recursive function `merge`, which Dynaplex analyzes as follows.

Recurrence Relations. For the recursive `mergesort` function, Dynaplex computes a recurrence relation (or simply *recurrence*), which describes the relationship between the sizes of the inputs of the function and its recursive calls, from program execution traces. To obtain program traces, we instrument the program to introduce the new variables `id` and `t` to keep track of the recursive calls as shown in Figure 2. We use the `trace` function to record the recursive call `id` and its input size as program traces.

We then run `mergesort` on randomly generated inputs of various sizes and collect the program execution traces, represented as an execution tree. For example, the tree in Figure 2 shows the program traces when applying `mergesort` to a list of 7 elements. The root node (7, [1]) is the first `mergesort` call with `id` [1] on the list of 7 elements. The first child node (3, [1, 1]) represents the first recursive call with `id` [1,1] on the first 3 elements of the original list. Similarly, the second child node (4, [1, 2]) represents the second recursive calls with id `id` [1,2] on the remaining 4 elements of the original list.

From this execution tree, Dynaplex forms tuples of the form (t_0, t_1), where t_0 and t_1 are the sizes of the original problem and the subproblems, respectively. Dynaplex next applies linear regression to learn a model of the form $t_1 = ct_0$, which represents the relation between the sizes of the original problem and the first subproblem. For example, Dynaplex learns the model $t_1 = \frac{1}{2}t_0$ from the data $(7, 3), (3, 1), (2, 1), (4, 2)$ in the execution tree in Figure 2.

Similarly, Dynaplex infers $t_2 = \frac{1}{2}t_0$ as the relation between `mergesort` and its second recursive call (subproblem). The combination of these two relations gives the relation $T(n) = T(\frac{n}{2}) + T(\frac{n}{2})$, indicating that `mergesort` makes two recursive calls over inputs that are approximately half of the original input of size n.

Polynomial Relations. For the non-recursive `merge` function, which combines the results of the recursive calls, Dynaplex computes a polynomial relation to capture its runtime complexity with respect to the input of `mergesort` (not the input of `merge` itself). To achieve this, we instrument `merge` to take another input n representing the size of the input of the `mergesort` caller as shown in Figure 2. We also create a the counter variable k in `merge` and increment k in each loop to count the number of executed instructions as shown in Figure 2. Thus, when running `mergesort` on various inputs to collect traces, we also obtain the execution traces for `merge`.

Dynaplex next applies regression to learn a polynomial model fitting the trace data to represent the complexity of `merge`. For this example, the obtained model indicates k is linear in the input size n of the `mergesort` caller, i.e., the complexity bound of `merge` is $O(n)$.

Solving Recurrences. Dynaplex combines the recursive and non-recursive results to obtain $T(n) = 2T(\frac{n}{2}) + O(n)$ as the recurrence for `mergesort`. Finally, the tool applies a pattern matching approach on the recurrence to obtain a closed-form solution representing an asymptotic complexity bound. For example, Dynaplex computes $\mathtt{O(n \log n)}$ as the complexity of the obtained recurrence and therefore of the `mergesort` implementation in Figure 2.

2.2 Dynamic Inference

Dynaplex is essentially a dynamic invariant generation technique and tool that focuses on discovering recurrence and polynomial relations to represent program complexity. Such a dynamic analysis technique typically infers program invariants under certain forms or templates from program execution traces, observed from running a program on a sample of concrete inputs.

Dynamically inferred invariants are shown to be practical and useful by the Daikon work [Ernst et al. 2007], and recently have been used to aid complex static analyses, e.g., proving program correctness using nonlinear invariants [Nguyen et al. 2017b; Yao et al. 2020], reasoning about program termination/non-termination properties by inferring ranking functions and recurrent

sets [Le et al. 2020], and analyzing heap programs with dynamically inferred invariants in separation logic [Le et al. 2019]. In contrast to these works, Dynaplex focuses on learning recurrences to analyze the runtime complexity of recursive programs.

Recurrence Relations. A recurrence relation is a recursive description of a function in terms of itself. In many cases, we can naturally express the running time of an algorithm as a recurrence, where the recursive cases of the recurrence correspond to the recursive cases of the algorithm [Erickson [n.d.]]. For example, the recurrence $T(n) = 2T(\frac{n}{2}) + n$ of `mergesort` in Figure 2 defines the work of sorting `n` elements in terms of the work of sorting two sublists of $\frac{n}{2}$ elements and merging the results. In this work, we focus on two popular forms of recurrences: *divide-and-conquer* recurrences such as the recurrence of `mergesort` and *linear* recurrence such as $T(n) = T(n-2) + T(n-1) + 1$ of `fibonacci`.

By itself, a recurrence does not produce the program complexity. Thus, after learning a recurrence, we need to solve it to obtain a closed-form solution, which is a non-recursive description of the program. We develop pattern-matching techniques that efficiently map various forms of divide-and-conquer and linear recurrences to different complexity classes. An advantage of using recurrences is that even simple recurrences, which can be inferred efficiently, can be solved to obtain expressive complexity classes involving logarithmic and exponential terms. For example, the solution of $T(n) = 2T(\frac{n}{2}) + n$ of `mergesort` is $O(n \log n)$, the solution of $T(n) = T(n-2) + T(n-1) + 1$ of `fibonacci` is $O(2^n)$, and the solution of $T(n) = 7T(\frac{n}{2}) + n^2$ of the matrix multiplication algorithm `Strassen` is $O(n^{\lg_2 7})$.

Inputs and Result Quality. Dynaplex can efficiently discover recurrence and polynomial relations by analyzing traces generated by running the program on a small set of inputs. However, just as with other data-driven dynamic analyses, Dynaplex can produce spurious results depending on the quality of the inputs. For example, randomly generated inputs might not help find the worst-case complexity of a program if such a complexity can only be observed when the program runs under certain inputs.

Finding such WCET (worst-case execution time) inputs is a challenging problem of its own, and there are many promising research on automatically generating them (e.g., [Lemieux et al. 2018; Petsios et al. 2017]). In this work, we do not attempt to generate WCET inputs, but instead focus on the problem of inferring program complexity from traces generated from given inputs.

Even with just randomly generated inputs, our evaluation in §5 shows that Dynaplex can still infer precise recurrences and capture correct worst-case program complexity. This is encouraging because in many cases we can just use cheap randomly generated inputs instead of having to rely on more expensive WCET inputs. Interestingly, in some cases, we obtained inaccurate recurrences from traces but our technique for solving them was able to "recover" from these inaccuracies and produce correct complexity bounds.

3 INFERRING RECURRENCES

Dynaplex infers recurrences to capture the asymptotic time complexity of recursive programs (or functions). We use the standard definition of *asymptotic complexity*, which describes the growth of the program runtime as its inputs get larger. As shown in §2.1, simple recurrences can capture expressive complexity bounds such as those involving logarithmic or nonlinear information. Dynaplex uses dynamic analysis to efficiently learn such recurrences from program traces and solve them to obtain expressive complexity bounds (described in §4).

```
   input  : program execution traces
   output : a recurrence of the form in Eq 1
1  tree ← get_exe_tree(traces)
2  valss ← []
3  foreach node f ∈ tree do
4      foreach i, g ∈ enumerate(get_children(n)) do
5          if |valss| ≤ i + 1 then valss.append(∅)
6          valss[i] ← valss[i] ∪ {(f.val, g.val)}
7  vals ← ∅
8  foreach i ∈ enumerate(|valss|) do
9      val ← compute_ratio(valss[i])
10     if val = null then return null
11     vals ← vals ∪ {val}
12 fn ← compute_poly(traces)
13 rec ← combine(vals, fn) // T(n) = T(n/c_1) + ··· + T(n/c_k) + f(n)
14 return rec
```

Fig. 3. Learning recurrences from program traces.

3.1 Learning Recurrence Relations

We support two forms of recurrences:

$$T(n) = T\left(\frac{n}{b_1}\right) + \cdots + T\left(\frac{n}{b_a}\right) + f(n) \tag{1}$$

$$T(n) = T(n - d_1) + \cdots + T(n - d_a) + f(n) \tag{2}$$

Here n is the size of the input, a the number of subproblems in the recursion, b_i the factor and d_i the difference by which the size of the i^{th} subproblem is reduced in each recursive calls, and $f(n)$ is the work done outside of the recursive call (e.g., the cost of dividing the problem and merging the results in a divide-and-conquer algorithm). Also, a, b_i, d_i are natural numbers, $a \geq 1$, $b_i > 1$, $d_i \geq 1$, and $f(n)$ is ≥ 1 and *monotonically increasing* (i.e., $f(n+1) > f(n)$).

We focus on these two forms of recurrence because they are sufficiently expressive to represent the complexity of many recursive programs. The first form represents *divide-and-conquer* recurrences defining recursive programs with subproblems that are some constant factors of the original problem. These recurrences can be solved to obtain complexity bounds such as $O(\lg n)$, $O(n \lg n)$, and $O(n^c)$. The second form represents *linear* recurrences defining recursive programs with subproblems that are smaller than the original problem by some constant values. These recurrences can be solved to obtain exponential complexity bounds such as $O(2^n)$.

3.1.1 Divide-and-Conquer Recurrences. Figure 3 shows the algorithm to infer divide-and-conquer recurrences of the form in Eq 1. In §3.1.2, we reuse this algorithm to infer linear recurrences of the form in Eq 2.

Collecting Input Sizes. We start by extracting an execution tree from the input traces (line 1). This tree represents the recursive calls of the program: a non-leaf node contains the size of the input to a caller function and the children nodes contain the sizes of the inputs to the recursive calls of that function. For example, Figure 2 shows an execution tree representing the execution of `mergesort` on a list of 7 elements.

We next traverse the execution tree and collect the size of the input problem of each function call f and the sizes of the input subproblems of the recursive calls of f, to the values list `valss` (lines 3-6). The example in Figure 2 has `valss` = [[(7,3), (3, 1), (2,1), (4,2), (2,1), (2,1)]; [(7,4),(4,2), (2,1), (2,1), (3,2), (2,1)]].

Computing Ratios. From each i^{th} value sublist `valss[i]` we compute the ratio between the problem of the function call f and the subproblem of the recursive call g_i of f (lines 8-11). This ratio value represents the factor b_i in the term $T(\frac{n}{b_i})$ in Eq 1. For example, for `valss[0]`=[(7,3), (3, 1), (2,1), (4,2), (2,1), (2,1)], the computed ratio is approximately 2, indicating the subproblem of the first recursive call of `mergesort` is about a half of the original input problem of `mergesort`.

Dynaplex then uses linear regression to compute a model $x = c \cdot y$ from a list of (x,y) tuples, where the coefficient c represents the ratio of x to y. Regression also produces an r-squared value representing how well the computed model fits the given data (r-squared value that is close to 1.0 is better). If the computed model has a low r-squared value, the algorithm cannot determine a clear relationship between x and y and stops (line 10).

For the current example with just 7 data points, we obtain 2.6 for the coefficient with a low r-squared value (close to 0.0). However, when running `mergesort` with larger inputs, e.g., lists of 100 elements, we have more trace data and obtain 2 for the coefficient with a high r-squared value (close to 1.0).

Forming Recurrences. Divide-and-conquer algorithms often consist of non-recursive components that analyze and combine subproblem results (e.g., the `merge` function in `mergesort`). Thus, we need to infer the runtime complexity `fn` of the imperative code (line 12). This technique, described in §3.2, analyzes program traces to learn polynomial models representing the runtime complexity of imperative programs and functions. For example, we infer $O(n)$ as the complexity of `merge`.

Finally, we combine the computed coefficients c_i and complexity `fn` to form a recurrence of the form in Eq 1. For `mergesort`, we obtain $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + O(n)$, corresponding to the recursive structure of `mergesort`: it makes two recursive calls, each with half the size of the original input and the merge operation takes linear time. In §4 we describe how to solve recurrences to obtain program complexity bounds.

3.1.2 Linear Recurrences. If Dynaplex cannot find a divide-and-conquer recurrence of the form in Eq 1, it next attempts to find a linear recurrence of the form in Eq 2, i.e., $T(n) = T(n-d_1) + \cdots + T(n-d_a) + f(n)$. These recurrences define recursive programs whose subproblems are smaller than the original problems by some constant values d_i. For example, the computation of `fibonacci` numbers has a linear recurrence $T(n) = T(n-1) + T(n-2) + 1$.

To infer linear recurrences, we reuse the algorithm shown in Figure 3 and described in §3.1.1. The main difference is that we compute the difference $x - y$ instead of the ratio $\frac{x}{y}$ for each subproblem (line 9). Specifically, for each sublist `valss[i]` of (x, y) tuples, we obtain a list of difference values $x - y$. From this list, we then select a representative value that occurs most frequently. By default Dynaplex sets this frequency parameter to be 95%, i.e., if 95% of the computed differences of a subproblem i is d_i, then d_i is the representative difference value (i.e., the d_i in the term $T(n-d_i)$ of the recurrence). Finally, the algorithm proceeds to find the complexity `fn` as described and combine it with the differences d_i to form a recurrence of the form $T(n) = T(n-d_0) + \cdots + f(n)$ in Eq 2.

For the `mergesort` example with 7 data points, we obtain the difference list $[4, 2, 1, 1, 1, 1]$ from `valss[0]` = $[(7, 3), (3, 1), (2, 1), (4, 2), (2, 1), (2, 1)]$. However, we could not select a representative difference value because 1, the most frequently occurred value, only occurs 4/6 times. Even when running `mergesort` on larger inputs to obtain more traces, we would not be able to find any difference value for any subproblem. The reason is because `mergesort` only has the recurrence

$T(n) = 2T(\frac{n}{2}) + O(n)$ of the form in Eq 1 as shown above and does not have linear recurrences of the form in Eq 2. In general, most recursive programs only have one of these forms but not both.

3.2 Learning Polynomial Relations

Dynaplex uses a dynamic analysis to infer polynomial relations, which represent the runtime complexity of a non-recursive functions or programs from execution traces. This allows us to compute the complexity of the combination or "conquer" step in divide-and-conquer algorithms, e.g., the $f(n)$ part of the recurrences.

First, to obtain traces, we instrument the considered function to capture the number of executed statements by creating a counter variable k and incrementing it in each loop in the function. We also capture the size of the *original* input problem to the program (not necessarily the input problem to the considered function) as our goal is to determine the complexity of the function with respect to the original problem. For example, as shown in Figure 2 we instrument the function `merge` to collect k, which represents the numbers of executed statements in `merge`, and n, the size of the input list of `mergesort`. We note that counter instrumentation is common in many static program analyses (e.g., [Breck et al. 2020; Gulwani 2009; Gulwani et al. 2009b; Hoffmann et al. 2011; Hoffmann and Hofmann 2010; Le et al. 2020; Nguyen et al. 2017b]).

From the obtained traces we use polynomial regression to learn a model capturing the relationship between k and n. Dynaplex uses the regression implementation in `numpy`, which supports nonlinear models by creating nonlinear terms up to a maximum degree value Δ (e.g., $t_1 = n, t_2 = n^2, \cdots, t_\Delta = n^\Delta$) and then finding a linear model over these nonlinear terms (e.g., $k = t_2 + 5t_1$ indicates $k = n^2 - 5n$). Thus, we learn linear models of the form

$$k = c_0 t_0 + c_1 t_1 + \cdots + c_\Delta t_\Delta + d, \tag{3}$$

where t_i are terms and c_i, d are real-valued coefficients.

Dynaplex uses a parameter Δ to indicate the maximum degree that it would consider (by default Dynaplex sets $\Delta = 5$). Also, if the user hypothesizes that the complexity might involve other nonlinear values, e.g., $\lg$, $\sqrt{}$, 2^n, $\cdots$, they can specify additional terms to represent those values for Dynaplex to consider. By default, Dynaplex automatically includes terms representing $\lg$ and $n \lg n$ as they are common program complexity. We note that existing invariant generation techniques [Le et al. 2020; Nguyen et al. 2012] also use this idea of creating terms representing interesting information and find relations among terms.

Selecting Models. A regression technique often produces multiple different models on the same set of data. The technique also gives each computed model an `r-squared` value to indicate how fit that model is with respect to the data. In many cases, however, models with high r-squared values, while fitting the given data well, do not generalize to unseen data and are thus spurious. For example, the model $k = -5.36e^{-21}n^3 + 0.5n^2 - 0.5n$ does not likely indicate that $k = O(n^3)$ because n^3 is associated with a negligible coefficient $-5.36e^{-21}$.

To avoid potentially spurious models, Dynaplex uses a simple heuristic that considers both r-squared values and term coefficients. First, we discard models with low r-squared values because such models do not fit the given data well. We also discard models that have the highest degree terms associated with coefficients that are close to 0. Finally, among the remaining models, we select the one with the highest r-squared value. Note that this heuristics does not always produce the best models, however, it is easy to implement and works well in practice. Moreover, in several cases shown in Section 5, even if we select the incorrect models (either due to the heuristics or poor quality trace data), the technique to solve the combined recurrence described in §4 can still derive the correct asymptotic complexity solutions.

Note that we can also compute polynomial relations to directly capture the complexity of a recursive program. However, a recurrence resembles the structure of a recursive program and thus can represent the program complexity more precisely (e.g., recurrences can capture logarithmic and exponential information such as $\lg_3 n, n^{\lg_2 7}$). We would not be able to achieve such complexity using polynomial relations unless we know a priori that the result would involve such information and thus manually specify terms representing them (e.g., $t = n^{\lg_2 7}$).

4 SOLVING RECURRENCES

By itself, a recurrence does not reveal the asymptotic complexity of a program. For example, we cannot tell that mergesort runs in $O(n \lg n)$ from its recurrence $T(n) = 2T(\frac{n}{2}) + n$ or that fibonacci has an exponential complexity from $T(n) = T(n-1) + T(n-2) + 1$.

To obtain asymptotic complexity, we need to solve the recurrence to obtain a *closed-form* solution—a non-recursive description of a program that satisfies the recurrence. However, solving an arbitrary recurrence is in general difficult. For example, in the *guess-and-check* method [Erickson [n.d.]], we guess the solution and use induction to show that solution is correct. While this method works for many forms of recurrences, it requires the user to manually provide good guesses.

Recurrence trees [Cormen et al. 2009; Erickson [n.d.]] is a common method of guessing the closed form of a recurrence. In this method, we represent the recurrence as a tree and compute the total cost of all nodes in the tree to a closed-form solution. For example, the tree for the recurrence $T(n) = 2T(\frac{n}{2}) + n$ of `mergesort` consists of a root node at level $i = 0$ having the cost n and each node at level $i \geq 1$ having the cost $\frac{n}{2^i}$ (each child of a non-leaf node with a cost m has the cost $\frac{m}{2}$). Moreover, at each level i the tree has 2^i nodes and thus the cost $2^i \cdot \frac{n}{2^i} = n$. Finally, since the tree has $\lg n$ levels (input n is halved each time), the total cost is then $n \cdot \lg n$, corresponding to the complexity $O(n \lg n)$.

Below, we use recurrence trees to develop the complexity bounds for various forms of divide-and-conquer and linear recurrences. Dynaplex implements these results in a pattern-matching method that maps inferred recurrences to different classes of complexity bounds.

4.1 Solving Divide-and-Conquer Recurrences

We use the well-known Master theorem to obtain the complexity bounds of a specific variant of divide-and-conquer algorithm and overapproximation to obtain the upper complexity bounds of general divide-and-conquer recurrences.

4.1.1 Master Theorem. Many divide-and-conquer algorithms have similar subproblem sizes and therefore can be defined using a specific variant of the recurrence of form in Eq 1 in which the factors b_i are the same. Recurrences of this form can be solved using the Master theorem, which is a popular "cookbook" technique that maps specific forms of divide-and-conquer recurrences to different classes of complexity.

Theorem 4.1. *The Master theorem [Cormen et al. 2009; Erickson [n.d.]] defines the solutions representing the* ***tight*** *(Θ) complexity bounds for recurrences of the form*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n). \tag{4}$$

The technique consists of three cases based on the relationships between the cost of solving the subproblems (defined as the ***critical component*** *$\hat{c} = \log_b a$) and the cost of $f(n)$:*

$$complexity = \begin{cases} \Theta(n^{\hat{c}}) & when \;\; f(n) = O(n^c) \;\wedge\; c < \hat{c} \\ \Theta(f(n)) & when \;\; f(n) = \Omega(n^c) \;\wedge\; c > \hat{c} \\ \Theta(n^{\hat{c}} \lg n) & when \;\; f(n) = \Theta(n^{\hat{c}}) \;\wedge\; c = \hat{c} \end{cases}$$

Proof. In the first case, if the cost of solving the subproblems at each tree level increases by a constant factor, then the value of $f(n)$ becomes polynomially smaller than $n^{\hat{c}}$ and thus the complexity is dominated by the cost of the last level, i.e., $n^{\hat{c}}$. In the second case, if the cost of solving the subproblems at each level decreases by a constant factor, the value of $f(n)$ becomes polynomially larger than $n^{\hat{c}}$ and thus the complexity is dominated by the cost of $f(n)$. In the last case, if the cost of solving the subproblem at each level is nearly equal, the value of $f(n)$ is comparable to $n^{\hat{c}}$ and thus the complexity is the product of $f(n)$ and the number of levels in the recurrence tree, i.e., $n^{\hat{c}} \lg n$. □

Using these three cases, Dynaplex implements a pattern matching technique that recognizes recurrences of the form in Eq 4, computes the values of c and $\hat{c}$, and compares them to determine the proper case and the corresponding program complexity. For example, Dynaplex maps the recurrence (i) $T(n) = 8T(\frac{n}{2}) + 1000n^2$ to $\Theta(n^3)$ because $c = 2$ is less than $\hat{c} = \log_2 8 = 3$ (1st case); (ii) $T(n) = 2T(\frac{n}{2}) + n^2$ to $\Theta(n^2)$ because $c = 2$ is greater than $\hat{c} = \log_2 2 = 1$ (2nd case); and (iii) the recurrence of `mergesort` to $\Theta(n \lg n)$ because $c = 1$ is equal to $\hat{c} = \log_2 2 = 1$ (3rd case).

While this Master Theorem-based technique is relatively simple, it allows us to obtain a wide range of complexity bounds, some of which are complex and involve non-polynomial terms. For example, as discussed in §5 Dynaplex got the complexity $O(n^{\lg_2 3}) = O(n^{1.58})$ from the recurrence $T(n) = 3T(\frac{n}{2}) + 1$ of the `Karatsuba` algorithm (1st case of the Master Theorem).

Moreover, the technique is tolerant to certain imprecisions during the learning process (e.g., due to insufficient traces). For example, instead of the recurrence $T(n) = 8T(\frac{n}{2}) + 1000n^2$, we could inaccurately infer $T(n) = 8T(\frac{n}{2}) + n$, but still obtain the correct complexity $\Theta(n^3)$ because both recurrences fall under the 1st case of the Master theorem. §5 provides additional examples from our benchmark programs.

4.1.2 General Divide-and-Conquer Recurrences. The Master theorem only solves recurrences defining divide-and-conquer algorithms with similar problem sizes and does not apply to those of the form in Eq 1. Thus, because Dynaplex supports more general recurrences, it can infer recurrences that the Master theorem cannot solve (e.g., $T(n) = T(\frac{n}{3}) + T(\frac{n}{2}) + O(n)$).

To solve such a general linear recurrence, we perform an overapproximation to obtain an upper bound complexity of the recurrence. More specifically, Dynaplex uses the result in Theorem 4.2 below to convert a general recurrence into a new once that has a form supported by the Master theorem and represents the upper bound complexity of the original recurrence. Then Dynaplex applies the Master theorem to the new recurrence to obtain the upper (O) bound complexity. For example, we convert $T(n) = T(\frac{n}{3}) + T(\frac{n}{2}) + O(n)$ to $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + O(n)$ and apply case 1 of the Master theorem to map it to $O(n \lg n)$.

Theorem 4.2. *For recurrences of the form in Eq 1, i.e., $T(n) = T(\frac{n}{b_1}) + \cdots + T(\frac{n}{b_a}) + f(n)$, we can apply the Master Theorem to the recurrences*

- $L(n) = a \cdot L\left(\frac{n}{\max(b_i)}\right) + 1$ *to obtain the lower (Ω) complexity bound and*
- $U(n) = a \cdot U\left(\frac{n}{\min(b_i)}\right) + f(n)$ *to obtain the upper (O) complexity bound.*

Proof. For recurrences of the form in Eq 1, notice that $T(n)$ is asymptotically positive (i.e., $T(n) > 0$ when n is sufficiently large) and monotonically increases (i.e., $T(n+1) > T(n)$). This is because $d_i \geq 1$ and $f(n) \geq 1$ and monotonically increases (assumptions given in Eq 3.1.2).

We now define the upper and lower bounds of the recurrences of the form in Eq 1 as the following recurrences $L(n)$ and $U(n)$, respectively:

$$L(n) = a \cdot L\left(\frac{n}{\max(b_i)}\right) + 1$$
$$U(n) = a \cdot U\left(\frac{n}{\min(b_i)}\right) + f(n)$$

These are the upper and lower bounds because (i) as $T(n)$ is increasing and $b_i \geq 1$, we have $T\left(\frac{n}{\max(b_i)}\right) \leq T\left(\frac{n}{b_i}\right) \leq T\left(\frac{n}{\min(b_i)}\right)$ and (ii) because $f(n) \geq 1$, $L(n) \leq T(n) \leq U(n)$.

Then, by applying the Master Theorem to $L(n)$ and $U(n)$ (both of which have the form in Eq 1), we get the **lower** (Ω) and **upper** (O) complexity bounds, respectively. □

4.2 Solving Linear Recurrences

The technique described in §4.1 focuses on divide-and-conquer recurrences and does not apply to linear recurrences of the form in Eq 3.1.2, where a subproblem is not a factor of the original problem. Examples of such recurrences include $T(n) = T(n-1) + T(n-2) + 1$ and $T(n) = 2T(n-1) + 1$ of the Fibonacci and Tower of Hanoi programs, respectively. Dynaplex maps recurrences of this form and several of its variants to different classes of asymptotic complexity using the results established below.

4.2.1 General Linear Recurrences. Here we establish the lower and upper complexity bounds for linear recurrences of the form in Eq 2.

Theorem 4.3. *For the recurrences of the form in Eq 2, i.e., $T(n) = T(n-d_1) + \cdots + T(n-d_a) + f(n)$, the **lower** ($\Omega$) and **upper** ($O$) complexity bounds are:*

$$\textit{complexity} = \begin{cases} \Omega(n) & O(nf(n)) \quad \textit{when } a = 1 \\ \Omega\left(a^{\lfloor \frac{n}{\max(d_i)} \rfloor}\right) & O(a^n f(n)) \quad \textit{when } a > 1 \end{cases}$$

Proof. Note that the first part of the proof is similar to the proof given for Theorem 4.2. First, note that $T(n)$ is asymptotically positive and monotonically increases because $d_i \geq 1$ and $f(n) \geq 1$ and monotonically increases. We define the following recurrences $L(n)$ and $U(n)$ as the upper and lower bounds of the recurrences of the form in Eq 2, respectively:

$$L(n) = a \cdot L(n - \max(d_i)) + 1$$
$$U(n) = a \cdot U(n - \min(d_i)) + f(n)$$

These are the upper and lower bounds because (i) as $T(n)$ is increasing and $d_i \geq 1$, we have $T(n - \max(d_i)) \leq T(n - d_i) \leq T(n - \min(d_i))$ and (ii) because $f(n) \geq 1$, $L(n) \leq T(n) \leq U(n)$.

Next, we compute the upper bound of the recurrence $U(n)$. The recursion tree of $U(n)$ has the height $h = \lfloor \frac{n}{\min(d_i)} \rfloor + 1$, and each non-leaf node has a children. Hence, the closed form of $U(n)$ is

$$U(n) = \sum_{j=0}^{h-1} \left(a^j \cdot f(n - i \cdot \min(d_i))\right)$$

Because $n - i \cdot \min(d_i) \leq n$ and $f(n)$ is increasing, $f(n - i \cdot \min(d_i)) \leq f(n)$. Hence,

$$U(n) \leq \sum_{j=0}^{h-1} \left(a^j \cdot f(n)\right) = \begin{cases} \frac{a^h \cdot f(n)}{a-1} & \text{if } a > 1 \\ h \cdot f(n) & \text{if } a = 1 \end{cases}$$

Because $h \leq n$, the **upper bound** of $U(n)$ (and thus the recurrences of the form in Eq 2) is $O(n \cdot f(n))$ when $a = 1$ and $O(a^n \cdot f(n))$ when $a > 1$.

Finally, we use the similar reasoning to determine the lower bound of the recurrence $L(n)$. The recurrence tree of $L(n)$ has the height $g = \lfloor \frac{n}{\max(d_i)} \rfloor + 1$, thus the closed form of $L(n)$ is

$$L(n) = \sum_{j=0}^{g-1} \left(a^j\right) = \begin{cases} \frac{a^g}{a-1} & \text{if } a > 1 \\ g & \text{if } a = 1 \end{cases}$$

Hence, the **lower bound** of $L(n)$ (and thus the recurrences of the form in Eq 2) is $\Omega(n)$ when $a = 1$ and $\Omega\left(a^{\lfloor \frac{n}{\max(d_i)} \rfloor}\right)$ when $a > 1$.

□

4.2.2 Linear Recurrence Variants. From Theorem 4.3, we can derive the complexity for several popular variants of the form in Eq 2. This allows Dynaplex to quickly solve linear recurrences that frequently appear in practice. We also can establish the tight (Θ) complexity bounds in some cases.

Corollary 4.4. *For the following forms of recurrences, we have the complexity bounds:*

Recurrence Form	*Complexity Bounds*	*Constraints on Eq 2*
$T(n) = T(n-d) + f(n)$	$O(nf(n))$	$a = 1$
$T(n) = T(n-d_1) + T(n-d_2) + f(n)$	$O(2^n f(n))$	$a = 2$
$T(n) = T(n-d_1) + 1$	$\Theta(n)$	$a = 1 \wedge f(n) = 1$
$T(n) = T(n-d_1) + \cdots + T(n-d_a) + 1$	$\Theta\left(a^{\lfloor \frac{n}{d_0} \rfloor}\right)$	$a > 1 \wedge f(n) = 1 \wedge d_i$ *are the same*

Proof. The worst-case complexity bounds of the first two forms are obtained directly from Theorem 4.3. For the third form when $a = 1$, the upper and lower bounds, as shown in Theorem 4.3, are $\Omega(n)$ and $O(nf(n))$, respectively. Moreover, because $f(n) = 1$, the tight bound is $\Theta(n)$. For the fourth form when $a > 1$, the upper and lower bounds are $\Omega\left(a^{\lfloor \frac{n}{\max(d_i)} \rfloor}\right)$ and $O\left(a^{\lfloor \frac{n}{\min(d_i)} \rfloor} f(n)\right)$ as shown in Theorem 4.3. Moreover, because $f(n) = 1$ and $d_0 = \min(d_i) = \max(d_i)$, the tight bound is $\Theta\left(a^{\lfloor \frac{n}{d_0} \rfloor}\right)$.

□

These forms of linear recurrences appear in many popular recursive algorithms. The first and second forms map to worst-case (O) complexity. The first maps recurrences such as $T(n) = T(n-1) + O(n)$ of popular sorting algorithms (e.g., `bubble` and `insertion`) to the quadratic complexity $O(n^2)$. The second maps recurrences with two subproblems to an exponential complexity. Note that even when $f(n)$ is a constant, these recurrences still have an exponential upper bound complexity $O(a^n)$. For example, `fibonacci` has the recurrence $T(n) = T(n-1) + T(n-2) + 1$ of this form and thus an exponential complexity.

The third and fourth forms map recurrences to tight (Θ) complexity bounds under a couple of additional constraints are common in many programs (e.g., $f(n) = 1$ and d_i's are the same). The third maps recurrences of common operations such as traversing lists or trees to a linear complexity. Finally, the fourth maps recurrences such as $T(n) = 2T(n-1) + 1$ of Hanoi to $\Theta(2^n)$.

5 EVALUATION

Dynaplex is implemented in Python and uses the regression function `polyfit` from `numpy` [Harris et al. 2020] to learn polynomial models. The tool is language-agnostic and works directly on given program traces. Dynaplex detects the types of traces and performs the corresponding computations (e.g., compute recurrences for traces represented by execution trees and polynomial models for

traces involving program counters and input sizes). After obtaining recurrences, Dynaplex solves them for closed-form solutions representing asymptotic complexity bounds.

Below we evaluate Dynaplex on its effectiveness at discovering program complexity and compare it with several complexity analysis tools. The experiments and results reported here were obtained on an AMD Ryzen 16-core machine with 32 GB of RAM running Linux. Dynaplex and all benchmark programs and results are available at https://github.com/caecus-commits/dynacom.

5.1 Benchmarks and Setup

Benchmarks. To evaluate Dynaplex we use 37 benchmark programs shown in Table 1. The programs are collected from various sources (e.g., OCaml's 99 problems [Ocaml.org 2021] and the literature [Breck et al. 2020; Chatterjee et al. 2019]) and written in C++, Python, and OCaml.

Most of these programs are classical recursive algorithms such as Fibonacci, Tower of Hanoi, different sorting algorithms, multiplication (Karatsuba and Strassen), and others [1]. The programs are relatively small (most are less than 100 lines of code, a couple are close to 200, e.g., `ConvexHull`, `Strassen`). However, they contain non-trivial data structures and a wide range of complexity bounds (e.g., logarithmic, nonlinear polynomial degrees, and exponential). The programs also have various forms of inputs: lists for `BinarySearch` or sorting algorithms, integers for `Factorial` and `Karatsuba`; trees for `Inorder`; stack for `ReverseStack`; logical expressions for `TruthTable`; 2-D matrices for `Strassen`, etc.

Setup. For these experiments, we instrument the programs to capture necessary traces as described in §3. Next, we run the instrumented programs on randomly generated inputs to obtain execution traces for Dynaplex to analyze. Because our benchmark programs take various forms of inputs (e.g., integers, strings, lists, matrix, etc), the generated inputs vary for each program (additional details in §5.2).

For each program, we run Dynaplex five times and report the median results. Dynaplex is highly automated but can be customized using the maximum degree value for regression learning discussed in §3.2 (set to 5 by default) and the frequency value for choosing the representative difference value discussed in §3.1.2 (set to 95% by default). All these parameters can be changed by Dynaplex's user; we chose these values based on our experience.

5.2 Dynaplex's Results

5.2.1 Table Format. Table 1 lists the results for the 37 benchmark programs. The first four columns show Dynaplex's results. Column **Inp** (S, N) shows information about the inputs used to generate traces: S is the maximum size of the inputs and N is the number of inputs. For example, for sorting algorithms, we randomly generate 100 lists of lists of various sizes with the maximum list having 500 elements, and for programs whose inputs are integers, e.g., `isPrime`, we run the programs with

[1] `PowerBinary` calculates 3^n mod 2^{64}. `ConvexHull` finds a convex hull of a set of 2D points. `BuildMSTree, QueryMSTree, MemMSTree` builds, queries, and reports memory usage of a Merge Sort Tree, respectively. `BalTenary` converts an integer to balanced tenary form (e.g., 2021 to 10Z100ZZ). `QuickSelect` finds the median in an unsorted list of integers. `PancakeSort` sorts an array a by reversing the subarray $a[0..i]$ for some i at each step. `PermGen` generates a permuatation length n from its alphabetical index. `PermIndex` gives the alphabetical index of a permutation. `Tenary012` finds all numbers not containing "012" in its tenary representation. `Compress` eliminates consecutive duplicates of list elements. `EulerTotient` calculates the number of coprime integers. `InOrder` and `InOrderBT` performs inoder traversal of a normal and balanced binary tree, respectively. `Sort/Reverse Stack` sort/reverse a stack. `TruthTable` computes the truth table of a given logical expression. `ClosestPair` finds a pair of points that have shortest Euclidean distance. `BallBins3` enumerates ways to place n labelled balls into 3 labelled bins. `IsPrime` checks if a integer is a prime number. `BSTCopy` copies a binary tree (complexity expressed in terms of tree height). `SubsetSum` computes the size of subset whose sum is equal to the given sum. `QsortSteps` implements quicksort (with the pivot being randomly selected) and `QsortCalls` models the recursive calls made by quicksort (and thus only has a linear $O(n)$ complexity).

Table 1. Results of Dynaplex on 37 recursive programs.

Programs	Inp (S,N)	T(s)	Dynaplex's Results Recurrence	Dynaplex's Results Complexity	Ground Truth Recurrence	Ground Truth Complexity
BinarySearch	500,100	0.67	✓	✓	$T(n) = T(\frac{n}{2}) + 1$	$O(\lg n)$
PowerBinary	500,100	1.83	✓	✓	$T(n) = T(\frac{n}{2}) + 1$	$O(\lg n)$
GCD	500,100	1.85	~~$T(n) = T(\frac{n}{6}) + 1$~~	✓	$T(n, m) = T(m, n\%m) + 1$	$O(\lg n)$
BalTernary	500,100	1.81	✓	✓	$T(n) = T(\frac{n}{3}) + 1$	$O(\lg_3 n)$
QueryMSTree	500,100	1.34	~~$T(n) = T(\frac{n}{6}) + T(\frac{n}{2}) + 1$~~	~~$O(n)$~~	–	$O(\lg^2 n)$
Factorial	20,100	1.1	✓	✓	$T(n) = T(n-1) + 1$	$O(n)$
QuickSelect	500,100	1.33	✓	✓	$T(n) = T(\frac{n}{2}) + n$	$O(n)$
Compress	500,100	0.76	✓	✓	$T(n) = T(n-1) + 1$	$O(n)$
IsPrime	500,100	0.72	✓	✓	$T(n) = T(n-1) + 1$	$O(n)$
EulerTotient	500,100	0.69	✓	✓	$T(n) = T(n-1) + 1$	$O(n)$
InOrder	500,100	7.09	✓	✓	$T(n) = 2T(\frac{n}{2}) + 1$	$O(n)$
InOrderBT	500,100	7.12	✓	✓	$T(n) = 2T(\frac{n}{2}) + 1$	$O(n)$
QsortCalls	500,100	1.57	~~$T(n) = T(\frac{n}{14}) + T(\frac{n}{3}) + n$~~	✓	$T(n) = T(n-1) + 1$	$O(n)$
ClosestPair	500, 100	0.98	✓	✓	$T(n) = 2T(\frac{n}{2}) + n$	$O(n \lg n)$
MergeSort	500,100	1.47	✓	✓	$T(n) = 2T(\frac{n}{2}) + n$	$O(n \lg n)$
Convex Hull	500,100	1.71	✓	✓	$T(n) = 2T(\frac{n}{2}) + n$	$O(n \lg n)$
BuildMSTree	500,100	1.81	✓	✓	$T(n) = 2T(\frac{n}{2}) + n$	$O(n \lg n)$
MemMSTree	500,100	1.58	✓	✓	$T(n) = 2T(\frac{n}{2}) + n$	$O(n \lg n)$
HeapSort	500,100	2.97	~~$T(n) = T(n/2) + n$~~	~~$O(n)$~~	$T(n) = T(n-1) + \lg n$	$O(n \lg n)$
Karatsuba	20,100	1.11	✓	✓	$T(n) = 3T(\frac{n}{2}) + 1$	$O(n^{\lg_2 3})$
Strassen	32,100	11.80	✓	✓	$T(n) = 7T(\frac{n}{2}) + n^2$	$O(n^{\lg_2 7})$
BubbleSort	500,100	0.97	✓	✓	$T(n) = T(n-1) + n$	$O(n^2)$
InsertionSort	500,100	1.26	✓	✓	$T(n) = T(n-1) + n$	$O(n^2)$
SelectionSort	500,100	1.28	✓	✓	$T(n) = T(n-1) + n$	$O(n^2)$
PancakeSort	500,100	1.44	✓	✓	$T(n) = T(n-1) + n$	$O(n^2)$
PermGen	500,100	1.44	✓	✓	$T(n) = T(n-1) + n$	$O(n^2)$
PermIndex	20,100	2.00	✓	✓	$T(n) = T(n-1) + n$	$O(n^2)$
SortStack	500,100	4.05	✓	✓	$T(n) = T(n-1) + n$	$O(n^2)$
ReverseStack	500,100	3.85	✓	✓	$T(n) = T(n-1) + n$	$O(n^2)$
QsortSteps	500,100	1.57	~~$T(n) = T(\frac{n}{15}) + T(\frac{n}{3}) + 1$~~	~~$O(n)$~~	$T(n) = T(n-1) + n$	$O(n^2)$
BSTCopy	10, 100	3.2	✓	✓	$T(n) = 2T(n-1) + 1$	$O(2^n)$
Fibonacci	10,100	4.35	✓	✓	$T(n) = T(n-2) + T(n-1) + 1$	$O(2^n)$
Hanoi	10,100	4.10	✓	✓	$T(n) = 2T(n-1) + 1$	$O(2^n)$
SubsetSum	10, 100	3.3	✓	✓	$T(n) = 2T(n-1) + 1$	$O(2^n)$
TruthTable	10,100	5.2	✓	✓	$T(n) = 2T(n-1) + 1$	$O(2^n)$
BallBins3	10, 100	3.2	✓	✓	$T(n) = 3T(n-1) + 1$	$O(3^n)$
Tenary012	10,100	2.15	✓	✓	$T(n) = 3T(n-1) + 1$	$O(3^n)$

100 random integers whose values are at most 500. For some programs, especially those that are exponential in the input, we randomly generate integers with small values, e.g., 10 for `Fibonacci` and `Hanoi`, because applying these programs on large values would cause the program to run slowly or produce a large number of traces. Column **T**(s) shows the total run time in seconds (both the runtime of the program on inputs and of Dynaplex).

The next two columns **Recurrence** and **Complexity** show the discovered recurrences and complexity bounds: ✓ indicates Dynaplex's results match the *ground truth* (i.e., actual, recurrences and complexity bounds of these programs) while ✗ indicates that Dynaplex found the incorrect result R. The last two columns **Recurrence** and **Complexity** show the ground truth. Most of these programs are standard algorithms with well-known recurrences and worst-case complexity bounds. For others, e.g., `QueryMSTree` and `TruthTable` we manually analyze the programs to determine the ground truths. Note that `QueryMSTree` cannot be defined by a recurrence (and hence is indicated by a –). This program runs in $O(\lg^2 n)$ and has different recurrences depending on recursive depths.

```
def karatsuba(x, y):

  lenx = len(str(x))
  leny = len(str(y))

  if lenx==1 or leny==1:
    return x*y

  n = max(lenx,leny)
  n2 = n / 2

  a = x / 10 ** n2
  b = x % 10 ** n2

  c = y / 10 ** n2
  d = y % 10 ** n2

  ac = karatsuba(a,c)
  bd = karatsuba(b,d)

  ad_plus_bc =
    karatsuba(a+b, c+d)
    - ac - bd

  prod = ac * 10**(2*n2)
    + (ad_plus_bc * 10**n2)
    + bd

  return prod
```

Fig. 4. Karatsuba algorithm

Also note we present the complexity results of Dynaplex as upper bounds (O), even though Dynaplex produces more precise bounds (Θ) in many cases as shown in §4. This presentation makes it easier to compare Dynaplex's results with worst-case complexity ground truths.

5.2.2 Results. In summary, out of 37 programs, Dynaplex got correct results for both recurrences and complexity bounds for 32 programs; got incorrect results for both recurrences and complexity bounds for 3 programs (`QueryMStree`, `HeapSort`, `QsortSteps`); and got incorrect recurrences but correct complexity results for 2 programs (`GCD`, `QsortCalls`. The running time for most programs are about 2 seconds. `Strassen` takes 11.80s because the program has much longer runtime than others (inputs of `Strassen` are 32x32 matrices).

Success. Dynaplex found the correct recurrences and complexity for 32/37 programs. These programs have a wide range of complexity bounds including logarithmic, nonlinear, and exponential polynomials. Dynaplex also discovers complexities involving non-polynomial terms, e.g., $O(n^{\lg_2 3})$ of `Karatsuba` and $O(n^{\log_2 7})$ for `Strassen`.

Dynamic analysis allows Dynaplex to handle complex programs and obtain precise results. For example, consider the `karatsuba` algorithm shown in Figure 4, adapted from [Jayaraman 2015]. This algorithm multiplies two n-digit numbers x, y using at most $n^{\lg_2 3} \approx n^{1.58}$ single-digit multiplications and thus `karatsuba` is asymptotically faster than standard multiplication algorithms, which typically require n^2 single-digit multiplications. As shown in Figure 4, the program uses non-trivial modulo and exponential operations, which can be difficult to analyze using static analysis. However, by analyzing the program runs on random inputs, Dynaplex was able to infer the correct recurrence $T(n) = 3T(\frac{n}{2}) + 1$ and solve it to obtain the correct complexity $O(n^{\log_2 3})$ or $O(n^{1.58})$.

We note that other dynamic and polynomial-based techniques, e.g., the mentioned SymInfer tool [Nguyen et al. 2017b] and the regression learning technique described in §3.2 would not be to obtain such a result because $n^{1.58}$ is *not* a polynomial. Interestingly, the template-based approach described in [Chatterjee et al. 2019] gives $O(n^{1.6})$ for `karatsuba`. While this is correct, it is less precise than Dynaplex's $O(n^{1.58})$ result (additional comparisons are given in §5.3).

Fail. Dynaplex failed to generate correct results for both recurrences and complexity bounds for 3 programs (`HeapSort`, `QsortSteps`, `QueryMStree`). The main reason is because the recursive calls and subproblems in these programs vary. In `HeapSort` the recursive calls are non-deterministic and are executed only under certain conditions. In `QsortSteps`, the sizes of the subproblems depend on the randomly selected pivot. For `QueryMSTree`, the program exhibits different recurrences in different recursion depths. In short, for these programs, the obtained execution trees are not

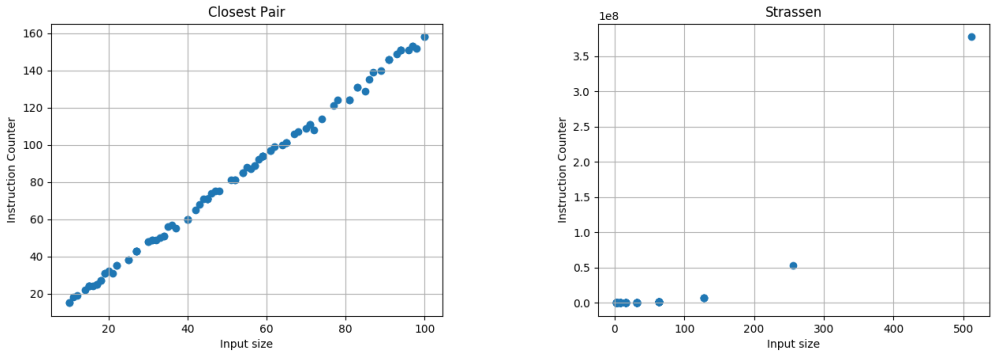

Fig. 5. Difficulty in reasoning about program complexity from growth plots

consistent (e.g., some levels have m children while others have n children, etc) and therefore cause Dynaplex to learn imprecise recurrences that lead to incorrect complexity bounds.

Note that for `QsortSteps` and `QueryMSTree`, Dynaplex detected that the obtained divide-and-conquer recurrence are not support by the Master theorem and applied the overapproximation techinque described in §4.1.2 to solve them.

Incorrect Recurrences but Correct Complexity. For 2 programs (`GCD` and `QsortCalls`, Dynaplex obtained incorrect recurrences, but still can solve them to get the correct complexity.

The Python program on the right computes the greatest common divisor (`GCD`). While being our smallest benchmark program, `GCD` has a non-trivial recurrence $T(n,m) = T(m, n\%m) + 1$ that is *not* support by Dynaplex. Thus, it is not suprise that Dynaplex generates an incorrect recurrence $T(n) = T(\frac{n}{6}) + 1$. Nonetheless, Dynaplex was able to solve this incorrect recurrence using the Master theorem and got the correct complexity $O(n)$.

```
def gcd(a,b):
    if(b==0):
      return a
    else:
      return gcd(b,a%b)
```

A similar situation happened with `QsortCalls`, where we inferred an incorrect recurrence from inconsistent traces due to the use of random pivot selection for partitioning. Note that this recurrence $T(n) = T(\frac{n}{14}) + T(\frac{n}{3}) + n$ is also not supported by the Master theorem, and thus Dynaplex converted it to $T(n) = T(\frac{n}{3}) + T(\frac{n}{3}) + n$ and applied the Master theorem to get $O(n)$, which is the correct complexity bound of the program.

Thus, Dynaplex was able to recover certain inaccuracies from the inferred recurrences. There is no guarantee that this would work (e.g., we likely get "lucky" that the incorrect recurrences mapped to correct solutions), but this is still an interesting behavior that we observe in the work.

5.2.3 Analyzing Growth Plots. Instead of using a technique such as Dynaplex, it is tempting to plot and analyze the relationship between the size of the inputs and the number of instructions or statements executed when running the program on random or even WCET inputs (e.g., as done in several works [Goldsmith et al. 2007; Zaparanuks and Hauswirth 2012]). However, while such plots can help the user gain visual intuition about the programs' growth, they can also mislead the user in many cases, especially those that involve subtle complexity bounds.

For example, Figure 5 shows that it is not easy to reason from the plots that the complexity bounds of `ClosestPair` and `Strassen` are $O(n \lg n)$ and $O(n^{\lg_2 7})$, respectively. In contrast, an

Table 2. Comparison with other tools.

Benchmark	NPWCARP	CHORA	Dynaplex	Ground Truth
QsortCalls	✓	~~$O(2^n)$~~	✓	$O(n)$
Mergesort	✓	✓	✓	$O(n \lg n)$
ClosestPair	✓	–	✓	$O(n \lg n)$
Karatsuba	~~$O(n^{1.6})$~~	✓	✓	$O(n^{\lg_2 3})$
Strassen	~~$O(n^{2.9})$~~	✓	✓	$O(n^{\lg_2 7})$
QsortSteps	✓	~~$O(n2^n)$~~	~~$O(n)$~~	$O(n^2)$
Fibonacci	–	✓	✓	$O(2^n)$
Hanoi	–	✓	✓	$O(2^n)$
SubsetSum	–	✓	✓	$O(2^n)$
BSTCopy	–	✓	✓	$O(2^n)$
BallBins3	–	✓	✓	$O(3^n)$
Ackermann	–	–	–	$O(iA(i,n))$

approach such as Dynaplex explicitly provides closed-form solutions representing complexity bounds and thus mitigates potential misunderstandings from the user.

5.3 Comparing to Others

We compare Dynaplex with two static complexity analysis tools: CHORA [Breck et al. 2020], which uses templates and constraint solving to find recurrences and runtime complexity, and NPWCARP [Chatterjee et al. 2019], which uses templates and ranking functions to compute worst-case complexity. For this comparison, we use the 12 benchmark programs in CHORA.

Note that we were not able to fully run CHORA (even when using the provided virtual machine, we still cannot run the tool on its own benchmark programs) and thus used the reported results in [Breck et al. 2020], which states that CHORA outperformed several other complexity analyzers (e.g., ICRA [Kincaid et al. 2017]). For NPWCARP, we were not able to run the tool on many programs because it does not support the languages of these programs (e.g., modulo operations or complex data structures).

Results. Table 2 shows the results of the three tools comparing to the ground truth complexity. Similarly to the format of Table 1, ✓ and ~~✗~~ indicate that the result R matches and does not match the ground truth, respectively.

In summary, other than the `Ackermann` program that no tool can analyze, among the remaining 11 programs, Dynaplex got one mismatch (`QsortSteps`); CHORA fails to produce result for 1 program (`ClosestPair`) and got 2 mismatches (the two quicksort programs `QsortCalls` and `QsortSteps`); and NPWCARP did not run on 5 programs and got 2 mismatches (`Karatsuba` and `Strassen`).

For 8 programs, both CHORA and Dynaplex produced precise complexity bounds that match the ground truths. For `QsortSteps`, both tools failed to obtain the ground truth complexity $O(n^2)$. Dynaplex got an incorrect complexity $O(n)$ due to random pivot section as explained in §5.2 while CHORA got a correct but imprecise and strange exponential $O(n2^n)$ result [2]. CHORA also got this exponential complexity for `QsortCalls`. For `closestPair`, both Dynaplex and `NPWCARP` got the ground truth $O(n \lg n)$ while `CHORA` did not produce a result.

[2]The CHORA paper [Breck et al. 2020] also shows that the expected complexity is quadratic but does not explain why CHORA produced an exponential bound.

Comparing to NPWCARP, both CHORA and Dynaplex worked with more programs. NPWCARP got less precise bounds for `strassen` and `karatsuba` as these involve non-polynomial terms such as $n^{\lg_2 7}$. Moreover, NPWCARP does not support exponential complexity such as those of `Fibonacci` and `Hanoi` and other exponential programs listed in Table 2.

All three tools failed on the `Ackermann` function, which has the unusual recurrence $A(i, n) = A(i-1, A(i, n-1))$ for $i, n > 0$ and the $O(iA(i, n))$ complexity [Grossman and Zeitman 1988]. NPWCARP does not have a template supporting this complexity and CHORA's generated recurrence invariants were not able to establish this complexity. Dynaplex was not able to obtain execution traces for this program because its computation is extremely intensive, even on small inputs.

In general, we found that it is difficult to directly compare these tools. Static analysis techniques such as CHORA and NPWCARP aim to reason about all program paths soundly, but doing so is often expensive and is only possible for restricted classes of properties, recurrences, or programs (e.g., NPWCARP does not support many programs). Dynamic analyses such as Dynaplex limit their attention to only some of a program's paths, and thus provide no guarantee that those invariants are correct, but can often be more efficient and produce more expressive results (e.g., non-polynomial complexity bounds of `Karatsuba` and `Strassen`). Our experiments show that Dynaplex is in general efficient and produce accurate results, even when using just random inputs.

6 RELATED WORKS

Dynamic Invariant Generation. Dynaplex is inspired by approaches that use dynamically inferred invariants for complexity analysis. SymInfer and NumInv [Nguyen et al. 2017a,b] compute nonlinear polynomial invariants for Java programs from execution traces and use either a symbolic execution tool or symbolic states extracted from symbolic execution to refute spurious invariants and generate counterexample inputs. An application of these invariants is that they can be used to capture the program complexity of several imperative programs. More recently, Dynamite [Le et al. 2020] extends these nonlinear invariants to represent ranking functions and recurrence sets for the analysis of program termination and non-termination behaviors.

Nguyen et al. [2020] propose using dynamic analysis to infer recurrences and applies the Master theorem to solve them for complexity bounds. However, this work mainly provides the high-level ideas and does not provide any algorithms or formal analyses. It also does not have an implementation, i.e., all processes from analyzing traces, inferring and solving recurrences, etc are done by hand. Dynaplex fully developed, implemented, and thoroughly evaluated the idea of using dynamically inferred recurrences to discover program complexity.

Static Analyses. Several works use static analyses to discover the complexity of imperative programs. LOOPUS [Sinn et al. 2014, 2017] uses *difference constraints* for complexity and resource bound analysis. KoAT [Brockschmidt et al. 2016] uses polynomial ranking functions by combining symbolic runtime and size bounds to generate invariants for complexity analysis of imperative integer programs. SPEED [Gulwani 2009; Gulwani et al. 2009a,c] generates linear invariants to compute non-linear and disjunctive bounds. SPEED also uses abstract interpretation to compute non-linear bounds involving operators such as logarithm, exponentiation, multiplication, square-root, and max [Gulavani and Gulwani 2008].

Several works develop static complexity analyses for recursive programs. The NPWCARP tool [Chatterjee et al. 2019] mentioned in §5.3 uses ranking functions and linear programming to compute non-polynomial worst-case upper bounds of both imperative and recursive programs. This technique requires the user to input the invariant template and ranking functions to work. de Oliveira et al. [2016]; Farzan and Kincaid [2015]; Kincaid et al. [2019] compute recurrence relations from pre- and post-states of loops and use them to generate loop invariants to establish

complexity bounds. ICRA [Kincaid et al. 2017] uses tensor product for recurrence-based invariant generation for linear recursion. The CHORA tool [Breck et al. 2020] used in §5.3 builds on ICRA and combines templates-based and recurrence-based techniques to generate complexity bounds. Several other techniques focus on recurrence relations for worst-case complexity analysis [Albert et al. 2009, 2008, 2007; Flajolet et al. 1991; Grobauer 2001]. For example, Albert et al. [2008] solve recurrence relations using evaluation trees and can derive the complexity bound of merge sort.

In contrast to these works, Dynaplex uses dynamic analysis to support programs written in various languages and focuses on generating and solving recurrence invariants to capture the complexity of a recursive program. In §5.3 we also compare Dynaplex to NPWCARP and CHORA.

WCET Inputs and Verification. Another direction taken in complexity analysis is the generation of input that triggers the worst-case execution behavior of programs. The work in [Lemieux et al. 2018; Petsios et al. 2017] uses evolutionary fuzzing techniques to generate inputs with higher total execution path length. Users can run these inputs through a standard profiling tool to produce profiles or plots visualizing the growth [Goldsmith et al. 2007; Graham et al. 1982; Nethercote and Seward 2003; Zaparanuks and Hauswirth 2012].

There are also works on verifying given complexity bounds [Srikanth et al. 2017]. In particular, the TiML functional language [Wang et al. 2017] allows a user to specify time complexity as types and then uses type checking to verify the specified complexity.

In this work, we apply dynamic analysis on randomly generated inputs and therefore have no soundness guarantee on the inferred recurrence results. In future work, we will explore using these verification techniques and WCET inputs to improve the confidence level of Dynaplex's results.

7 CONCLUSION

We propose a Dynaplex, a new dynamic analysis technique and tool to analyze the asymptotic runtime complexity of recursive programs. Dynaplex learns divide-and-conquer and linear recurrences describing recursive programs and then uses pattern matching to map the recurrences to closed-form solutions representing various asymptotic complexity bounds.

Our evaluation shows that Dynaplex is effective in inferring accurate recurrences and complexity bounds for challenging recursive programs, even when just using randomly generated inputs. We make Dynaplex and all benchmark programs and results available at https://github.com/caecus-commits/dynacom.

REFERENCES

Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, German Puebla, Diana Ramírez, G Román, and Damiano Zanardini. 2009. Termination and cost analysis with COSTA and its user interfaces. *Electronic Notes in Theoretical Computer Science* 258, 1 (2009), 109–121.

Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic inference of upper bounds for recurrence relations in cost analysis. In *International Static Analysis Symposium*. Springer, 221–237.

Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. 2007. Cost Analysis of Java Bytecode. In *European Symposium on Programming*. Springer, 157–172.

Thomas Ball and Sriram K. Rajamani. 2001. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN Symposium on Model Checking of Software*. Springer, 103–122.

Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. 2020. Templates and Recurrences: Better Together. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 688–702.

Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2016. Analyzing runtime and size complexity of integer programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 38, 4 (2016), 1–50.

Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 463–473.

Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2019. Non-polynomial worst-case analysis of recursive programs. *ACM Transactions on Programming Languages and Systems* 41, 4 (2019), 1–52.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices* 37, 5 (2002), 57–68.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Steven de Oliveira, Saddek Bensalem, and Virgile Prevosto. 2016. Polynomial invariants by linear algebra. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 479–494.

Jeff Erickson. [n.d.]. Solving Recurrences. http://jeffe.cs.illinois.edu/teaching/algorithms/notes/99-recurrences.pdf, accessed on 2021-04-15.

Michael D. Ernst. 2000. *Dynamically detecting likely program invariants*. Ph.D. Dissertation. University of Washington.

Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* (2007), 35–45.

Azadeh Farzan and Zachary Kincaid. 2015. Compositional recurrence analysis. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 57–64.

Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. 1991. Automatic average-case analysis of algorithms. *Theoretical Computer Science* 79, 1 (1991), 37–109.

Simon F Goldsmith, Alex S Aiken, and Daniel S Wilkerson. 2007. Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 395–404.

Susan L Graham, Peter B Kessler, and Marshall K McKusick. 1982. Gprof: A call graph execution profiler. *ACM Sigplan Notices* 17, 6 (1982), 120–126.

Bernd Grobauer. 2001. Cost recurrences for DML programs. *ACM SIGPLAN Notices* 36, 10 (2001), 253–264.

Jerrold W Grossman and R Suzanne Zeitman. 1988. An inherently iterative computation of Ackermann's function. *Theoretical computer science* 57, 2-3 (1988), 327–330.

Bhargav S Gulavani and Sumit Gulwani. 2008. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In *International Conference on Computer Aided Verification*. Springer, 370–384.

Sumit Gulwani. 2009. SPEED: Symbolic Complexity Bound Analysis. In *Computer Aided Verification*. Springer-Verlag, 51–62.

Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009a. Control-flow Refinement and Progress Invariants for Bound Analysis. In *Programming Language Design and Implementation*. 375–385.

Sumit Gulwani, Krishna K Mehra, and Trishul Chilimbi. 2009b. Speed: precise and efficient static estimation of program computational complexity. *ACM Sigplan Notices* 44, 1 (2009), 127–139.

Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009c. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages*. ACM, 127–139.

Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernàndez del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. 2002. Lazy Abstraction. In *Principles of Programming Languages*. ACM, 58–70.

Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate amortized resource analysis. In *Principles of Programming Languages*. 357–370.

Jan Hoffmann and Martin Hofmann. 2010. Amortized resource analysis with polynomial potential. In *European Symposium on Programming*. Springer, 287–306.

Anirudh Jayaraman. 2015. Karatsuba Multiplication Algorithm - Python Code. https://pythonandr.com/2015/10/13/karatsuba-multiplication-algorithm-python-code/, accessed on 2021-04-05.

Zachary Kincaid, Jason Breck, Ashkan Forouhi Boroujeni, and Thomas Reps. 2017. Compositional recurrence analysis revisited. *ACM SIGPLAN Notices* 52, 6 (2017), 248–262.

Zachary Kincaid, Jason Breck, John Cyphert, and Thomas Reps. 2019. Closed forms for numerical loops. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. 2020. DynamiTe: dynamic termination and non-termination proofs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–30.

Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

788–801.

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 254–265.

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Principles of Programming Languages*. ACM, 42–54.

Nicholas Nethercote and Julian Seward. 2003. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science* 89, 2 (2003), 44–66.

ThanhVu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017a. A Counterexample-guided Approach to Finding Numerical Invariants. In *Foundations of Software Engineering*. ACM, 605–615.

ThanhVu Nguyen, Matthew Dwyer, and William Visser. 2017b. SymInfer: Inferring Program Invariants using Symbolic States. In *Automated Software Engineering*. IEEE, 804–814.

ThanhVu Nguyen, Didier Ishimwe, Alexey Malyshev, Timos Antonopoulos, and Quoc-Sang Phan. 2020. Using dynamically inferred invariants to analyze program runtime complexity. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Security from Design to Deployment*. 11–14.

ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using Dynamic Analysis to Discover Polynomial and Array Invariants. In *International Conference on Software Engineering*. IEEE, 683–693.

Ocaml.org. 2021. 99 Problems in OCaml. https://ocaml.org/learn/tutorials/99problems.html, accessed on 2021-04-15.

Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2155–2168.

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *International Conference on Computer Aided Verification*. Springer, 745–761.

Moritz Sinn, Florian Zuleger, and Helmut Veith. 2017. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of automated reasoning* 59, 1 (2017), 3–45.

Linhai Song and Shan Lu. 2014. Statistical debugging for real-world performance problems. *ACM SIGPLAN Notices* 49, 10 (2014), 561–578.

Akhilesh Srikanth, Burak Sahin, and William R Harris. 2017. Complexity verification using guided theorem enumeration. *ACM SIGPLAN Notices* 52, 1 (2017), 639–652.

Peng Wang, Di Wang, and Adam Chlipala. 2017. TiML: a functional language for practical complexity analysis with invariants. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 79.

Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Programming Language Design and Implementation*. ACM, 106–120.

Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic profiling. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 67–76.