Review

Ari Mermelstein

May 19, 2019

1 Review Material

1.1 Discrete math

1. Induction
2. Sets
3. Logic

1.2 Calculus

1. Limits
2. L'Hôpital's rule
3. derivatives: rules
   (a) $(fg)' = f'g + fg'$
   (b) $(\frac{f}{g})' = \frac{gf' - fg'}{g^2})$
4. log rules
   (a) $\log(ab) = \log a + \log b$
   (b) $\log(\frac{a}{b}) = \log a - \log b$
   (c) $\log_b a = \frac{\log_c b}{\log_c a}$
   (d) $n^{\log_b a} = a^{\log_b n}$

1.3 Data structures

1. Recursion
2. Linked lists
3. stacks
4. queues
5. BSTs

2 Binary search

Input: An array A[1..n]
Input: An integer n
Input: An integer to search for called target
Output: An index $1 \leq i \leq n$ such that A[i] = target
start = 1;
end = n;
while *start* $\leq$ *end* **do**
  mid = $\lfloor \frac{start+end}{2} \rfloor$;
  if *target = A[mid]* **then**
    return mid;
  end
  else if *target* $< A[mid]$ **then**
    end = mid-1;
  end
  else
    start = mid+1;
  end
end
return NOT_FOUND;

2.1 How long does binary search take?

Let $T(n)$ be the worst case running time for binary search on an array of size n. Then

$$T(n) = T(\frac{n}{2}) + 1$$

is an equation that represents the running time.
How do we solve this? One way is by substitution.

2.2 Substitution method

1. Guess a solution
2. Prove it's correct by induction.

$T(n) = T(\frac{n}{2}) + 1$

$T(\frac{n}{2}) = T(\frac{n}{4}) + 1$

so $T(n) = T(\frac{n}{4}) + 1 + 1$

...

$T(n) = T(\frac{n}{2^k}) + k$

We Want $2^k = n \Rightarrow k = \log_2 n$.

$T(n) = 1 + \log_2 n$.

2. Prove by induction.
We want to prove that $T(n) \leq c \log_2 n$ for some positive c and for all n.

Base case: $T(k) = \log_2 k \leq c$ as long as c is sufficiently large.

Inductive case:

$T(n) = T(\frac{n}{2}) + 1$

By the inductive hypothesis

$\Rightarrow T(n) \leq c \log_2(\frac{n}{2}) + 1$

$\Rightarrow T(n) \leq c(\log_2 n - \log_2 2) + 1$

$\Rightarrow T(n) \leq c \log_2 n - c + 1$

$\Rightarrow T(n) \leq c \log_2 n - (c + 1)$

$\Rightarrow T(n) \leq c \log_2 n$ as long as $c + 1 > 0 \Rightarrow c \geq 1$

2.3 Questions similar to binary search

3 Sorting

3.1 InsertionSort pseudocode

```
Input: An array A[1..n]
Input: An integer n
for i=2 to n do
    key = A[i];
    location = i-1 ; // location represents the locations we are looking at to see
      where the key goes
    while location > 0 and A[location] > key do
        A[location+1] = A[location];
        location = location-1;
    end
    A[location + 1] = key;
end
```

3.2 Analyzing insertion sort

The outer loop happens $n-1$ times. The inner loop, in the worst case has to run $i-1$ steps. So the worst case time is given by:

$$\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \Theta(n^2)$$

3.3 MergeSort Pseudocode

3.4 Merge

```
Input: An array A
Input: integers start1, end1, start2, end2
; // We will view A as having 2 subarrays A[start1..end1] and
  A[start2..end2]
Let result[1..end2-start1+1] be a new array;
indexL = start1;
indexR = start2;
indexRes = 1;
; // Go through both subarrays and pick the minimum element.
  Stop when one of the subarrays runs out.
while indexL ≤ end1 and indexR ≤ end2 do
    if A[indexL] < A[indexR] then
        result[indexRes] = A[indexL];
        indexL = indexL + 1;
    end
    else
        result[indexRes] = A[indexR];
        indexR = indexR + 1;
    end
    indexRes = indexRes + 1;
end
; // Copy the remaining subarray over
while indexL ≤ end1 do
    result[indexRes] = A[indexL];
    indexL = indexL + 1;
    indexRes = indexRes + 1;
end
while indexR ≤ end1 do
    result[indexRes] = A[indexR];
    indexR = indexR + 1;
    indexRes = indexRes + 1;
end
; // Copy the result back
indexRes=1;
for i = start1 to end2 do
    A[i] = result[indexRes];
end
```

3.5 Mergesort

Input: An array A
Input: integers start, end
if *end-start* ≤ 0 **then**
| **return**;
end
mid = floor((start + end) /2) ;
mergesort(A, start, mid);
mergesort(A, mid+1, end);
merge(A, start, mid, mid+1, end);

4 Algorithm complexities

4.1 Running Time and Space Complexity

Definition: The running time of an algorithm A on a particular input is the number of primitive operations A performs when run on the input.

Definition: The space complexity of an algorithm A on a particular input is the amount of extra space the algorithm requires to run. Note: local variables count as $\Theta(1)$ space.

4.2 Best, Worst, Average Case

Definition: The best case time of an algorithm is the minimum number of primitive operations the algorithm requires over all possible inputs.

Definition: The worst case time of an algorithm is the maximum number of primitive operations the algorithm requires over all possible inputs.

Definition: The average case running time is the expectation of the running time over all possible inputs given a particular probability distribution.

4.3 $O, \Omega, \Theta, \omega, o$

Definition: We say that one function $f(n)$ is $O(g(n))$ if there exist constants $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

Example: $n \in O(n^2)$ why?

Definition: We say that one function $f(n)$ is $\Omega(g(n))$ if there exist constants $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$ such that $f(n) \geq cg(n)$ for all $n \geq n_0$.

Definition: We say that one function $f(n)$ is $\Theta(g(n))$ if there exist constants $c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

limit definitions

Definition: We say that one function $f(n)$ is $O(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, for some $c \geq 0$

Definition: We say that one function $f(n)$ is $\Omega(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, for some $c > 0$ or $\lim_{n\to\infty} \frac{f(n)}{g(n)} \to \infty$.

Definition: We say that one function $f(n)$ is $\Theta(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, for some $c > 0$

Talk about o and ω.

5 Theorems

1. $\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$
2. Θ is an equivalence relation.

6 Recurrences

1. substitution
2. recursion trees + substitution
3. Master theorem

7 Heaps

1. definition
2. inserting
3. removing
4. building a heap
5. heapsort

8 Heaps pseudocode

insert(A, heapsize, value)

Input: A heap A
Input: integers heapsize, value
A[heapsize+1] = value;
heapsize = heapsize+1 ;
index = heapSize;
while $index \geq 0 and A[parent(index)] < A[index]$ **do**
  swap(A[index], A[parent(index)]);
  index = parent(index);
end

Remove(A, heapsize)

Input: A heap A
Input: integer heapsize
swap(A[heapsize], A[1];
heapsize = heapsize - 1 ;
fixheap(A, 1, heapSize);

fixheap(A, i, heapSize)

Input: A heap A
Input: integer heapsize, i
max = i;
if $leftChild(i) \leq heapSize\ and\ A[leftChild(i)] > A[max]$ **then**
  max = leftChild(i);
end
if $rightChild(i) \leq heapSize\ and\ A[rightChild(i)] > A[max]$ **then**
  max = rightChild(i);
end
if $max = i$ **then**
  return;
end
swap(A[max], A[i]);
fixheap(A, max, heapSize);

Heapsort(A, n)

Input: A heap A
Input: integer n
for i=$\lfloor \frac{n}{2} \rfloor downto 1$ **do**
  fixHeap(A, i, n);
end
n = heapSize;
while $heapSize > 0$ **do**
  swap(A[1], A[heapSize]);
  heapSize = heapSize-1;
  fixHeap(A, 1, heapSize);
end

9 Quicksort

9.1 Partition

The partition algorithm selects an element as the pivot, and the moves everything smaller or equal to the left and everything greater than or equal to the right. The invariant is that every number in $A[start..b] \leq pivot$ and every number in $A[b+1..i-1] \geq pivot$. Everything in $A[i..end]$ is unknown.

```
Partition(A, start, end)

Input: an array A
Input: integers start and end
Output: The pivot index
b = start;
for i=start+1 to end do
    if A[i] < pivot then
        b = b+1 ;
        swap(A, i, b);
    end
end
swap(A, start, b);
return b;
```

9.2 Quicksort

The idea is to run partition on the array and then recursively sort both subarrays.

```
Quicksort(A, start, end)

Input: an array A
Input: integers start and end
if end ≤ start then
    return;
end
p = partition(A, start, end);
Quicksort(A, start, p-1);
Quicksort(A, p+1, end);
```

9.3 Analysis

The worse case for quicksort is when the array is already sorted or reverse sorted. The pivot will always not give us a good split and we will wind up with a running time of $1+2+3+\cdot+n = \Theta(n^2)$. However, if we pick a random pivot, then the running time will be $\Theta(n \log n)$ with high probability.

10 Selection

10.1 Finding the max or the min

The straightforward algorithm for finding the min or the max is the following:

Min(A, n)

Input: an array A
Input: an integer n, the length of A
Output: The minimum of A
min = A[1];
for *i=2* **to** n **do**
  if $A[i] < min$ **then**
    min = A[i];
  end
end
return min;

This takes n-1 comparisons in the worst case. Max is similar.

10.2 Simultaneous max and min

If we want to find the max and min at the same time, we could do it in n-1 + n-1 = 2n-2 comparisons. Or we can find them simultaneously with the following recursive algorithm.

Min-And-Max(A, start, end)

Input: an array A
Input: integers start and end
Output: An ordered pair (min, max)
if $end - start \leq 1$ **then**
  if $A[start] < A[end]$ **then**
    return (A[start], A[end]);
  end
  else
    return (A[end], A[start]);
  end
end
$mid = \lfloor \frac{start+end}{2} \rfloor$;
(minLeft, maxLeft) = Min-And-Max(A, start, mid);
(minRight, maxRight) = Min-And-Max(A, mid+1, end);
min = minLeft;
max = maxLeft;
if $minRight < min$ **then**
  min = minRight;
end
if $maxRight > max$ **then**
  max = maxRight;
end
return (min, max);

10.2.1 Analysis

How long does this take?

It obeys the occurrence: $T(n) = 2T(n/2) + 1$, $T(2) = 2$.

$T(n) = 2T(\frac{n}{2}) + 1$
$\Rightarrow= 4T(\frac{n}{4}) + 2 + 1$
$\Rightarrow= 8T(\frac{n}{8}) + 4 + 2 + 1$
$\ldots$
$\Rightarrow= 2^k T(\frac{n}{2^k}) + 1 + 2 + \cdots + 2^{k-1}$
Let $\frac{n}{2^k} = 2 \Rightarrow 2^k = \frac{n}{2}$, and $1 + 2 + \cdots + 2^{k-1} = 2^k - 1 = \frac{n}{2} - 1$.
$\Rightarrow= 2\frac{n}{2} + \frac{n}{2} - 1$.
This has solution $T(n) = \frac{3n}{2} - 1$.

10.3 Finding the Kth smallest

We can find the kth smallest number in an array as follows: Take the array A and turn it into a min heap. Then pop the heap k times. This takes $\Theta(n + k \log n)$. If k is small, this is a good algorithm, but if $k = \Theta(n)$, this is no longer all that good.

A better algorithm is based on partition. Call partition using the pivot. Once that happens, if $k < pivotindex$ then you look in the left subarray. Otherwise, you look in the right subarray. This takes $\Theta(n)$ time with high probability.

11 Graphs

11.1 Definitions

Definition: A Graph is a pair $G = (V, E)$ where V is a set of vertices and E is a set of edges. If G is a directed graph, then each $e \in E$ is an ordered pair (u, v) representing that u and v are related. If G is undirected, then each $e \in E$ is an unordered pair, $\{u, v\}$. (Abuse of notation: most books and papers use (u, v) even for undirected graphs, but this is technically not true.)

Definition: A path is a sequence of vertices $(v_0, v_1, v_2, \cdots, v_k)$. The length of a path is the number of edges in the path.

Definition: A cycle is a path that starts and ends at the same vertex. A simple cycle is a cycle that doesn't repeat vertices. **Definition**: A graph is connected if every vertex is reachable from every other vertex. Connected components of a graph are parts of the graph in which every vertex is reachable from every other vertex.

Definition: The outdegree of a vertex v is the number of edges that leave v. The indegree of a vertex v is the number of edges that enter v. In an undirected graph, we just talk about degree.

Definition: a forest is an acyclic, undirected graph. A tree is an acyclic, undirected, connected graph. A directed, acyclic graph is called a "dag."

11.2 BFS

```
Input: the adjacency lists of a graph G = (V, E)
Let visited be a new array;
for v ∈ V do
  | visited[v] = false;
end
for v ∈ V do
  | if not visited[v] then
  |   | BFS-Visit(G, v);
  | end
end
```

Input: the adjacency lists of a graph G = (V, E)
Input: The source vertex v
Input: The visited array
Let $Q = \emptyset$;
; `// Q is the empty queue`
Q.enqueue(v);
visited[v] = true;
visit(v);
while $Q \neq \emptyset$ **do**
    w = Q.dequeue();
    for $u \in Adj[w]$ **do**
        if *not visited[u]* **then**
            Q.enqueue(u);
            visited[u] = true;
            visit(u);
        end
    end
end

Time Complexity: $\Theta(V + E)$.

11.3 DFS

Input: the adjacency lists of a graph G = (V, E)
Let visited be a new array;
for $v \in V$ **do**
    visited[v] = false;
end
for $v \in V$ **do**
    if *not visited[v]* **then**
        DFS-Visit(G, v);
    end
end

Input: the adjacency lists of a graph G = (V, E)
Input: The source vertex v
Input: The visited array
visited[v] = true;
visit(v);
for $u \in Adj[v]$ **do**
    if *not visited[u]* **then**
        DFS-Visit(G, u, visited);
    end
end

11.4 Dijkstra-Prim

```
Input: the adjacency lists of a graph G = (V, E, w)
Input: s, the source vertex
Let F = V;
; // F is the fringe
Let parent[|V|] be a new array;
Let H be a min heap;
Let T = ∅;
; // T is the MST
Let dist[|V|] be a new array;
; // dist[i] is the smallest edge to add vertex i into the
 tree
for v ∈ V do
    H.insert((v, ∞));
    parent[v] = null;
    dist[v] = ∞;
end
H.decreaseKey(s, 0);
for v ∈ Adj[s] do
    H.decreaseKey(v, w(s, v));
    parent[v] = s;
    dist[v] = w(u,v);
end
while F ≠ ∅ do
    u = H.removeMin();
    T = T ∪ {parent[u], u, w(u, parent(u))};
    for v ∈ Adj[u] and v ∈ T do
        if w(u, v) < dist[v] then
            H.decreaseKey(v, w(u,v));
            dist[v] = w(u,v);
        end
    end
    F = F \ {u};
end
```

N.B.: You need to remember where in the heap each v is so that you can implement decreaseKey() in $\Theta(\log V)$ time.

11.5 Kruskal

Input: the adjacency lists of a graph G = (V, E, w)
Input: s, the source vertex
$T = \emptyset$;
Let $Edges[|E|]$ be a new array;
Copy all edges into the array;
sort(Edges);
for $e \in Edges$ **do**
    if *adding e doesn't create a cycle* **then**
        $T = T \cup \{e\}$;
    end
end

How do you decide if adding e creates a cycle? Use a Union-Find D.S. = Disjoint sets D.S.

Make-Set(n):
Create an array parent[n]. set parent[i] = null for all $1 \leq i \leq n$

Union(x, y): attach the smaller of the two trees to the larger.
make the root of the larger tree the representative.

Find(x): return the representative of x, but when you do it, bring up all of the pointers to the root.

12 Dynamic Programming

The idea of dynamic programming is that some problems have recursive structures, but that the straightforward recursive algorithms take exponential time. Intuitively, this is because many subproblems get computed many times. There are two fixes: top-down recursion with memoization and bottom up with a table. I'll demonstrate both on fibonacci.

12.1 Fibonacci

Recall that the Fibonacci sequence is defined recursively as:

$$\text{fib(n)} = \begin{cases} 1 & n \leq 2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

One could program the naïve algorithm as follows:

fib(n):
Input: an integer n
Output: the n$^{\text{th}}$ fibonacci number
if $n \leq 2$ **then**
  return 1;
end
else
  return fib(n-1) + fib(n-2);
end

It's easy to see why this will take exponential time. In running this algorithm, we will call fib(3) many times for large n.

The running time is $\Theta(\Phi^n)$, where $\Phi = \frac{1+\sqrt{5}}{2}$. = Bad!

12.1.1 Dynamic programming solution

The solution is either to memoize or to solve this bottom up. The bottom up is as follows:
fib(n):
Input: an integer n
Output: the n$^{\text{th}}$ fibonacci number
Let results[1..n] be a new array ;
results[1] = 1;
results[2] = 1;
for *i=3* **to** n **do**
  results[i] = results[i-1] + results[i-2];
end
return results[n];

This takes $\Theta(n)$ time. This is a far cry for exponential. (Technically, you can find fibonacci numbers in $\Theta(\log n)$ time.)

12.2 Binomial Coefficients

Suppose we wanted to compute the number of ways to choose r objects from a group of n, denoted $\binom{n}{r}$. You can theoretically do this with factorials, but this is unrealistic since the size of the numbers grow huge very quickly and cannot be stored in a word in the computer. So let's leverage a recurrence:

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

$$\binom{n}{0} = 1, \binom{n}{n} = 1$$

Conceptually, this means that you have 2 choices: you can choose the first object, or not. If you choose it, you need to pick $r-1$ more objects out of the remaining $n-1$. If you choose to leave it alone, you still need to pick r from $n-1$.

Proof: Look at the class notes!! It's way too annoying to type up.

If we implement this algorithm directly, we will get the same horrible behavior as fibonacci.

12.2.1 Memoization

We can use the recursive algorithm above and add a hash table to remember while still recursing. Example:
binom(n,r):

Input: integers n and r
Output: $\binom{n}{r}$
Let map be a new, empty hash map;
return binom(n,r,map);

binom(n, r, map):

Input: integers n and r
Input: a hash map called map
Output: $\binom{n}{r}$
int answer;
if *map.containsKey((n,r))* **then**
| **return** map.get((n,r));
end
else if *r=0 or r=n* **then**
| ans = 1;
end
else
| ans = binom(n-1, r-1, map) + binom(n-1, r, map);
end
map.put((n,r), ans);
return ans;

12.2.2 Dynamic programming

Let's instead build a table:
binom(n. r)

Input: integers n and r
Output: $\binom{n}{r}$
Let binom[0..n][0..r] be a new 2D array;
for *i = 0* **to** *n* **do**
    for *j = 0* **to** *min{i, r}* **do**
        if *j= 0 or j = i* **then**
            binom[i][j] = 1;
        end
        else
            binom[i][j] = binom[i-1][j-1] + binom[i-1][j];
        end
    end
end
return binom[n][r];

How long does this take? $\Theta(nr) \approx \Theta(n^2)$. This is much better than exponential.

12.3 Edit Distance

Given 2 strings S[1..n] and T[1..m], find the number of moves necessary to transform S into T. A move is an insertion, deletion, or a switch of characters.

Naïve recursive algorithm The naïve recursive algorithm is to notice that we can look at all prefixes of S and T, and figure out what to do with the new character. If the new characters match, don't do anything, and keep the answer you used to have. Otherwise, find the minimum of insertion, deletion, and swap, and add 1.

EditDistance(S, T, n, m)
Input: Strings S and T
Input: integers n and m, denoting the size of the prefixes
Output: the edit distance of S and T
if $n = 0$ **then**
    return m;
end
else if $m = 0$ **then**
    return n;
end
else if $S[n] = T[m]$ **then**
    return EditDistance(S, T, n-1, m-1);
end
else
    return $1 + min\{EditDistance(S, T, m-1, n-1), EditDistance(S, T, m, n-1), EditDistance(S, T, m-1, n)$;
end

This algorithm takes exponential time. To make it efficient, either memoize or build a table. I'll give you the table version.

Dynamic Programming

Input: Strings S and T
Input: integers n and m, the sizes of S and T
Output: the edit distance of S and T
Output: the dist array
Let dist[0..n][0..m] be a new 2D array;
for *i=0* **to** *n* **do**
  dist[i, 0] = i;
end
for *i=0* **to** *m* **do**
  dist[0, i] = i;
end
for *i = 1* **to** *n* **do**
  for *j = 1* **to** *m* **do**
    if *S[i] = T[j]* **then**
      dist[i,j] = dist[i-1,j-1];
    end
    else
      dist[i,j] = 1 + $min\{dist[i-1, j-1], dist[i-1, j], dist[i, j-1]\}$;
    end
  end
end
return dist[n,m] and dist;

This takes $\Theta(nm)$ time.

12.3.1 Backtracking

Given the table, you can find an optimal alignment as follows:

Start with dist[n,m]. Then follow the rules in reverse until you get to dist[0,0]. This only takes an additional $\Theta(max\{n, m\}$ time.