COMP26120: Divide and Conquer (2019/20)

Lucas Cordeiro

lucas.cordeiro@manchester.ac.uk
Divide-and-Conquer (Recurrence)

- References:
  - *Algorithm Design and Applications*, Goodrich, Michael T. and Roberto Tamassia (Chapter 8)
  - *Introduction to Algorithms*, Cormen, Leiserson, Rivest, Stein (Chapters 2 and 4)
Intended Learning Outcomes

• Understand the **divide-and-conquer** paradigm and how **recurrence** can be obtained

• Solve recurrences using **substitution** method

• Describe **various examples** to analyse divide-and-conquer algorithms and how to solve their recurrences
Divide-and-Conquer

• The **divide-and-conquer** paradigm involves three steps at each level of the **recursion**:
  - **Divide** the problem into some subproblems that are smaller instances of the same problem \((D(n)))\)
  - **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, straightforwardly solve the subproblems \((aT(n/b))\)
  - **Combine** the solutions to the subproblems into the solution for the original problem \((C(n))\)

\[
T(n) = \Theta(1) \text{ if } n \leq c, \\
T(n) = D(n) + aT(n/b) + C(n) \text{ otherwise.}
\]
Illustrative Example (Merge Sort)

- The merge sort algorithm closely follows the **divide-and-conquer** paradigm
  - **Divide:** Divide the \( n \)-element sequence to be sorted into two subsequences of \( n=2 \) elements each (\( \Theta(1) \))
  - **Conquer:** Sort the two subsequences recursively using merge sort (\( 2T(n/2) \))
  - **Combine:** Merge the two sorted subsequences to produce the sorted answer (\( \Theta(n) \))

\[
T(n) = \Theta(1) \quad \text{if } n = 1, \\
T(n) = \Theta(1) + 2T(n/2) + \Theta(n) \quad \text{otherwise.}
\]
How Does Merge Sort Work?

• **Idea:** suppose we have two piles of cards face up on a table; **each pile is sorted**, with the smallest cards on top

• We wish to merge the two piles into a single sorted output pile, which is to be face down on the table

  1. choose the smaller of the two cards on top of the face-up piles
  2. remove it from its pile (which exposes a new top card)
  3. place this card face down onto the output pile
  4. repeat this until one input pile is empty, at which time we take the remaining input pile and place it face down onto the output pile
Merge Sort Algorithm

MergeSort(A, left, right) {
    if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);
    }
}

// Merge() takes two sorted subarrays of A and
// merges them into a single sorted subarray of A
// (how long should this take?)
**Merge()**

**Merge**(A, left, mid, right)

\[ n_1 = \text{mid-left} + 1 \]
\[ n_2 = \text{right-mid} \]

create two sorted subarrays \( L[0..n_1] \) and \( R[0..n_2] \)

*for* \( i=0 \) to \( n_1-1 \)
  *do* \( L[i] = A[\text{left}+i] \)

*for* \( j=0 \) to \( n_2-1 \)
  *do* \( R[j] = A[\text{mid}+j+1] \)

\( L[n_1] = \infty \)
\( R[n_2] = \infty \)

\( i=0 \)
\( j=0 \)

*for* \( k=\text{left} \) to \( \text{right} \)
  *do if* \( L[i] \leq R[j] \)
    *then* \( A[k] = L[i] \)
    \( i = i+1 \)
    \( i = i+1 \)
  *else* \( A[K] = R[j] \)
    \( j = j+1 \)
Merge()

\[ \text{Merge}(A, \text{left}, \text{mid}, \text{right}) \]

\[
\begin{align*}
n_1 &= \text{mid-left} + 1 \\
n_2 &= \text{right-mid}
\end{align*}
\]

create two sorted subarrays \( L[0..n_1] \) and \( R[0..n_2] \)

\[
\begin{align*}
\text{for } i &= 0 \text{ to } n_1-1 \\
&\quad \text{do } L[i] = A[\text{left}+i] \quad \Theta(n_1)
\end{align*}
\]

\[
\begin{align*}
\text{for } j &= 0 \text{ to } n_2-1 \\
&\quad \text{do } R[j] = A[\text{mid}+j+1] \quad \Theta(n_2)
\end{align*}
\]

\[
\begin{align*}
L[n_1] &= \infty \\
R[n_2] &= \infty \\
i &= 0 \\
j &= 0
\end{align*}
\]

\[
\begin{align*}
\text{for } k &= \text{left} \text{ to } \text{right} \\
&\quad \text{do if } L[i] \leq R[j] \\
&\qquad \text{then } A[k] = L[i] \quad \Theta(n) \\
&\qquad \quad i = i + 1 \\
&\qquad \text{else } A[K] = R[j] \\
&\qquad \quad j = j + 1
\end{align*}
\]
MergeSort() running on a array

Diagram showing the process of MergeSort on an array of numbers.
MergeSort() running on a array

85  24  63  45

17  31  96  50
MergeSort() running on an array
MergeSort() running on a array
MergeSort() running on an array
MergeSort() running on an array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on an array
MergeSort() running on a array
MergeSort() running on a array
MergeSort() running on a array

24  85

45  63

17  31  96  50
MergeSort() running on a array
MergeSort() running on a array

24 45 64 85

17 31 96 50
MergeSort() running on a array
MergeSort() running on a array

24  45  64  85

17  31  96  50
MergeSort() running on a array

24 45 64 85

17 31 50 96
MergeSort() running on a array
MergeSort() running on a array

![Diagram of MergeSort process]

- Top level: 17, 24, 31, 45, 50, 63, 85, 96
- Middle level:
  - Left: 17, 24, 31
  - Right: 45, 50, 63
- Lower level:
  - Left: 17, 24, 31
  - Right: 17, 24, 31
Complexity Analysis of Merge Sort

<table>
<thead>
<tr>
<th>Statement</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>MergeSort(A, left, right) {</td>
<td></td>
</tr>
<tr>
<td>\hspace{2cm} if (left &lt; right) {</td>
<td></td>
</tr>
<tr>
<td>\hspace{4.5cm} mid = floor((left + right) \div 2);</td>
<td></td>
</tr>
<tr>
<td>\hspace{4.5cm} MergeSort(A, left, mid);</td>
<td></td>
</tr>
<tr>
<td>\hspace{4.5cm} MergeSort(A, mid+1, right);</td>
<td></td>
</tr>
<tr>
<td>\hspace{4.5cm} Merge(A, left, mid, right);</td>
<td></td>
</tr>
<tr>
<td>\hspace{2cm} }</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

- So $T(n) = \Theta(1)$ when $n = 1$, and
  
  \[ 2T(n/2) + \Theta(n) \text{ when } n > 1 \]

- So what (more succinctly) is $T(n)$?
Recurrences

• The expression that represents the **merge sort**: 

\[
T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 
\end{cases}
\]

• is a **recurrence**.
  - Recurrence: an **equation** or **inequality** that describes a function in terms of its **value on smaller functions**
Recurrences: Factorial

• What is the recurrence equation for this algorithm?

\[
\text{fac}(n) \text{ is} \\
\text{if } n = 1 \text{ then return } 1 \\
\text{else return fac}(n-1) * 1
\]

• A recurrence defines \( T(n) \) in terms of \( T \) for smaller values

\[
T(n) = c \quad \text{if } n=1 \\
T(n) = T(n-1) + c \quad \text{if } n>1
\]
Recurrences: Binary Search

• What is the recurrence equation for this algorithm?

```plaintext
BinSearch(A[1...n], q)
  if n = 1
    then if A[n] = q then return n
    else return 0
  k ← (n+1)/2
  if q < A[k] then BinSearch(A[1...k-1], q)
  else BinSearch(A[k...n], q)

T(n) = c if n = 1
T(n) = c + T(n/2) if n > 1
```
Other Recurrence Examples

\[ s(n) = \begin{cases} 
0 & n = 0 \\
0 & n > 0 
\end{cases} \]

\[ s(n) = \begin{cases} 
0 & n = 0 \\
 n + s(n-1) & n > 0 
\end{cases} \]

\[ T(n) = \begin{cases} 
c & n = 1 \\
2T\left(\frac{n}{2}\right) + c & n > 1 
\end{cases} \]

\[ T(n) = \begin{cases} 
c & n = 1 \\
aT\left(\frac{n}{b}\right) + cn & n > 1 
\end{cases} \]
Solving Recurrences

• Substitution method
• Iteration method
• Master method
Proof by Induction (COMP11120)

• Why is *proof by induction* essential to learn?

```c
unsigned int N=*;
unsigned int i = 0;
long double x=2;
while( i < N ){
    x = ((2*x) - 1);
    ++i;
}
assert( i == N );
assert(x>0);
```

```c
unsigned int N=*
unsigned int i = 0;
long double x=2;
if( i < N ){
    x = ((2*x) - 1);
    ++i;
}
    } k copies
...
assert(! ( i < N ) );
assert( i == N );
assert(x>0);
Proof by Induction of Programs

Handling Unbounded Loops with ESBMC 1.20
(Competition Contribution)

Jeremy More 1, Lucas 1
1 Electronics and Co
2 Department of Comp
a

Model Checking Embedded C Software using k-Induction and Invariants

Herbert Rocha 1, Hussama Ismail 2, Lu 3
1 Federal University of Roraima, E-mail: herbert.rocha@ufrbr
2 lucascordeiro@ufam.edu.br

Abstract. We extended E symbolic model checking and multi-threaded ANSI verify by induction that it
search for a bounded read

1 Overview

ESBMC is a context-bounded single- and multi-threaded C or
ESBMC can only be used to prove pro regret, unless we kno,
ever, this is generally not the c
prove safety properties in 

The details of ESBMC are desp

2 Differences to ESBM

Except for the loop handling
version. The main changes co
were replaced CBMC’s

I. INTRODUCTION

The Bounded Model Checking (BMC) techniques based on
Boolean Satisfiability (SAT) or Satisfiability Module Theories
(SMT) have been applied to verify single- and multi-threaded
programs and to find subtle bugs in real programs 1, 2, 3.

The idea behind the BMC techniques is to check the negation
of a given property at a given depth, i.e., given a transition
system M, a property ϕ, and a limit of iterations k, BMC
unfolds the system k times and converts it into a Verification
Condition (VC) ϕ such that ϕ is satisfiable if and only if ϕ
has a counterexample of depth less than or equal to k.

Typically, BMC techniques are only able to falsify pro

properties up to a given depth k; they are not able to prove the
correctness of the system, unless an upper bound of k is
known, i.e., a bound that unfolds all loops and recursive
functions to their maximum possible depth. In particular, BMC techniques

DepthK: A k-Induction Verifier Based on Invariant
Inference for C Programs

William Rocha 4, Heri 5
4 Electronic and Information Research

5 Department of Computer Science,
6 Division of Computer Science,

Abstract. DepthK is a software verification tool that combines
k- and induction based on program invariant
polyhedral constraints. DepthK uses a
checker that verifies single- and multi-
tthreaded and#

In this paper, we present a novel proof by induction algorithm,
which is based on the top of a symbolic context-bounded
model checker and uses an iterative deepening approach
to verify, for each step k up to a given maximum, whether a
given property ϕ holds in the program. The pro-
posed k-induction algorithm consists of three different
cases that are handled separately:
forward, condition, and inductive step, respec-
tively for a counterexample with up to k loops and

usual loops. The first attempts to apply the k-induction
method to software verification are only recent. In this
paper, we present a novel proof by induction algorithm,
which is built on top of a symbolic context-bounded
model checker and uses an iterative approach to verify.

DepthK is a verification tool that combines
k- and induction based on program invariant
polyhedral constraints. DepthK uses a
checker that verifies single- and multi-
tthreads.

In this paper, we present a novel proof by induction algorithm,
which is built on top of a symbolic context-bounded
model checker and uses an iterative approach to verify.

DepthK is a verification tool that combines
k- and induction based on program invariant
polyhedral constraints. DepthK uses a
checker that verifies single- and multi-
tthreads.

In this paper, we present a novel proof by induction algorithm,
which is built on top of a symbolic context-bounded
model checker and uses an iterative approach to verify.

DepthK is a verification tool that combines
k- and induction based on program invariant
polyhedral constraints. DepthK uses a
checker that verifies single- and multi-
tthreads.

In this paper, we present a novel proof by induction algorithm,
which is built on top of a symbolic context-bounded
model checker and uses an iterative approach to verify.

DepthK is a verification tool that combines
k- and induction based on program invariant
polyhedral constraints. DepthK uses a
checker that verifies single- and multi-
tthreads.

In this paper, we present a novel proof by induction algorithm,
which is built on top of a symbolic context-bounded
model checker and uses an iterative approach to verify.

DepthK is a verification tool that combines
k- and induction based on program invariant
polyhedral constraints. DepthK uses a
checker that verifies single- and multi-
tthreads.

In this paper, we present a novel proof by induction algorithm,
which is built on top of a symbolic context-bounded
model checker and uses an iterative approach to verify.

DepthK is a verification tool that combines
k- and induction based on program invariant
polyhedral constraints. DepthK uses a
checker that verifies single- and multi-
tthreads.

In this paper, we present a novel proof by induction algorithm,
which is built on top of a symbolic context-bounded
model checker and uses an iterative approach to verify.
Review: Proof by Induction

• Show that a property $P$ is true for $n \geq k$
  ▪ Base case
  ▪ Step case or inductive step

• Suppose that
  ▪ $P(k)$ is true for a fixed constant $k$
    o Often $k = 0$
  ▪ $P(n) \implies P(n+1)$ for all $n \geq k$

• Then $P(n)$ is true for all $n \geq k$
Induction Example: Gaussian Closed Form

• Prove $0+1 + 2 + 3 + \ldots + n = n(n+1) / 2$

  - **Basis:** If $n = 0$, then $0 = 0(0+1) / 2$
  
  - **Inductive hypothesis:** Assume that $0 + 1 + 2 + 3 + \ldots + k = k(k+1) / 2$
  
  - **Inductive step:** show that if $P(k)$ holds, then also $P(k+1)$ holds

    \[(0+1+2+\ldots+k)+(k+1) = (k+1)((k+1)+1) / 2.\]

    - Using the induction hypothesis that $P(k)$ holds:
      \[
k(k+1)/2 + (k+1).
      \]
      \[
k(k+1)/2 + (k+1) = [k(k+1)+2(k+1)]/2 = (k^2 +3k+2)/2 = (k+1)(k+2)/2
      \]
      \[= (k+1)((k+1)+1)/2 \quad \text{hereby showing that indeed } P(k+1) \text{ holds}\]
Induction Example: Geometric Closed Form

• Prove \( a^0 + a^1 + \ldots + a^n = (a^{n+1} - 1)/(a - 1) \) for all \( a \neq 1 \)

  ▪ **Basis:** show that \( a^0 = (a^{0+1} - 1)/(a - 1) \)
    
    \[ a^0 = 1 = (a^1 - 1)/(a - 1) \]

  ▪ **Inductive hypothesis:**
    
    o Assume \( a^0 + a^1 + \ldots + a^k = (a^{k+1} - 1)/(a - 1) \)

  ▪ **Inductive step:** show that if \( P(k) \) holds, then also \( P(k+1) \) holds:
    
    \[ a^0 + a^1 + \ldots + a^{k+1} = a^0 + a^1 + \ldots + a^k + a^{k+1} \]
    
    \[ = (a^{k+1} - 1)/(a - 1) + a^{k+1} = (a^{k+1+1} - 1)/(a - 1) \]
The Substitution Method for Solving Recurrences

• The substitution method
  - A.k.a. the “making a good guess method”
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Examples:

Recurrence: $T(n) = 2T([n/2]) + n$

Guess the solution is: $T(n) = O(n \log n)$

Prove that $T(n) \leq cn \log n$ for some $c > 0$ using induction and for all $n \geq n_0$
The Substitution Method

• Induction requires us to show that the solution remains valid for the limit conditions

• Base case: show that the inequality holds for some $n$ sufficiently small
  - If $n = 1 \rightarrow T(1) \leq c \cdot 1 \cdot \log 1 = 0$
  - However, $T(n) = 2T(\lfloor n/2 \rfloor) + n \therefore T(1) = 1$
  - But

  $n=2 \rightarrow T(2) = 2T(1) + 2 = 4 \quad \text{and} \quad cn \log n = c \cdot 2 \cdot \log 2 = 2c$

  $n=3 \rightarrow T(3) = 2T(1) + 3 = 5 \quad \text{and} \quad cn \log n = c \cdot 3 \cdot \log 3$
• We can start from \( T(2) = 4 \) or \( T(3) = 5 \) using some \( c \geq 2 \), given that

\[
\begin{align*}
T(n) &\leq cn \log n & T(2) &\leq c2 \log 2 & T(3) &\leq c3 \log 3
\end{align*}
\]

• Base case holds w.r.t. the asymptotic notation:
  - \( T(n) \leq cn \log n \) for \( n \geq n_0 \)

• Hint: extend the boundary conditions to make the inductive hypothesis count for small values of \( n \)
The Substitution Method

- Inductive hypothesis:
  - Assume that $T(n) \leq c \ n \ \log \ n$ for $c > 0$ holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding

  $$T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)$$
The Substitution Method

• Induction: Inequality holds for $n$

**Original recurrence**

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + n$$

$$\leq 2\left( c \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor \right) + n$$

$$\leq cn \log(n/2) + n$$

$$= cn (\log n - \log 2) + n$$

$$= cn \log n - cn + n$$

$$\leq cn \log n$$
The Substitution Method

- Induction: Inequality holds for $n$

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + n$$
$$\leq 2(c \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor) + n$$
$$\leq cn \log(n/2) + n$$
$$= cn(\log n − \log 2) + n$$
$$= cn \log n − cn + n$$
$$\leq cn \log n \ (\text{holds for } c \geq 1, \text{ upper bound analysis})$$
Making a good guess

• Guessing a solution takes experience / creativity
  ▪ Use heuristics to help you become a good guesser
  ▪ Use recursion trees

• Consider this example: \( T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \)
  ▪ What is your guess here? \( T(n) = O(n \log n) \)
  ▪ The term “17” cannot substantially affect the solution

• Prove loose upper and lower bounds, e.g.:
  ▪ Start with \( T(n) = \Omega(n) \), then \( T(n) = O(n^2) \) until we converge to \( T(n) = O(n \log n) \)
Changing Variables

• A little algebraic manipulation can make unknown recurrence similar to one you have seen before

\[ T(n) = 2T(\sqrt{n}) + \log n \quad m = \log n : \quad n = 2^m \]

\[ T(2^m) = 2T(2^{m/2}) + m \quad S(m) = T(2^m) \]

\[ S(m) = 2S(m/2) + m \quad \text{similar to} \quad T(n) = 2T(n/2) + n \]

\[ T(n) = T(2^m) = S(m) \quad S(m) = O(m \log m) \]

\[ = O(m \log m) = O(\log n \log \log n) \]

Note that \( \log 2^m = m \) and \( \log 2^{m/2} = m/2 \)
Exercise: Recursive Binary Search

• Given the recurrence equation of the binary search:

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1; \\
1 + T(n/2) & \text{if } n > 1; 
\end{cases} \]

• Guess the solution is: \( T(n) = O(\log n) \)

• Prove that \( T(n) \leq c \log n \) for some \( c > 0 \) using induction and for all \( n \geq n_0 \)
Summary

• Many useful algorithms are recursive in structure:
  ▪ to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems

• We also analysed recurrences using the substitution method

• We have demonstrated that $T(n)$ of merge sort is $\Theta(n \log n)$, where $\log n$ stands for $\log_2 n$