CS 303
Design and Analysis of Algorithms

Review For Midterm
Dong Xu

(Based on class note of David Luebke)

Mid-term

- 12:55-1:55pm, Friday, March 19
- Close book
- Bring your calculator
- 30% of your final score
- Office hours during March 15-19
  - Dong: 11am-noon and 3pm-4pm, Wed, Mar 17 (no office hour on Mar 19 due to travel)
  - Ashwin: additional office hours at 9:30-noon, Fri, Mar 19.

Review Of Topics
- Asymptotic notation
- Solving recurrences
- Sorting algorithms
  - Insertion sort
  - Merge sort
  - Heap sort
  - Quick sort
  - Counting sort
  - Radix sort
  - Bucket sort

Review of Topics
- Structures for dynamic sets
  - Priority queues
  - Hash tables

Proof By Induction
- Claim:S(n) is true for all n >= k (e.g., k = 0)
- Basis:
  - Show formula is true when n = k
- Inductive hypothesis:
  - Assume formula is true for an arbitrary n
- Step:
  - Show that formula is then true for n+1

Review: Analyzing Algorithms
- We are interested in asymptotic analysis:
  - Behavior of algorithms as problem size gets large
  - Constants, low-order terms don’t matter
Insertion Sort

Statement | Effort
---|---
InsertionSort(A, n) { | |
for i = 2 to n { | c_i 
key = A[i] | c_{i(n-1)} 
j = i - 1; | c_{i(n-1)} 
while (j > 0) and (A[j] > key) { | T 
j = j - 1 | (T-(n-1)) 
} | 0 
A[j+1] = key | c_{(n-1)} 
} | 0 
} | T = t_1 + t_2 + … + t_n where t_i is number of while expression evaluations for the i-th for loop iteration

Analyzing Insertion Sort

- T(n) = c_{1n} + c_{2(n-1)} + c_{3(n-1)} + c_{4T} + c_{5(T-(n-1))} + c_{6(T-(n-1))} + c_{7(n-1)}
- What can T be?
  - Best case – inner loop body never executed
    \[ t_i = 1 \] \( T(n) \) is a linear function
  - Worst case – inner loop body executed for all previous elements
    \[ t_i = i \] \( T(n) \) is a quadratic function
  - If T is a quadratic function, which terms in the above equation matter?

Upper Bound Notation

- We say InsertionSort’s run time is \( O(n^2) \)
- Properly we should say run time is in \( O(n^2) \)
- Read O as “Big-O” (you’ll also hear it as “order”)
- In general a function
  - \( f(n) \) is \( O(g(n)) \) if there exist positive constants \( c \) and \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \)
- Formally
  - \( O(g(n)) = \{ f(n): \exists \text{ positive constants } c \text{ and } n_0 \text{ such that } f(n) \leq c \cdot g(n) \forall n \geq n_0 \} \)

Big O Fact

- A polynomial of degree k is \( O(n^k) \)
- Proof:
  - Suppose \( f(n) = b_k n^k + b_{k-1} n^{k-1} + \ldots + b_1 n + b_0 \)
  - Let \( a_i = |b_i| \)
  - \( f(n) \leq a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \)
  - \( \leq n^k \sum a_i \frac{n^i}{n^i} \leq n^k \sum a_i \leq cn^k \)

Lower Bound Notation

- We say InsertionSort’s run time is \( \Omega(n) \)
- In general a function
  - \( f(n) \) is \( \Omega(g(n)) \) if \( \exists \) positive constants \( c \) and \( n_0 \) such that \( 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0 \)

Asymptotic Tight Bound

- A function \( f(n) \) is \( \Theta(g(n)) \) if \( \exists \) positive constants \( c_1, c_2, \) and \( n_0 \) such that
  \[ c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n \geq n_0 \]
Other Asymptotic Notations

- A function $f(n)$ is $o(g(n))$ if $\exists$ positive constants $c$ and $n_0$ such that $f(n) < c \cdot g(n)$ $\forall$ $n \geq n_0$

- A function $f(n)$ is $\omega(g(n))$ if $\exists$ positive constants $c$ and $n_0$ such that $c \cdot g(n) < f(n)$ $\forall$ $n \geq n_0$

Notation Summary

Notation Summary

- $o()$ is like $<$
- $O()$ is like $\leq$
- $\omega()$ is like $>$
- $\Omega()$ is like $\geq$
- $\Theta()$ is like $=$

Review: Recurrences

Recurrence: an equation that describes a function in terms of its value on smaller functions

\[
\begin{align*}
s(n) &= \begin{cases} 
0 & n = 0 \\
n + s(n-1) & n > 0 
\end{cases} \\
T(n) &= \begin{cases} 
c & n = 1 \\
aT\left(\frac{n}{b}\right) + cn & n > 1 
\end{cases}
\end{align*}
\]

Review: Solving Recurrences

- Substitution method
- Recursion tree method
- Master method

Review: Substitution Method

- Substitution Method:
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Examples:
    - $T(n) = 2T(n/2) + \Theta(n)$ $\neq$ $T(n) = \Theta(n \log n)$
    - We can show that this holds by induction

Substitution Method

- Our goal: show that $T(n) = 2T(n/2) + n = O(n \log n)$
- Thus, we need to show that $T(n) \leq c \cdot n \log n$ with an appropriate choice of $c$
  - Inductive hypothesis: assume $T\left(\lfloor n/2 \rfloor \right) \leq c \cdot \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor$
  - Substitute back into recurrence to show that $T(n) \leq c \cdot n \log n$ follows, when $c \geq 1$
Review: Recursion Tree

- Recursion tree method:
  - Expand the recurrence into a tree form
  - Work some algebra to express as a summation
  - Evaluate the summation

Review: The Master Theorem

- Given: a divide and conquer algorithm
  - An algorithm that divides the problem of size $n$ into $a$ subproblems, each of size $n/b$
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function $f(n)$
  - Then, the Master Theorem gives us a cookbook for the algorithm’s running time:

\[
\begin{align*}
\text{if } T(n) &= aT(n/b) + f(n) \text{ then} \\
T(n) &= \begin{cases} \\
\Theta(n^{\log_a b}) & f(n) = \Theta(n^{\log_a b}) \\
O(n^{\log_a b}) & f(n) = \Theta(n^{\log_a b+\epsilon}) \\
\Omega(n^{\log_a b-\epsilon}) & f(n) = \Theta(n^{\log_a b-\epsilon}) \quad \text{AND} \quad af(n/b) \leq cf(n) \text{ for large } n \end{cases}
\end{align*}
\]

Review: Merge Sort

\[
\text{MergeSort}(A, \text{left}, \text{right}) \quad \\
\text{if } (\text{left} < \text{right}) \{ \\
\text{mid} = \text{floor}((\text{left} + \text{right}) / 2); \\
\text{MergeSort}(A, \text{left}, \text{mid}); \\
\text{MergeSort}(A, \text{mid+1}, \text{right}); \\
\text{Merge}(A, \text{left}, \text{mid}, \text{right}); \\
\}
\]

// Merge() takes two sorted subarrays of A and // merges them into a single sorted subarray of A. // Merge() takes $O(n)$ time, $n =$ length of A

Review: Analysis of Merge Sort

<table>
<thead>
<tr>
<th>Statement</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>MergeSort$(A, \text{left}, \text{right}) { }$</td>
<td>$T(n)$</td>
</tr>
<tr>
<td>if (left &lt; right) {}</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>mid = floor((left + right) / 2);</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>MergeSort$(A, \text{left}, \text{mid});$</td>
<td>$T(n/2)$</td>
</tr>
<tr>
<td>MergeSort$(A, \text{mid+1}, \text{right});$</td>
<td>$T(n/2)$</td>
</tr>
<tr>
<td>Merge$(A, \text{left}, \text{mid}, \text{right});$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>}</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

- So $T(n) = \Theta(1)$ when $n = 1$, and
  - $2T(n/2) + \Theta(n)$ when $n > 1$
- Solving this recurrence (row?) gives $T(n) = n \log n$

Review: Heaps

- A heap is a “complete” binary tree, usually represented as an array:

\[
\begin{array}{c}
4 \\
10 \\
7 \\
8 \\
9 \\
3 \\
2 \\
16 \\
14 \\
10 \\
8 \\
7 \\
9 \\
3 \\
2 \\
4 \\
1
\end{array}
\]

$A = 16, 14, 10, 8, 7, 9, 3, 2, 4, 1$
Review: Heaps

- To represent a heap as an array:
  
  - `Parent(i)` { return ⌊i/2⌋; }
  - `Left(i)` { return 2*i; }
  - `right(i)` { return 2*i + 1; }

Review: The Heap Property

- Heaps also satisfy the heap property:
  
  - `A[Parent(i)] ≥ A[i]` for all nodes `i > 1`
  - In other words, the value of a node is at most the value of its parent
  - The largest value is thus stored at the root (`A[1]`)
  - Because the heap is a binary tree, the height of any node is at most Θ(lg n)

Review: Heapify()

- **Heapify()**: maintain the heap property
  
  - Given: a node `i` in the heap with children `l` and `r`
  - Given: two subtrees rooted at `l` and `r`, assumed to be heaps
  - Action: let the value of the parent node “float down” so subtree at `i` satisfies the heap property
    - Recurse on that subtree
  - Running time: `O(h)`, `h = height of heap = O(lg n)`

Review: BuildHeap()

- **BuildHeap()**: build heap bottom-up by running **Heapify()** on successive subarrays
  
  - Walk backwards through the array from `n/2` to 1, calling **Heapify()** on each node.
  - Order of processing guarantees that the children of node `i` are heaps when `i` is processed
  - Easy to show that running time is `O(n lg n)`
  - Can be shown to be `O(n)`
  - Key observation: most subheaps are small

Review: Heapsort()

- **Heapsort()**: an in-place sorting algorithm:
  
  - Maximum element is at `A[1]`
  - Discard by swapping with element at `A[n]`
    - Decrement heap size `A`
    - `A[n] now contains correct value`
  - Restore heap property at `A[1]` by calling **Heapify()**
  - Running time: `O(n lg n)`
  - **BuildHeap**: `O(n)`, **Heapify**: `n * O(lg n)`

Review: Priority Queues

- The heap data structure is often used for implementing priority queues
  
  - A data structure for maintaining a set `S` of elements, each with an associated value or `key`
  - Supports the operations **Insert()**, **Maximum()**, and **ExtractMax()**
  - Commonly used for scheduling, event simulation
### Priority Queue Operations
- **Insert(S, x)** inserts the element x into set S
- **Maximum(S)** returns the element of S with the maximum key
- **ExtractMax(S)** removes and returns the element of S with the maximum key

### Implementing Priority Queues

#### HeapInsert(A, key)
```c
    heap_size[A] ++;
    i = heap_size[A];
    while (i > 1 AND A[Parent(i)] < key) {
        i = Parent(i);
    }
    A[i] = key;
```

#### HeapMaximum(A)
```c
    // This one is really tricky:
    return A[i];
```

#### HeapExtractMax(A)
```c
    if (heap_size[A] < 1) { error; }
    max = A[1];
    heap_size[A] --;
    Heapify(A, 1);
    return max;
```

### Review: Quicksort
- **Quicksort pros:**
  - Sorts in place
  - Sorts $O(n \log n)$ in the average case
  - Very efficient in practice
- **Quicksort cons:**
  - Sorts $O(n^2)$ in the worst case
  - Naïve implementation: worst-case = sorted
  - Even picking a different pivot, some particular input will take $O(n^2)$ time

- **Another divide-and-conquer algorithm**
  - The array $A[p..r]$ is *partitioned* into two non-empty subarrays $A[p..q]$ and $A[q+1..r]$
  - Invariant: All elements in $A[p..q]$ are less than all elements in $A[q+1..r]$
  - The subarrays are recursively quicksorted
  - No combining step: two subarrays form an already-sorted array
Review: Quicksort Code

Quicksort(A, p, r)
{
    if (p < r)
    {
        q = Partition(A, p, r);
        Quicksort(A, p, q);
        Quicksort(A, q+1, r);
    }
}

Review: Partition Code

Partition(A, p, r)

x = A[p];
i = p - 1;
j = r + 1;
while (TRUE)
    repeat
        j--;
    until A[j] <= x;
    repeat
        i++;
    until A[i] >= x;
    if (i < j)
        Swap(A, i, j);
    else
        return j;

partition() runs in O(n) time

Review: Analyzing Quicksort

- What will be the worst case for the algorithm?
  - Partition is always unbalanced
- What will be the best case for the algorithm?
  - Partition is perfectly balanced
- Which is more likely?
  - The latter, by far, except...
- Will any particular input elicit the worst case?
  - Yes: Already-sorted input

In the worst case:
T(1) = Θ(1)
T(n) = T(n - 1) + Θ(n)
Works out to
T(n) = Θ(n²)

Average case works out to T(n) = Θ(n lg n)
Key idea: A limited number of unbalanced Partition() is OK

In the best case:
T(n) = 2T(n/2) + Θ(n)
Works out to
T(n) = Θ(n lg n)
Review: Improving Quicksort

- The real liability of quicksort is that it runs in $O(n^2)$ on already-sorted input.
- Book discusses two solutions:
  - Randomize the input array, OR
  - Pick a random pivot element
- *How do these solve the problem?*
  - By insuring that no particular input can be chosen to make quicksort run in $O(n^2)$ time.

Sorting Summary

- Insertion sort:
  - Easy to code
  - Fast on small inputs (less than ~50 elements)
  - Fast on nearly-sorted inputs
  - $O(n^2)$ worst case
  - $O(n^2)$ average (equally-likely inputs) case
  - $O(n^2)$ reverse-sorted case

Sorting Summary

- Merge sort:
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort subarrays
    - Linear-time merge step
  - $O(n \lg n)$ worst case
  - Doesn’t sort in place

Sorting Summary

- Heap sort:
  - Uses the very useful heap data structure
  - Complete binary tree
  - Heap property: parent key > children’s keys
  - $O(n \lg n)$ worst case
  - Sorts in place
  - Fair amount of shuffling memory around

Sorting Summary

- Quick sort:
  - Divide-and-conquer:
    - Partition array into two subarrays, recursively sort
    - All of first subarray < all of second subarray
    - No merge step needed!
  - $O(n \lg n)$ average case
  - Fast in practice
  - $O(n^2)$ worst case
  - Naive implementation: worst case on sorted input
  - Address this with randomized quicksort

Review: Comparison Sorts

- Comparison sorts: $O(n \lg n)$ at best
  - Model sort with decision tree
  - Path down tree = execution trace of algorithm
  - Leaves of tree = possible permutations of input
  - Tree must have $n!$ leaves, so $O(n \lg n)$ height
### Review: Comparison Sorts

<table>
<thead>
<tr>
<th>Sorting</th>
<th>Time</th>
<th>Space</th>
<th>Stability</th>
<th>In place</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble</td>
<td>(O(n^2))</td>
<td>(O(1))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Insertion</td>
<td>(O(n^2))</td>
<td>(O(n^2))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Selection</td>
<td>(O(n^2))</td>
<td>(O(n^2))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Merge</td>
<td>(O(n \log n))</td>
<td>(O(n \log n))</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Heap</td>
<td>(\Omega(n \log n))</td>
<td>(O(n \log n))</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Quick</td>
<td>(O(n \log n))</td>
<td>(O(1))</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Counting Sort

- **Counting sort:**
  - Assumption: input is in the range 1..k
  - Basic idea:
    - Count number of elements \(k \leq i\) for each element \(i\)
    - Use that number to place \(i\) in position \(k\) of sorted array
  - No comparisons! Runs in time \(O(n + k)\)
  - Stable sort
  - Does not sort in place:
    - \(O(n)\) array to hold sorted output
    - \(O(k)\) array for scratch storage

```plaintext
CountingSort(A, B, k)
for i=1 to k
    C[i] = 0;
for j=1 to n
    C[A[j]] += 1;
for i=2 to k
    C[i] = C[i] + C[i-1];
for j=n downto 1
    B[C[A[j]]] = A[j];
    C[A[j]] -= 1;
```

### Radix Sort

- **Radix sort:**
  - Assumption: input has \(d\) digits ranging from 0 to \(k\)
  - Basic idea:
    - Sort elements by digit starting with least significant
    - Use a stable sort (like counting sort) for each stage
  - Each pass over \(n\) numbers with \(d\) digits takes time \(O(n+k)\), so total time \(O(dn+dk)\)
    - When \(d\) is constant and \(k=O(n)\), takes \(O(n)\) time
  - Fast! Stable! Simple!
  - Doesn’t sort in place

### Hashing Tables

- **Motivation:** symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
  - We typically don’t care about sorted order

### Hash Tables

- **More formally:**
  - Given a table \(T\) and a record \(x\), with key (= symbol) and satellite data, we need to support:
    - Insert \((T, x)\)
    - Delete \((T, x)\)
    - Search\((T, x)\)
  - Don’t care about sorting the records
  - *Hash tables* support all the above in \(O(1)\) expected time
Review: Direct Addressing

- **Suppose:**
  - The range of keys is \(0..m-1\)
  - Keys are distinct
- **The idea:**
  - Use key itself as the address into the table
  - Set up an array \(T[0..m-1]\) in which
    - \(T[i] = x\) if \(x \in T\) and \(\text{key}[x] = i\)
    - \(T[i] = \text{NULL}\) otherwise
- This is called a *direct-address table*

---

Review: Hash Functions

- **Next problem:** *collision*

---

Review: Resolving Collisions

- **How can we solve the problem of collisions?**
  - **Open addressing**
    - To insert: if slot is full, try another slot, and another, until an open slot is found (*probing*)
    - To search, follow same sequence of probes as would be used when inserting the element
  - **Chaining**
    - Keep linked list of elements in slots
    - Upon collision, just add new element to list

---

Review: Chaining

- Chaining puts elements that hash to the same slot in a linked list

---

Review: Analysis Of Hash Tables

- **Simple uniform hashing:** each key in table is equally likely to be hashed to any slot
- **Load factor** \(\alpha = n/m = \text{average # keys per slot}\)
  - Average cost of unsuccessful search = \(O(1+\alpha)\)
  - Successful search: \(O(1 + \alpha/2) = O(1 + \alpha)\)
  - If \(n\) is proportional to \(m\), \(\alpha = O(1)\)
  - So the cost of searching = \(O(1)\) if we size our table appropriately

---

Review: Choosing A Hash Function

- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- **Methods:**
  - Division method
  - Multiplication method
Review: The Division Method

- \( h(k) = k \mod m \)
  - In words: hash \( k \) into a table with \( m \) slots using the slot given by the remainder of \( k \) divided by \( m \)
- Elements with adjacent keys hashed to different slots: good
- If keys bear relation to \( m \): bad
- Upshot: pick table size \( m \) = prime number not too close to a power of 2 (or 10)

Review: The Multiplication Method

- For a constant \( A, 0 < A < 1 \):
  \[ h(k) = \lfloor m \left( kA - \lfloor kA \rfloor \right) \rfloor \]
  - Fractional part of \( kA \)
- Upshot:
  - Choose \( m = 2^p \)
  - Choose \( A \) not too close to 0 or 1
  - Knuth: Good choice for \( A = (\sqrt{5} - 1)/2 \)