Goals of the Course

- Cover fundamental data structures/algorithms including: trees, hashing, sorting, etc.
- Teach tricks of the trade (time/space trade-off).
- Analyze algorithms/data structures – learn to argue the merits of a particular algorithm.
- Analyze the complexity or growth rate of algorithm running time.

Math Review

Be sure to know all the formulas described in §1.2 of the book.

What is the number of nodes in a complete binary tree of depth N?

Analysis of List Insertion

Problem: Insert n elements into a linked list checking for duplicates (the list is unsorted).

$$S = 1 + 2 + 3 + \ldots + (n-1) + n$$

$$S = n + (n-1) + (n-2) + \ldots + 2 + 1$$

$$2S = (n+1) * n, \quad S = [(n+1) * n]/2$$

Is sorting the list helpful? in the worst case? in the average case?

Computational Complexity

How to measure "performance" of an algorithm?

- We consider the growth of running "time" as a function of the **size** of the input to the algorithm. What units for time? which input?
- Typically, we consider worst case running time. Why worst case? (Average is sometimes considered as well.)
- What does "average" mean?
  What is the "average input"?
  Who is the average person in this building?
- Reading the data.
- Writing the data!

Growth Rates (chapter 2)

$2n + 5 \approx n$

Consider: larger and larger values of n.

In contrast, consider: 2^n vs n^3.

	$2n+5$	n^3	2^n
$n = 5$	15	125	32
$n = 10$	25	1,000	1,024
$n = 20$	45	8,000	1,048,576

To formally analyze growth rates, we introduce $big - O$ notation. (§2.1 in the book.)

$t(n) = O(f(n))$

if there are constants c and n_0 such that $t(n) \leq c * f(n)$ when $n \geq n_0$.

Examples

- $n + 7 = O(n)$
- $10n + 7 = O(n)$
- Is $10^{-4}n^2 = O(10^3 n)$?
- Is $1000n = O(.001n^2)$?
- For any $f(n)$, is it always the case that $f(n) = O(f(n))$?
- Intuition: $O(f) \equiv$ upper bound on f
- $6n^2 = O(n^2) = O(n^3) = O(2^n)$?
- "style:" $O(f) \equiv$ tight upper bound on f
- Are growth rates transitive, i.e.,
  if $f(n) = O(g(n))$ and $G(n) = O(h(n))$,
  is it the case that $f(n) = O(h(n))$?

Growth Rate Rules

Rule 1:

$$If T_1(n) = O(f(n))\text{and } T_2(n) = O(g(n)), \text{then}$$

$$(a) T_1(n) + T_2(n) = max(O(f(n)), O(g(n))),$$

$$(b) T_1(n) * T_2(n) = O(f(n) * g(n))$$

Examples:

- $n^3 + 2n^2 = O(n^3)$
- $6n^3 * 20n^4 = O(n^7)$

In general, **Rule 2:**
If $T(x)$ is a polynomial of degree n,
then $T(x) = O(x^n)$.
Order notation style: No constants, No lower-order terms, Tight upper bound.

Analyzing Loops

Here is a program to calculate $\Sigma_{i=1}^{n} i^3$:

```
    function sum(n: integer): integer;
        var   i, partial_sum: integer;

   begin
{1}  partial_sum := 0;
{2}    for i := 1 to n do
{3}      partial_sum := partial_sum+i*i*i;
{4}    sum := partial_sum;
   end;
```

- Line-by-line analysis.
- Growth order analysis.
- Analyzing **for** loops: the running time of a **for** loop is at most the running time of the statement inside the loop **times the number of iterations**.

Nested For Loops

The following program fragment is $O(n^2)$:

```
for i := 1 to n do
     for j := 1 to n do
          k := k + 1;
```

The following program fragment, which has $O(n)$ work followed by $O(n^2)$ work, is also $O(n^2)$:

```
for i := 1 to n do
     a[i] := 0;
for i := 1 to n do
     for j := 1 to n do
          a[i] := a[i] + a[j] + i + j;
```

The Fibonacci Sequence

Each number is the sum of the previous two numbers:

$$
\begin{aligned}
f(n) &= f(n-1) + f(n-2) \\
\text{Define} \quad f(0) &= 0 \\
f(1) &= 1
\end{aligned}
$$

and the entire sequence is: $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$

This equation is actually a recursive procedure for computing the Fibonacci sequence.

Fibonacci Procedure

```
    function fib(n: integer): integer;
    begin
{1}        if  (n=1) then
{2}        fib := 1
       else
{3}        fib := fib(n-1) + fib(n-2);
    end;
```

The procedure is elegant, but what is its efficiency?

How many times is $f(n-2)$ computed?

How many times is $f(n-3)$ computed?

To analyze the complexity of Fibonacci precisely, we need to understand recurrence relations.

Recurrence Equations

Recurrence relation is an equation that relates the value of a function to the values of the same function for smaller arguments.

Example:

$t(0) = 1$

$t(n) = 2t(n-1)$ if $n > 0$

$t(n) = 2t(n-1) = 4t(n-2) = \ldots = 2^n t(0) = 2^n$

Running Time of fib

To analyze the Fibonacci sequence, we compute a lower bound on $t(n)$ as follows:

$t(n) = t(n-1) + t(n-2)$

$t(n-1) \geq t(n-2)$

so $t(n) \geq 2t(n-2)$

If we carry out a similar expansion as above, we have

$t(n) \geq 2t(n-2) \geq 4t(n-4) \geq 2^k t(n-2k)$

Set $k = n/2$ and we get that

$t(n) \geq 2^{n/2}$

Thus, the Fibonacci sequence recursion grows exponentially in n.
Observe, $t(n) \leq 2t(n-1)$, so $t(n) \leq 2^n$

$\log$s in Running Time

How many times can you cut a number in half until it becomes 1? (e.g., 16 or 32)

In general, ?

Suppose the input to an algorithm is a list of n numbers. Then, can we have an order $\log n$ algorithm?

Binary Search

Given an integer x and integers $a_1, a_2, \ldots, a_n$, which are presorted and **already in memory**, find i such that $a_i = x$, or return $i = 0$ if x is not in the input.

What is a linear-time solution?

What is a $\log n$-time solution?

Binary Search Example

INDEX: 1 ... 5 ... 9

ARRAY: A ... E ... I

```
FIND(B)

ARRAY(5) = E > B
           H = 5 - 1 = 4
           M = [(4 + 1) / 2] = 2
           ARRAY(2) = B           DONE
```

L	M	H
1	5	9
1	2	4

Efficient Exponentiation

What is the obvious algorithm for computing x^n for arbitrary x and n?

What is the running time of the algorithm as a function of n?

Can you see an algorithm that takes order $\log n$-time?

Hint: $x^n = x^{n/2} * x^{n/2} \quad (n \text{ is even})$

and $x^n = x^{(n-1)/2} * x^{(n-1)/2} * x \quad (n \text{ is odd})$.