Running Time of Algorithms

CSE/IT 122 ~ Algorithms & Data Structures

FindMax of Sequence

- Find the max sequence of a finite sequence $a_1, a_2, a_3, \ldots, a_n$
- Algorithm

```
max := a_1                  // assign max to first element
for i:= 2 to n    // n = number of elements in sequence
    if max < a_i
        max = a_i
```

Analysis of FindMax

```
max := a_1              // cost c_1; number of times = 1
// check for fencepost error
for i:= 2 to n          // cost c_2; number of times = (n-2+1)+1 = n
    if max < a_i        // cost c_3; number of times = n - 1
        max = a_i       // does this execute? Depends?
```

➔ Best Cast, Worst Case, Average Case

Analysis of FindMax

- ➔ Best Case:
  - Loop doesn’t execute at all. Largest element is the first element
  - Then $T(n) = c_1+c_2n+c_3(n-1)$ which is $O(n)$
- ➔ Worst Case:
  - Loop executes every single time and largest element is the last element. (i.e. the sequence is SORTED)
  - Body of loop executes at cost of c_4 and $n - 1$ times
  - Then $T(n) = c_1+c_2n+c_3(n-1)+c_4(n-1)$ which is $O(n)$
- ➔ Average Case
  - Hase to be bounded between $O(n)$ and $O(n)$ which is of course $O(n)$

Linear Search

- Search a sequence of ordered integers $a_1, a_2, \ldots, a_n$ looking for a term in an ordered sequence that matches x. Common problem locate an element in an ordered list. Does the list have to be ordered?
- Algorithm

```
i := 1    // i is the location, subscript of the term that matches
while(i <= n and x ≠ a_i)
     i := i+1
if( i <= n)
     location := i
else
     Location := 0
```

Analysis of Linear Search

- ➔ Best Case:
  - First element matches. The while loop only runs once. The if loop only runs once and it's body is executed.
  - Which if statement to use? Pick the WORST case.
    - Why? Trying to find an upper bound
  - In this case both happen 1 time. So constant time!
- ➔ Worst Case
  - While loop executes n+1 times and body executes n times. Occurs when the item is not in the list. O(n)

Analysis of Linear Search

- ➔ Average Case
  - Assume that x is in the list (sequence). Then x = a_k, then the while loop happens k+(k-1)=2k-1 times. The condition of the while is done k times, thh body of the loop k-1 times
  - The other stuff happens 3 times. (Assume a cost of 1)
  - The average then is $\frac{(3+(2k-1))k}{k} = \frac{(3k+2k^2-k)}{k} = \frac{(2k^2+2k)}{k} = O(k)$

Binary Search

➔ Algorithm:

```
left := 1                  // the left endpoint of search interval
right := n                 // the right endpoint of the search interval
while left <= right
    mid := floor((left+right)/2)
    if x > a_m
        left := mid+1
    else
        right := mid
if x == a_left
    return left
else
    return 0
```

Analysis of Binary Search

- ➔ Worst Case
  - Assume that the number of terms in your sequence is $n = 2^k$, if not make $n = 2^k$ by adding zeros
  - For each time through the loop you get a count of 4. The while loop, the floor, the if, and the assignment.
  - But now each time through the sequence is halved. So $2^k/2 = 2^{k-1}$
  - So at each step you keep having the sequence $2^k, 2^{k-1}, 2^{k-2}, \ldots, 2^1, 2^0$ so will execute k times. So cost is 4k. Remember $n = 2^k$ and want cost in terms of k, so log n = k
  - And $4k = 4\log(n) = O(\log(n))$

Sorting Problem

- Formal Definition
  - Input: A sequence of n numbers $\langle a_1, a_2, a_3, \ldots, a_n \rangle$
  - Output: a permutation (reordering) $\langle a`_1, a`_2, a`_3, \ldots, a`_n \rangle$ of the input sequence such that $\langle a`_1 \leq a`_2 \leq a`_3 \leq \ldots \leq a`_n \rangle$
- How many permutations are there for n items?
- This is why brute force methods don't work.

Insertion Sort

- Works the way you sort a hand of cards based on value. Pick up the first card. Then pick up the next card and reading the cards from left to right you place the card in the right order. Continue on in this process.
- Key idea: sorted pile and an unsorted pile. Mathematically you partition the set into two parts.
  - Sorted partition
  - Unsorted partition
- Insertion sort does the sorting in-place. Assume you have an Array A consisting of n elements, an insertion sort re-arranges the numbers within the Array.

Insertion Sort

- Algorithm

```
for j := 2 to n        // n is the length of the array A[1..n]
    key := A[j]
    // insert A[j] into the sorted sequence A[1..j-1]
    i := j - 1
    while i > 0 and A[i] > key
        A[i + 1] := A[i]
        i := i = 1;
    A[i + 1] := key
```

- Example:
  - A = ⟨10,7,9,3,4⟩

Analysis of the Insertion Sort

Insertion Sort	cost	times
`for j := 2 to n`	c_1	n
`  key := A[j]`	c_2	$n-1$
`  //insert A[j] into the sorted sequence A[1..j-1]`	0	$n-1$
`  i := j - 1`	c_3	$n-1$
`  while i > 0 and A[i] > key`	c_4	$?$
`    A[i+1] := A[i]`	c_5	$?$
`    i := i - 1;`	c_6	$?$
`  A[i + 1] := key`	c_7	$n-1$

Analysis of Insertion Sort

➔ Best Case:

- The best case occurs if A is already sorted.
- Then A[i] <= key and i = j-1.
- So the while loop executes 1 times in this case for 1 to (n-1) times or 2 to n times
- The body of the while loop doesn't happen at all
- And A[i + 1] := key happens n - 1 times
- To figure out the best-case running time $T(n)_{best\_case}$ sum all the values:
  - $T(n)_{best\_case} = c_1n+c_2(n-1)+c_s(n-1)+c_4(n-1)+c_7(n-1)$
  - $= (c_1+c_2+c_3+c_4+c_7)n - (c_2+c_3+c_4+c_7) = k_1n - k_2$
  - So $T(n)_{best\_case} = O(n)$

Analysis of Insertion Sort

➔ Worst Case:

- The worst case happens when the array is sorted in reverse sort order
- The body of the while loop will execute until i = 0 in this case
- So when j = 2, the loop executes once for A[1] > key and once for the test i > 0. For a total of two times
- When j = 3, the loop executes once for A[2] > key and once for A[1] > key and once for test i > 0 for a total of 3 times
- And so on. In general, the loop happens j times. So the while loop happens

$$\sum_{j=2}^{n} j = n(n+1)/2$$

Analysis of Insertion Sort

- Worst Case, cont.
  - Likewise the elements inside the loop execute one time less than the loop, so both of those statements happen
  - $\sum_{j=2}^{n}(j-1) = (n-1)((n-1)+1)/2 = n(n-1)/2$
  - So summing all these terms yields

$$T(n)_{worst\_case} = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\tfrac{n(n+1)}{2} - 1) + c_5(\tfrac{n(n-1)}{2}) + c_6(\tfrac{n(n-1)}{2}) + c_7(n-1)$$
$$= (\tfrac{c_4}{2} + \tfrac{c_5}{2} + \tfrac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \tfrac{c_4}{2} - \tfrac{c_5}{2} - \tfrac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7)$$
$$\text{So } T(n)_{worst_case} = an^2 + bn + c = O(n^2)$$

Analysis of Insertion Sort

- Average Case:
  - The while loop is going to execute some t_j times and
  - $$\sum_{j=2}^{n} t_j = n(n+1)/2 - k_j$$
  - And the body of the while loop executes t_j - 1 times so they will execute a total of $$\sum_{j=2}^{n} t_j = n(n+1)/2 - k_j$$

- When summing the total number of times the statements will execute, you will end up still being $T(n)_{average\_case} = O(n^2)$