Problem 1

Consider the following program

Algorithm Sort($L[0 \dots N]$)

Require: L is an array.
  while L is not sorted **do**
    $L \leftarrow$ a random permutation of L.
  end while
Ensure: L is sorted.

Assume we can test that L is sorted $\Theta(|L|)$ time, that we can compute a random permutation of L in $\Theta(|L|)$ time.

⍰ P1.1

Does the $SORT$ program sort correctly? If yes, then provide an invariant for the while-loop and provide
a bound function that can be used to prove the correctness of the program. If no, then argue why the program is not correct.

Solution

The program does sort correctly, but inefficiently, given enough time. This is known as Bogosort, as there are finite permutation in the list, one of them are the sorted array.

The invariant for the while-loop is as follow:

$$\text{Invariant: } 0 < i < S \mid L_i \text{ is a permutation of } L_0 \wedge S = n!$$

This invariant holds through the while-loop because the permutation of the list is a permutation of the original list, and the number of permutations is $n!$.

The bound function for the while-loop is as follow:

- The algorithm is probabilistic, and the expected number of iterations is $n!$.
- The probability for get the sorted list is $p = \frac{1}{n!}$.

One can use Bernoulli's trial to find success probability p. The cumulative probability of having sorted the list after k attempts is $1 - (1 - \frac{1}{n!})^k$

P1.2

Assume the program $SORT$ is correct. Is the program stable? Explain why.

Solution

The program is not stable since SORT is non-deterministic.

- Each iteration generates a random permutation of the list. If the list contains duplicate elements, their relative order is then not reserved between permutations.

- The algorithm does not consider the original order of elements when determine the list is sorted.

P1.3

What is the worst case runtime complexity of this program? What is the best case runtime complexity of this program? Is this program optimal? Explain your arguments.

Solution

Worst case scenario occurs when the algorithm goes through all $n!$ permutations before finding the sorted list. The worst case runtime complexity is $\Theta(n! \cdot n)$.

Best case scenario occurs when the first generated permutation is the sorted list, which has the probability of $\frac{1}{n!}$, which has the runtime of $\Theta(n)$ as it only needs to generate one random permutation and check for sorted.

No, the program is not optimal, since it is based on random permutation, and the expected number of iterations is $n!$. There are no guarantee that the algorithm is reliable to sort the list. Definitely not as optimal as other sorting algorithm such as merge-sort or heap-sort.

P1.4

What is the expected case runtime complexity of this program? Explain your answer.

Solution

Similar to previous mentioned, one can use Bernoulli's trial to find success probability p.

This probability p of success of each run is $\frac{1}{n!}$, where n is the number of elements in the list.

The expected case runtime complexity is $\Theta(n! \cdot n)$, since the expected number of iterations is $n!$. (since each permutation will take $\Theta(n)$ time to check if array is sorted.

Problem 2

The median of a list L of distinct values is the middle value $v \in L$: an equal number of values in L are smaller and larger than v. For example, in the list $L = [1, 5, 4, 2, 3]$, the median is 3. Consider two sorted lists $\mathcal{A}[0 \ldots N)$ and $\mathcal{B}[0 \ldots M)$ with $N + M$ distinct values. You may assume that the total number of values in $\mathcal{A}$ and $\mathcal{B}$ is odd ($N + M$ is odd). Hence, there is a value $v \in (\mathcal{A} \cup \mathcal{B}$ such that an equal amount $E = [\frac{N+M}{2}]$ of other values smaller and larger than v .

P2.1

Provide an algorithm `Median(A, B)` that computes the median of the combined list $\mathcal{A} \cup \mathcal{B}$ in $\mathcal{O}(\log_2(N+M))$ time.

Solution

Algorithm Median$(A[0 \ldots N), B[0 \ldots M])$

Require: N < M $|A| \leq |B|$
  $N \leftarrow |A|$
  $M \leftarrow |B|$
  $L \leftarrow A \cup B$
  $low \coloneqq 0$
  $high \coloneqq N$
  while $low \leq high$ **do**
    $i \coloneqq \lfloor \frac{low+high}{2} \rfloor \leftarrow$ index of A
    $j :- \lfloor \frac{N+M+1}{2} \rfloor - i \leftarrow$ index of B
    $A_{\text{left}} = i > 0 \ ? \ A[i-1] \ : \ -\infty$
    $A_{\text{right}} = i < N \ ? \ A[i] \ : \ \infty$
    $B_{\text{left}} = j > 0 \ ? \ B[j-1] \ : \ -\infty$
    $B_{\text{right}} = j < M \ ? \ B[j] \ : \ \infty$
    if $A_{\text{left}} \leq B_{\text{right}} \wedge B_{\text{left}} \leq A_{\text{right}}$ **then**
      if $(N+M) \mod 2 == 1$ **then**
        return $\max(A_{\text{left}}, B_{\text{left}})$
      else
        return $\frac{\max(A_{\text{left}}, B_{\text{left}}) + \min(A_{\text{right}}, B_{\text{right}})}{2}$
      end if
    else if $A_{\text{left}} > B_{\text{right}}$ **then**
      $high \leftarrow i - 1$
    else
      $low \leftarrow i + 1$
    end if
  end while

P2.2

Explain why your algorithm is correct and why the complexity is $\Theta(\log_2(N+M))$.

Solution

The median of combined list $\mathcal{A} \cup \mathcal{B}$ is the value v such that it either the maximum value of left elements or minimum value of right elements (since $N+M$ is odd). Additionally, it partitions the array such that left side will always contains $\lfloor \frac{M+N}{2} \rfloor$ elements.

Since it employs binary search on two smaller arrays and adjusting the partition $A[i-1], A[i], B[j-1], B[j]$, it halves the search space through each iteration to smaller array.

Thus, one can then achieve the complexity (of binary search) as $\Theta(\log_2(N+M))$.

P2.3

Let $\mathcal{P}$ be an algorithm with complexity $\Theta(\log_2(N+M))$ that computes the middle value $A \cup B$. Argue how we can use P to break up the Merge-step necessary to merge two sorted lists with $N+M = 2E+1$ values into two independent Merge-steps that each merge only E values.

Solution

After using $\mathcal{P}$ to find median of $\mathcal{A} \cup \mathcal{B}$, given that $N + M = 2E + 1$, median will split the list into two halves, each with E elements.

Partition $\mathcal{A}$ and $\mathcal{B}$ into two subsets $\mathcal{A}_{\text{left}}$, $\mathcal{A}_{\text{right}}$ and $\mathcal{B}_{\text{left}}$, $\mathcal{B}_{\text{right}}$ such that left subsets contains items $\leq$ median, right subsets contains *gee* median.

Proceed with two independent Merge-steps that each merge only E values for both lower and higher sets. Finally concatenate these two arrays into one sorted lists.

Overall complexity for the merge ops is $O(2E)$ as each sub-problem involves merging E elements.