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 14, 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 $i \leftarrow 1$ *to* $n-1$ **do**
  for $j \leftarrow i+1$ *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 $i \leftarrow 2$ *to* n **do**
  $j \leftarrow i-1$ **while** $j \geq 1$ *and* $A[j] > A[j+1]$ **do**
    swap $A[j]$ and $A[j+1]$
  end
  $j \leftarrow j-1$
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
$q \leftarrow [(p + r)/2]$
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
$q \leftarrow [(p + r)/2]$
`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 $L \leftarrow A[p...q]$ and $R \leftarrow A[q + 1...r]$ of elements in sorted order
`Merge`(A, p, q, r)
Append ∞ at the end of L and R
$i \leftarrow 1,\ j \leftarrow 1$
for $k \leftarrow p$ *to* r **do**
    if $L[i] \leq R[j]$ **then**
        $A[k] \leftarrow L[i]$
        $i \leftarrow i + 1$
    end
    else
        $A[k] \leftarrow R[j]$
        $j \leftarrow j + 1$
    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:

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

3.3 Inversion Counting

Definition of the Inversion Numbers: Given array $A[1...n]$, two elements $A[i]$ and $A[j]$ are inverted if $i < j$ but $A[i] > A[j]$. The inversion number of A is the number of inverted pairs.

Theorem:
The number of swaps used by Insertion Sort = Inversion Number (Proved by induction on the size of the array)
Algorithm to Compute Inversion Number:
Algorithm 1: Check all $\Theta(n^2)$ pairs.
Algorithm 2: Run Insertion Sort and count the number of swaps –Also $\Theta(n^2)$ time.
Algorithm 3: Divide and Conquer

3.3.1 Counting Inversions: Divide-and-Conquer

Principle of the Algorithm:
• Divide: divide array into two halves
• Conquer: recursively count inversions in each half
• Conbine: count inversions where a_i and a_j are in different halves, and return sum of three quantities
Inversion counting during the combine step is very similar to the Merge Algorithm (Algorithm 6), by counting the sum of each inversion number of the right array (indicated by $I[j]$) comparing to the left array.

Algorithm 7: Inversion Count during Combination

Input: Two Arrays $L \leftarrow A[p...q]$ and $R \leftarrow A[q+1...r]$ of elements in sorted order
`Count`(A, p, q, r)
$i \leftarrow 1,\ j \leftarrow 1,\ c \leftarrow 0$
while $(i \leq q-p+1)\&\&(j \leq r-q)$ **do**
  if $L[i] \leq R[j]$ **then**
    $i \leftarrow i+1$
  end
  else
    $I[j] = q-p-i+2$
    $c \leftarrow c+I[j]$
    $j \leftarrow j+1$
  end
end

The time-complexity of the algorithm is $\Theta(n \log n)$, same as the Merge Sort.

3.3.2 Implementation of the Algorithm

Algorithm 8: Main Algorithm

`Sort-and-Count`(A, p, r)
if $p = r$ **then**
  return *0*
end
$q \leftarrow \lfloor (p+r)/2 \rfloor$
$c_1 \leftarrow$ `Sort-and-Count`(A, p, q)
$c_2 \leftarrow$ `Sort-and-Count`$(A, q+1, r)$
$c_3 \leftarrow$ `Merge-and-Count`(A, p, q, r)
return $c_1 + c_2 + c_3$
First Call: `Sort-and-Count`$(A, 1, n)$

Algorithm 9: Merge-and-Count

Input: Two Arrays $L \leftarrow A[p...q]$ and $R \leftarrow A[q+1...r]$ of elements in sorted order
`Merge-and-Count`(A, p, q, r)
Append ∞ at the end of L and R
$i \leftarrow 1,\ j \leftarrow 1, c \leftarrow 0$
for $k \leftarrow p$ *to* r **do**
  if $L[i] \leq R[j]$ **then**
    $A[k] \leftarrow L[i]$
    $i \leftarrow i + 1$
  end
  else
    $A[k] \leftarrow R[j]$
    $j \leftarrow j + 1$
    $c \leftarrow c + q - p - i + 2$
  end
end
return c

3.4 D&C: Observations on Problem Size and Number of Problems

Observations of D&C in Logarithmic Patterns:
- Break up problem of size n into p parts of size n/q.
- Solve parts recursively and combine solutions into overall solution.
- At level i, we break i times and we have p^i problems of size n/q^i.
- When we cannot break up any more, usually when the problem size becomes 1. Usually $i \approx \log_q n$.

The number of problems at (bottom) level $\log_q n$ is $p^i = p^{\log_q n} = n^{\log_q p}$.

Observations of D&C in Non-Logarithmic Patterns:
- Break up problem of size n into $p(\leq 2)$ parts of size $n - q$. (e.g. $q = 1$ for Hanoi Problem)
- Assume that $q = 1$
- At level i, we break i times and we have p^i problems of size $n - i$.
- If we stop when the problem size becomes 1, then $n - i = 1 \implies i = n - 1$.

The number of problems at (bottom) level $n - 1$ is: $p^i = p^{n-1}$.

3.5 Maximum Contiguous Subarray

Example: The Maximum Subarray Problem
Input: An array of numbers $A[1, ..., n]$, both positive and negative
Output: Find the maximum $V(i, j)$, where $V(i, j) = \sum_{k=i}^{j} A[k]$

Brute-Force Algorithm

Idea: Calculate the value of $V(i, j)$ for each pair $i \leq j$ and return the maximum value.
Requires three nested for-loop, time complexity: $\Theta(n^3)$.

A Data-Reuse Algorithm

Idea: $V(i, j) = V(i, j-1) + A[j]$
Requires two nested for-loop, time complexity: $\Theta(n^2)$.

A D&C Algorithm

Idea: Cut the array into two halves, all subarrays can be classified into three cases: entirely in the first/second half, or crosses the cut.

Compare with the merge sort: Whole algorithm will run in $\Theta(n \log n)$ time if the cross-cut can be solved in $O(n)$ time.

Algorithm 10: Maximum Subarray

```
MaxSubArray(A, p, r)
if p = r then
|  return A[p]
end
q ← ⌊(p + r)/2⌋
M_1 ← MaxSubArray(A, p, q)
M_2 ← MaxSubArray(A, q + 1, r)
L_m, R_m ← −∞
V ← 0
for i ← q to p do
|  V ← V + A[i]
|  if V > L_m then
|  |  L_m ← V
|  end
end
V ← 0
for i ← q + 1 to r do
|  V ← V + A[i]
|  if V > R_m then
|  |  R_m ← V
|  end
end
return max M_1, M_2, L_m + R_m
First Call: MaxSubArray(A, 1, n)
```

Recurrence: $T(n) = 2T(n/2) + n \implies T(n) = \Theta(n \log n)$