COMP 3711 Course Notes

Design and Analysis of Algorithms

LIN, Xuanyu

ALGORITHMS

COMP 3711 Design and Analysis of Algorithms

THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

September 13, 2023

1 Asymptotic Notation

Upper Bounds $T(n) = O(f(n))$
if exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $T(n) \leq c \cdot f(n)$.
Lower Bounds $T(n) = \Omega(f(n))$
if exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$, $T(n) \geq c \cdot f(n)$.
Tight Bounds $T(n) = \Theta(f(n))$
if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.
Note: Here ”=” means ”is”, not equal.

2 Introduction - The Sorting Problem

2.1 Selection Sort

Algorithm 1: Selection Sort

```
Input: An array A[1...n] of elements
Output: Array A[1...n] of elements in sorted order (asending)
for i ← 1 to n − 1 do
    for j ← i + 1 to n do
        if A[i] > A[j] then
            swap A[i] and A[j]
        end
    end
end
```

Running Time: $\frac{n(n-1)}{2}$
Best-Case = Worst-Case: $T(n) = \Theta(\frac{n(n-1)}{2}) = \Theta(n^2)$

2.2 Insertion Sort

Algorithm 2: Insertion Sort

```
Input: An array A[1...n] of elements
Output: Array A[1...n] of elements in sorted order (asending)
for i ← 2 to n do
    j ← i − 1 while j ≥ 1 and A[j] > A[j + 1] do
        swap A[j] and A[j + 1]
    end
    j ← j − 1
end
```

Running Time: Depends on the input array, ranges between $(n-1)$ and $\frac{n(n-1)}{2}$
Best-Case: $T(n) = n - 1 = \Theta(n)$ (Useless)
Worst-Case: $T(n) = \Theta(\frac{n(n-1)}{2}) = \Theta(n^2)$ (Commonly-Used)
Average-Case: $T(n) = \Theta(\sum_{i=2}^{n} \frac{i-1}{2}) = \Theta(\frac{n(n-1)}{4}) = \Theta(n^2)$ (Sometimes Used)

2.3 Wild-Guess Sort

Running Time: Depends on the random generation, could be faster than the insertion sort.

2.4 Worst-Case Analysis

The algorithm’s worst case running time is $O(f(n)) \implies$ On all inputs of (large) size n, the running time of the algorithm is $\leq c \cdot f(n)$.

Algorithm 3: Wild-Guess Sort

Input: An array $A[1...n]$ of elements

Output: Array $A[1...n]$ of elements in sorted order (asending)

$\pi \leftarrow [4, 7, 1, 3, 8, 11, 5, ...]$ Create random permutation Check if $A[\pi[i]] \leq A[\pi[i+1]]$ for all $i = 1, 2, ..., n-1$ If yes, output A according to π and terminate else $Insertion - Sort(A)$

The algorithm's worst case running time is $\Omega(f(n)) \implies$ There exists at least one input of (large) size n for which the running time of the algorithm is $\geq c \cdot f(n)$.

Thus, Insertion sort runs in $\Theta(n^2)$ time.

Notice

Selection sort, insertion sort, and wild-guess sort all have worst-case running time $\Theta(n^2)$. How to distinguish between them?

- Closer examination of hidden constants
- Careful analysis of typical expected inputs
- Other factors such as cache efficiency, parallelization are important
- Empirical comparison

Stirling's Formula

Prove that $\log(n!) = \Theta(n \log n)$

First $\log(n!) = O(n \log n)$ since:

$$\log(n!) = \sum_{i=1}^{n} \log i \leq n \times \log n = O(n \log n)$$

Second $\log(n!) = \Omega(n \log n)$ since:

$$\log(n!) = \sum_{i=1}^{n} \log i \geq \sum_{i=n/2}^{n} \log i \geq n/2 \times \log n/2 = n/2(\log n - \log 2) = \Omega(n \log n)$$

Thus, $\log(n!) = \Theta(n \log n)$

3 Divide & Conquer

Main idea of D & C: Solve a problem of size n by breaking it into one or more smaller problems of size less than n. Solve the smaller problems recursively and combine their solutions, to solve the large problem.

3.1 Binary Search

Example: Binary Search

Input: A sorted array $A[1, ..., n]$, and an element x

Output: Return the position of x, if it is in A; otherwise output nil

Idea of the binary search: Set $q \leftarrow$ middle of the array. If $x = A[q]$, return q. If $x < A[q]$, search $A[1, ..., q-1]$, else search $A[q+1, ..., n]$.

Algorithm 4: Binary Search

Input: Array $A[1...n]$ of elements in sorted order

```
BinarySearch(A[], p, r, x) (p, r being the left & right iteration, x being the element being searched) if p > r then
  | return nil
end
q ← [(p + r)/2]
if x = A[q] then
  | return q
end
if x < A[q] then
  | BinarySearch(A[], p, q − 1, x)
end
else
  | BinarySearch(A[], q + 1, r, x)
end
```

Recurrence of the algorithm, supposing $T(n)$ being the number of the comparisons needed for n elements: $T(n) = T(\frac{n}{2}) + 2$ if $n > 1$, with $T(1) = 2$.

$\implies T(n) = 2\log_2 n + 2 \implies O(\log n)$ algorithm

Example: Binary Search in Rotated Array

Suppose you are given a sorted array A of n distinct numbers that has been rotated k steps, for some unknown integer k between 1 and $n-1$. That is, $A[1...k]$ is sorted in increasing order, and $A[k+1...n]$ is also sorted in increasing order, and $A[n] < A[1]$.

Design an $O(\log n)$-time algorithm that for any given x, finds x in the rotated sorted array, or reports that it does not exist.

Algorithm:

First conduct a $O(\log n)$ algorithm to find the value of k, then search for the target value in either the first part or the second part.

$Find - x(A, x)$

$k \leftarrow Find - k(A, 1, n)$ (First find k)

$if\ x \geq A[1]\ then\ return\ BinarySearch(A, 1, k, x)$

$Else\ return\ BinarySearch(A, k+1, n, x)$

Example: Finding the last 0

You are given an array $A[1...n]$ that contains a sequence of 0 followed by a sequence of 1 (e.g., 0001111111). A contains k 0(s) ($k > 0$ and $k << n$) and at least one 1.

Design an $O(\log k)$-time algorithm that finds the position k of the last 0.

Algorithm:

$i \leftarrow 1$
while $A[i] = 0$
  $i \leftarrow 2i$
$find - k(A[i/2...i])$

3.2 Merge Sort

Principle of the Merge Sort:

- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

Algorithm 5: Merge Sort

```
MergeSort(A, p, r) (p, r being the left & right side of the array to be sorted)
if p = r then
    return
end
q ← [(p + r)/2]
MergeSort(A, p, q)
MergeSort(A, q + 1, r)
Merge(A, p, q, r)
First Call: MergeSort(A, 1, n)
```

Algorithm 6: Merge

```
Input: Two Arrays L ← A[p...q] and R ← A[q + 1...r] of elements in sorted order
Merge(A, p, q, r)
Append ∞ at the end of L and R
i ← 1, j ← 1
for k ← p to r do
    if L[i] ≤ R[j] then
        A[k] ← L[i]
        i ← i + 1
    end
    else
        A[k] ← R[j]
        j ← j + 1
    end
end
```

Let $T(n)$ be the running time of the algorithm on an array of size n.

Merge Sort Recurrence:

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n), \quad n > 1, \quad T(1) = O(1)$$

Simplification:

$$\implies T(n) = 2T(n/2) + n, \quad n > 1, \quad T(1) = 1$$

Result:

$$T(n) = n \log_2 n + n = O(n \log n)$$