1. Introduction

...it is necessary to make the most efficient use of the available memory. We impose a restriction on the amount of available memory, so that we may be assured that the time required to execute the program does not exceed a predetermined limit. The program is executed within a fixed amount of time, and the program is externally accessible. This is considered essential to avoid the overhead time to execute the program. We study the execution of a program when it is loaded into a fixed amount of memory. In the section on external storage, we consider the problem of selecting and sorting data of limited storage...
A simple estimation from this lemma yields the next upper bound.

The following pass number is $O(\log \log N)$.

**Theorem 1**. If most $c$ elements of the data are contained in a single bucket, then after $c$-pass PRAM ($O(\log \log N)$) time, the entire data set can be partitioned into two sets with $\epsilon$-approximation.

**Proof.** The last sparse graph $G$ that counts the time complexity for $N$.

**Theorem 2**. If $O(N)$ is the number of edges in a graph $G$, then the expected number of edges in $G$ is $\Omega(N)$.

**Proof.** The above result is obtained by counting the number of edges in the graph $G$. In view of the information provided by the section, we consider the following.

In Section 2, we present several constructions of the PRAM model that may be obtained in the next section. We suppress the proof of any practical application. If we suppose that the work distribution is not known a priori, the result is not clear.

We refer to this as the primary issue. The following result is of interest in order to determine the cost of the algorithm.

The sequel to this section is devoted to the selection problem.

**Section 1. Introduction**

There are $\Omega(N)$ elements in the data set. Since compositions can only be made in these locations, we will assume that these compositions are only made in these locations.

2. Elementary results

Theorem 3. The numbers of edges and vertices in a graph $G$ are $\Omega(N)$.

**Proof.** The above result is obtained by counting the number of edges in the graph $G$.
Selection problem is in the complexity class $\mathcal{NP}$. The total number of comparisons to find the smallest element in a list of $n$ elements is $\Omega(n \log n)$. In practice, however, we usually refer to the number of comparisons made in the algorithm. A simple algorithm for finding the minimum element requires $\Omega(n \log n)$ comparisons. This is achieved by sorting the list using a sorting algorithm such as quicksort or mergesort. When $n > 1$, a comparison can be made in constant time.

When $n = 1$, a comparison can be made in constant time.

**Theorem 2.** For $n > 2$, there is a class of selection algorithms which can find the median of $n$ numbers in $O(n \log n)$ time.

**Proof.** The algorithm for finding the median is as follows:

1. Choose a random pivot $p$ from the list.
2. Partition the list into two sublists: one containing elements less than $p$, and the other containing elements greater than or equal to $p$.
3. Recursively apply the algorithm to the sublist containing the smaller elements.
4. Recursively apply the algorithm to the sublist containing the larger elements.

The algorithm runs in $O(n \log n)$ time on average.

For some fixed even $k$, a sample of size $k$ will be a sorted sample of $k$ elements in sorted order.

**Theorem 3.** A $k$-pass algorithm which selects the top $k$ elements of a set of $n$ elements requires $O(k n)$ time.

**Algorithm:**

1. Scan the list once, selecting the top $k$ elements.
2. Sort the selected elements.
3. Select the top $k$ elements from the sorted list.

The algorithm runs in $O(k n)$ time.
**Lemma 5.2.2** 2/2

Suppose we are given a sequence of data points that are sorted in ascending order. The algorithm for finding the median of a sorted sequence is to take the middle element (if the sequence has an odd number of elements) or the average of the two middle elements (if the sequence has an even number of elements).

**Proof:**

We can use binary search to determine the median of the sorted sequence. The idea is to find the partition point that divides the sequence into two equal halves. This is achieved by repeatedly dividing the sequence into two halves and choosing the partition point that ensures the median is in the left half.

1. **Selection of a two-pass algorithm for selecting the median**

   a. Selection algorithm that has an average complexity of \(O(\log N)\)

   b. The median of a sorted sequence can be found in linear time by a single-pass algorithm.

   c. The algorithm iterates through the sequence and compares each element with the median. If the element is greater than the current median, it is exchanged with the last element of the sequence. If the element is less than the current median, it is exchanged with the first element of the sequence.

2. **Proposition:**

   - If the sequence is already sorted, the algorithm will correctly select the median.
   - If the sequence is not sorted, the algorithm may select a value other than the median.

**Theorem 4.1.4**

4. Lower bounds for multi-pass selection

   a. For a two-pass algorithm to determine the median, it must process at least one of the two rows of the sorted sequence.

   b. For a three-pass algorithm to determine the median, it must process at least two rows of the sorted sequence.

   c. For a four-pass algorithm to determine the median, it must process at least three rows of the sorted sequence.

Theorem 4.1.4 follows from Theorem 1 and Theorem 2.
References

The authors wish to thank A. Schubring and a referee for suggesting improvements.

Acknowledgment

The proofs of the main results are contained in the technical report [1] by the author and the referee. It seems likely that the upper bound may be reduced to slightly more than the mean of the distribution of the number of elements in a random subset of size at least half of the median, using a median-finding algorithm which finds the median in expected time $O(n \log n)$. A short proof is given in the appendix.

Theorem 5. For any $p > 0$, there is a single-pass median-finding algorithm with expected running time $O(n \log n)$.

Proof. Consider the algorithm after the first $\epsilon n$ elements have been read. The algorithm is correct with probability at least $1 - \epsilon$ by a Chernoff bound argument.

Theorem 6. There is an $\epsilon > 0$ such that any one-pass algorithm which finds the median in expected time $O(n \log n)$ must have expected running time $O(n \log^2 n)$.

Proof. Consider the algorithm after the first $\epsilon n$ elements have been read. The algorithm is correct with probability at least $1 - \epsilon$ by a Chernoff bound argument.

Theorem 7. For any $p > 0$, there is a single-pass median-finding algorithm with expected running time $O(n \log n)$.

Proof. Consider the algorithm after the first $\epsilon n$ elements have been read. The algorithm is correct with probability at least $1 - \epsilon$ by a Chernoff bound argument.