1.3.1 Introduction to Algorithms

Chapter 13

Example 13.2

1. The function $f(x) = x^2$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^2 > x$ holds for all $x > 1$.

Example 13.3

Let $f(x) = x^3$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^3 > x$ holds for all $x > 1$.

Example 13.4

Let $f(x) = x^4$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^4 > x$ holds for all $x > 1$.

Example 13.5

Let $f(x) = x^5$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^5 > x$ holds for all $x > 1$.

Example 13.6

Let $f(x) = x^6$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^6 > x$ holds for all $x > 1$.

Example 13.7

Let $f(x) = x^7$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^7 > x$ holds for all $x > 1$.

Example 13.8

Let $f(x) = x^8$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^8 > x$ holds for all $x > 1$.

Example 13.9

Let $f(x) = x^9$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^9 > x$ holds for all $x > 1$.

Example 13.10

Let $f(x) = x^{10}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{10} > x$ holds for all $x > 1$.

Example 13.11

Let $f(x) = x^{11}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{11} > x$ holds for all $x > 1$.

Example 13.12

Let $f(x) = x^{12}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{12} > x$ holds for all $x > 1$.

Example 13.13

Let $f(x) = x^{13}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{13} > x$ holds for all $x > 1$.

Example 13.14

Let $f(x) = x^{14}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{14} > x$ holds for all $x > 1$.

Example 13.15

Let $f(x) = x^{15}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{15} > x$ holds for all $x > 1$.

Example 13.16

Let $f(x) = x^{16}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{16} > x$ holds for all $x > 1$.

Example 13.17

Let $f(x) = x^{17}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{17} > x$ holds for all $x > 1$.

Example 13.18

Let $f(x) = x^{18}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{18} > x$ holds for all $x > 1$.

Example 13.19

Let $f(x) = x^{19}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{19} > x$ holds for all $x > 1$.

Example 13.20

Let $f(x) = x^{20}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{20} > x$ holds for all $x > 1$.

Example 13.21

Let $f(x) = x^{21}$. Then $f(x)$ is greater than $x$ for all $x > 1$. Therefore, the inequality $x^{21} > x$ holds for all $x > 1$.
In the book, we use the terms "speedup" and "speedup degradation".

**Definition:**
If the speedup of an algorithm is said to have been degraded, then the degraded speedup is defined as the ratio of the degraded speedup to the original speedup.

**Example:**
If the original speedup is 10 and the degraded speedup is 5, then the speedup degradation is 2.

**Note:**
In the book, we use the terms "speedup" and "speedup degradation".

In this context, we are considering the speedup of an algorithm. The speedup is defined as the time it takes for a sequential program to run divided by the time it takes for a parallel program to run. A speedup of 2 means that the parallel program is running twice as fast as the sequential program.

Also, the speedup of a program is defined as the ratio of the time it takes for the program to run sequentially to the time it takes for the program to run in parallel.

The speedup of a program is a measure of how much faster a program runs in parallel compared to how fast it runs sequentially.

In general, the speedup of a program can be larger than 1. This means that the program runs faster in parallel than it does sequentially.

In this context, we are considering the speedup of an algorithm. The speedup is defined as the time it takes for a sequential program to run divided by the time it takes for a parallel program to run. A speedup of 2 means that the parallel program is running twice as fast as the sequential program.

Also, the speedup of a program is defined as the ratio of the time it takes for the program to run sequentially to the time it takes for the program to run in parallel.

The speedup of a program is a measure of how much faster a program runs in parallel compared to how fast it runs sequentially.

In general, the speedup of a program can be larger than 1. This means that the program runs faster in parallel than it does sequentially.

In this context, we are considering the speedup of an algorithm. The speedup is defined as the time it takes for a sequential program to run divided by the time it takes for a parallel program to run. A speedup of 2 means that the parallel program is running twice as fast as the sequential program.

Also, the speedup of a program is defined as the ratio of the time it takes for the program to run sequentially to the time it takes for the program to run in parallel.

The speedup of a program is a measure of how much faster a program runs in parallel compared to how fast it runs sequentially.

In general, the speedup of a program can be larger than 1. This means that the program runs faster in parallel than it does sequentially.

In this context, we are considering the speedup of an algorithm. The speedup is defined as the time it takes for a sequential program to run divided by the time it takes for a parallel program to run. A speedup of 2 means that the parallel program is running twice as fast as the sequential program.

Also, the speedup of a program is defined as the ratio of the time it takes for the program to run sequentially to the time it takes for the program to run in parallel.

The speedup of a program is a measure of how much faster a program runs in parallel compared to how fast it runs sequentially.

In general, the speedup of a program can be larger than 1. This means that the program runs faster in parallel than it does sequentially.

In this context, we are considering the speedup of an algorithm. The speedup is defined as the time it takes for a sequential program to run divided by the time it takes for a parallel program to run. A speedup of 2 means that the parallel program is running twice as fast as the sequential program.

Also, the speedup of a program is defined as the ratio of the time it takes for the program to run sequentially to the time it takes for the program to run in parallel.

The speedup of a program is a measure of how much faster a program runs in parallel compared to how fast it runs sequentially.

In general, the speedup of a program can be larger than 1. This means that the program runs faster in parallel than it does sequentially.
CHAPTER 14. PROAM ALGORITHMS

13.2 COMPUTATIONAL MODEL

Figure 13.1: Examples of fixed connection networks

(a) Butterfly (b) Mesh

EXERCISES

\[ \text{Lemma 13.1: Maximum Speedup} \]

Prove that the maximum speedup is equal to the ratio of the speedup of the fastest processor to the speedup of the slowest processor. Assume that the processors are identical and that the problem can be partitioned without loss of generality.

Example 13.2: Consider the following problem: Given a set of tasks, each with a different execution time, find the schedule that minimizes the total execution time. Assume that the processors are identical and that the problem can be partitioned without loss of generality.

\[ \frac{\sum_{i=1}^{n} t_i}{\text{Max speedup}} = \text{Optimal schedule time} \]
The idea of the compiler. The primary role of the compiler is to convert the source code into an intermediate form called the Abstract Syntax Tree (AST). The AST is then used by the language processor to generate the final executable code or code that can be executed by the target machine.

### Compiler Overview

1. **Preprocessing**
   - Removes comments from the source code.
   - Performs macro expansions.
   - Does not change the syntax of the code.

2. **Syntax Analysis**
   - Converts the source code into a series of tokens.
   - Checks the syntax of the code.

3. **Semantic Analysis**
   - Checks the semantics of the code.
   - Determines the meaning of the code.

4. **Code Generation**
   - Converts the AST into machine code.
   - Optimization of the generated code.

### Example 1.3.2

#### A. Program Flow Transition Table

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{m} \text{m} \text{m} \text{m} \text{m} \text{m}</td>
</tr>
<tr>
<td>\text{1} \text{2} \text{3} \text{4} \text{5} \text{6}</td>
</tr>
</tbody>
</table>

#### B. Program Variables

- \textbf{Global Variables:}
  - \textbf{m} (Memory)

- \textbf{Processors:}
  - \textbf{d} (Data)
  - \textbf{3} (Index)
  - \textbf{2} (Execution)
  - \textbf{1} (Execution)

### Chapter 13: PL/1 Algorithms

611
In the domain of parallel computation, the concept of the parallel prefix problem is crucial. It involves the computation of the prefix sums of a sequence of numbers in parallel. The parallel prefix problem can be formalized as follows:

**Theorem 1.1.** Let \( P \) be a prefix of a list \( L \) of \( n \) elements. Then, the parallel prefix problem can be solved with a processor CREW PRAM in time \( \Omega \left( \frac{n}{p} \right) \) using \( p \) processors.

To solve the parallel prefix problem, we use the CREW PRAM model, which allows concurrent reads and writes on the same memory location. The following algorithm demonstrates how to solve the parallel prefix problem on a CREW PRAM:

1. **Initialization:** Each processor \( P_i \) holds the value of the \( i^{th} \) element of the list \( L \).
2. **Computation:** Each processor computes the prefix sum of its own element and the prefix sum of the element to the left. This is done in a parallel fashion by concurrent reads and writes.
3. **Termination:** The final result is stored in the memory locations corresponding to each processor.

The complexity of this algorithm is \( \Omega \left( \frac{n}{p} \right) \), which matches the lower bound for solving the parallel prefix problem on a CREW PRAM.

**Example 1.2.** To illustrate the CREW PRAM model, consider the following example:

- **Input:** List \( L = \{a_1, a_2, a_3, a_4, a_5\} \)
- **Processor:** \( P_1 \) holds \( a_1 \), \( P_2 \) holds \( a_2 \), \( P_3 \) holds \( a_3 \), \( P_4 \) holds \( a_4 \), \( P_5 \) holds \( a_5 \)

Using the CREW PRAM model, each processor can read and write to memory concurrently, allowing the prefix sums to be computed efficiently.

**Algorithm 1.1.** The following algorithm illustrates the process of computing prefix sums on a CREW PRAM:

```
for i = 1 to n do
    P_i computes prefix sum of P_i and P_{i-1}
end for
```

This algorithm runs in \( \Omega \left( \frac{n}{p} \right) \) time on a CREW PRAM, demonstrating the efficiency of the CREW PRAM model in solving the parallel prefix problem.
In this section we consider more complex problems that were not covered in the preceding section.

13.31 Prefix Combination

The problem is to construct a combination of the given number of elements. The initial element is the empty combination. The goal is to construct a combination of the size of the desired number of elements.

Example 13.31: Let the given elements be 1, 2, and 3. We want to construct a combination of size 2.

10.32 Prefix Combination

The problem is to construct a combination of the given number of elements. The initial element is the empty combination. The goal is to construct a combination of the size of the desired number of elements.

Example 13.32: Let the given elements be 1, 2, and 3. We want to construct a combination of size 2.

13.33 Fundamentals Techniques

In this section we consider more complex problems that were not covered in the preceding sections.

Chapter 13: Fundamentals Techniques

1. Discuss the basic principles of prefix combination.

2. Explain how to construct a combination of the given number of elements.

3. Solve the problem of constructing a combination of the given number of elements.

4. Develop an algorithm to construct a combination of the given number of elements.

Exercises

Only one of the following questions is correct. If you select more than one answer, your program will be marked as incorrect.

1. Discuss the basic principles of prefix combination.

2. Explain how to construct a combination of the given number of elements.

3. Solve the problem of constructing a combination of the given number of elements.

4. Develop an algorithm to construct a combination of the given number of elements.

Exercises
Example 13.12
Let I be the input to the peak computation be 6, 7, 8, 9.

Theorem 13.2
The peak computation can be performed in \( O(n) \) time using

\[
\begin{align*}
I &= (1, 1, I') \\
F &= \left( \frac{2}{n} \right) I
\end{align*}
\]

Algorithm 13.2
Prefactor calculation (O(n) time)

Step 1. Let \( I = \frac{2}{n} I \). Let the main processor perform the main computation:

\[
\begin{align*}
&\text{Step 2. Each processor inserts the processor name computed in Step 1. Each processor inserts the processor name computed in Step 1. Each processor inserts the processor name computed in Step 1. Each processor inserts the processor name computed in Step 1.}
&\text{Step 2. Each processor inserts the processor name computed in Step 1. Each processor inserts the processor name computed in Step 1.}
\end{align*}
\]
Deterministic time reductions

The time algorithm is not

The second algorithm is a work-division

The time algorithm is a work-division

The time algorithm is a work-division

In this order: In this order: In this order: In this order:

process

process

process

process

The time algorithm is a work-division

process

process

process

process

This machine can be done sequentially in these times. First, the first hand:

The time algorithm is a work-division

process

process

process

process

This machine can be done sequentially in these times. First, the first hand:

The time algorithm is a work-division

process

process

process

process

This machine can be done sequentially in these times. First, the first hand:
13.4 SELECTION

We now turn our attention to the selection problem, which is the process of choosing an element from a list based on some criteria. In this section, we will study the algorithm for selection, which is a fundamental problem in computer science.

13.4.1 Maximum Selection with k Processors

In this section, we consider the problem of selecting the maximum element from a list of numbers using k processors. We assume that each processor has access to a shared memory and can communicate with other processors.

The problem can be defined as follows: given a list of n numbers, we want to find the maximum number using k processors in parallel.

To solve this problem, we can divide the list into k parts, each part containing n/k numbers. Each processor then calculates the maximum number in its part and returns it to a central location.

The central processor then compares the maximum numbers returned by the processors and returns the overall maximum.

13.4.2 Selection Algorithms

There are several algorithms for solving the selection problem, including quickselect, heapsort, and mergesort. Each of these algorithms has its own advantages and disadvantages.

Quickselect is a randomized algorithm that works by partitioning the list into two parts, one containing elements smaller than the target element and the other containing elements larger than the target element. The algorithm then recursively applies itself to the appropriate part until the target element is found.

Heapsort is a comparison-based sorting algorithm that works by building a heap from the list and then repeatedly removing the largest element from the heap and placing it at the end of the list.

Mergesort is a divide-and-conquer algorithm that works by dividing the list into two halves, sorting each half, and then merging the two sorted halves.

In conclusion, the selection problem is a fundamental problem in computer science, and there are many algorithms available for solving it. The choice of algorithm depends on the specific requirements of the problem at hand.
13.2.1 Maximum Selection Among Integers

Theorem 13.2. The maximum of $n$ numbers can be found in $O(n)$ time.

Algorithm 13.2: Maximum Selection in $O(n)$ time

1. Find the maximum of the first $k$ numbers, where $k = \lceil \sqrt{n} \rceil$.
2. Use a divide-and-conquer approach to find the maximum between the two resulting maximums.
3. Repeat steps 1 and 2 until the maximum is found.


13.2.2 Finding the Maximum Using $p$ Processes

Theorem 13.3. The maximum of $n$ numbers can be found in $O(n/p)$ time using $p$ processes.

Algorithm 13.3: Finding the maximum in parallel

1. Divide the numbers into $p$ equal parts.
2. Find the maximum in each part.
3. Find the maximum of the $p$ maximums found in step 2.


13.2.3 Comparing $n$ Numbers

Theorem 13.4. The comparison of $n$ numbers can be done in $O(n)$ time.

Algorithm 13.4: Comparing $n$ numbers

1. Compare each pair of numbers.
2. Repeat step 1 until all comparisons are made.


13.2.4 Sorting $n$ Numbers

Theorem 13.5. The sorting of $n$ numbers can be done in $O(n \log n)$ time.

Algorithm 13.5: Sorting $n$ numbers

1. Divide the array into two halves.
2. Sort each half separately.
3. Merge the sorted halves.


13.2.5 Merging $n$ Lists

Theorem 13.6. The merging of $n$ sorted lists can be done in $O(n)$ time.

Algorithm 13.6: Merging $n$ lists

1. Combine the lists into pairs.
2. Merge each pair.
3. Repeat step 2 until all lists are merged.


13.2.6 Finding the Median

Theorem 13.7. The median of $n$ numbers can be found in $O(n)$ time.

Algorithm 13.7: Finding the median

1. Sort the numbers.
2. Find the middle element.


13.2.7 Finding the Mode

Theorem 13.8. The mode of $n$ numbers can be found in $O(n)$ time.

Algorithm 13.8: Finding the mode

1. Count the frequency of each number.
2. Find the number(s) with the highest frequency.


13.2.8 Approximating the Median

Theorem 13.9. The median of $n$ numbers can be approximated in $O(n)$ time.

Algorithm 13.9: Approximating the median

1. Use a selection algorithm to find the median.
2. Use a divide-and-conquer approach to find the median in each part.
3. Repeat step 1 until the median is found.


13.2.9 Approximating the Mode

Theorem 13.10. The mode of $n$ numbers can be approximated in $O(n)$ time.

Algorithm 13.10: Approximating the mode

1. Use a selection algorithm to find the mode.
2. Use a divide-and-conquer approach to find the mode in each part.
3. Repeat step 1 until the mode is found.
Theorem 1.3.6. The maximum of a finite set can be found in O(n) time.

Theorem 1.3.7. The maximum of an infinite set can be found in O(1) time.

Figure 1.3.8. The integer maximization problem.

Algorithm 1.3.8. Integer maximization.

1. \text{Input one of the above keys.}
2. \text{Step 1. Find the maximum of the above keys with a binary search.}
3. \text{Step 2. Decide which key to use.}
4. \text{Output the key.}
EXERCISES

1. Perform an insertion sort on the following array by hand:
   
   [5, 2, 9, 1, 5, 6, 3, 4]

   Show each step of the process.

2. Write a C code to sort a given array using the quicksort algorithm. The array may contain duplicate elements. Ensure your code handles duplicates correctly.

THEOREM 13.5 (Quicksort)

Theorem 13.5 states that the quicksort algorithm is an efficient sorting algorithm. The proof of this theorem is omitted here for brevity.

LEMMA 13.3 (Partitioning)

The lemma 13.3 describes a partitioning technique used in quicksort. It states that the partitioning process divides the array into two parts. The proof of this lemma is also omitted for brevity.

1.3.5 A Work-Optimal Randomized Algorithm

The algorithm of Theorem 13.3 has a work of $\Theta(n \log n)$. The algorithm is work-optimal.

1.3.6 Selection Sort

The expression in Theorem 13.3 has a speedup of $\Theta(n)$ in expected time.

Problems

1. What is the time complexity of the quicksort algorithm?
2. How does the quicksort algorithm handle duplicate elements in an array?
3. Demonstrate the quicksort algorithm on the given array: [5, 2, 9, 1, 5, 6, 3, 4].
13.8 SORING

In short, the SORING is the corresponding subject of

*"A in the list of SORING is the corresponding subject of

  (1) 3/4 + 2/3 = 11/12
  (2) 5/6 + 7/8 = 19/24
  (3) 1/2 + 3/4 = 5/4
  (4) 2/3 + 1/6 = 1/2
  (5) 1/4 + 3/8 = 1/2

The SORING is the corresponding subject of

Step 1: Partition X into 3 parts with appropriate weights.

Algorithm 13.11: Sorting in O(log n) time.

The number of processors used in each step is of the form:

- Step 1: \( \frac{1}{2^n} \) processors
- Step 2: \( \frac{1}{2^{n-1}} \) processors
- Step 3: \( \frac{1}{2^{n-2}} \) processors
- Step 4: \( \frac{1}{2^{n-3}} \) processors
- Step 5: \( \frac{1}{2^{n-4}} \) processors

The process continues until all processors are used.

The algorithm works as follows:

1. Partition the data into subparts.
2. Each subpart is sorted independently.
3. The sorted subparts are combined to form the final sorted list.

This process is repeated recursively until the data is completely sorted.

The key to the algorithm is the efficient use of parallel processors to sort large datasets.
13.6.3. Processor Algorithms

Theorem 13.17

A NEW PRAM Processor can be programmed in O((u)log n) time using

Algorithm 13.12: Odd-even merge sort

1. Step 2. Choose X and X such that Algorithm 13.10 and n = 2 where

to sort k processors. Let X be the final

be the input.

Step 3. Choose k processors to sort X. Each processor has

n = 2k

Step 1. Let X = X and X = X. The input.

Step 0. If n > 1, stop X.

Step 1: (n/2) log n

Theorem 13.16: We can sort n elements in O((u)log n) time using

Algorithm 13.10: Odd-even merge sort

Theorem 13.14 (u)log n

The proof of the theorem is similar to Theorem 13.12.

Chapter 13: PRAM Algorithms
Chapter 13: Randomized Algorithms

13.4 Randomized Algorithms

Theorem 13.12

Corollary 13.12

Theorem 13.13

Corollary 13.13

Theorem 13.14

Algorithm 13.15: Preliminary Sorting Algorithm

Step 1: Allocate log₂ n processors to compute the rank of each element in the order of their ranks. If one of the elements is equal, the rank of both the elements is the same. (E.g., (1/2, 1/2) is the same as (1/2, 1/2)).

Step 2: Move on to the next step if the rank is 0 in each layer. If the rank is 0 in each layer, then the processor is sorted.

Step 3: The algorithm is now complete, and the keys are sorted.

End of Chapter 13: Randomized Algorithms