Part 5. Computational Complexity (2)

Outline

- Complexity of Algorithms
- Efficiency of Searching Algorithms
- Sorting Algorithms and Their Efficiencies

(revisit) Properties of Growth-rate functions (1/3)

1. You can ignore low-order terms in an algorithm’s growth-rate function.
   - $O(n^3 + 4n^2 + 3n)$ it is also $O(n^3)$

(revisit) Properties of Growth-rate functions (2/3)

2. You can ignore a multiplicative constant in the high-order term of an algorithm’s growth-rate function
   - $O(5n^3)$, it is also $O(n^3)$

(revisit) Properties of Growth-rate functions (3/3)

3. You can combine growth-rate functions
   - $O(n^2) + O(n)$, it is also $O(n^2)$
   - Which you write as $O(n^2)$

Examples

- Determine whether each of these functions is $O(x^2)$
  - $f(x) = 17x + 11$
  - $f(x) = x^2 + 1000$
  - $f(x) = x\log x$
  - $f(x) = x^4/2$
  - $f(x) = 2^x$
### Examples

- Is \((x^2 + 1)/(x + 1) = O(x)\)?

### What is the Satisfied Solution?

- Algorithm should provide **correct** answer.
- Algorithm should be **efficient**.

### Demonstrating Efficiency

- Computational complexity of the algorithm
  - Time complexity
  - Space complexity
    - Analysis of the computer memory required
    - Data structures used to implement the algorithm

### Best, Average, and Worst Cases

- **Worst case**
  - Just how bad can it get:
    - The maximal number of steps
- **Average case**
  - Amount of time expected "usually"
- **Best case**
  - The smallest number of steps

### Outline

- Complexity of Algorithms
- **Efficiency of Searching Algorithms**
- Sorting Algorithms and Their Efficiencies

### Sequential Search

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
</table>

- Array of \(n\) items
  - From the first one until either you find the item or reach the end of the array.
  - Best case: \(O(1)\)
  - Worst case: \(O(n)\) (\(n\) times of comparison)
  - Average case: \(O(n)\) (\(n/2\) comparison)
Binary Search (1/3)

• Searches a sorted array for a particular item by repeatedly dividing the array in half.
• Determines which half the item must be in and discards other half.
• Suppose that \( n = 2^k \) for some \( k \). \((n=1,2,4,8,16,...)\)
  1. Inspect the middle item of size \( n \)
  2. Inspect the middle item of size \( n/2 \)
  3. Inspect the middle item of size \( n/2^2 \)
  4. .
  5. .
  6. .

Binary Search (2/3)

• Dividing array in \textit{half} \( k \) times.
• Worst case
  • Algorithm performs \( k \) divisions and \( k \) comparisons.
  • Since \( n = 2^k \), \( k = \log_2 n \)
  • \( O(\log_2 n) \)

Binary Search (3/3)

• What if \( n \) is not a power of 2?
• We can find the smallest \( k \) such that,

\[
2^{k-1} < n < 2^k
\]

\[
k - 1 < \log_2 n < k
\]

\[
1 + \log_2 n < k + 1
\]

\[
k = 1 + \log_2 n \text{ rounded down}
\]

Therefore, the algorithm is still \( O(\log_2 n) \).

Is Binary Search is more Efficient than Linear Search? (1/2)

• For large number, \( O(\log_2 n) \) requires significantly less time than \( O(n) \)
• For small numbers such as \( n < 25 \), does not show big difference.

Is Binary Search is more Efficient than Linear Search? (2/2)

Outline

• Complexity of Algorithms
• Efficiency of Searching Algorithms
• \textit{Sorting Algorithms and Their Efficiencies}
### Sorting Algorithm

- Organize a collection of data into either **ascending** or **descending** order.

  - **Internal sort**
    - Collection of data fits entirely in the computer’s main memory
  - **External sort**
    - Collection of data will not fit in the computer’s main memory all at once.

- We will only discuss **internal sort**.

### Selection Sort (1/2)

<table>
<thead>
<tr>
<th>Initial Array</th>
<th>After 1st swap</th>
<th>After 2nd swap</th>
<th>After 3rd swap</th>
<th>After 4th swap</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 10 14 37 13</td>
<td>29 10 14 13 37</td>
<td>13 10 14 29 37</td>
<td>13 10 14 29 37</td>
<td>10 13 14 29 37</td>
</tr>
</tbody>
</table>

### Selection Sort (2/2)

- Comparison
  
  \[(n - 1) + (n - 2) + (n - 3) + \ldots + 1 = \frac{n(n - 1)}{2}\]

- Exchange
  
  \[3 \times (n - 1)\]

- Comparison + Exchange
  
  \[\frac{n(n - 1)}{2} + 3 \times (n - 1) = n^2/2 + 3n/2 - 3\]

- Therefore, complexity is \(O(n^2)\)

### Bubble Sort (1/6)

- Compares adjacent items and exchanges them if they are out of order.

- It requires several passes over the data.

- After the first pass, the biggest item has “bubbled” to the end of the array.

### Bubble Sort (2/6)

**Pass 1**

<table>
<thead>
<tr>
<th>Initial Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>29 10 14 37 13</td>
</tr>
</tbody>
</table>

Performs comparisons 4 times

<table>
<thead>
<tr>
<th>Performs comparisons 4 times</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 29 14 37 13</td>
</tr>
<tr>
<td>10 14 29 37 13</td>
</tr>
<tr>
<td>10 14 29 37 13</td>
</tr>
<tr>
<td>10 14 29 13 37</td>
</tr>
</tbody>
</table>

### Bubble Sort (3/6)

**Pass 2**

<table>
<thead>
<tr>
<th>Performs comparisons 3 times</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 14 29 13 37</td>
</tr>
<tr>
<td>10 14 29 13 37</td>
</tr>
<tr>
<td>10 14 13 29 37</td>
</tr>
</tbody>
</table>

**Maximum number of passes?**
Bubble Sort (4/6)

- Analysis
  - At most \( n - 1 \) passes
    - Pass 1 requires \( n - 1 \) comparisons and maximum \( n - 1 \) exchanges
    - Pass 2 requires \( n - 2 \) comparisons and maximum \( n - 2 \) exchanges
    - ... 
    - Pass \( i \) requires \( n - i \) comparisons and maximum \( n - i \) exchanges
  - Until there is no exchange happens in the pass (sorted)

Bubble Sort (5/6)

- Worst case
  - Comparison:
    \( (n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2 \)
  - Exchanges:
    \( (n - 1) + (n - 2) + \ldots + 1 = n(n - 1)/2 \) times.
    Each exchange requires three data moves.
    Therefore, \( 3n(n - 1)/2 \)
  - Total
    \( 4n(n - 1)/2 + 2n(n - 1) - 2n \)
    \( \Theta(n^2) \), in the worst case

Bubble Sort (6/6)

- Best case (original data is already sorted)
  - Comparison: \( n - 1 \)
  - Exchanges: no exchange
  - Total
    \( \Theta(n) \), in the best case

Insertion Sort (1/6)

- Partitions the array into two regions: sorted and unsorted.
- Get the first unsorted item and find where to insert to make sorted array.
- Increase sorted part.

Insertion sort (2/6)

If there is only ONE item, it is sorted.

Initial Array 29 10 14 37 13

Copy 10
Shift 29
Insert 10; copy 14
Shift 29
Insert 14; copy 37

Sorted Array 10 14 29 37 13

Insert 37 on top of itself

Insertion sort (3/6)

Insert 37 on top of itself

10 14 29 37 13

Copy 13
Shift 37, 29, 14

10 14 13 29 37

Insert 37 on top of itself

10 13 14 29 37

Shift 37, 29, 14

Insert 37 on top of itself
Inser1on	
 sort	
 (4/6)
1: public static <T extends Comparable>
  void insertionSort (T[] theArray, int n){
3: for (int unsorted = 1; unsorted < n; ++unsorted){
4: T nextItem = theArray[unsorted];
5: int loc = unsorted;
6: while ((loc >0) && (theArray[loc-1].compareTo(nextItem)>
7: theArray[loc] = theArray[loc-1];
8: loc--;
9: }
10: theArray[loc] = nextItem;
11: }
12: }

Inser1on	
 sort	
 (5/6)
• Analysis
  – (Line 3 in the code) Unsorted/sorted point will move (n-1) times
    for each of the points
    • First unsorted data (mth) is copied (1 move)
    • (n-1)
  – (Line 6 in the code) For (m-1) sorted items, check the sorted part if any item is bigger than copied item (m-1 times) and if there is sorted item larger than the copied mth item, shift sorted item to the next position in the array (maximum m -1 times).
    • Insert copied mth item.(1 move)
    • m grows from 1 to n-1.
    • (1+2+…+(n-1))+(1+2+…+(n-1))+(n-1)

Total,
(n-1)+(1+2+…+(n-1))+(1+2+…+(n-1))+(n-1)
= 2n(n-1)/2 + 2(n-1)
= n^2 + n - 2
• O(n^2) in the worst case

Mergesort (1/12)
• Recursive sorting algorithm
• Gives the same performance
• Divide-and-conquer
  – Divide the array into halves
  – Sort each half
  – Merge the sorted halves into one sorted array

Mergesort (3/12)
public static
void mergesort(Comparable[] theArray, Comparable[] tempArray, int first, int last){
 if (first < last){
  int mid = (first + last)/2;
  mergesort(theArray, tempArray, first, mid);
  mergesort(theArray, tempArray, mid+1, last);
  merge (theArray, tempArray, first, mid, last);
 }
Mergesort (5/12)

```java
private static void merge (Comparable[] theArray, Comparable[] tempArray, int first, int mid, int last({
    int first1 = first;
    int last1 = mid;
    int first2 = mid+1;
    int last2 = last;
    int index = first1;
    while ((first1 <= last1) && (first2 <= last2)){
        if( theArray[first1].compareTo(theArray[first2])<0) {
            tempArray[index] = theArray[first1];
            first1++;
        } else{
            tempArray[index] = theArray[first2];
            first2++;
        }
        index++;
    }
    for (index = first; index <= last; ++index){
        theArray[index ] = tempArray[index];
    }
}) //end merge
```
**Mergesort (10/12)**

- If $n$ is a power of 2 (i.e. $n = 2^k$), then the recursion goes $k = \log_2 n$ levels deep.
- If $n$ is not a power of 2, there are $1 + \log_2 n$ (rounded down) levels of recursive calls to mergesort.

**Mergesort (10/12)**

- At level 0, the original call to mergesort calls merge once. (requires $3n - 1$ operations)
- At level 1, two calls to mergesort and each of them will call merge.
  - Total $2 \cdot (2 \cdot (n/2) - 1)$ operations required
- At level $m$, $2^m$ calls to merge occur.
  - Each of them will call merge with $n/2^m$ items and each of them requires $3(n/2^m)$-operations. Together, $3n \cdot 2^m$ operations are required.
- Because there are $\log_2 n$ or $1 + \log_2 n$ levels, total $O(n \cdot \log_2 n)$

**Mergesort (11/12)**

- Since there are either $\log_2 n$ or $1 + \log_2 n$ levels, mergesort is $O(n \cdot \log_2 n)$ in both the worst and average cases.
- **Significantly faster** than $O(n^2)$

**Quicksort (1/9)**

1. Select a **pivot** item.
2. Subdivide array into 3 parts
   - Pivot in its sorted position
   - Subarray with elements $< pivot$
   - Subarray with elements $\geq pivot$
3. Recursively apply to each sub-array

**Quickstart**

```c
void quicksort( void *a, int low, int high ) {
    int pivot;
    /* Termination condition! */
    if ( high > low ) {
        pivot = partition( a, low, high );
        quicksort( a, low, pivot-1 );
        quicksort( a, pivot+1, high );
    }

    int pivot_item;
    pivot = low + high/2;
    pivot_item = a[pivot];
    right = high; left = low;
    while ( left < right ) {
        /* Move left while item $< pivot */
        while (a[left] <= pivot_item) left++;
        /* Move right while item $> pivot */
        while (a[right] > pivot_item) right--;
        if ( left < right ) {
            /* Move left while item $< pivot */
            while (a[left] $< pivot_item) left++;
            /* Move right while item $> pivot */
            while (a[right] $> pivot_item) right--;
            if ( left < right ) SWAP(a,left,right);
        }
    }
    /* right is final position for the pivot */
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
}
```

**Step 1. Partition**

- **LEFT**
- **RIGHT**
# Algorithm Analysis

## Average Case

- Each level involves,
  - Maximum \( (n - 1) \) comparisons.
  - Maximum \( (n - 1) \) swaps. (3\( (n - 1) \) data movements)
  - \( \log n \) levels are required.
- Average complexity \( O(n \log_2 n) \)

## Worst Case!

### Before the partition

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### After the partition

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>30</td>
<td>40</td>
<td>50</td>
<td>60</td>
<td>70</td>
</tr>
</tbody>
</table>

- This case involves \( (n-1)+(n-2)+(n-3)+\ldots+1+0 = n(n-1)/2 \) comparisons
- Quicksort is \( O(n^2) \) for the worst-case.

## How about this?

```c
int partition( void *a, int low, int high )
{
    int left, right;
    void *pivot_item;
    pivot = low + high/2;
    pivot_item = a[pivot];
    right = high; left = low;
    while ( left < right ) {
        /* Move left while item < pivot */
        while( a[left] <= pivot_item ) left++;
        /* Move right while item > pivot */
        while( a[right] > pivot_item ) right--;
        if ( left < right ) SWAP(a,left,right);
    }
    /* right is final position for the pivot */
    a[low] = a[right];
    a[right] = pivot_item;
    return right;
}
```

## Worst Case analysis

- This case involves \( (n-1)+(n-2)+(n-3)+\ldots+1+0 = n(n-1)/2 \) comparisons
- Quicksort is \( O(n^2) \) for the worst-case.
Selecting pivot

- Strategies for Selecting pivot
  - First value: worst case if the array is sorted.
- Middle value
  - Better for the sorted data (less exchange)
- Median of 3 sample values
  - Worst case can still happen but less likely

Radix Sort

1. Take the least significant digit
2. Sort the numbers based on the digit
3. Concatenate the buckets together in order
4. Recursively sort each bucket, starting with the next digit to the right

Example

0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150
(1560, 1250) (1061) (0222) (0123, 0283) (2154, 0004)
1560, 1250, 1061, 0222, 0123, 0283, 2154, 0004

(0004)(0222,0123)(2150,2154)(1560,1061)(0283)
0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283

(0004,1061)(0123,2150,2154)(0222, 0283)(1560)
0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560

(0004,0123, 0222, 0283)(1061, 1560)(2150, 2154)
0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154

A radix sort of n decimal integers of d digits each

```
radixSort(inout theArry:ItemArray, in n:integer, in d:integer){
for (j=d down to 1)
  initialize 10 groups to empty
  initialize a counter for each group to 0
  k = jth digit of theArray[i]
  place theArray[i] at the end of group k
  increase kth counter by 1
}
replace the items in theArray with all the items in group 0, followed by all the items in group 1, and so on.
}
```

Radix sort

- Analysis
  - n moves each time it forms groups
  - n moves to combine them again into one group.
  - Total 2n^d (for the strings of d characters)
  - Radix sort is O(n) for d << n

Next reading

- Rosen Ch 8.1 (for 6th edition: 7.1) Recurrence Relations
- Rosen Ch 8.3 (for 6th edition: 7.3) Divide and Conquer