Design and Analysis of Algorithms

- Understanding of Algorithm
- Upper bounds on specific solutions
- Lower bounds on problems

Algorithms – What Are They?

Definition. An _____ is a well-defined computational procedure that transforms inputs into outputs, achieving the desired input-output relationship.

Definition. A computational _____ is a specification of the desired input-output relationship.

Definition. An _____ of a problem is all the inputs needed to compute a solution to the problem.

Definition. A _____ algorithm halts with the correct output for every input instance. We can then say the algorithm _____ the problem.

Example: Sorting

Sorting is a common operation

Many sorting algorithms available, best choice depends on application

PROBLEM:

- INPUT: Sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- OUTPUT: Permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

INSTANCE: $\langle 6, 4, 3, 7, 1, 4 \rangle \longrightarrow \langle 1, 3, 4, 4, 6, 7 \rangle$

Algorithm: Insertion Sort

In-Place Sort: uses only a fixed amount of storage beyond that needed for the data.

Pseudocode:

```
Insertion-Sort(A)              ; A is an array of numbers
1     for j = 2 to length(A)
2         key = A[j]
3         i = j - 1
4         while i > 0 and A[i] > key
5             A[i+1] = A[i]
6             i = i - 1
7         A[i+1] = key
```

Analyzing Algorithms

- Predict resource utilization

  _____

  _____

- Dependent on architecture

  Model of computation

  Sequential (RAM model)

  Parallel (PRAM model)

- Running Time = F(Problem Size)
  = F(Input Size)
  = number of primitive operations used to solve problem

- Input Size:

  Sorting: _____

  Multiplication: _____

  Graphs: _____

Operations

Examples: additions, multiplications, comparisons

Constant time C_i per i^{th} line of pseudocode

In reality each operations takes different amount of time

_____ constraints on the input, other than size, resulting in the fastest possible running time

_____ constraints on the input, other than size, resulting in the slowest possible running time

_____ average running time over every possible type of input (usually involves probabilities of different types of input)

Example: Insertion Sort

- $n = \text{length}(A)$
- t_j = number of times the while loop *test* is executed for that value of j

Insertion-Sort(A)		**Cost**	**Times (Iterations)**
1	for j = 2 to length(A)	c_1	n
2	key = A[j]	c_2	$n-1$
3	i = j - 1	c_3	$n-1$
4	while i > 0 and A[i] > key	c_4	$\Sigma_{j=2}^{n} t_j$
5	A[i+1] = A[i]	c_5	$\Sigma_{j=2}^{n} t_j - 1$
6	i = i - 1	c_6	$\Sigma_{j=2}^{n} t_j - 1$
7	A[i+1] = key	c_7	$n-1$

Analysis

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Best Case: Array already sorted, $t_j = 1$ for all j

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5(0) + c_6(0) \\ &\quad + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) * n - (c_2 + c_3 + c_4 + c_7) \\ &= an + b \quad \text{(linear in } n\text{)} \end{aligned}$$

Worst Case: Array in reverse order, $t_j = j$ for all j

Note that $\Sigma_{j=1}^{n} j = \frac{n(n+1)}{2}$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{2} - 1) + c_5 \frac{n(n-1)}{2} \\ &\quad + c_6 \frac{n(n-1)}{2} + c_7(n-1) \\ &= (c_4/2 + c_5/2 + c_6/2)n^2 \\ &\quad + (c_1 + c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_7)n \\ &\quad - (c_2 + c_3 + c_4 + c_7) \\ &= an^2 + bn + c \quad \text{(quadratic in } n\text{)} \end{aligned}$$

Average Case: Check half of array on average, $t_j = \mathrm{j}/2$ for all j

$T(n) = an^2 + bn + c$

Analysis

- Concentrate on worst-case running time
- Upper bound
- Average case not much better

Order of Growth

The _____ of a running-time function Θ is the fastest growing term, discarding constant factors.

Insertion Sort:

- Best Case: $an + b \longrightarrow \Theta(n)$
- Worst Case: $an^2 + bn + c \longrightarrow \Theta(n^2)$

XYZ-Sort:

- Worst Case: $\Theta(n^3)$
- Now we can say Insertion-Sort better than XYZ-Sort *for large inputs*.

Designing Algorithms

- Incremental Design
  - Iterative

	Size n					
Complexity	10	20	30	40	50	
n	.00001 s	.00002 s	.00003 s	.00004 s	.00005 s	
n^2	.0001 s	.0004 s	.0009 s	.0016 s	.0025 s	
n^3	.001 s	.008 s	.027 s	.064 s	.125 s	
n^5	.1 s	3.2 s	24.3 s	1.7 min	5.2 min	
2^n	.001 s	1.0 s	17.9 min	12.7 days	35.7 years	
3^n	.059 s	58 min	6.5 years	3855 centuries	$2x10^8$ centuries	$1.3x$

  - Example: Insertion Sort

- Divide-and-Conquer
  - Recursive
  - Example: Merge Sort
  - 1. _____ problem into smaller subproblems
  - 2. _____subproblems by solving them recursively
  - 3. _____ solutions of subproblems

Example: Merge Sort

1. _____ the n element sequence to be sorted into two subsequences of n/2 elements each.
2. _____ (sort) the two subsequences recursively using merge sort

3. _____ (merge) the two sorted subsequences to produce the sorted answer

---

Example: Merge Sort

Recursion bottoms out when subproblem contains only one element (p = r)

```
MergeSort(A,p,r)
1     if p < r
2     then q = ⌊ (p+r)/2 ⌋
3          MergeSort(A,p,q)
4          MergeSort(A,q+1,r)
5          Merge(A,p,q,r)
```

Example: Merge Sort

Merge(A,p,q,r) procedure has $\Theta(n)$ running time, n = r - p + 1

```
Merge(A,p,q,r)                ; Exercise 1.3-2
1     for j = p to r
2          B[j] = A[j]
3     i = p
4     w = q + 1
5     while (p ≤ q) and (w ≤ r)
6          if B[p] ≤ B[w]
7          then A[i] = B[p]
8               p = p + 1
9          else A[i] = B[w]
```

```
10              w = w + 1
11          i = i + 1
12      if p > q
13      then p = w
14          q = r
15      for j = p to q
16          A[i] = B[j]
17          i = i + 1
```

Analyzing Divide-and-Conquer Algorithms

Definition: A _____ or _____ describes the running time of a recursive algorithm on a problem of size n in terms of the running time of the algorithm on smaller inputs.

For small enough input size (n $\leq$ c, for example), running time is constant.

- Example: T(n) = $\Theta(n^0) = \Theta(1)$
- Sorting one element in Merge Sort

For larger input size:

- D(n) = _____
  each of size ____

- C(n) = _____

Thus we generate the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Analysis of Merge Sort

Note if n = 1, Then T(n) = $\Theta(1)$

1. DIVIDE computes middle of array in constant time: D(n) = $\Theta(1)$
2. CONQUER sorts 2 subarrays of size n/2 in time: 2T(n/2)
3. COMBINE (Merge) procedure: C(n) = $\Theta(n)$

$$\text{T(n)} = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(1) + \Theta(n) & \text{if } n > 1 \end{cases}$$

Note that $\Theta(n)$ dominates $\Theta(1)$.

We will show next class that T(n) = $\Theta(n \lg n)$, where lg = $\log_2 n$.

Summary

Does the computational difference amount to much?

- Supercomputer running efficient insertion sort

  - 100 x 10^6 instructions per second
  - $2n^2$ instructions to sort n numbers
- Personal computer running inefficient merge sort
  - 1 x 10^6 instructions per second
  - $50n \lg n$ instructions to sort n numbers
- Sorting one million numbers (n = 10^6)
- Supercomputer: $\frac{2(10^6)^2 \text{ instructions}}{10^8 \text{ instructions/sec}} =$ _____
- Personal Computer: $\frac{50(10^6)\lg(10^6) \text{ instructions}}{10^6 \text{ instructions/sec}} =$ _____
- MORAL: A little computational shrewdness can go a long way.

Growth of Functions

Definition: The _____ efficiency of an algorithm is the order of growth of the algorithm as the input size approaches the limit (increases without bound).

The asymptotically more efficient algorithm is usually the better choice for all but small inputs.

Theta

f(n) = $\Theta(g(n))$, g(n) is the Asymptotically Tight Bound for f(n)

$\Theta(g(n)) = \{$f(n): there exist positive constants c_1, c_2, and n_0 such that
$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$ $\}$

"f(n) $= \Theta(g(n))$" means f(n) is an element of the set of functions g(n).

Here is a graphical depiction of Θ.

Click mouse to advance to next frame.

Example:

Show that $n^2 - 2n = \Theta(n^2)$ and $200n^2 - 100n = \Theta(n^2)$.

$c_1 n^2 \leq n^2 - 2n \leq c_2 n^2$	$c_1 n^2 \leq 200n^2 - 100n \leq c_2 n^2$
$c_1 \leq 1 - 2/n \leq c_2$	$c_1 \leq 200 - 100/n \leq c_2$
$c_1 \leq 1/3$, $c_2 \geq 1$	$c_1 \leq 100$, $c_2 \geq 200$
$n \geq 3$, $n \geq 1$	$n \geq 1$, $n \geq 1$
$n_0 = 3$	$n_0 = 1$

Because some choice for c_1, c_2, and n_0 exists, then the functions are both $\Theta(n^2)$.

Because coefficients on the high-order term only affect constants, they are dropped from the Θ notation.

$\Theta(n^0) = \Theta(1)$ constant

O(g(n))

f(n) = O(g(n)), g(n) is an Asymptotic Upper Bound for f(n)

O(g(n)) = {f(n): there exist positive constants c and n_0 such that $0 \leq$ f(n) $\leq$ c*g(n) for all $n \geq n_0$ }

See Figure 2.1b, page 25, for graphical depiction of O.

Examples:

- n^2 - 2n = O(n^2)
- 200 n^2 - 100n = O(n^2) = O(n^3) = ...
- n = O(n^2)
- Does f(n) = $\Theta(g(n))$ imply f(n) = O(g(n))? ____
- Does f(n) = O(g(n)) imply f(n) = $\Theta(g(n))$? ___

Omega(g(n))

f(n) = $\Omega(g(n))$, g(n) is an Asymptotic Lower Bound for f(n)

$\Omega(g(n))$ = {f(n): there exist positive constants c and n_0 such that $0 \leq$ c*g(n) $\leq$ f(n) for all $n \geq n_0$ }

See Figure 2.1c, page 25, for graphical depiction of Ω.
Examples:

- n^2 - 2n = $\Omega(n^2)$

- 200 n^2 - 100 n = $\Omega(n^2) = \Omega(n) = \Omega(1)$
- $n^2 = \Omega(n)$
- Does f(n) = $\Theta(g(n))$ imply f(n) = $\Omega(g(n))$? ____
- Does f(n) = $\Omega(g(n))$ imply f(n) = $\Theta(g(n))$? ___

Theorem 2.1

f(n) = $\Theta(g(n))$ if and only if f(n) = O(g(n)) and f(n) = $\Omega(g(n))$.

Asymptotic Notation in Equations

Example: $2n^2 + 3n + 1 = 2n^2 + \Theta(n)$

In other words, there is some function f(n) element of $\Theta(n)$ that makes the equation true; namely, 3n + 1.

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n) = \Theta(n^2).$$

In other words, for any function f(n) element of $\Theta(n)$, there is some function h(n) element of $\Theta(n^2)$; namely, $2n^2$ + f(n).

Summary

o notation f(n) = o(g(n)), g(n) is an upper bound of f(n) that is not asymptotically tight

ω **notation** f(n) = $\omega(g(n))$, g(n) is a lower bound of f(n) that is not asymptotically tight

Example: f(n) = $3n^3 + 4$

- f(n) = $\Theta(n^3)$
- f(n) = $O(n^3) = O(n^4) = ...$
- f(n) = $\Omega(n^3) = \Omega(n^2) = \Omega(n) = \Omega(1)$
- f(n) = $o(n^4) = o(n^5) = ...$
- f(n) = $\omega(n^2) = \omega(n) = \omega(1)$

Some Useful Mathematical Tools To Remember

- monotonically increasing, strictly increasing
- monotonically decreasing, strictly decreasing
- lg n = $log_2 n$ = binary logarithm
- ln n = $log_e n$ = natural logarithm, e = 2.7182...
- $log_b a^n = n log_b a$
- $log_c(ab) = log_c a + log_c b$
- a = $b^{log_b a}$
- $a^{log_b n} = n^{log_b a}$
- $log_a b = \frac{log_c\, b}{log_c\, a}$
- Stirling's approximation to n! = $\sqrt{2\pi n}\ (n/e)^n\ (1 + \Theta(1/n))$
- Splitting summations

- Mathematical Induction
- Review other tools starting on page 34
