EE 360C - ALGORITHMS

Lecture 6
Algorithm Complexity 2

Vallath Nandakumar
University of Texas at Austin

A mathematical formula should never be "owned" by anybody! Mathematics belong to God.

-- *Donald Knuth*

Agenda

- ☞ Last class
  - Big-O, Big-Ω, Big-Θ, and little-o and little-ω
- ☞ This class
  - Review of last class material
  - More properties of complexity sets
  - Decision trees (if there is time)

ANALOGIES TO TRADITIONAL COMPARISONS

☞ Analogies between asymptotic comparisons of two functions *f* and *g* and comparison of two real numbers *a* and *b*

- $f(n) = O(g(n)) \approx a \leq b$
- $f(n) = \Omega(g(n)) \approx a \geq b$
- $f(n) = \Theta(g(n)) \approx a = b$
- $f(n) = o(g(n)) \approx a < b$
- $f(n) = \omega(g(n)) \approx a > b$

LIMIT THEOREMS

☞ Theorems

- $f(n) = o(g(n))$ implies $\lim_{n\to\infty} f(n)/g(n) = 0$
- $f(n) = \omega(g(n))$ implies $\lim_{n\to\infty} f(n)/g(n) = \infty$
- $f(n) = O(g(n))$ implies $\lim_{n\to\infty} f(n)/g(n) = 0$ or c
- $f(n) = \Omega(g(n))$ implies $\lim_{n\to\infty} f(n)/g(n) = \infty$ or c
- $f(n) = \Theta(g(n))$ implies $\lim_{n\to\infty} f(n)/g(n) = c$

COMPARISON OF FUNCTIONS

- Transitivity
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$ imply $f(n) = \Theta(h(n))$
- Reflexivity
  - $f(n) = \Theta(f(n))$
- Symmetry
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose Symmetry
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

SUM OF FUNCTIONS

- ☞ If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$
- ☞ Asymptotic bound for algorithm
  - Comparable to asymptotic bound for slowest part
- ☞ If $g = O(f)$, then $f + g = \Theta(f)$

If $f = O(h)$ and $g = O(h)$, then $f + g = O(h)$:

There exists a c1 and n01 for f/h, and c2 and n02 for g/h, such that for all n> n0, f <= h, or g <= h. Pick n0 = max(n0f, n0g), and pick c1 + c2 for the new c.

If $g = O(f)$, then $f + g = \Theta(f)$

Show that g+f is O(f) and Omega(f).

Since g = O(f), there exists a c, n0 such that g <= cf when n > n0. For f+g, pick c+1 as the new constant. So g + f <= (c+1)f when n >= n0; => g+f is O(f).

Also, g+f >= 1.f, for n >= n0, since g is asymptotically positive. Hence g+f is Omega(f).

Hence the result.

iClicker

What is wrong with this statement? Pick the best answer.

Any comparison based sorting algorithm requires at least *O*(*n* log *n*) comparisons.

A. Mergesort almost always performs better, so it is not true.
B. Insertion sort performs better with nearly sorted input, so it is not true.
C. There is no such thing as a comparison-based sorting algorithm.
D. The lower-bound is O(n) comparisons for comparison-based sorting algorithms

STANDARD NOTATIONS

☞ Monotonicity

- *f(n) monotonically increasing* if $m \le n \Rightarrow f(m) \le f(n)$
- *f(n) monotonically decreasing* if $m \le n \Rightarrow f(m) \ge f(n)$
- *f(n) strictly increasing* if $m < n \Rightarrow f(m) < f(n)$
- *f(n) strictly decreasing* if $m < n \Rightarrow f(m) > f(n)$

☞ Floors and Ceilings

- For any real number x, greatest integer less than or equal to x denoted by $\lfloor x \rfloor$
- For any real numbers x, least integer greater than or equal to x denoted by $\lceil x \rceil$

STANDARD NOTATIONS

- ☞ Modular Arithmetic
  - For any integer *a* and positive integer *n*
    - *a* mod *n* is remainder of quotient *a/n*
    - $a \bmod n = a - \lfloor a/n \rfloor n$
- ☞ Polynomials
  - Given nonnegative integer *d*
  - Polynomial in *n* of degree *d* is function *p*(*n*)
    - $p(n) = \sum_{i=0}^{d} a_i n^i$
  - Where constants $a_0, a_1, \ldots, a_d$ are coefficients of polynomial and $a_d \neq 0$
  - For asymptotically positive polynomial *p*(*n*) of degree *d*
    - $p(n) = \Theta(n^d)$

EXPONENTIALS

☞ For all real *a* > 0, *m*, and *n*, these identities hold

- $a^0 = 1$
- $a^1 = a$
- $a^{-1} = 1/a$
- $(a^m)^n = a^{mn}$
- $(a^m)^n = (a^n)^m$
- $a^m a^n = a^{m+n}$

☞ Any exponential function with base strictly greater than 1 grows faster than any polynomial function

- $\lim_{n \to \infty} n^b / a^n = 0$

LOGARITHM NOTATIONS

☞ Logarithm Notations

- $\lg n = \log_2 n$
- $\ln n = \log_e n$
- $\lg^k n = (\lg n)^k$
- $\lg \lg n = \lg(\lg n)$

LOGARITHM NOTATIONS

☞ For all real $a > 0$, $b > 0$, $c > 0$, and n

- $a = b^{\log_b a}$
- $\log_c (ab) = \log_c a + \log_c b$
- $\log_b a^n = n \log_b a$
- $\log_b a = \log_c a / \log_c b$
- $\log_b (1/a) = -\log_b a$
- $\log_b a = 1 / \log_a b$
- $a^{\log_b c} = c^{\log_b a}$

BOUNDS FOR COMMON FUNCTIONS

- ☞ Polynomials
  - $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$
- ☞ Polynomials Time Algorithm
  - Algorithm whose running time is $O(n^d)$ for some constant d (where d is independent of input size)
- ☞ Logarithms
  - $O(\log_a n) = O(\log_b n)$ for any constants a, $b > 0$
    - Can ignore base in logarithms
  - For every $x > 0$, $\log n = O(n^x)$
    - Any log grows slower than polynomial
- ☞ Exponentials
  - For every $r > 1$ and $d > 0$, $n^d = O(r^n)$
    - Every exponential grows faster than every polynomial

LINEAR TIME: $O(n)$

☞ Linear Time: $O(n)$

- Running time is constant factor times size of input

☞ Example – Computing maximum of n numbers

```
1  $max \leftarrow a_1$
2  for $i$ = 2 to $n$
3      do if ($a_i > max$)
4          then $max \leftarrow a_i$
```

LINEAR TIME: $O(n)$

☞ Merge

- Combine two sorted lists
  - $A = a_1, a_2, \ldots, a_n$ and $B = b_1, b_2, \ldots, b_n$

```
1   $i = 1, j = 1$
2   while (both lists are nonempty)
3       do if ($a_i < b_j$) append $a_i$ to output and increment $i$
4           else append $b_j$ to output and increment $j$
5   append remainder of nonempty list to output
```

LINEARITHMIC TIME: $O(n \log n)$

☞ Commonly arises in divide and conquer algorithms

- Mergesort

☞ Example – Largest Empty Interval

- Given n time stamps $x_1, \ldots, x_n$ when files arrive at server, find largest time interval when no files arrive
  - Sort time stamps
  - Scan sorted list in order subtracting time stamps to get intervals and identify maximum interval

QUADRATIC TIME: $O(n^2)$

- ☞ Commonly arise when enumerating all pairs of elements
- ☞ Example – Closest Pair of Points
  - Given list of n points in plane $(x_1,y_1),\ldots,(x_n,y_n)$, find closest pair
- ☞ $O(n^2)$ solution – Try all pairs

```
1   $min \leftarrow (x_1 - x_2)^2 + (y_1 - y_2)^2$
2   for $i = 1$ to $n$
3       do for $j = i + 1$ to $n$
4           do $d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2$
5               if $d < min$
6                   then $min \leftarrow d$
```

CUBIC TIME: $O(n^3)$

☞ Set Disjointness

- Given n sets $S_1,\ldots,S_n$ each of which is subset of elements $1,2,\ldots,n$ check if some pair is disjoint

```
1  for each set $S_i$
2      do for each other set $S_j$
3          do for each element $p \in S_i$
4              do determine if $p \in S_j$
5              if no element of $S_i$ belongs to $S_j$
6                  then report $S_i$ and $S_j$ are disjoint
```

POLYNOMIAL TIME: $O(n^k)$

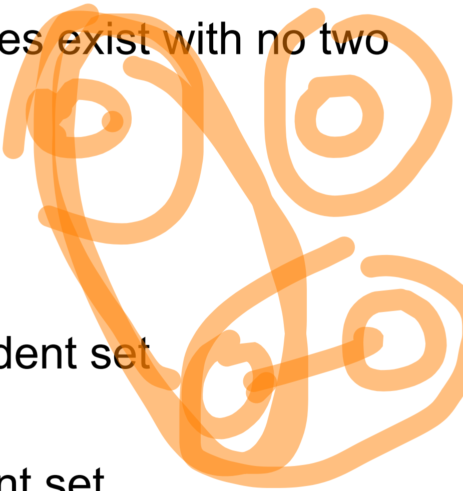
☞ Independent Set of Size *k*

- Given graph, check if some *k* nodes exist with no two joined by an edge

☞ $O(n^k)$ Solution

- Enumerate all subsets of *k* nodes

```
1  for each subset S of k nodes
2      do check whether S is independent set
3          if S is independent set
4              then report S as independent set
```



EXPONENTIAL TIME

- ☞ Independent Set
  - Given graph, find largest independent set
- ☞ $O(n^2 2^n)$ Solution
  - Enumerate all subsets

```
1  $S^* \leftarrow \varnothing$
2  for each subset $S$ of nodes
3      do check whether $S$ independent set
4          if $S$ largest independent set seen so far
5              then update $S^* \leftarrow S$
```

SUBLINEAR TIME

☞ Binary Search – $O(\log n)$

- Given sorted array A of size n, check whether p in array
- Start with BinarySearch(A,1,n,p)

```
BinarySearch($A$,$i$,$j$,$p$)
1  $m \leftarrow \lfloor (j-i)/2 \rfloor$
2  if $A[m] = p$
3      then return true
4  else if $p < A[m]$
5      then return BinarySearch($A$,$i$,$m$-1,$p$)
6  else return BinarySearch($A$,$m$+1,$j$,$p$)
```

DECISION TREES FOR COMPARISON SORTING

☞ Comparison Sorting
- Direct Comparisons of Elements

☞ View Abstractly as Decision Trees
- Full Tree
- Each internal node annotated by $i : j$ for some i and j in $1 \le i, j \le n$, for n elements in input sequence
- Each leaf annotated by permutation
  - $\langle \pi(1), \pi(1), \ldots \pi(n) \rangle$
- Executing sorting algorithm equates to tracing path in decision tree

DECISION TREE EXAMPLE

- ☞ Any correct sorting must be able to generate each of the *n*! permutation on *n* elements
  - Each permutation must appear as leaf in decision tree

LOWER BOUND FOR WORST-CASE

- Worst-case number of comparisons
  - Length of longest path from root of decision tree to any reachable leaf
  - Height of tree
- Lower bound on any comparison sort algorithm
  - Minimum height of all decision trees where each permutation appears as reachable leaf

```
            1:2
          /     \
       < /     > \
        /         \
     2:3           1:3
     / \           / \
  < / > \       < / > \
   /     \       /     \
1,2,3    1:3  2,1,3    2:3
         / \           / \
      < / > \       < / > \
       /     \       /     \
    1,3,2   3,1,2 2,3,1   3,2,1
```

LOWER BOUND FOR WORST-CASE

☞ Theorem

- Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in worst case

Decision Trees

- ☞ Given a comparison sorting algorithm A, and some particular number n, we draw a tree corresponding to the different sequences of comparisons A might make on an input of length n.
- ☞ If the first comparison the algorithm makes is between the objects at positions a and b, then it will make the same comparison no matter what other list of the same length is input, because in the comparison model we do not have any other information than n so far on which to make a decision.
- ☞ Then, for all lists in which a<b, the second comparison will always be the same, but the algorithm might do something different if the result of the first comparison is that a>b.
- ☞ So we can draw a tree, in which each node represents the positions involved at some comparison, and each path in the tree describes the sequence of comparisons and their results from a particular run of the algorithm. Each node will have two children, representing the possible behaviors of the program depending on the result of the comparison at that node.

- ☞ This tree describes an algorithm in which the first comparison is always between the first and second positions in the list (this information is denoted by the "1:2" at the root of the tree). If the object in position one is less than the object in position two, the next comparison will always be between the second and third positions in the list (the "2:3" at the root of the left subtree). If the second is less than the third, we can deduce that the input is already sorted, and we write "1,2,3" to denote the permutation of the input that causes it to be sorted. But if the second is greater than the third, there still remain two possible permutations to be distinguished between, so we make a third comparison "1:3", and so on.
- ☞ Any comparison sorting algorithm can always be put in this form, since the comparison it chooses to make at any point in time can only depend on the answers to previously asked comparisons. And conversely, a tree like this can be used as a sorting algorithm: for any given list, follow a path in the tree to determine which comparisons to be made and which permutation of the input gives a sorted order. This is a reasonable way to represent algorithms for sorting very small lists (such as the case n=3 above) but for larger values of n it works better to use pseudo-code. However this tree is also useful for discovering various properties of our original algorithm A.