Wuhan University
Journal of Natural Sciences

2023, Vol.28 No.6, 483-492

Article ID 1007-1202(2023)06-0483-10
DOI https://doi.org/10.1051/wujns/2023286483

# Program Construction Method for Sequential Statistics Class Algorithm Based on Bidirectional Scanning Induction

ZUO Zhengkang[1], WANG Yuekun[1], LIANG Zanyang[1,3], SU Wei[1], HUANG Qing[1], WANG Yuan[2], WANG Changjing[1†]

1. College of Computer Information Engineering, Jiangxi Normal University, Nanchang 330022, Jiangxi, China;
2. College of Software, Jiangxi Normal University, Nanchang 330022, Jiangxi, China;
3. Nanchang Government Service Data Administration, Nanchang 330038, Jiangxi, China



**Abstract:** The program construction process is based on rigorous mathematical reasoning, which leads to a fully correct algorithmic program via step-by-step refinement of the program specifications. The existing program construction methods' refinement process is partly based on individual subjective speculation and analysis, which lacks a precise guidance method. Meanwhile, efficiency factors have usually been ignored in the construction process, and most of the constructed abstract programs cannot be run directly by machines. In order to solve these problems, a novel program construction method for the sequence statistical class algorithms based on bidirectional scan induction is proposed in this paper. The method takes into account the efficiency factor and thus improves the Morgan's refinement calculus. Furthermore, this paper validates the method's feasibility using an efficiency-sensitive sequential statistics class algorithm as a program construction example. The method proposed in this paper realizes the correctness construction process from program specifications to efficient executable programs.

**Key words:** program construction; bidirectional scanning induction; sequential statistics; Morgan's refinement calculus
**CLC number**: TP311

## 0 Introduction

The goal of software development is to create a correct and efficient program. Software defects frequently have significant economic and social consequences particularly in critical systems. The practice demonstrates that logical correctness can be guaranteed only when the algorithm program is developed using the formal method[1]. As a formal method, program construction combines the program specification, refinement rules,

**Received date:** 2023-02-16
**Foundation item:** Supported by the National Natural Science Foundation of China (62262031), the Jiangxi Provincial Natural Science Foundation (20232BAB202010), the Science and Technology Project of Education Department of Jiangxi Province (GJJ210307,GJJ2200302), and the Cultivation Project for Academic and Technical Leader in Major Disciplines in Jiangxi Province (20232BCJ22013)
**Biography:** ZUO Zhengkang, male, Ph. D., Professor, Senior member of CCF, research direction: software formal methods and generic programming. E-mail: zhengkang2005@iscas.ac.cn
† To whom correspondence should be addressed. E-mail: wcj771006@163.com

and abstract language to produce concrete program code[2]. Ensuring the correctness of program code, this process relies on strict mathematical reasoning[3].

The method of program construction is a hot issue in the research of formal methods. It is challenging to apply existing relevant technologies to the software development process[4]. Currently, the weakest pre-predicate method, the PAR method, and Morgan's refinement calculus are typical methods[5]. These existing construction methods lack complete guidance methods in the process, and mainly rely on subjective speculation and analysis. This makes the program building process tedious. At the same time, the program construction methods that are completely oriented to obtain the object program usually only consider the correctness of the object program, but ignores the efficiency of the algorithm. This results in the low efficiency of the constructed algorithm. Furthermore, most existing program constructors can only be derived from abstract programs, but accurately converting abstract programs into executable programs has become challenging[6].

This paper proposes a program construction method based on induction and an efficiency-driven approach to deal with the above mentioned issues. The method emphasizes problem induction prior to construction and proposes efficient strategies for the sequential statistics class of problems during the induction process, yielding efficient inductive recurrence relations. The Morgan's refinement calculus is improved in the construction[7]. The inductive recursive relation is used as a guide to gradually refine the program to obtain a highly reliable abstract program, which is then converted into an executable C++ program via the conversion platform. This program construction method implements a correctness construction process from a program specification to efficient executable programs.

## 1 Related Work

Dijkstra's weakest pre-condition method has provided program construction for assignment statements, selection statements, and looping statements[8] that start with the precise program specification of the given problem and generate a program that satisfies the specification under the guidance of a program verification system. Because correctness proofs are required before deriving each piece of code, Dijkstra's weakest precondition method combines program construction with correctness proofs[9]. The method focuses on developing programs rather than designing algorithms. It incorporates program correctness proof theory into the development process and ensures program correctness while it is being developed, resulting in a low level of automation[10]. Furthermore, using Dijkstra's weakest precondition method, it is impossible to derive program statements directly and formally.

Chaudhari *et al*[11] focused on the typical steps in the program derivation process, starting from the program specifications. Ref. [11] divided the post-assertion into multiple specifications based on the weakest predicate method and then formed a loop invariant for derivation[12]. Refs. [11,12] are based on the decomposition of post-assertions. However, in the face of more complex program specifications, finding sub-specifications becomes difficult, and the derivation process becomes more complex as the number of sub-specifications increases.

Kourie *et al* proposed a program development style based on the correctness-by-construction approach in Ref. [13]. It combined Dijkstra's GCL language with Morgan's refinement calculus, starting with a formal problem specification and progressing to a code with small steps and simple management[14]. The difficulty in deriving these algorithms stems from the fact that they are based on loop invariants inferred at the start of the derivation. The difficulty of loop programs is the development of loop invariants, and loop invariants of complex programs are frequently challenging to obtain.

Manber developed and obtained abstract algorithms for two classical sequential statistical problem classes in Ref. [15]. The concept of induction is incorporated into the algorithm development process. However, because this development method is still non-formal and relies on analysis, it is difficult to guarantee the correctness of the developed algorithmic programs.

## 2 Proposed Program Construction Method

Typically, algorithms can be implemented by various methods with varying degrees of efficiency[16]. This paper aims to design efficient algorithms based on inductive and efficiency-driven program construction methods.

The sequence statistics problem, also known as the selection problem, is common in algorithm design and

analysis. It involves solving the problem of ordering the elements of a sequence without prior ordering, so it usually aims at unordered sequences and is highly sensitive to efficiency. The *K*-value problem, extreme value problem, and median problem are a few examples of selection problems.

This paper uses the sequential statistics problem as an example, and an inductive construction method for the sequential statistics problem is proposed. It consists of four steps, as depicted in Fig. 1.

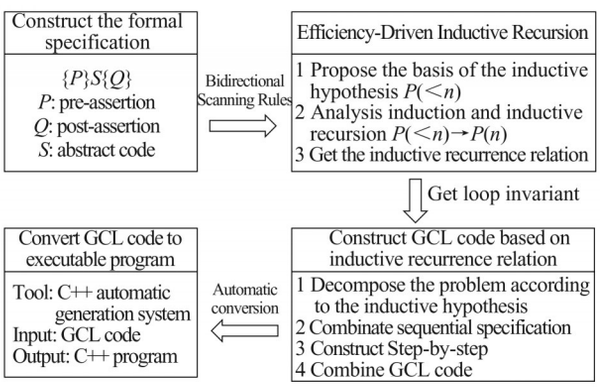


**Fig. 1 Construction method flow chart**

## 2.1 Constructing the Formal Specification

Formal specification is a formal language description of a problem to be solved that specifies the problem's goal[17]. It consists of a pre-assertion "*P*" and a post-assertion "*Q*" that precisely describes the algorithm's function. Formal specification describes the program's purpose — "what to do" — as well as how the program implements the program specification [18].

The program specification is represented by the Hoare triplet $\{P\}S\{Q\}$, where *S* stands for abstract code. If *P* is True before *S* is executed, then *Q* must also be True after *S* is executed[19].

## 2.2 Efficiency-Driven Inductive Recursion

The induction method is an advantageous technique for proving theorems. It is very widespread in computer science. Induction reduces the original problem to a smaller subproblem (or a set of smaller sub-problems). To solve the problem using this approach, only two guarantees are required.

1) A small-scale example of problem-solving is possible.

2) The solution to each of these problems can be derived from the solution to the smaller-scale problem.

Satisfying the above two points, the problem can be solved using induction, which generally does not require proof[20]. The following two steps summarize efficiency-driven inductive recursion:

**Step 1** Propose the induction hypothesis

Firstly we give a smaller problem $P(<n)$ as the basis for the hypothesis. $P(<n)$ satisfies the result $P(n)$ obtained after several deductive reasoning.

**Step 2** Induction and recursion

We first analyze the given assumptions to find the relationship between $P(<n)$ and $P(n)$, and then provide the inductive recursive procedure. Finally, we show how to derive $P(<n) \Rightarrow P(n)$ from the answers to small-scale problems.

The procedure of standard induction is given below. $P(1)$ is the basis of induction, and the abstract procedure Proc ensures that each induction part Pcd() satisfies the problem condition without fulfilling the end of induction Guard. Proc meets $P(i) \wedge \text{Proc} \Rightarrow P(i+1)$, and the induction ends when the end of the induction condition is satisfied.

$P(<n) \rightarrow P(n)$:

**Indt**(Pcd([1,1])/\Utd([2,$n$])/\Guard,Proc)

**Indt**(Pcd([1,2])/\Utd([3,$n$])/\Guard,Proc)

⋮

**Indt**(Pcd([1,$n$−1])/\Utd([$n$,$n$])/\Guard,Proc)

**Indt**(Pcd([1,$n$])/\Utd({∅})/\Guard,Proc)

Meaning of symbols:

$P(n)$ : original problem

$P(<n)$ : sub-problem

Indt() : execution of an induction step

Pcd() : the processed part of the sequence

Utd() : the pending part of the sequence

Proc: abstraction processes that ensure that inductive recursion proceeds

Guard: induction ending condition

The problem division method is the primary factor affecting the efficiency of the algorithm[21]. Because different division methods can be used to develop algorithms of varying complexity, selecting a superior problem division method significantly improve the efficiency of solving algorithms. In order to pursue the efficiency of algorithms, we need more heuristic rules for problem division. This part is crucial, and creative work that requires different strategies for different problems.

**Step 3** Derive inductive recurrence relation

For sequential statistical class problems, this paper applies bidirectional scanning rules as an induction strategy to improve the efficiency of the developed algo-

rithm. The method sets initial anchor points $i_0$ and $j_0$ at each end of the sequence. It uses these two inductive bases for inductive recursion to establish a process Proc that ensures inductive recursion proceeds. Compared with the traditional induction method of dealing with one element at a time, this method increases $i_0$ by $x$ length. It reduces $j_0$ by $y$ length, processing as many elements as possible in one induction. Thus, this approach to induction leads to relatively efficient inductive recurrence relations, which in turn leads to the development of efficient algorithmic programs. The schematic diagram of the bidirectional scanning method is shown in Fig. 2.

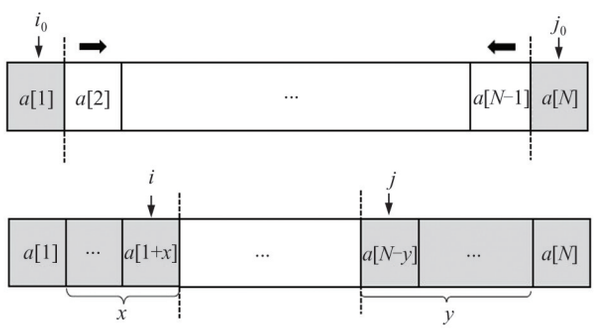


**Fig. 2 Schematic diagram of bidirectional scanning method**

The induction process of bidirectional scanning is as follows:

$P(<n)\rightarrow P(n)$:

**Indt**(Pcd($[1,i_0]\cup[j_0,n]$)$\wedge$Utd($[i_0+1, j_0-1]\wedge$ !Guard$\wedge i_0<j_0$, Proc)

**Indt**(Pcd($[1,i_1]\cup[j_1,n]$)$\wedge$Utd($[i_1+1, j_1-1]\wedge$ !Guard$\wedge i_1=i_0+x\wedge j_1=j_0-y\wedge i_1<j_1$, Proc)

⋮

**Indt**(Pcd($[1,i_n]\cup[j_n,n]$)$\wedge$Utd($\{\varnothing\}\wedge$ !Guard$\wedge i_n=i_{n-1}+x\wedge j_n=j_{n-1}-y\wedge i_n=j_n$, Proc)

## 2.3 Constructing GCL Code Based on Inductive and Recursive Relationships

This subsection is guided by inductive recursive relations and applies Morgan's refinement rule to construct abstract program. The core idea of refinement is to use an Spec($P$,$S$,$Q$) to represent $\{P\}S\{Q\}$. To make Spec($P$,$S$,$Q$) true, the GCL program $C$ is one of the $S$ that conforms to the specification. It is denoted as Sat($C$,Spec($P$,$S$,$Q$)), where $C$ is a refinement of the abstract program $S$.

The expression $\text{Spec}(P,S,Q)\subseteq\text{Spec}(P',S,Q')$ represents $\text{Spec}(P',S',Q')$ as a refinement of $\text{Spec}(P,S,Q)$, if and only if the following conditions hold.

$$\forall C{:}\text{GCL}\cdot\text{Sat}(C,\text{Spec}(P',S',Q'))\Rightarrow(\text{Sat}(C,\text{Spec}(P,S,Q))$$

Any GCL program C that satisfies $\text{Spec}(P',S',Q')$ must also satisfy $\text{Spec}(P,S,Q)$. Morgan listed several refinement rules in Ref. [22]. Since our goal is not to develop a complete refinement theory, we can use some rules[22] to assist us in developing the algorithm. The whole process can be summarized as follows:

$\{P\}S\{Q\}$

**Step 1** Decompose problems according to inductive hypothesis

$\subseteq$\{Strengthened post-assertion Inv$\wedge$!Guard=>$Q$\}

$\{P\}S\{\text{Inv}\wedge!\text{Guard}\}$

**Step 2** Sequential combination specification

$\subseteq$\{Use Inv as an intermediate assertion \}

$\{P\}$SInit\{Inv\};SLoop\{Inv$\wedge$!Guard \}

**Step 3** Construction step by step

$\subseteq$\{Refine the initialization section \}

$\{P\}$Init_GCL\{Inv\};do Guard$\rightarrow B$ od\{Inv$\wedge$Guard\}

$\subseteq$\{Find the variant $V$\}

$\{P\}$Init_GCL\{Inv\};do Guard$\rightarrow$

$\{\text{Inv}\wedge V=V_0\}B\{\text{Inv}\wedge V<V_0\}$od

$\{\text{Inv}\wedge\text{Guard}\}$

$\subseteq$\{Continue refining part $B$ \}

$\{P\}$Init_GCL\{Inv\};do Guard$\rightarrow$

$\{\text{Inv}\wedge V=V_0\}B$_GCL$\{\text{Inv}\wedge V<V_0\}$od

$\{\text{Inv}\wedge!\text{Guard}\}$

**Step 4** Combine GCL code

$\subseteq$\{Combine GCL code \}

Sequential combination {Init_GCL$\cdots B$_GCL}

The basic steps in the process are as follows:

1) Decompose problems according to an inductive hypothesis

Because the problem can be decomposed and abstracted into several independent modules, we begin by decomposing the problem according to the given $\{P\}S\{Q\}$. The decomposition is typically accomplished in two ways:

A. If a recursive structure is required, then the recursive and refinement parts are decomposed. Subsequently, find the refinement part of the specification $\{P\}S\{Q\}$ for the construction and the recursive part is combined at the end.

B. Decompose the loop part and initialization part of the refinement part, respectively.

This decomposition allows us to focus on solving a small range of problems and facilitates the construction of algorithmic programs.

2) Sequential combination specification

Sequential combinatorial specification necessitates loop invariant, which is a computer science concept. A loop invariant is required for understanding algorithmic programs, proving, constructing, and automatically generating algorithmic programs, and building programs. It refers to the fact that a property of a looping program (typically the program's correctness) remains constant before and after the loop[23]. A loop invariant holds true throughout all three steps of the loop:

a. Initialization (before the first iteration of the loop);

b. During each iteration of the loop;

c. At the end of the loop.

In computer science, loop invariant is an extension and implementation of inductive reasoning, where the process of inductive reasoning is analogous to designing an algorithm. In traditional Morgan's refined calculus, loop invariants often rely on deductive reasoning. In contrast, in the approach of this paper, it is more convenient to find loop invariants by constructing inductive and recursive relations in advance.

For a kind of sequential problem, by analyzing the Proc in the inductive recursive relation $P(<n)=>P(n)$ and some other constraints, we can quickly obtain the loop invariant in general form as follows. The "Problem conditions" are non-essential supplementary conditions that reflect the laws of loop variables and can be deemed necessary depending on the problem's complexity.

$$\text{Inv}\equiv\text{Pcd}()\wedge\text{Problem condition}$$

The method's inductive and recursive relations provide a complete set of algorithmic logic. It enables the program construction process to be guided by the correct theory and to avoid erroneous refinements.

After finding the correct Inv and ! Guard, get the Inv∧ ! Guard $Q$, then the problem is refined as $\{P\}S\{\text{Inv}\wedge!\text{Guard}\}$. Inv is used to divide the initial specification $\{P\}S\{Q\}$ into an initial and loop specification. Its general form is as follows .

$$\{P\}\text{SInit}\{\text{Inv}\};\text{SLoop}\{\text{Inv}\wedge!\text{Guard}\}$$

where $\{P\}$SInit{Inv} is the initialization specification. SInit refers to the variables needed for the initialization part. $\{P\}$ is the pre-assertion of the initialization specification, {Inv} is the post-assertion. In the expression {Inv}SLoop{Inv∧ ! Guard}, SLoop refers to variables that may be used in the loop part, {Inv} is the pre-assertion of the loop specification, and {Inv∧!Guard} is a post-assertion.

3) Construction step by step

After sequential combination specification, the GCL program can be obtained by constructing and combining these two specifications. Morgan's refinement calculus is applied to refine each specification, and the key steps are verified during the refinement process to ensure the correctness of the algorithm. Firstly, refine the initialization part: $\{P\}$SInit{Inv}$\subseteq\{P\}$Init_GCL{Inv}. Secondly, refine the loop part: {Inv}SLoop{Inv∧!Guard} $\subseteq$ do Guard$\rightarrow\{\text{Inv}\wedge V=V_0\}B\{\text{Inv}\wedge V<V_0\}$od. If $B$ contains a loop part, continue problem decomposition until it can be refined into basic statements.

4) Combine GCL code

The Apla language developed by our team is similar to GCL, and can be a semantically one-to-one correspondence with GCL. Rewriting the GCL code to Apla code and then using the C++ automatic program generation system developed by our team, we can generate executable C++ programs.

### 2.4 Converting GCL Code to Executable Code

The GCL code is still an abstract program and cannot be compiled and run directly on the computer. The C++ program automatic generation system developed by our team transforms the GCL code into an executable C++ program so that the running result can be obtained through compilation. Since the compilation into a machine code is entrusted to a third-party compiler, the machine independence of the algorithm program is realized, and the reliability of the conversion system is easily guaranteed[24]. The following is the overall structure of the system, as shown in Fig. 3.

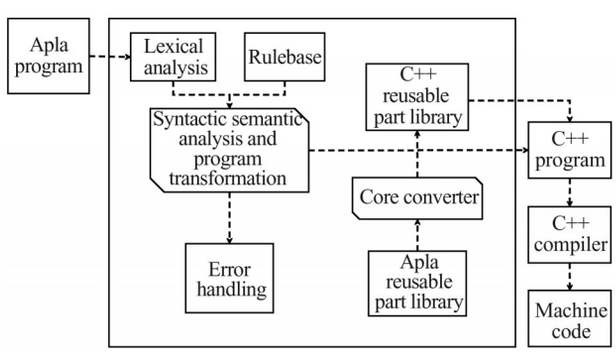


**Fig. 3 Apla to C++ generation system diagram**

## 3 Case Study: Search for the *k*-th Smallest Number Problem

Problem description: Given a sequence $A$ of length

$N$, we search for the $k$-th smallest number in the sequence.

### 3.1 Constructing the Formal Specification of the Case

Define the expression partsort ($A'$,1,$N$,$k$,pivot) to indicate that pivot is the $k$-th smallest number in sequence $A'$. The $A'$ is the permutation of the $A$, $a'[k]$ represents the $k$-th element of the sequence $A'$. This indirectly determines that the pivot is also the $k$-th smallest number in sequence $A$. Due to the specificity of the problem, the subscript of the sequence, in this case, starts from 1.

partsort ($A'$,1,$N$,$k$,pivot)≡

($a'[k]$=pivot)∧($\forall x$:1≤$x$<$k$:$a'[x]$<pivot)∧

($\forall x$:$k$<$x$≤$N$:$a'[x]$>pivot)

The final formal specification is as follows:

$P$: ($\forall p,q$ : 1≤$p$≤$N$∧1≤$q$≤$N$ : $a[p]$≠$a[q]$)

$Q$: partsort ($A'$,1,$N$,$k$,pivot)

### 3.2 Efficiency-Driven Inductive Recursion of the Case

For a kind of sequential problem, we can choose a value in the sequence $A$ as the pivot and determine its order statistic $i$. The solution to the problem is obtained by selecting whether $i$ is equal to $k$. If it is not equal, the pivot is reselected until $i$=$k$. This process obviously employs recursion, and the crux of the problem is determining how to select the pivot and the order statistic of the pivot $i$. This process involves a method that needs to be called repeatedly in subsequent steps, and we try to do induction.

(a)Standard induction

The focus of the sequential problem is on which method to use. To that end, we begin with a standard inductive recursion, specifying that the first element in $A$ is pivot to determine its order statistic $i$.

1) Propose the basis of inductive hypothesis

Assume that the pivot is the first smallest number in the range [1,1] of the array interval.

2) Analytical induction and inductive recursion

The assumption obviously holds since there is only one element of pivot in the range. To determine the order statistic $i$ of pivot in $A$, we need to discuss the case of increasing from 1 to $N$ at the boundary of the array interval. We can use pointer $j$ to represent the boundary, and pointer $j$ is initialized to 1.

Each induction step moves one unit to the right, yielding the induction recurrence relation in Table 1.

At the end of the recursion, we can determine the order statistic $i$ of pivot, and if $i$=$k$, pivot is the $k$-th decimal we need to find. If $i$≠$k$, the second element of the interval is taken as pivot and $j$ is recursive from 2 onwards. After calculation, the worst time complexity of this method is $O(n!)$, which is an impractical algorithm. It would be faster to obtain the $k$-th smallest number directly using the traditional sorting algorithm in sequential problems. However, a more efficient algorithm can be derived using the bidirectional scanning induction method proposed in the standard induction approach.

**Table 1 The standard inductive recursive relationship of the problem**

| $i$-th | $i$ | $j$ | Operation |
|---|---|---|---|
| 1 | $i_0$ | 1 | $a[j]$<pivot→$i_1$=$i_0$+1∨$a[j]$>pivot)→$i_1$=$i_0$ |
| 2 | $i_1$ | 2 | $a[j]$<pivot→$i_2$=$i_1$+1∨$a[j]$>pivot)→$i_2$=$i_1$ |
| ⋮ | ⋮ | ⋮ | ⋮ |
| $N$ | $i$ | $N$ | $a[j]$<pivot→$i$=$i_{n-1}$+1∨$a[j]$>pivot)→$i$=$i_{n-1}$ |

(b)Bidirectional scanning induction

Applying the bidirectional scanning method to reperform the inductive recursion, we set the anchor points $i$ and $j$ at the two ends of the sequence and then specify pivot=$a$[1]; the pointers $i$ and $j$ travel along the sequence in opposite directions.

1) Propose the basis of inductive hypothesis

Suppose that at step $n$ of the algorithm, the sequence $A'$ satisfies the following expression:

($\forall x$ : 1≤$x$<$i$ : pivot≤$a[x]$)∧($\forall y$ : $j$≺$y$<$N$ : pivot≥$a[y]$)

2) Analytical induction and inductive recursion

Let the base case $P$(<$n$) be $i$=1∧$j$=$N$, $i$ and $j$ should be moved towards each other at the $n$+1 time to ensure the hypothesis is still valid until $i$=$j$. We first discuss the case of $i$<$j$. According to the inductive hypothesis, in the base case, the interval $[1,1)$ and $[N,N)$ are both empty sets by substituting $i$ and $j$ into the inductive hypothesis base, and the expression is true. To make the hypothesis still hold after $i$ and $j$ move, we require $i$ and $j$ to stop when they encounter $a[i]$ >pivot∧$a[j]$ <pivot in the process of moving in opposite directions, and then swap $i$ and $j$ to ensure that the hypothesis holds. This process can be described in the following form.

Proc≡

**if** $a[i]$<pivot∧$a[j]$>pivot → $i$++; $j$−−;

**[]** If $a[i]$>pivot∧$a[j]$>pivot → $j$−−;

**[]** If $a[i]$<pivot∧$a[j]$<pivot → $i$++;

**[]** If $a[i]$>pivot∧$a[j]$<pivot → swap($i$, $j$);

**fi**

According to the above process, the inductive recur-

rence relationship in Table 2 can be obtained.

**Table 2 The bidirectional scanning induction recursive relationship of the problem**

| $i$-th | $i$ | $j$ | Operation |
|---|---|---|---|
| 1 | 1 | $N$ | $(a[i]<\text{pivot}\wedge a[j]>\text{pivot})\rightarrow i_1=i+1\vee$<br>$(a[i]>\text{pivot}\wedge a[j]>\text{pivot})\rightarrow j_1=j-1\vee$<br>$(a[i]<\text{pivot}\wedge a[j]<\text{pivot})\rightarrow i_1=i+1, j_1=j-1\vee$<br>$(a[i]>\text{pivot}\wedge a[j]<\text{pivot})\rightarrow i_1=j, j_1=i$ |
| 2 | $i_1$ | $j_1$ | $(a[i_1]<\text{pivot}\wedge a[j_1]>\text{pivot})\rightarrow i_2=i_1+1\vee$<br>$(a[i_1]>\text{pivot}\wedge a[j_1]>\text{pivot})\rightarrow j_2=j_1-1\vee$<br>$(a[i_1]<\text{pivot}\wedge a[j_1]<\text{pivot})\rightarrow i_2=i_1+1, j_2=j_1-1\vee$<br>$(a[i_1]>\text{pivot}\wedge a[j_1]<\text{pivot})\rightarrow i_2=j_1, j_2=i_1$ |
| ⋮ | ⋮ | ⋮ | ⋮ |

When the induction proceeds to the state $i=j$, which means all scans are completed, only the values of the pivot and $a[i]$ need to be exchanged to satisfy the induction hypothesis. When using this kind of inductive recurrence relation, we compare the ending value $i$ with $k$, using the permutation sequence $A'$ to narrow down the range of each comparison with the following possibilities.

1. $i=k$, the current value of pivot is the $k$-th smallest number.

2. $i>k$, the $k$-th smallest number is in the subscript interval $[1,i)$ of $A'$.

3. $i<k$, the $k$-th smallest number is in the subscript interval $[i, N]$ of $A'$.

Obviously, the above process is very suitable for recursive description so that the following inductive recursive relation can be obtained, as shown in Table 3.

**Table 3 Recursive part of the problem inductive recursive relations**

| | Pivot | Perm | $i$ | Operation |
|---|---|---|---|---|
| $A$ | $A[1]$ | $\text{perm}(A, A')$ | $i_1$ | $i_1>k\rightarrow A_1\equiv A'[1,i_1)\vee i_1<k\rightarrow A_1\equiv A'(i_1,N]\vee i_1=k\rightarrow\text{End}$ |
| $A_1$ | $A_1[1]$ | $\text{perm}(A_1, A_1')$ | $i_2$ | $i_1>k\rightarrow A_2\equiv A_1'[1,i_2)\vee i_1<k\rightarrow A_2\equiv A_1'(i_2,N]\vee i_2=k\rightarrow\text{End}$ |
| $A_2$ | $A_2[1]$ | $\text{perm}(A_2, A_2')$ | $i_3$ | $i_1>k\rightarrow A_3\equiv A_2'[1,i_3)i_1<k\rightarrow A_3\equiv A_2'(i_3,N]\vee i_3=k\rightarrow\text{End}$ |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

The recursive induction range is smaller than the previous one each time, which can ensure that the $k$-th smallest number is found during the execution. The average time complexity of this approach is measured to be $O(n)$. Compared with the standard induction method, the efficiency of the algorithm is significantly improved by this inductive recursive approach.

### 3.3 Constructing GCL Code of the Case Based on Inductive Recursion Relations

1) Decompose problems according to inductive hypothesis

$S$: $[(\forall p, q: 1\leqslant p\leqslant n\wedge 1\leqslant q\leqslant n: a[p]\neq a[q]), \text{perm}(A, A')\wedge \text{partsort}(A',1,N,k,\text{pivot})]$

According to the inductive recursion relation, we can decompose the specification into a refinement part and a recursive part:

//Refinement part

$S_1$: $[(\forall p, q: 1\leqslant p\leqslant n\wedge 1\leqslant q\leqslant n: a[p]\neq a[q]), \text{perm}(A, A')\wedge \text{partsort}(A',1,N,i,\text{pivot})]$;

//Recursive part

$S_2$: $[\ \text{perm}(A, A')\wedge\text{partsort}\ (A', 1, N, i, \text{pivot}), \text{perm}(A, A')\wedge\text{partsort}\ (A',1,N,k,\text{pivot})]$

After decomposing the problem, we define the following procedural framework, with $S_1$ and $S_2$ requiring separate refinement:

**func** Select (Array $a$, Integer $l$, Integer $r$, Integer $k$):

<Integer result>

$\{ N>0\}$

$S_1$ （1）

$\{ \text{perm}(A,A')\wedge\text{partsort}\ (A',1,N,i,\text{pivot})\}$

$S_2$ （2）

$\{ \text{perm}(A,A')\wedge\text{partsort}\ (A',1,N,k,\text{pivot})\}$

**cnuf**

2) Sequential combination specification

For $S_1$, the value of $A[1]$ is chosen as a pivot. To achieve the method, the element positions in the sequence are manipulated to construct $A'$. The following loop invariant can be constructed based on the inductive recurrence relation.

$$\text{Inv}\equiv A'[1,i-1]\leqslant\text{pivot}\wedge A'[j+1,N]\geqslant\text{pivot}$$

When $i=j$, the method is partially completed, so Guard is defined as $i\neq j$, and we can get $\text{Inv}\wedge i=j$ is equivalent to $\text{partsort}(A', 1, N, i, \text{pivot})$. After refining the problem to $\{N>0\}S_1\{\text{Inv}\wedge!\text{Guard}\}$, the specification can be divided into the initialization part: $\text{SInit}: [P, \text{Inv}]$ and the loop part: $\text{SLoop}: [\text{Inv}, \text{Inv}\wedge i=j]$, so the following sequential combinations can be made.

$$\text{SInit}:[N>0,\text{Inv}];\text{Sloop}:[\text{Inv},\text{Inv}\wedge i=j]$$

3) Construction step by step

① Constructing the initialization part

According to the induction hypothesis, $i$ and $j$ are

pointed to both ends of the sequence, and the initialization part is refined as follows.

SInit:[$P$,Inv]⊆$i$:=1, $j$:=$N$, pivot:= $a$[1];

Substituting Inv, since the intervals [1,1) and ($N$,$N$] are False, Inv=True and this refinement clearly holds.

Inv≡$A'$[1,0]⩽pivot∧$A'$[$N$+1,$N$]⩾pivot

② Constructing the loop part

Since ! Guard is $i$=$j$ and the loop hold condition Guard is $i$≠$j$, we can write the following loop body.

**do** $i$≠$j$→[Inv∧$i$≠$j$,Inv∧(0⩽$V$<$V_0$)] **od**

The value of the variant $V$ decreases with each execution of the loop body compared to the previous value $V_0$, ensuring that the loop can proceed. In this example,$V$≡$j$−$i$,0⩽$V$<$V$, 0⩽$j$−$i$<$j_0$−$i_0$.

To move $i$ and $j$ towards each other, the following refinement is performed.

SLoop:[Inv∧$i$≠$j$,Inv∧(0⩽$j$−$i$<$j_0$−$i_0$)][$i$,$j$\$i$+1,$j$−1];

⊆{ replace postcondition }

SLoop:[Inv∧$i$≠$j$,Inv[$i$,$j$\$i$+1,$j$−1]∧(0⩽($j$−$i$−2)<$j_0$−$i_0$)];

⊆{ Simplification }

SLoop:[Inv∧$i$≠$j$,Inv[$i$,$j$\$i$+1,$j$−1]∧True];

⊆{ Simplification }

SLoop:[Inv∧$i$≠$j$,Inv[$i$,$j$\$i$+1,$j$−1]];

⊆{ Substitute Inv }

SLoop:[Inv∧$i$≠$j$,($A'$[1,$i$−1]⩽pivot∧$A'$[$j$+1,$N$]⩾pivot)[$i$,$j$\$i$+1,$j$−1]];

⊆{ replace $i$ and $j$ }

The substitution operation is crucial in this case, and we refine it using the inductive recurrence relation as shown below.

$A'$[1,$i$−1]⩽pivot∧$A'$[$j$+1,$N$]⩾pivot

⊆

**If** ($i$≠$j$∧$a$[$i$]⩽pivot∧$a$[$j$]⩾pivot) →

$A'$[1,$i$]⩽pivot∧$A'$[$j$,$N$]⩾pivot

**If** ($i$≠$j$∧$a$[$i$]⩽pivot∧$a$[$j$]<pivot) →

$A'$[1,$i$]⩽pivot∧$A'$[$j$+1,$N$]⩾pivot

**If** ($i$≠$j$∧$a$[$i$]>pivot∧$a$[$j$]⩾pivot) →

$A'$[1,$i$−1]⩽pivot∧$A'$[$j$,$N$]⩾pivot

**If** ($i$≠$j$∧$a$[$i$]>pivot∧$a$[$j$]<pivot) →

$A'$[1,$i$−1]⩽pivot∧$A'$[$j$+1,$N$]⩾pivot

For the variation of $i$ and $j$, the loop invariant condition is split to facilitate the refinement, and two loops Inv1 and Inv2 are used to refine, where Inv1∧Inv2=Inv.

Inv1≡$A'$[1,$i$−1]⩽pivot; Inv2≡$A'$[$j$+1,$N$]⩾pivot;

Inv1:

**do** $i$≠$j$∧$a$[$i$]⩽pivot→

SLoop1:[Inv1∧($i$≠$j$∧$a$[$i$]⩽pivot),Inv∧0<$N$−$i$⩽$N$−$i_0$]

**od**

Inv2:

**do** $i$≠$j$∧$a$[$j$]⩾pivot→

SLoop2:[Inv2∧($i$≠$j$∧$a$[$j$]⩾pivot),Inv∧0<$j$⩽$j_0$]

**od**

Obviously, the above loop body can be refined as:

**do** ($i$≠$j$)∧($a$[$j$]⩾pivot)→$j$:=$j$−1 **od**

**do** ($i$≠$j$)∧($a$[$i$]⩽pivot)→$i$:=$i$+1 **od**

Both inner loops stop when and only when $a$[$i$]> pivot∧$a$[$j$]<pivot or $i$=$j$. For the first case, we can directly swap $a$[$i$] and $a$[$j$] to make the loop continue. For the second case, $i$ and $j$ meet at a position smaller than the pivot. In order to satisfy the loop invariant, the values of the pivot and $a$[$i$] should be exchanged.

Select Statement Rule:

SLoop:[Inv∧$i$≠$j$,($A'$[1,$i$−1]⩽pivot∧$A'$[$j$+1,$N$]⩾pivot)[$i$,$j$\$i$+1,$j$−1]];

⊆

**do** ($i$≠$j$)∧($a$[$j$]⩾pivot) → $j$:=$j$−1 **od**

**do** ($i$≠$j$)∧($a$[$i$]⩽pivot) → $i$:=$i$+1 **od**

**if** ($a$[$i$]>pivot∧$a$[$j$]<pivot) → [$a'$[$i$],$a'$[$j$]\$a$[$j$],$a$[$i$]]

**[]** ($i$=$j$)→[$a'$[1],$a'$[$i$]\$a$[$i$],pivot]

**fi**

Finally, the $S_1$ part is refined to:

$i$,$j$,pivot:=1,$N$,$a$[1];

**do** $i$≠$j$→

**do** ($i$≠$j$)∧($a$[$i$]⩽pivot)→$i$:=$i$+1 **od**

**do** ($i$≠$j$)∧($a$[$j$]⩾pivot)→$j$:=$j$−1 **od**

**if** ($i$=$j$)→$a$[$i$]=pivot;$a$[$l$]=$a$[$i$];

**[]**→$t$:=$a$[$i$];$a$[$i$]:=$a$[$j$];$a$[$j$]:=$t$;

**fi**

**od**

For the recursive part $S_2$, we generalize the bound to $l$, $r$. Modify the $l$,$r$ range recursively until the pivot is the $k$-th decimal. The following code is derived from the inductive recurrence relation:

Select($A$,$l$,$r$,$k$)

⊆

**if** (partsort($A'$,$l$,$r$,$i$,pivot)∧$i$=$k$)→pivot;

[] (partsort($A'$,$l$,$r$,$i$,pivot)∧$i$>$k$)→Select($A'$,$l$,$i$−1,$k$);

[] (partsort ($A'$,$l$,$r$,$i$,pivot)∧$i$<$k$)→Select($A'$,$i$+1,$r$,$k$);

**fi**

4) Combine GCL code

$S_1$ represents the execution of the partsort method once, and $S_2$ represents the selection of the recursive procedure. The final GCL code of the process can be obtained by combining $S_1$ and $S_2$ as Algorithm 1.

**Algorithm 1 GCL Code: Search for the *k*-th smallest number problem**

```
1 func Select (Array a, Integer l, Integer r, Integer k)
2 <Integer result>
3 i, j, pivot := l, r, a[l];
4 do i≠j →
5   do (i≠j)∧(a[i]≤pivot) → i := i+1 od
6   do (i≠j)∧(a[j]≥pivot) → j := j−1 od
7   if (i=j) →a[i] = pivot; a[l] = a[i];
8   [] →t := a[i]; a[i] := a[j]; a[j]:= t;
9   fi
10 od
11 if (i=k) → result := pivot
12   [] (i>k) → result := Select(A,l,i−1,k);
13   [] (i<k) → result := Select(A,i+1,r,k);
14 fi
15 cunf
```

### 3.4 Converting GCL Code of the Case to Executable Code

Inputting the GCL abstract code from the previous step into the C++ automatic conversion system, we obtain the conversion result shown as in Fig. 4. The code on the left corresponds to the algorithm's GCL code. The code on the right is the C++ executable program that was generated automatically after verification.

Inputting test data randomly into the generated C++ code, we obtain the running result as shown in Fig. 5. After verification, the program is ready to run. Because the Apla to C++ generation system can ensure the semantic equivalence of converting Apla code to C++ code. As a result, it is possible to ensure that the generated C++ program is completely correct, eliminating the need to verify cumbersome C++ programs, dramatically improving software development efficiency, and ensuring the software's reliability and correctness.

## 4 Conclusion

This paper proposes an induction-based and efficiency-driven program construction method. This method reflects the effect of efficient partitioning on algorithm efficiency and the guiding role of induction throughout the program construction process. The correctness of order statistics problems such as the largest and smallest number and the *k*-th decimal have been constructed using the method . Due to limited pages, this paper only provides a detailed construction process for the *k*-th decimal problem. In summary, the proposed method has the following benefits:

(1) Inductive reasoning is performed on the problem prior to program construction, and the recursive relationship of the problem can be obtained, which can quickly obtain the loop invariant and provide a clear direction for the construction process.

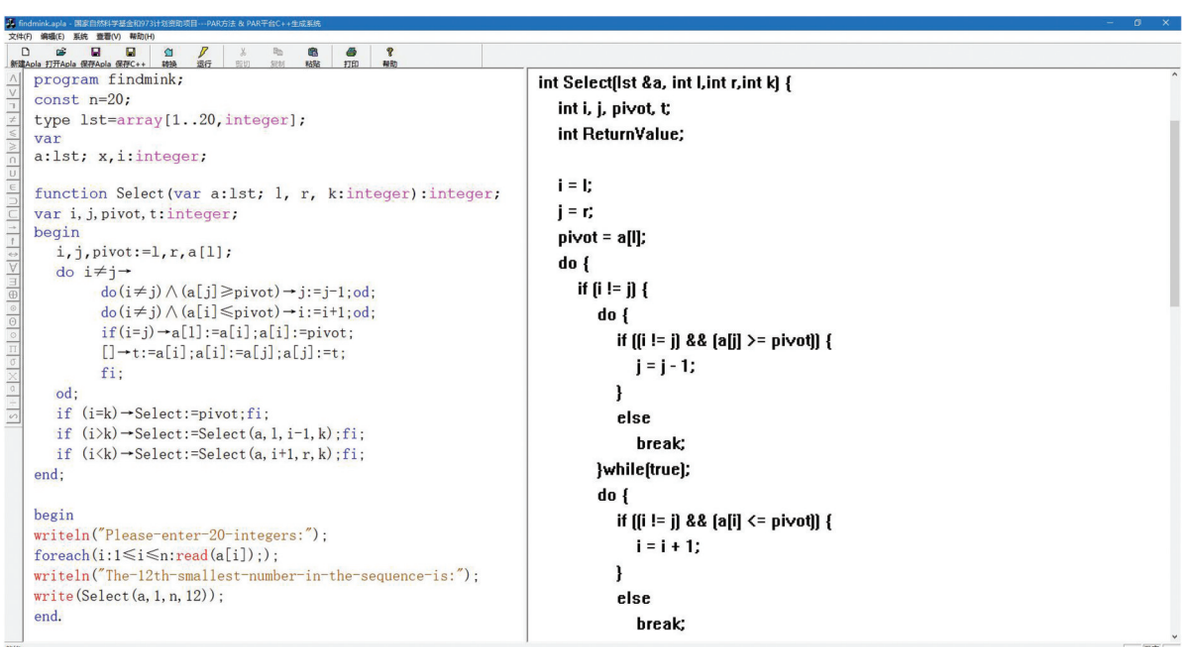


**Fig. 4 Search for the *k*-th smallest number problem conversion diagram**

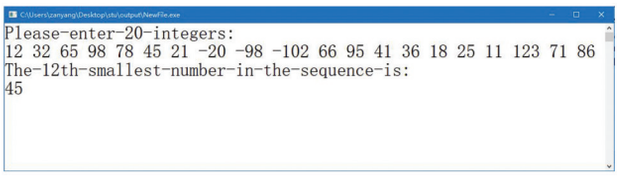


**Fig. 5 Generated C++ code running result**

(2) The generalization and construction process is driven by efficiency, with a focus on algorithm design and a greater emphasis on the algorithm's internal logic. Experiments have shown that using the bidirectional scanning induction method as a construction guide in the program construction of sequential statistics problems is more efficient than the standard induction method.

(3) The entire program construction process from program specification to concrete executable program has been completed.

## References

[1] Wang J, Zhan N J, Feng X Y, *et al*. Overview of formal methods [J]. *Journal of Software*, 2019, **30**(1): 33-61(Ch).

[2] Gu T L. *Formal Methods of Software Development*[M]. Beijing: Higher Education Press, 2005(Ch).

[3] CAS. *Chinese Disciplinary Development Strategy·Software Science and Engineering* [M]. Beijing: Science Press, 2021 (Ch).

[4] You Z, Xue J Y, Zuo Z K. Unified formal derivation and automatic verification of three binary-tree traversal nonrecursive algorithms [J]. *Cluster Computing*, 2016, **19**(4): 2145-2156.

[5] Wang C J, Cao Z X, Yu C L,*et al*. Nonlinear program construction and verification method based on partition recursion and Morgan's refinement rules[J]. *Wuhan University Journal of Natural Sciences*, 2023, **28**(3):246-255..

[6] Shi H P, Shi H H, Xu S H. Algorithm design through the optimization of reuse-based generation [C]// *Proc of National Conference of Theoretical Computer Science.* Singapore: Springer-Verlag, 2020: 14-32.

[7] Zuo Z K, Hu Y, Huang Q, *et al*. Automatic algorithm programming model based on the improved Morgan's refinement calculus[J]. *Wuhan University Journal of Natural Sciences*, 2022, **27**(5):405-414.

[8] Dijkstra E W. *A Discipline of Programming* [M]. Englewood Cliffs: Prentice Hall, 1976.

[9] Zuo Z K, Huang Z P, Fang Y, *et al*. A unified strategy for formal derivation and proof of binary tree nonrecursive algorithms[J]. *Wuhan University Journal of Natural Sciences*, 2022, **27**(5): 415-423.

[10] Shi H H, Xue J Y. Formal development of algorithm based on PAR [J]. *Chinese Journal of Computers*, 2009(5): 138-147(Ch).

[11] Chaudhari D L, Damani O. Introducing formal methods via program derivation [C]// *Proc of ACM Conference on Innovation & Technology in Computer Science Education.* New York: ACM Press, 2015: 266-271.

[12] Chaudhari D L, Damani O. Combining top-down and bottom-up techniques in program derivation [C]// *Proc of Inter Sym on Logic-Based Program Synthesis and Transformation*. Cham: Springer-Verlag, 2015: 244-258.

[13] Kourie D G, Watson B W. *Correctness-by-Construction Approach to Programming* [M]. Berlin: Springer-Verlag, 2012.

[14] Watson B W, Cleophas L G, Kourie D G. Using correctness-by-construction to derive dead-zone algorithms [C]// *Prague Stringology Conference* 2014 *Computer Science.* Prague: Prague Stringology Club, 2014: 84-95.

[15] Manber U. *Introduction to Algorithms–A Creative Approach* [M]. Boston: Addison-Wesley Longman Publishing Co. Inc, 2010.

[16] Shi H H, Xue J Y. Development of a set of highly reliable search algorithm programs based on PAR [J]. *Computer Research and Development*, 2010(S1): 212-216(Ch).

[17] Michael J B, Dinolt G W, Drusinsky D. Open questions in formal methods [J]. *IEEE Annals of the History of Computing*, 2020, **53**(5): 81-84.

[18] Xue J Y, You Z, Hu Q M, *et al*. PAR: A practicable formal method and its supporting platform [C]// *Proc of Inter Con on Formal Engineering Methods.* Cham: Springer-Verlag, 2018: 70-86.

[19] Zuo Z K, Su W, Liang Z Y, *et al*. A formal method for developing algebraic and numerical algorithms[J]. *Wuhan University Journal of Natural Sciences*, 2021, **26**(2): 191-199.

[20] Bundy A. The automation of proof by mathematical induction —ScienceDirect [J]. *Handbook of Automated Reasoning*, 2001, **1**: 845-911.

[21] Xue J Y. A unified approach for developing efficient algorithmic programs [J]. *Comput Sci & Technol*, 1997, **12**(4): 314-329.

[22] Morgan C. *Programming from Specifications*[M]. Englewood Cliffs: Prentice-Hall Inc, 1994.

[23] Gries D. A note on a standard strategy for developing loop invariants and loops [J]. *Science of Compute Programming*, 1982, **2**(3): 207-214.

[24] Lai Y. *Development of APLA to C++ Automatic Program Transformation System* [D]. Nanchang: Jiangxi Normal University, 2002(Ch).

□