ABBOTTABAD UNIVERSITY OF SCIENCE AND TECHNOLOGY

Assignment # 1

Submitted To: Sir Jamal Abdul Ahad

Submitted By: Ayesha Jadoon

Roll no: 14638

Subject: Data Structures

Semester: BSCS (3rd A)

Date: 31/10/2024

Chapter #1

Question: 1.1-1

Describe your own real-world example that requires sorting. Describe one that requires finding the shortest distance between two points.

Answer:

Real World Example: Organizing a Library Catalog

Scenario: A library needs to organize its collection of books efficiently to facilitate easy access for patrons.

1. **Purpose of Sorting:** The library has received a large number of new books that need to be cataloged. Sorting helps in organizing these books by various criteria, such as genre, author, or title, making it easier for library visitors to find the materials they need.
2. **Sorting Process:**
    - **Genre Sorting:** First, the books are sorted into broad categories, such as Fiction, Non-Fiction, Science, History, etc.
    - **Author Sorting:** Within each genre, books are then sorted alphabetically by the author's last name.
    - **Title Sorting:** Finally, for books by the same author, they are further sorted by title in alphabetical order.
3. **Outcome:** This systematic sorting process enhances user experience by allowing patrons to quickly locate books. For example, if a patron is looking for a book by "J.K. Rowling," they can easily find it under the Fiction section, sorted alphabetically by the author's last name, followed by the title.

Finding the Shortest Distance: Map Applications

Step 1:

As we know, the sorting technique is used in many fields or in society. If we should describe one that requires determining the shortest distance between two points, it can be maps applications. When we open the map application and write the destination address, it will show us some possible vehicles to reach the destination. They can be these options:

- **On foot**
- **By bus or train**
- **By car**

Step 2:

If we choose by bus or train option it will show the shortest distance to the destination address according to the bus and train routes. It gives priority to the routes according to the time spent on travel, not the shortest distance.

Step 3:

But if we choose on foot or by car options, it shows the shortest distance according to the way on the map. The process of choosing the best route happens in this way:

1. Map application starts to track the route to the destination address. If the road is divided into multiple ways, the application will track all of them.
2. These divided roads can be also divided in some ways. This application tracks all these possibilities till it reaches the destination address.
3. After reaching the destination, this map application compares all these possibilities and chooses the best one according to the shortest distance.

Question: 1.1-2

Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

Answer:

In a real-world setting, there are several measures of efficiency to consider beyond just speed. Here are some key factors that can affect the performance and effectiveness of an algorithm or system:

1. **Memory Usage**: The amount of memory required by the algorithm is crucial, especially in environments with limited resources. High memory consumption can lead to slower performance or even system crashes.
2. **Energy Consumption**: In mobile and embedded systems, the energy efficiency of an algorithm can significantly impact battery life. Algorithms that consume less energy while maintaining performance are often preferred.
3. **Scalability**: An algorithm's ability to handle an increasing amount of work or its capacity to be enlarged to accommodate growth is important, especially in cloud computing and big data environments.
4. **Complexity**: This includes both time complexity (how the run time increases with input size) and space complexity (how the memory requirement increases with input size). Understanding the complexity can help in predicting performance as data scales.

5. **Robustness**: The algorithm's ability to handle errors or unexpected input gracefully is vital. A robust algorithm can recover from errors without crashing.
6. **Maintainability**: This reflects how easily an algorithm can be understood, modified, or extended in the future. Well-structured code is easier to maintain.
7. **Parallelizability**: The extent to which an algorithm can be parallelized can affect its efficiency on multi-core or distributed systems. Algorithms that can effectively distribute their workload are often more efficient in modern computing environments.
8. **Latency**: This is the time taken for a system to respond to an input. In real-time applications, minimizing latency can be more critical than raw processing speed.
9. **Throughput**: This measures the number of tasks that can be completed in a given time frame. High throughput is often more desirable than low latency in batch processing systems.
10. **Data Transfer Rate**: For algorithms that involve significant data movement (e.g., between memory and storage), the efficiency of data transfer can impact overall performance.

By considering these measures, a more comprehensive understanding of efficiency can be developed, helping to make informed decisions about algorithm design and implementation in various real-world scenarios

---

Question: 1.1-3

Select a data structure that you have seen, and discuss its strengths and limitations**.**

Answer:

Data Structure: Hash Tables

Strengths:

1. **Fast Average Case Performance:**
   Hash tables offer $O(1)$ time complexity for insertion, deletion, and search operations, making them efficient for quick data access.
2. **Flexible Key Types:**
   They can handle various key types, such as strings and numbers, which is useful in diverse applications.

3. **Dynamic Sizing:**
   Many implementations can resize dynamically to maintain performance and reduce collisions.

4. **Simplicity of Use:**
   The straightforward interface allows for easy implementation across programming languages.

Limitations:

1. **Worst-case Performance:**
   Operations can degrade to $O(n)$ in case of excessive collisions, affecting efficiency.

2. **Memory Overhead:**
   Hash tables can consume more memory than other data structures due to maintaining an underlying array.

3. **Collision Resolution Complexity:**
   Handling collisions can complicate implementation and potentially increase time complexity.

4. **Non-Ordered Data:**
   They do not maintain any order among elements, which can be a drawback if order is needed.

5. **Hash Function Dependency:**
   Performance relies on the quality of the hash function, and a poor hash can lead to increased collisions.

Conclusion

Hash tables are efficient for key-based access but have limitations in performance, memory usage, and order maintenance. Understanding these factors is essential for choosing the right data structure for specific applications.

Question: 1.1-4

How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

Answer:

Similarities:

1. **Graph Use**:
   Both problems can be represented using graphs, where points (vertices) are locations and lines (edges) are the paths between them.

2. **Optimization Goal**:
   Both aim to find the best (shortest or cheapest) path. The shortest-path problem looks for the quickest route between two specific points, while the traveling-salesperson problem tries to find the shortest route to visit multiple points and return to the start.

3. **Complexity**:
   Both problems can be complex, but the traveling-salesperson problem is harder to solve than the shortest-path problem.

Differences:

1. **Objective**:
   The shortest-path problem finds the best way from one point to another. The traveling-salesperson problem finds the best way to visit a group of points and come back.

2. **Path Requirements**:
   In the shortest-path problem, you can go directly from point A to point B without visiting others. In the traveling-salesperson problem, you must visit every point exactly once.

3. **Difficulty**:
   The shortest-path problem can be solved quickly using algorithms like Dijkstra's. In contrast, the traveling-salesperson problem is more complicated and often needs special methods to find a good solution.

Question: 1.1-5

Suggest a real-world problem in which only the best solution will do. Then come up with one in which <approximately= the best solution is good enough.

Answer:

1. Problem Requiring the Best Solution:

Air Traffic Control:

When managing flights, controllers need to find the safest and best routes for planes to avoid collisions and delays. Here, only the best solution is acceptable because safety is the priority.

2. Problem Where Approximately the Best Solution is Good Enough:

Delivery Route Planning:

For a delivery service, finding the exact best route for every delivery can take too much time. Instead, a route that is close to the best is often good enough to ensure packages arrive on time. Using a simple method to get a good route works well without needing to find the perfect one every time.

Question: 1.1-6

Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

Answer:

Real-World Problem: Traffic Management Systems

1. Entire Input Available:
When there's a big event like a concert or a sports game, traffic managers can get all the information ahead of time. They know how many people are coming, what time the event starts, and if any roads will be closed. With this information, they can plan the best way to handle the extra cars, set up detours, and adjust traffic lights before the event begins.

2. Input Arriving Over Time:
On a regular day, traffic data comes in little by little. For example, they receive updates about how many cars are on the road, if there are any accidents, or if the weather changes. Traffic management systems need to use this real-time information to make quick decisions. They might change traffic light timings or suggest new routes for drivers based on the latest data to keep traffic flowing smoothly.

This example shows how traffic management can deal with situations where they have all the needed information upfront and situations where they must adjust to new data as it comes in.

Question: 1.2-1

Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

Answer:

Application: Social Media Apps (like Facebook or Instagram)

Function of Algorithms:

1. **Feed Ranking**:
   This algorithm decides which posts you see first in your news feed. It looks at what you like, share, and comment on to show you the most interesting content.

2. **Friend and Page Suggestions**:
   The app suggests new friends or pages to follow based on what you like and who you already follow. For example, if you like cooking pages, it might recommend other cooking accounts.

3. **Spam Detection**:
   Algorithms help find and block spammy or harmful posts. They check the content to keep your feed clean and safe.

4. **Ad Targeting**:
   The app shows you ads that match your interests. If you often look at travel posts, you might see ads for travel deals.

In short, social media apps use algorithms to help you see the posts you care about, connect with people, keep your feed safe, and show you relevant ads.

Question: 1.2-2

Suppose that for inputs of size n on a particular computer, insertion sort runs in $8n^2$ steps and merge sort run in $64\ n\ lg\ n$ steps. For which values of n does insertion sort beat merge sort?

Answer:

To find out for which values of n insertion sort is faster than merge sort, we need to compare the two performance formulas:

1. **Insertion Sort**: $8n^2$
2. **Merge Sort**: $64nlog_2(n)$

We want to find when:

$$8n^2 < 64\text{n}\log_2(\text{n})$$

Step 1: Simplifying the Inequality

We can simplify this by dividing both sides by 8:

$$n^2 < 8n\log_2(\text{n})$$

Step 2: Rearranging the Inequality

Next, we can divide both sides by n (as long as $n > 0$):

$$n < 8\log_2(\mathrm{n})$$

Step 3: Testing Values of n

Now, we can check different values of n to see when this inequality holds:

- **For $n = 1$:**

  $1 < 8\log_2(1)$ (False, since 0 on the right)

- **For $n = 2$:**

  $2 < 8\log_2(2)$ (True, since $2 < 8$)

- **For $n = 3$:**

  $3 < 8\log_2(3)$ (True, since $3 < 12.68$)

- **For n= 4:**

  $4 < 8\log_2(4)$ (True, since $4 < 16$)

- **For $n = 5$:**

  $5 < 8\log_2(5)$ (True, since $5 < 18.568$)

Continuing this way, we find:

- **For $n = 43$:**

  $43 < 8\log_2(43)$ (True, since $43 < 43.408$)

- **For $n = 44$**:

  $44 < 8\log_2(44)$ (False, since $44 < 43.672$)

Conclusion

From the above checks, insertion sort is faster than merge sort when n is less than or equal to 43. So, insertion sort beats merge sort for:

$$n \leq 43$$

In other words, for sorting up to 43 items, insertion sort is the better choice. Beyond that, merge sort is more efficient.

Question: 1.2-3

What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

Answer:

To find the smallest value of n for which $100n^2 < 2^n$, we can analyze the inequality directly.

Step 1: Set Up the Inequality

We want to solve the inequality:

$$100n^2 < 2^n$$

Step 2: Testing Values of n

Let's check various values of n:

- **For $n = 14$**:

$$100(14)^2 = 100 \times 196 = 19600$$

$$2^{14} = 16384$$

So, $19600 > 16384$ (False)

- **For $n = 15$**:

$$100(15)^2 = 100 \times 255 = 19600$$

$$2^{15} = 32768$$

So, $22500 < 32768$ (True)

Conclusion

From the above calculations, we see that:

- When $n = 14$, $100n^2$ is greater than 2^n.
- When $n = 15$, $100n^2$is less than 2^n.

Therefore, the smallest value of n such that $100n^2 < 2^n$ is:

$$\text{Smallest } n = 15$$

So, the answer is $n = 15$.

Question: 1-1 Comparison of running times

For each function $f\ (n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t , assuming that the algorithm to solve the problem takes $f\ (n)$ microseconds.

Answer:

	1 Second	**1 Minute**	**1 Hour**	**1 Day**	**1 Month**	**1 Year**	**1 Century**
$Lg\ n$	$2^{1\times10^6}$	$2^{6\times10^7}$	$2^{3.6\times10^9}$	$2^{8.64\times10^{10}}$	$2^{2.592\times10^{12}}$	$2^{3.1536\times10^{13}}$	$2^{3.15576\times10^{15}}$
$\sqrt{n}$	1×10^{12}	3.6×10^{15}	1.29×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.96×10^{30}
n	1×10^{6}	6×10^{7}	3.6×10^{9}	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.16×10^{15}
$nlg\ n$	62746	2801417	133378058	2755147513	71870856404	797633893349	6.86×10^{13}
n^2	1000	7745	60000	293938	1609968	5615692	56176151
n^3	100	391	1532	4420	13736	31593	146679
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

Question: 2.1-1

Using Figure 2.2 as a model, illustrate the operation of I NSERTION-SORT on an array initially containing the sequence (31, 41, 59, 26, 41, 58).

Answer:

To illustrate the operation of the Insertion Sort algorithm on the array (31,41,59,26,41,58)(31, 41, 59, 26, 41, 58)(31,41,59,26,41,58), we can break it down into the following steps:

Initial Array:

31	41	59	26	41	58

Steps of Insertion Sort:

1. **Look at 31**:

- The first element is considered sorted.

31	41	59	26	41	58

2. **Look at 41**:

- Compare 41 with 31. Since 41 is greater, it stays in place.

31	41	59	26	41	58

3. **Look at 59**:

- Compare 59 with 41. Since 59 is greater, it stays in place.

31	41	59	26	41	58

4. **Look at 26**:

- Compare 26 with 59, then 41, and then 31.
- 26 is less than 31, so we shift 31, 41, and 59 to the right and place 26 in the first position.

26	31	41	59	41	58

5. **Look at 41**:

- Compare 41 with 59. Move 59 to the right.
- Compare 41 with 41 (the previous one); no changes needed.
- Place the new 41 in its correct position.

26	31	41	41	59	58

6. **Look at 58**:

- Compare 58 with 59. Move 59 to the right.
- Compare 58 with 41. It stays in place.
- Place 58 in the correct position.

26	31	41	41	58	59

Final Sorted Array:

The final sorted array after applying insertion sort is:

26	31	41	41	58	59

Summary:

After using insertion sort, the sorted array is [26,31,41,41,58,59]. Each number is placed in the right order by comparing it to the already sorted numbers.

Chapter # 2

Question: 2.1-2

Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array $A[1:n]$. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUM-ARRAY procedure returns the sum of the numbers in $A[1:n]$.

```
SUM-ARRAY (A, n)
1  sum = 0
2  for i = 1 to n
3        sum = sum + A[i]
4  return sum
```

Answer:

- **Loop invariant:**

  At the start of the $i-th$ iteration of the for loop, the variable sum holds the value:

$$sum = \sum_{j=1}^{i-1} A[j]$$

  This means sum equals the sum of the first $i-1$ elements of the array.

- **Initialization**:

Before the first iteration ($i = 1$):

- The variable `sum` is initialized to 0 (line 1).
- There are no elements before the first element, so

$$sum = \sum_{j=1}^{0} A[j] = 0$$

- **Maintenance**:

Assuming the loop invariant holds at the start of iteration i:

- During the $i - th$ iteration, `sum` is updated:

$$sum \leftarrow sum + A[i]$$

- This results in:

$$sum = \left(\sum_{j=1}^{i-1} A[j]\right) + A[i] = \sum_{j=1}^{i} A[j]$$

- After incrementing i for the next iteration, the loop invariant is maintained because:

$$sum = \sum_{j=1}^{(i)} A[j] = \sum_{j=1}^{(i+1)-1} A[j]$$

- **Termination**:

  The loop terminates when $i > n$. Since i increments by 1 each iteration, at termination $i = n + 1$. At this point:

$$sum = \sum_{j=1}^{i-1} A[j] = \sum_{j=1}^{n} A[j]$$

  Therefore, sum contains the total sum of all elements in $A[1 \ldots n]$.

Conclusion:

Your explanation effectively uses the loop invariant to show that the SUM-ARRAY procedure returns the correct sum of the array elements. It's concise and covers all necessary aspects: initialization, maintenance, and termination. You can confidently present this as your solution.

Question: 2.1-3

Rewrite the INSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

Answer:

To modify the INSERTION-SORT procedure to sort an array in monotonically decreasing order, change the comparison operator in the while loop from > to <.

INSERTION-SORT-DECREASING(A)

```
1       for j = 2 to length[A]
2           key = A[j]
3           i = j − 1
4           while i > 0 and A[i] < key
5               A[i + 1] = A[i]
6               i = i − 1
7           A[i + 1] = key
```

Explanation:

- **Comparison Operator Change:** In line 4, replace $A[i] > key$ with $A[i] < key$.
- **How It Works:** This change shifts elements smaller than key to the right, placing larger elements at the front.
- **Result:** The array is sorted in decreasing order because each key is inserted where all preceding elements are greater or equal.

Question: 2.1-4

Consider the ***searching problem***:

Input: A sequence of n numbers ha $(a_1, a_2, \ldots\ldots, a_n)$ stored in array $A[1:n]$ and a value x.

Output: An index i such that x equals $A[i]$ c or the special value NIL if x does not

appear in A.

Write pseudocode for ***linear search***, which scans through the array from beginning to end, looking for x . Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

Answer:

Pseudocode for Linear Search:

LINEAR-*SEARCH* (A, x)

```
1    for i = 1 to A.length()
2        if A[i] == x
3            return i
4    return NIL
```

Loop Invariant:

At the start of each iteration (before line 2), x has not been found in the subarray $A[1..i-1]$.

Proof of Correctness:

- **Initialization:** Before the first loop, $i = 1$, and we haven't checked any elements yet. So, x is not found in an empty list.
- **Maintenance:** If $A[i]$ is not x, then x is not in $A[1..i]$. The invariant holds for the next i.
- **Termination:** If we find x, we return its position i. If we reach the end without finding x, the invariant confirms x isn't in A. So, we return NIL.

Conclusion:

The algorithm correctly returns the index of x if it's in the array, or NIL if it's not.

Question: 2.1-5

Consider the problem of adding two n-bit binary integers a and b, stored in two n-element arrays A[0: n − 1] and $B[0: n-1]$, where each element is either 0 or $a = \sum_{i=0}^{n-1} A[i] \ . 2^i$ and $b = \sum_{i=0}^{n-1} B[i] \ . 2^i$ The sum $c = a + b$ of the two integers should be stored in binary form in an (n +1) element array $C[0: n]$ where $c = \sum_{i=0}^{n-1} C[i] \ . 2^i$ .Write a procedure ADD-BINARY-INTEGERS that takes as input arrays A and B, along with the length n, and returns array C holding the sum.

Answer:

ADD-BINARY-INTEGERS (A, B, n)

1. $Initialize\ array\ C[0..n]$
2. $carry = 0$
3. $for\ i = 0\ to\ n - 1$:
4. $\quad sum = A[i] + B[i] + carry$
5. $\quad C[i] = sum\ \%\ 2$
6. $\quad carry = sum\ //\ 2$
7. $C[n] = carry$
8. $return\ C$

Explanation:

- Initialize result array $C[0..n]$.
- Set carry to 0.
- Loop from $i = 0$ to $n - 1$ (least significant bit to most significant bit):
  - Calculate $sum = A[i] + B[i] + carry$.
  - Set $C[i] = sum\ \%\ 2$ (current bit of the sum).
  - Update $carry = sum\ //\ 2$ (integer division).
- After the loop, set $C[n] = carry$ (the final carry).
- Return the array C containing the sum of A and B.

Question: 2.2-1

Express the function $n^3/1000 + 100n^2 - 100n + 3$ in terms of Θ-notation

Answer:

The function $f(n) = n^3/1000 + 100n^2 - 100n + 3$ can be expressed in Θ-notation as:

$$f(n) = \Theta(n^3)$$

Question: 2.2-2

Consider sorting n numbers stored in array $A[1: n]$ by first finding the smallest element of $A[1: n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2: n]$, and exchange it with $A[2]$. Then find the smallest element of $A[3: n]$, and exchange it with $A[3]$. Continue in this manner for the first $n - 1$ elements of A. Write pseudocode for this algorithm, which is known as **selection sort**. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n - 1$ elements, rather than for all n elements? Give the worst-case running time of selection sort in Θ-notation. Is the best-case running time any better?

Answer:

Pseudocode for Selection Sort:

SELECTION_SORT(A)

```
1.    for i = 1 to A.length() − 1
2.        min_index = i
3.        for j = i to A.length()
4.            if A[j] < A[min_index]
5.                min_index = j
6.        swap(A[i], A[min_index])
```

Loop Invariant:

At the start of the $i - th$ iteration, the subarray $A[1 \ldots i - 1]$ consists of the $i - 1$ smallest elements of $A[1 \ldots n]$, sorted in ascending order.

Why Only First $n - 1$ Elements:

After placing the smallest $n - 1$ elements in their correct positions, the remaining element A[n] is the largest and is already in its correct place. Therefore, the algorithm doesn't need to process the last element.

Worst-Case Running Time:

- In each iteration, the inner loop runs from $j = i$ to n, resulting in $n - i$ comparisons.
- Total running time:

$$Total\ Comparisons = \sum_{i=1}^{n-1} (n - i)$$

$$= n(n-1) - \sum_{i=1}^{n-1} i$$

$$= n(n-1) - \frac{n(n+1)}{2}$$

$$= \frac{n(n+1)}{2}$$

$$= \frac{n^2 - n}{2}$$

$$= \Theta(n^2)$$

Best-Case Running Time:

- The best-case running time is also $\Theta(n^2)$.
- Selection sort always performs the same number of comparisons regardless of the initial order of the elements.

Question: 2.2-3

Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? sing , Θ notation, give the average-case and worst-case running times of linear search. Justify your answers.

Answer:

- **Average Number of Elements Checked:**
  - On average, linear search checks $\frac{n+1}{2}$ elements.
  - This is because the target element is equally likely to be in any position from 1 to n.

- **Worst-Case Number of Elements Checked:**
  - In the worst case, linear search checks $\boldsymbol{n}$ elements.
  - This happens when the target is the last element or not present in the array.
- **Running Times in Θ-Notation:**
  - **Average-Case Running Time:** $\Theta(n)$
  - **Worst-Case Running Time:** $\Theta(n)$
- **Justification:**
  - **Average Case:**
    - Since the target is equally likely to be anywhere, on average, half of the array is checked.
    - $\frac{n+1}{2}$ simplifies to $\Theta(n)$ because constants are ignored in Θ-notation.
  - **Worst Case:**
    - The algorithm may need to check every element, resulting in n comparisons.
    - This directly translates to $\Theta(n)$ time complexity.

Question: 2.2-4

How can you modify any sorting algorithm to have a good best-case running time?

Answer:

Modified Selection Sort for Good Best-Case Running Time:

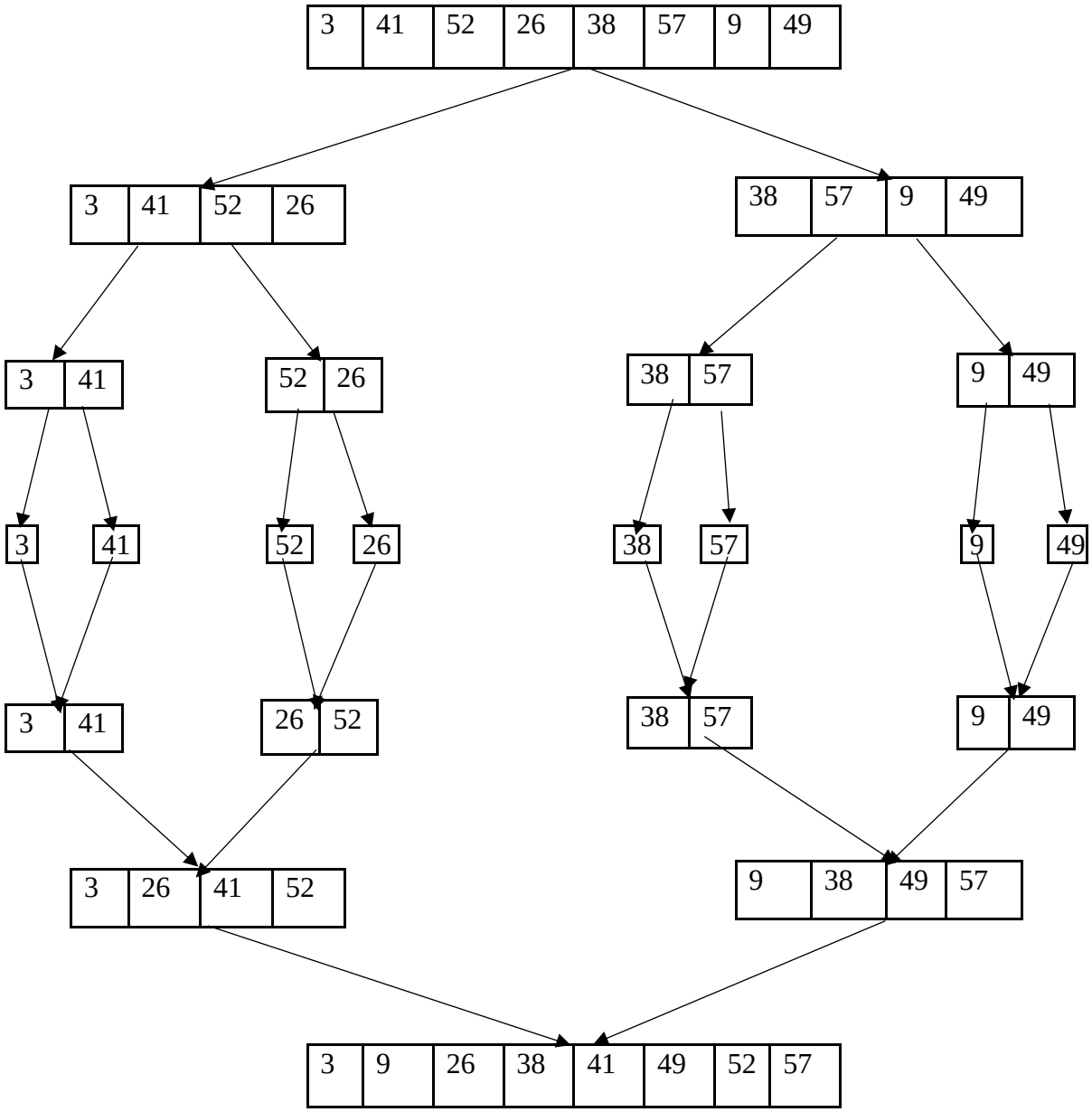
To achieve a good best-case running time for any sorting algorithm, we can design it to handle its best-case scenario as a special case, returning a predetermined solution if the conditions are met. To do this, modify the algorithm so that it first randomly generates a possible solution, then checks if this solution satisfies the sorting goal. If it does, the algorithm outputs the result and halts. Otherwise, it proceeds with the original sorting process. Although this approach is unlikely to succeed on every attempt, in the best-case scenario, the running time will only be as long as it takes to verify the solution.

For example, for selection sort, we can check whether the input array is already sorted and if it is, we can return without doing anything. We can check whether an array is sorted in linear time. So, selection sort can run with a best-case running time of $\Theta(n)$.

Question: 2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence (3, 41, 52, 26, 38, 57, 9, 49).

Answer:



Question: 2.3-2

The test in line 1 of the **MERGE-SORT** procedure reads "if $p \geq r$ " rather than "if $p \neq r$." If **MERGE-SORT** is called with $p > r$, then the subarray $A[p:r]$ is empty. Argue that as long as the initial call of **MERGE-SORT** (A, 1, n) has $n \geq 1$, the test "if $p \neq r$ " suffices to ensure that no recursive call has $p > r$.

Answer:

When modifying the **MERGE-SORT** procedure to use the condition **"if $p \neq r$"** instead of **"if $p \geq r$,"** here's why no recursive call will have **p > r** when the initial call is **MERGE-SORT (A, 1, n)** with **$n \geq 1$**:

1. **Initial Call:**
    - **MERGE-SORT (A, 1, n)** is called with **p = 1** and **r = n**.
    - Since **$n \geq 1$**, we have **p ≤ r**.
2. **Splitting the Array:**
    - The array is split into two parts:
        - **Left Half: MERGE-SORT (A, p, q)** where **q = floor ((p + r)/2)**.
        - **Right Half: MERGE-SORT (A, q + 1, r)**.
    - This ensures:
        - **For Left Half: p ≤ q**.
        - **For Right Half: q + 1 ≤ r**.
3. **Maintaining p ≤ r:**
    - In both recursive calls:
        - **MERGE-SORT (A, p, q):** Here, **p ≤ q**.
        - **MERGE-SORT (A, q + 1, r):** Since **q < r**, **q + 1 ≤ r**.
    - Therefore, in all recursive calls, **p ≤ r** is always true.
4. **Preventing p > r:**
    - Because each split ensures that the starting index **p** is always less than or equal to the ending index **r**, there's no situation where **p > r**.
    - The condition **"if $p \neq r$"** effectively checks for subarrays with more than one element (**p < r**), avoiding empty subarrays.

Conclusion:

By starting with **MERGE-SORT (A, 1, n)** where $\mathbf{n \geq 1}$ and using the condition **"if $\mathbf{p \neq r}$,"** all recursive calls maintain $\mathbf{p \leq r}$. This ensures that no subarray is empty ($\mathbf{p > r}$), allowing MERGE-SORT to function correctly without encountering any invalid recursive calls.

Question: 2.3-3

State a loop invariant for the **while** loop of lines 12-18 of the MERGE procedure.

Show how to use it, along with the **while** loops of lines 20-23 and 24-27, to prove

that the MERGE procedure is correct.

Answer:

Loop Invariant for the While Loop (Lines 12-18) of MERGE:

- **At the start of each iteration:**
    - The subarray $A[p..k-1]$ contains the smallest elements from the union of L ($A[p..q]$) and R ($A[q+1..r]$), sorted in ascending order.

Using the Loop Invariant to Prove MERGE is Correct:

1. **After the While Loop (Lines 12-18) Finishes:**
    - $A[p..k-1]$ contains the smallest elements from L and R, sorted.
    - Either L or R still has remaining elements.

2. Handling Remaining Elements (Lines 20-23 or 24-27):

- **If L has remaining elements:**
    - The while loop in lines 20-23 copies the remaining elements from L to A.
- **If R has remaining elements:**
    - The while loop in lines 24-27 copies the remaining elements from R to A.
- These remaining elements are already sorted, so appending them maintains the overall sorted order.

3. **Final State:**

- After both loops, $A[p..r]$ is fully sorted, containing all elements from L and R in order.

Question: 2.3-4

Use mathematical induction to show that when ≥ is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & if\ n = 2, \\ 2T(n/2) + n & if\ n > 2, \end{cases}$$

is $T(n) = nlgn$.

Answer:

To prove that $T(n) = nlgn$ is the solution to the recurrence:

$$T(n) = \begin{cases} 2 & if\ n = 2, \\ 2T(n/2) + n & if\ n > 2, \end{cases}$$

where n is an exact power of 2, we use mathematical induction.

Base Case:

When $n = 2$,

$$T(2) = 2 = 2log2$$

Thus, the solution holds for $n = 2$.

Inductive Step:

Assume that for some integer $k \geq 1$, $T(2^k) = 2^k log 2^k$ holds. We need to show that it also holds for $n = 2^{k+1}$, i.e., $T(2^{k+1}) = 2^{k+1} log\ 2^{k+1}$.

From our recurrence formula,

$$T(2^{k+1}) = 2T(2^{k+1}/2) + 2^{k+1}.$$

$$= 2T(2^k) + 2.2^k$$

$$= 2.2^k\ log\ 2^k + 2.2^k$$

$$= 2.2^k\ (log\ 2^k + 1)$$

$$= 2.2^k\ (log\ 2^k + log2)$$

$$= 2^{k+1} log 2^{k+1}$$

$$= nlogn$$

This completes the inductive step.

Conclusion:

Since both the base case and inductive step are verified, by mathematical induction, $T(n) = nlogn$ is the solution for all n that are exact powers of 2.

Question: 2.3-5

You can also think of insertion sort as a recursive algorithm. In order to sort $A[1:n]$, recursively sort the subarray $A[1:n-1]$ and then insert $A[n]$ into the sorted subarray $A[1:n-1]$. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

Answer:

Recursive Insertion Sort Pseudocode:

Recursive-Insertion-Sort (A, n)

```
1.  if n > 1
2.     Recursive − Insertion − Sort(A, n − 1)
3.     key = A[n]
4.     i = n − 1
5.     while i > 0 and A[i] > key
6.        A[i + 1] = A[i]
7.        i = i − 1
8.     A[i + 1] = key
```

Recurrence for Worst-Case Running Time

Let $T(n)$ represent the worst-case runtime of the recursive algorithm on an array of size n. The recurrence is:

$$T(n) = T(n-1) + O(n)$$

This solves to $\boldsymbol{T(n) = O(n^2)}$, similar to the iterative version of insertion sort.

Question: 2.3-6

Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further consideration. The ***binary search*** algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\theta lg\ n$.

Answer:

Binary Search Algorithm:

Binary search is an efficient search algorithm that works on a sorted array by repeatedly dividing the search interval in half. By comparing the midpoint element of the subarray to the target value v, binary search eliminates half of the remaining elements each time.

Pseudocode:

Iterative Version

BINARY_SEARCH_ITERATIVE (A, l, r, v)

```
while l <= r
    mid = (l + r) / 2
    if A[mid] == v
        return mid
    else if A[mid] > v
        r = mid − 1
    else
        l = mid + 1
return NIL  // v not found
```

BINARY_SEARCH_ITERATIVE (A, l, r, v)

```
while l <= r
    mid = (l + r) / 2
    if A[mid] == v
        return mid
    else if A[mid] > v
        r = mid − 1
    else
        l = mid + 1
return NIL // v not found
```

Recursive Version

BINARY_SEARCH_RECURSIVE (A, l, r, v)

```
    if l > r
        return NIL // v not found
    mid = (l + r) / 2
    if A[mid] == v
        return mid
    else if A[mid] < v
        return BINARY_SEARCH_RECURSIVE(A, mid + 1, r, v)
    else
        return BINARY_SEARCH_RECURSIVE(A, l, mid − 1, v)
```

Worst-Case Running Time Analysis

Binary search halves the search range each time. Starting with n elements, each step reduces it to $\frac{n}{2}$, $\frac{n}{4}$ and so on. After k steps, the size becomes $\frac{n}{2^k}$, reaching 1 when $k = log_2 n$. Therefore, the worst-case time complexity is:

$$\theta(logn)$$

This logarithmic complexity makes binary search very efficient for large, sorted arrays compared to linear search.

Question: 2.3-7

The while loop of lines 5-7 of the **INSERTION-SORT** procedure in Section *2.1* uses a linear search to scan (backward) through the sorted subarray $A[1: j-1]$. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to $\Theta(nlgn)$

Answer:

The loop in insertion sort has two tasks:

1. **Locate Position**: It uses a linear search to find where to insert key.
2. **Shift Elements**: It shifts elements greater than key to make room for insertion.

Using binary search reduces the number of comparisons (task 1) but does not avoid shifting elements (task 2), which still requires $\Theta(n)$ time. Therefore, even with binary search, the overall worst-case running time remains $\Theta(n^2)$, due to the shifting of elements.

Question: 2.3-8

Describe an algorithm that, given a set S of n integers and another integer x , determines whether S contains two elements that sum to exactly x . Your algorithm should take $\theta(n\ lg\ n)$ time in the worst case.

Answer:

To check if a set S of n integers contain two elements that sum to x, we can use sorting and a two-pointer technique.

Algorithm

1. **Sort S**: Sort the array S in $\theta(nlogn)$ time.

2. **Two-Pointer Search**:
    - Initialize pointers i (start of the array) and j (end of the array).
    - While $i < j$:
        - Compute $sum = S[i] + S[j]$.
        - If $sum == x$, return indices i and j.
        - If $sum < x$, increment i; if $sum > x$, decrement j.
    - If no pair is found, return NIL.

Pseudocode:

```
FUNCTION (S, x)
        n = S.length()
  MERGE_SORT (S, 1, n)  // θ(n log n) sorting step
        i = 1
        j = n
        while i < j
          sum = S[i] + S[j]
          if sum == x
            return i,j  // Pair found
          else if sum < x
            i = i + 1
          else
            j = j − 1
        return NIL  // No pair found
```

Time Complexity

- Sorting takes $\Theta(nlogn)$.
- The two-pointer search takes $\Theta(n)$.

Thus, the algorithm runs in $\Theta(nlogn)$ time.

Question: 2-1 Insertion sort on small arrays in merge sort

Although merge sort runs in $\Theta(nlogn)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to ***coarsen*** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which n/k sub lists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

Part (a): Show that insertion sort can sort the n/k sublists, each of length k, in Θ(nk) worst-case time.

Answer:

Sorting Sublists with Insertion Sort:

Each of the $\frac{n}{k}$ sublists of length k is sorted using insertion sort, which has a worst-case time complexity of $\theta(k^2)$ per sublist. Thus, the total time for sorting all sublists is:

$$\boldsymbol{\frac{n}{k} \cdot \theta(k^2) = \theta(nk)}$$

Part (b): Show how to merge the sublists in $\theta(n \lg(n/k))$ worst-case time.

Answer:

Merging Sublists:

To merge $\frac{n}{k}$ sorted sublists, we need log (n/k) levels of merging. Each level takes $O(n)$ time to merge all elements, resulting in a total merge time of:

$$\boldsymbol{\theta(nlog(n/k))}$$

Part (c): Given that the modified algorithm runs in $\theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of θ notation?

Answer:

Largest k for Same Complexity as Standard Merge Sort:

For the modified algorithm to match standard merge sort's $\Theta(nlog\ n)$ complexity, we set $k = \Theta(log\ n)$. This way, the overall time remains:

$$\boldsymbol{\Theta(nk + nlog(n/k)) = \Theta(nlogn)}$$

Part (d): How should you choose k in practice?

Answer:

Choosing k in Practice:

In practice, k should be a small constant (e.g., 10–50), as insertion sort is efficient on small arrays. Testing on the target machine can help determine the optimal k.

2-2 Correctness of bubblesort:

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order. The procedure BUBBLESORT sorts array $A[1:n]$.

```
BUBBLESORT(A, n)
 for i = 1 to n − 1
   for j = n downto i + 1
     if A[j] < A[j − 1]
       exchange A[j] with A[j − 1]
```

Part (a): Let A' denote the array A after **BUBBLESORT> (A, n)** is executed. To prove that it terminates and that

$$A'[1] \leq A'[2] \leq \cdots \leq A'[n]$$

In order to show that **BUBBLESORT** actually sorts, what else do you need to prove?

The next two parts prove inequality (2.5).

Answer:

Required Proof of Correctness:

To prove BUBBLESORT sorts correctly, we must show it terminates with A' as a sorted version of A, containing all the same elements in non-decreasing order. Since BUBBLESORT only swaps elements, the final array A' will be a permutation of the original array A.

Part (b): State precisely a loop invariant for the **for** loop in lines 2-4, and prove that this loop invariant holds. Your proof should use the structure of the loop-invariant proof presented in this chapter.

Answer:

Loop Invariant for Inner Loop (Lines 2-4):

Invariant: At the start of each iteration of the inner loop, the subarray $A[j:n]$ contains the elements originally in $A[j:n]$ in some order, with $A[j]$ being the smallest.

- **Initialization**: Before the first iteration, $A[n]$ is trivially the smallest in $A[n]$.
- **Maintenance**: In each iteration, if $A[j-1] > A[j]$, they are swapped, ensuring $A[j-1:n]$ has $A[j-1]$ as the smallest element.
- **Termination**: When $j = i + 1$, $A[i:n]$ contains the original elements, with $A[i]$ as the smallest.

Part (c): Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the **for** loop in lines 1-4 that allows you to prove inequal ity (2.5). Your proof should use the structure of the loop-invariant proof presented in this chapter.

Answer:

Loop Invariant for Outer Loop (Lines 1-4):

Invariant: At the start of each iteration of the outer loop, $A[1:i-1]$ contains the $i-1$ smallest elements in sorted order, with $A[i:n]$ containing the remaining unsorted elements.

- **Initialization**: At the start, $A[1:i-1]$ is empty, which is trivially sorted.

- **Maintenance**: After the inner loop, $A[i]$ becomes the smallest element in $A[i:n]$, so A[1:i] is sorted.
- **Termination**: When $i = n$, $A[1:n]$ is sorted.

Part (d): What is the worst-case running time of **BUBBLESORT**? How does it compare with the running time of **INSERTION-SORT**?

Answer:

Bubble Sort has a worst-case time complexity of $\theta(n^2)$ due to $n - i$ comparisons per pass. Although Insertion Sort also has a worst-case time complexity of $\theta(n^2)$, it performs fewer swaps than Bubble Sort, making it generally faster in practice.

Question: 2-3 Correctness of Horner's rule

You are given the coefficients $a_0, a_1, a_2, \ldots, a_n$ of a polynomial

$$P(x) = \sum_{k=o}^{n} a_k x^k$$

$$= a_0 + a_1 x, + a_2 x^2 + \ldots + a_{n-1} x^{n-1} + a_n x^n$$

and you want to evaluate this polynomial for a given value of x . ***Horner's rule*** says to evaluate the polynomial according to this parenthesization:

$$P(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) > \cdots)).$$

The procedure HORNER **implements** Horner**'s rule to > evaluate *P*(*x*), > given the coefficients $a_0, a_1, a_2, \ldots, > a_n$ in an array $A[0:n]$ and > the value of *x*.

```
HORNER (A,n,x)
  p = 0
  for i = n downto 0
    p = A[i] + x * p
  return p
```

Part (a): In terms of Θ-notation, what is the tunning time of this procedure?

Answer:

Running Time of HORNER:

From the pseudocode of Horner's Rule, the algorithm runs in a loop for all the elements, i.e. it runs at $\Theta(n)$ time.

Part (b): Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare with **HORNER**?

Answer:

Naive Polynomial Evaluation:

The naive method calculates each term $a_k x^k$ from scratch and sums them.

Pseudocode:

NAIVE_POLY_EVALUATE(A, n, x)

```
p = 0
for i = 0 to n
    term = A[i]
    for j = 1 to i
        term = term * x
    p = p + term
return p
```

Running Time:

This approach takes $\theta(n^2)$ time because calculating each power x^k requires multiple multiplications. HORNER is more efficient with $\Theta(n)$ time.

Part (c): Consider the following loop invariant for the procedure **HORNER**:

At the start of each iteration of the **for** loop of lines 2-3,

$$P(x) = \sum_{k=o}^{n-(i+1)} A[K+i+1] \,.\, x^k$$

Interpret a summation with no terms at equaling 0. Following the structure of th loop-invariant proof presented in this chapter, use this loop invariant to show that, at termination,

$p = \sum_{k=o}^{n} A[k] \,.\, x$

Answer:

Initialization: $i = n, p = \sum_{k=o}^{n} A[k+i+1] \,.\, x = 0$, loop invariant holds.

Maintenance: in the end of each iteration:

$$p = A[i] + x. \sum_{k=o}^{n-(i+1)} A[K+i+1] \,.\, x^k$$

$$= A[i] + \sum_{k=o}^{n-(i+1)} A[K+i+1] \,.\, x^{k+1}$$

$$= A[i].x^0 + \sum_{k=1}^{n-i} A[K+i] \,.\, x^k$$

$$= \sum_{k=o}^{n-i} A[K+i] \,.\, x^k$$

Decrementing i preserves the loop invariant.

Termination: The loop terminates at $i = -1$. If we substitute,

$$p = A[i] + x. \sum_{k=o}^{n-(i+1)} A[K+i+1] \,.\, x^k = \sum_{k=o}^{n} A[K] \,.\, x^k$$

Question:2-4 Inversions

Let $A[1:n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A.

Part (a): List the five inversions of the array $\langle 2,3,8,6,1 \rangle$.

Answer:

List of Inversions:

For the array $\langle 2,3,8,6,1 \rangle$, inversions occur when $i < j$ and $A[i] > A[j]$. The five inversions are:

$$(3,4), (3,5), (2,5), (1,5), (4,5)$$

Part (b): What array with elements from the set $\langle 1,2, \ldots, n \rangle$. has the most inversions? How many does it have?

Answer:

Array with Most Inversions:

The array in descending order, $\langle n, n-1, \ldots, 2,1 \rangle$, has the most inversions, with a total of:

$$\frac{n(n-1)}{2}$$

because every pair (i, j) with $i < j$ is an inversion.

Part (c): What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

Answer:

Relationship to Insertion Sort:

Insertion sort swaps elements to fix inversions, so its running time depends on the number of inversions. More inversions mean more swaps, making the time complexity $O(n + k)$, where k is the number of inversions. In the worst case, this is $O(n^2)$.

Part (d): Give an algorithm that determines the number of inversions in any permutation. on *n* elements in $\Theta(n \boldsymbol{lg} n)$ worse-case time. (Hint: Modify merge sort.)

Answer:

Counting Inversions in $\Theta(n \log n)$:

To count inversions efficiently, use a modified merge sort. During merging, count how many elements from the left side are greater than those on the right. Each time an element from the right subarray is placed before an element from the left subarray, it represents multiple inversions.

Modified Merge Sort Pseudocode:

COUNT_INVERSIONS(A, left, right)

$if\ left >= right$

$return\ 0$

$mid = (left + right) / 2$

$count = COUNT\_INVERSIONS(A, left, mid)$

$count \mathrel{+}= COUNT\_INVERSIONS(A, mid + 1, right)$

$count \mathrel{+}= MERGE\_AND\_COUNT(A, left, mid, right)$

$return\ count$

MERGE_AND_COUNT(A, left, mid, right)

$count = 0$

// **Merge and count inversions**

$return\ count$

This approach counts inversions during the merging process, resulting in $\Theta(nlogn)$ time complexity.

Chapter # 3

Question: 3.1-1

Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.

Answer:

To change the argument for insertion sort, here's a simple explanation:

1. **What Insertion Sort Does**:
   Insertion sort arranges a list by picking one item at a time and putting it in the right place among the already sorted items. It works worst when the list is sorted in reverse order.

2. **Counting Comparisons**:
   For a list with n items, the worst-case number of comparisons is:

$$\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}$$

OR

$$1 + 2 + 3 + \cdots + (n-1) = \frac{(n-1)n}{2}$$

This means it takes about $O(n^2)$comparisons.

3. **Any Size of n**:
   This argument works whether n is a multiple of 3 or not. Even if there are leftover items, insertion sort still needs to compare each new item with the sorted ones.

4. **Conclusion**:
   So, insertion sort always takes about $O(n^2)$ time, no matter how many items there are. This shows that insertion sort is not good for large lists.

In summary, insertion sort needs $O(n^2)$ time for any input size, making it inefficient for big lists.

Question: 3.1-2

Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2.

Answer:

Selection Sort Algorithm:

Selection sort arranges a list by repeatedly selecting the smallest element from the unsorted portion and moving it to the front. Here's how it works:

1. Start with the first element and assume it is the smallest.
2. Compare this element with all other elements to find the smallest one.
3. Swap the smallest found element with the first element.
4. Move to the next position and repeat until the entire array is sorted.

Pseudocode:

$for\ i\ from\ 0\ to\ n-1\ do$:

$min\_index\ =\ i$

$for\ j\ from\ i+1\ to\ n-1\ do$:

$if\ array[j]\ <\ array[min\_index]\ then$

$min\_index\ =\ j$

$swap\ array[i]\ with\ array[min\_index]$

Running Time Analysis:

1. **Comparisons**:
   - In the first pass, it compares n elements.
   - In the second pass, it compares n−1n - 1n−1 elements, and this continues until it compares just 1 element.
   - The total number of comparisons is:

$$1 + 2 + 3 + \cdots + (n-1) = \frac{(n-1)n}{2}$$

   This simplifies to $O(n^2)$.
2. **Swaps**:
   - There is one swap for each of the $n-1$ passes, so the number of swaps is also $O(n)$, but it is not the dominant factor.

Conclusion:

The overall running time of the selection sort algorithm is:

$$O(n^2)$$

This means selection sort is not efficient for large lists because its running time increases significantly as the number of elements grows.

Question: 3.1-3

Suppose that ˛ is a fraction in the range $0 < \alpha < 1$. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the αn largest values start in the first αn positions. What additional restriction do you need to put on α? What value of α maximizes the number of times that the αn largest values must pass through each of the middle $(1 - 2\alpha)n$ array positions?

Answer:

To analyze the lower-bound argument for insertion sort with the αn\alpha nαn largest values starting in the first αn positions, we can follow these steps:

Generalizing the Argument:

1. **Input Arrangement**:
    - We have an array of size n.
    - The first αn\alpha nαn positions contain the largest αn\alpha nαn values.
    - The remaining $(1 - \alpha)n$ values are mixed and can be smaller.
2. **Insertion Sort Process**:
    - Insertion sort processes the array from left to right, comparing each element with those that are already sorted.
    - Since the largest αn values start at the front, they will need to be placed correctly among the smaller values that follow.

Analyzing the Passes:

3. **Passes Through the Middle Positions**:
    - The middle positions consist of the values in the range from αn to $(1 - \alpha)n$.
    - As the αn largest values move to their final positions, they must pass through the middle $(1 - 2\alpha)n$ positions.

Restrictions on α\alphaα:

4. **Restrictions**:
    - To ensure that there are actual middle positions for the largest values to pass through, we need:

$$1 - 2\alpha > 0 \Rightarrow \alpha < \frac{1}{2}$$

Maximizing Passes:

5. **Maximizing the Number of Passes**:
    - The number of times the αn\alpha nαn largest values pass through the middle positions is maximized when α\alphaα is as large as possible, while still satisfying $\alpha < \frac{1}{2}$.
    - Therefore, the best value for α\alphaα to maximize the number of passes through the middle positions is:

$$\alpha \to \frac{1}{2}$$

Conclusion:

In summary, the lower-bound argument for insertion sort shows that when the αn largest values start in the first αn positions, they will still require multiple comparisons to find their correct place in the array. The restriction on α is that it must be less than $\frac{1}{2}$, and the value that maximizes the passes through the middle positions is α approaching $\frac{1}{2}$.

Question: 3.2-1

Let $f\ (n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition

of , -notation, prove that max $(f(n), g(n)) = \Theta(f(n) + g(n))$.

Answer:

To prove that $max(f(n), g(n)) = \Theta(f(n) + g(n))$:

1. Upper Bound:
Since $max(f(n), g(n)) \leq f(n) + g(n)$, we can set $c_2 = 1$ so:

$$max(f(n), g(n)) \leq c_2(f(n) + g(n))$$

2. Lower Bound:

We also have $max(f(n), g(n)) \leq f(n) + g(n)$, so we can set $c_1 = \frac{1}{2}$ such that:

$$c_1(f(n) + g(n)) \leq max(f(n), g(n))$$

Conclusion

With $c_1 = \frac{1}{2}$ and $c_2 = 1$, we get:

$$max(f(n), g(n)) \leq \theta(f(n) + g(n))$$

Question: 3.2-2

Explain why the statement, "The running time of algorithm A is at least $O(n^2)$" is meaningless.

Answer:

The statement "The running time of algorithm A is at least $O(n^2)$" is meaningless because $O(n^2)$is an upper bound, meaning the running time grows no faster than $O(n2)O(n\text{^}2)O(n2)$. To describe a minimum growth rate, we should use the lower bound notation $\Omega(n^2)$ instead.

Question: 3.2-3

Is $2^{n+1} = O(n^2)$? Is $2^{2n} = O(n^2)$?

Answer:

Is $2^{n+1} = O(n^2)$?

No, because 2^{n+1} grows exponentially, much faster than n^2.

Is $2^{2n} = O(n^2)$?

No, 2^{2n} also grows exponentially, far outpacing n^2.

Conclusion:

Both 2^{n+1} and 2^{2n} are not $O(n^2)$.

Question: 3.2-4

Prove Theorem 3.1.

Answer:

Proof of Theorem 3.1

Theorem 3.1 states that $f(n) = O(g(n))$ if there exist constants c> 0 and $n_0 > 0$ such that:

$$f(n) \leq c \cdot g(n)$$

for all $n \geq n_0$.

1. **Definition**: To prove $f(n) = O(g(n))$, we need constants c and n_0 so that $f(n)$is always less than or equal to $c \cdot g(n)$ for large n.

2. **Conclusion**: If such constants exist, then by definition, $f(n) = O(g(n))$.

Question: 3.2-5

Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case

running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

Answer:

To prove that $T(n) = \Theta(g(n))$ if and only if the worst-case running time is $O(g(n))$ and the best-case running time is $\Omega(g(n))$:

1. **If $T(n) = \Theta(g(n))$:**
    - By definition, $T(n)$ is bounded by $c_1 \cdot g(n) \leq T(n) \leq c_2 \cdot g(n)$ for constants c_1, c_2>0.
    - This implies $T(n)$ is $O(g(n))$ in the worst case (upper bound) and $\Omega(g(n))$ in the best case (lower bound).
2. **If worst-case is $O(g(n))$ and best-case is $\Omega(g(n))$:**

- The worst-case $O(g(n))$ gives an upper bound: $T(n) \leq c_2 \cdot g(n)$.
- The best-case $\Omega(g(n))$ gives a lower bound: $T(n) \geq c_1 \cdot g(n)$.
- Together, this implies $T(n) = \Theta(g(n))$, as it is both bounded above and below by $g(n)$.

Conclusion:

Thus, $T(n) = \Theta(g(n))$ if and only if the worst-case is $O(g(n))$ and the best-case is $\Omega(g(n))$.

Question: 3.2-6

Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

Answer:

To show $o(g(n)) \cap \omega(g(n))$ is empty:

1. $\boldsymbol{f(n) = o(g(n))}$:
   This means $f(n)$ grows strictly slower than $g(n)$.
2. $\boldsymbol{f(n) = \omega(g(n))}$:
   This means $f(n)$ grows strictly faster than $g(n)$.
3. **Conclusion**:
   A function cannot grow both strictly slower and strictly faster than $g(n)$ simultaneously, so $o(g(n)) \cap \omega(g(n))$ has no elements.

Question: 3.2-7

We can extend our notation to the case of two parameters n and m that can go to ∞ independently at different rates. For a given function $g(n, m)$, we denote by $O(g(n, m))$ the set of functions

$O(g(n,m)) = \{f(n,m)$: there exist positive constants c,n_o, and m_o

such that $0 \leq f(n,m) \leq cg(n,m)$

for all $n \geq n_o$ or $m \geq m_o$.}

Give corresponding definitions for $\Omega(g(n,m))$ and $(g(n,m))$.

Answer:

Here are the definitions for $\Omega(g(n,m))$ and $\Theta(g(n,m))$:

1. **$\Omega(g(n,m))$**:

   The set of functions $f(n,m)$ such that there exist constants $c > 0$, n_o, and m_o with:

   $$f(n,m) \geq c \cdot g(n,m) \qquad \text{for all } n \geq n_o \text{ or } m \geq m_o.$$

2. **$\Theta(g(n,m))$**:

   The set of functions $f(n,m)$ such that there exist constants c_1,c_2>0, n_o, and m_o with:

$cc_1 \cdot g(n,m) \leq f(n,m) \leq c_2 \cdot g(n,m)$for all $n \geq n_o$ *or* $m \geq m_o$.

These definitions ensure that $f(n,m)$ grows at least (for Ω) or exactly (for Θ) as fast as $g(n,m)$ when n or mmm becomes large.