Finding the Maximum Element in a Finite Sequence

The algorithm in pseudocode:

```
procedure max(a1, a2, ..., an: integers)
    max := a1
    for i := 2 to n
        if max < ai then max := ai
    return max[max is the largest element]
```
Binary Search

- the binary search algorithm in pseudocode.

```
procedure binary search(x: integer, a_1,a_2,..., a_n: increasing integers)
    i := 1 {i is the left endpoint of interval}
    j := n {j is right endpoint of interval}
    while i < j
        m := \lceil (i + j)/2 \rceil
        if x > a_m then i := m + 1
        else j := m
    if x = a_i then location := i
    else location := 0
    return location{location is the subscript i of the term a_i equal to x, or 0 if x is not found}
```

Binary Search

Find 10 in the following sequence

<table>
<thead>
<tr>
<th>Searching Sequence</th>
<th>a_m(as shown in the pseudocode)</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,3,4,5,6,7,8,9</td>
<td>5</td>
<td>none</td>
</tr>
</tbody>
</table>
Bubble Sort

- Bubble sort makes multiple passes through a list. Every pair of elements that are found to be out of order are interchanged.

Insertion Sort

- Insertion sort begins with the 2nd element. It compares the 2nd element with the 1st and puts it before the first if it is not larger.

{3, 7, 4, 9, 5, 2, 6, 1}

1. 3 7 4 9 5 2 6 1
2. 3 7 4 9 5 2 6 1
3. 3 4 7 9 5 2 6 1
4. 3 4 7 9 5 2 6 1
5. 3 4 5 7 9 2 6 1
6. 2 3 4 5 7 9 6 1
7. 2 3 4 5 6 7 9 1
8. 1 2 3 4 5 6 7 9
Definition 1: Let f and g be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $O(g(x))$ if there are constants $C$ and $k$ such that

$$|f(x)| \leq C|g(x)|$$

whenever $x > k$.

- This is read as “$f(x)$ is big-$O$ of $g(x)$” or “$g$ asymptotically dominates $f$.”
- The constants $C$ and $k$ are called witnesses to the relationship $f(x)$ is $O(g(x))$. Only one pair of witnesses is needed.
Using the Definition of Big-O Notation

Example: Show that $7x^2$ is $O(x^3)$.

Solution:
- When $x > 7$, $7x^2 < x^3$. Take $C = 1$ and $k = 7$ as witnesses to establish that $7x^2$ is $O(x^3)$.
- (Would $C = 7$ and $k = 1$ work?)

Using the Definition of Big-O Notation

Example: Show that $n^2$ is not $O(n)$.

Solution:
- Suppose there are constants $C$ and $k$ for which $n^2 \leq Cn$, whenever $n > k$. Then (by dividing both sides of $n^2 \leq Cn$ by $n$, then $n \leq C$ must hold for all $n > k$. A contradiction!)
Note the difference in behavior of functions as $n$ gets larger

Give a big-$O$ estimate for $(n \log n + 1)^2 + (\log n + 1)(n^2 + 1)$

Big-Omega Notation

**Definition 2:** Let $f$ and $g$ be functions from the set of integers or the set of real numbers to the set of real numbers. We say that $f(x)$ is $\Omega(g(x))$ if there are constants $C$ and $k$ such that

$$|f(x)| \geq C|g(x)|$$

when $x > k$.

- We say that “$f(x)$ is big-Omega of $g(x)$.”
- Big-$O$ gives an upper bound on the growth of a function, while Big-Omega gives a lower bound. Big-Omega tells us that a function grows at least as fast as another.
- $f(x)$ is $\Omega(g(x))$ if and only if $g(x)$ is $O(f(x))$. This follows from the definitions.
Big-Omega Notation

Example: Show that \( f(x) = 8x^3 + 5x^2 + 7 \) is \( \Omega(g(x)) \) where \( g(x) = x^3 \).

Solution:
- \( f(x) = 8x^3 + 5x^2 + 7 \geq 8x^3 \) for all positive real numbers \( x \).
- Is it also the case that \( g(x) = x^3 \) is \( O(8x^3 + 5x^2 + 7) \)?

Big-Theta Notation

Example: Show that \( f(x) = 3x^2 + 8x \log x \) is \( \Theta(x^2) \).

Solution:
- \( 3x^2 + 8x \log x \leq 11x^2 \) for \( x > 1 \), since \( 0 \leq 8x \log x \leq 8x^2 \).
  - Hence, \( 3x^2 + 8x \log x \) is \( O(x^2) \).
- \( x^2 \) is clearly \( O(3x^2 + 8x \log x) \)
- Hence, \( 3x^2 + 8x \log x \) is \( \Theta(x^2) \).
Example: Describe the time complexity of the algorithm for finding the maximum element in a finite sequence.

```
procedure max(a₁, a₂, ..., aₙ; integers)
max := a₁
for i := 2 to n
    if max < aᵢ then max := aᵢ
return max
```

Solution: Count the number of comparisons.
- The max < ai comparison is made n - 2 times.
- Each time i is incremented, a test is made to see if i ≤ n.
- One last comparison determines that i > n.
- Exactly 2(n - 1) + 1 = 2n - 1 comparisons are made.

Hence, the time complexity of the algorithm is Θ(n).