VE281

Data Structures and Algorithms

Analyzing Algorithms; Sorting

Announcement

- Written Assignment One Released
  - Find the description on Canvas
  - Due time: 3:40 pm, Sep. 26th, 2016

- Midterm exam time: in lecture on Oct. 31st, 2016

Outline

- Analyzing Time Complexity of Programs
- Sorting Basics
- Merge Sort

Review

- Asymptotic Analysis: Big-Oh
  - Common functions and their growth rates
- Relatives of Big-Oh
  - Big-Omega
  - Theta

Analyzing Time Complexity of Programs

- For atomic statement, such as assignment, its complexity is $\Theta(1)$.

- For branch statement, such as if-else statement and switch statement, its complexity is that of the most expensive Boolean expression plus that of the most expensive branch.

```
if(Boolean_Expression_1) {Statement_1}
else if (Boolean_Expression_2) {Statement_2}
…
else if (Boolean_Expression_n) {Statement _n}
else {Statement_For_All_Other_Possibilities}
```

Analyzing Time Complexity of Programs

- For subroutine call, its complexity is that of the subroutine.

- For loops, such as while and for loop, its complexity is related the number of operations required in the loop.

Time Complexity Example One

- What is the time complexity of the following code?

```
sum = 0;
for(i = 1; i <= n; i++)
  sum += i;
```

- The entire time complexity is $\Theta(n)$.

Time Complexity Example Two

- What is the time complexity of the following code?

```
sum = 0;
for(i = 1; i <= n; i++)
    for(j = 1; j <= i; j++)
        sum++;
```

- Note that the statements

```
j <= i;
j++;
sum++;
```

  all occur (roughly) $1 + 2 + \cdots + n = n(n+1)/2$ times.

- The time complexity is $\Theta(n^2)$.

Time Complexity Example Three

- What is the time complexity of the following code?

```
sum = 0;
for(i = 1; i <= n; i *= 2)
  for(j = 1; j <= n; j++)
    sum++;
```

- The outer loop occurs $\log n$ times.
- The statements `sum++ / j<=n / j++` occur $n \log n$ times.
- The time complexity is $\Theta(n \log n)$.

Time Complexity Example Four

- What is the time complexity of the following code?

```
sum = 0;
for(i = 1; i <= n; i *= 2)
   for(j = 1; j <= i; j++)
      sum++;
```

- The number of times that the statements **`sum++ / j<=i / j++`** occur is

$$1 + 2 + 4 + 8 + \cdots 2^{\log n} \approx 2n - 1$$

- The time complexity is $\Theta(n)$.

Multiple Parameters

- Example: Compute the rank ordering for all C (i.e., 256) pixel values in a picture of P (i.e., 64×64) pixels.

```
for(i=0; i<C; i++)    // Initialize count
    count[i] = 0;
```

$\Theta(C)$

```
for(i=0; i<P; i++)    // Look at all pixels
    count[value[i]]++; // Increment count
```

$\Theta(P)$

```
sort(count);           // Sort pixel counts
```

$\Theta(C \log C)$

- The time complexity is $\Theta(P + C \log C)$.

Space/Time Trade-off Principle

- One can often reduce time if one is willing to sacrifice space, or vice versa.

- Example: factorial
  - Iterative method: Get “n!” using a for-loop.
  - This requires $\Theta(1)$ memory space and $\Theta(n)$ runtime.

  - Table lookup method: Pre-compute the factorials for $1, 2, \cdots, N$ and store all the results in an array.
  - This requires $\Theta(n)$ memory space and $\Theta(1)$ runtime (fetching from an array).

Outline

- Analyzing Time Complexity of Programs
- Sorting Basics
- Merge Sort

Sorting

- Given array A of size N, reorder A so that its elements are in order.
  - "In order" with respect to a consistent comparison function, such as "$\leq$" or "$\geq$".

- Sorting order
  - Ascending order
  - Descending order
- Unless otherwise specified, we consider sorting in ascending order.

Characteristics of Sorting Algorithms

- Average case time complexity
- Worst case time complexity
- Space usage: **in place** or not?
  - **in place**: requires $O(1)$ additional memory.
  - Don't forget the stack space used in recursive calls.
- **Stability**: whether the algorithm maintains the relative order of records with equal keys.
  - Usually there is a secondary key whose ordering you want to keep. Stable sort is thus useful for sorting over multiple keys.

(4, b), (3, e), (3, b), (5, b) → (3, e), (3, b), (4, b), (5, b)

Sort on the first number

Stable!

Types of Sorting Algorithms

- Sorting algorithms can be classified as **comparison sort** and **non-comparison sort**.

- **Comparison sort**: each item is compared against others to determine its order.

- **Non-comparison sort**: each item is put into predefined "bins" independent of the other items presented.
  - No comparison with other items needed.
  - It is also known as **distribution-based sort**.

Types of Sorting Algorithms

- General types of comparison sort
  - Insertion-based: insertion sort
  - Selection-based: selection sort, heap sort
  - Exchange-based: bubble sort, quick sort
  - Merging-based: merge sort

- Non-comparison sort:
  counting sort, bucket sort, radix sort

Insertion Sort

- **`A[0]`** alone is a sorted array.
- For **`i=1`** to **`N-1`**
  - **Insert** **`A[i]`** into the appropriate location in the sorted array **`A[0], …, A[i-1]`**, so that **`A[0], …, A[i]`** is sorted.
  - To do so, save **`A[i]`** in a temporary variable **`t`**, shift sorted elements greater than **`t`** right, and then insert **`t`** in the gap.
- Time comlexity? $O(N^2)$
- In place? Yes. O(1) additional memory.
- Stable?
  - Yes, because elements are visited in order and equal elements are inserted after its equals.

Insertion Sort
Best Case Time Complexity

- For **`i=1`** to **`N-1`**
  - **Insert** **`A[i]`** into the appropriate location in the sorted array **`A[0], …, A[i-1]`**, so that **`A[0], …, A[i]`** is sorted.

- The **best case** time complexity is $O(N)$.
  - It happens when the array is already sorted.
  - For other sorting algorithms we will talk, their best case time complexity is $\Omega(N \log N)$.

Selection Sort

- For **`i=0`** to **`N-2`**
  - Find the smallest item in the array **`A[i], …, A[N-1]`**. Then, swap that item with **`A[i]`**.
- Finding the smallest item requires **linear search**.
- Time complexity?
  - $O(N^2)$ best case?
- In place?
  - Yes. O(1) additional memory.
- Stable?
  - No. (3, e), (3, b), (2, a) → (2, a), (3, b), (3, e)

Bubble Sort

```
For i=N-2 downto 0
  For j=0 to i
    If A[j]>A[j+1] swap A[j] and A[j+1]
```

- Compares two adjacent items and swap them to keep them in ascending order.
  - From the beginning to the end. The last item will be the largest.

- Time complexity? $O(N^2)$
- In place? Yes.
- Stable?
  - Yes, because equal elements will not be swapped.

Two Problems with Simple Sorts

- They learn only one piece of information per comparison and hence might compare every pair of elements.
  - Contrast with binary search: learns N/2 pieces of information with first comparison.
- They often move elements one place at a time (bubble sort and insertion sort), even if the element is "far" from its **final place**.
  - Contrast with selection sort, which moves each element exactly to its final place.

- Fast sorts attack these two problems.
  - Two famous ones: **merge sort** and **quick sort**.

Outline

- Analyzing Time Complexity of Programs
- Sorting Basics
- Merge Sort

Merge Sort
Algorithm

- Spilt array into two (roughly) equal subarrays.
- Merge sort each subarray recursively.
  - The two subarrays will be sorted.
- Merge the two sorted subarrays into a sorted array.

[8, 3, 13, 6, 2, 14, 5, 9, 10]

[8, 3, 13, 6, 2]   [14, 5, 9, 10]

⋮   ⋮

[2, 3, 6, 8, 13]   [5, 9, 10, 14]

[2, 3, 5, 6, 8, 9, 10, 13, 14]

Merge Sort

Pseudo-code

```
void mergesort(int *a, int left, int
  right) {
    if (left >= right) return;
    int mid = (left+right)/2;
    mergesort(a, left, mid);
    mergesort(a, mid+1, right);
    merge(a, left, mid, right);
}
```

Merge Two Sorted Arrays

- For example, merge A = (2, 5, 6) and B = (1, 3, 8, 9, 10).

- Compare the smallest element in the two arrays A and B and move the smaller one to an additional array C.
- Repeat until one of the arrays becomes empty.
- Then append the other array at the end of array C.

Merge Two Sorted Arrays

Implementation

- We actually do not “remove” element from arrays A and B.
  - We just keep a pointer indicating the smallest element in each array.
  - We “remove” element by incrementing that pointer.

```
i = j = k = 0;
while(i < sizeA && j < sizeB) {
  if(A[i]<=B[j]) C[k++]=A[i++];
  else C[k++]=B[j++];
}
if(i == sizeA) append(C, B);
else append(C, A)
```

Time complexity?

Time complexity is $O(sizeA + sizeB)$

Merge Sort

Time Complexity

```
void mergesort(int *a, int left, int
   right) {
      if (left >= right) return;
      int mid = (left+right)/2;
      mergesort(a, left, mid);
      mergesort(a, mid+1, right);
      merge(a, left, mid, right);
}
```

$T(N/2)$

$T(N/2)$

$O(N)$

- Let $T(N)$ be the time required to merge sort N elements.
- Merge two sorted arrays with total size N takes $O(N)$.

Recursive relation: $T(N) = 2T(N/2) + O(N)$

How to solve the recurrence?

Solve Recurrence: Master Method

- A "black box" for solving recurrence.
- However, there is an important assumption: all sub-problems have roughly **equal** sizes.
  - E.g., merge sort
  - Not apply to unbalanced division.

Solve Recurrence: Master Method

- Recurrence: $T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$
  - Base case: $T(n) \le constant$ for all sufficiently small n.
  - a = number of recursive calls (integer ≥ 1)
  - b = input size shrinkage factor (integer > 1)
  - $O(n^d)$: the runtime of merging solutions. d is real value ≥ 0.
  - a, b, d are independent of n.
- <u>Claim</u>:

base doesn't matter

$$T(n) = \begin{cases} O(n^d \log n) & if\ a = b^d \\ O(n^d) & if\ a < b^d \\ O(n^{\log_b a}) & if\ a > b^d \end{cases}$$

base matters!

Example of Merge Sort

Recurrence: $$T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$$

Claim: $$T(n) = \begin{cases} O\left(n^d \log n\right) & if\ a = b^d \\ O\left(n^d\right) & if\ a < b^d \\ O\left(n^{\log_b a}\right) & if\ a > b^d \end{cases}$$

- $a = 2, b = 2, d = 1 \quad \Rightarrow b^d = a$
- $T(n) = O(n \log n)$

Another Example: Binary Search

Recurrence: $$T(n) \le aT\left(\frac{n}{b}\right) + O(n^d)$$

Claim: $$T(n) = \begin{cases} O\left(n^d \log n\right) & if\ a = b^d \\ O\left(n^d\right) & if\ a < b^d \\ O\left(n^{\log_b a}\right) & if\ a > b^d \end{cases}$$

- Exercise: What is a, b, d?

Merge Sort

Characteristics

- Not in-place
  - For efficient merging two sorted arrays, we need an auxiliary $O(N)$ space.
  - Recursion needs up to $O(\log N)$ stack space.

- Stable if **`merge()`** **maintains** the relative order of equal keys.

Divide-and-Conquer Approach

- Merge sort uses the **divide-and-conquer** approach.

- Recursively **breaking** down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly.
  - For merge sort, split an array into two and sort them respectively.

- The solutions to the sub-problems are then **combined** to give a solution to the original problem.
  - For merge sort, merge two sorted arrays.