CMPE 180-92

Data Structures and Algorithms in C++

November 2 Class Meeting

Department of Computer Engineering
San Jose State University

SILICON VALLEY'S FIRST CHOICE FOR NEW ENGINEERING HIRES
— Silicon Valley Business Journal, 2013

Fall 2017

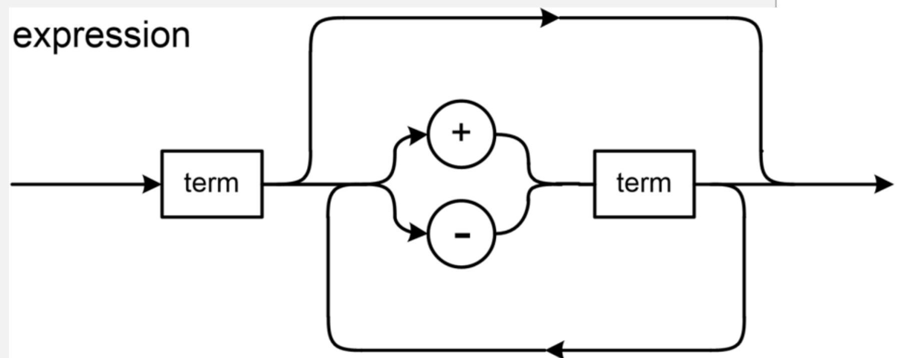
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

San José State UNIVERSITY

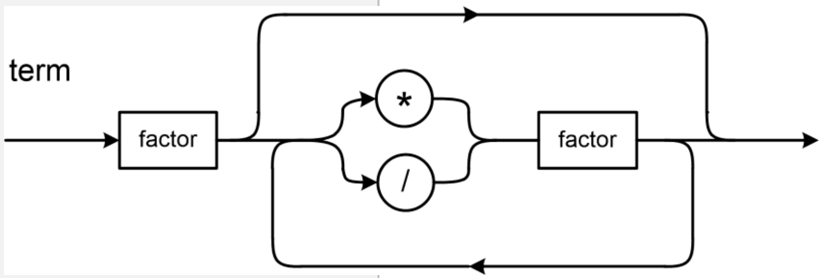
Assignment #10 Sample Solution

```
double Calculator::expression() const throw(string)
{
    double value = term();  // evaluate the first term
    bool done = false;
    char ch;

    do
    {
        cin >> ws;
        ch = cin.peek();

        switch (ch)
        {
            case '+':
            {
                cin >> ch;         // read the +
                value += term();   // evaluate the next term and add its value
                break;
            }

            case '-':
            {
                cin >> ch;         // read the -
                value -= term();   // evaluate the next term and subtract its value
                break;
            }

            default: done = true; // no more terms
        }
    } while (!done);


    return value;  // the expression's value
}
```



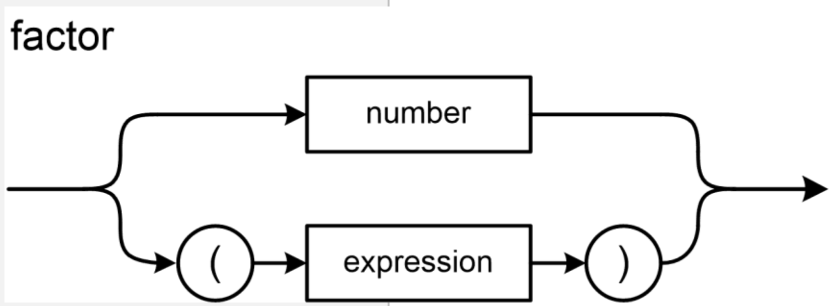
Assignment #10 Sample Solution, *cont'd*

```
double Calculator::term() const throw(string)
{
    double value = factor();  // evaluate the first factor
    bool done = false;
    char ch;

    do
    {
        cin >> ws;
        ch = cin.peek();

        switch (ch)
        {
            case '*':
            {
                cin >> ch;          // read the *
                value *= factor();  // evaluate the next factor and multiply its value
                break;
            }

            case '/':
            {
                cin >> ch;                    // read the /
                double value2 = factor();  // evaluate the next factor

                // Do the division unless the value is zero.
                if (value2 != 0) value /= value2;
                else
                {
                    throw string("Division by zero");
                    return 0;
                }

                break;
            }


            default: done = true;  // no more factors
        }
    } while (!done);


    return value;  // the term's value
}
```



Assignment #10 Sample Solution, *cont'd*

```
double Calculator::factor() const throw(string)
{
    cin >> ws;               // skip blanks
    char ch = cin.peek();    // what's the next input character?

    if (isdigit(ch))
    {
        return number();   // evaluate and return a number's value
    }

    else if (ch == '(')
    {
        cin >> ch;                     // read the (
        double value = expression();   // evaluate the subexpression

        ch = cin.peek();
        if (ch == ')') cin >> ch;      // read the )
        else
        {
            throw string("Missing )");
        }


        return value;  // return the parenthesized subexpression's value
    }

    else
    {
        throw string("Invalid factor");
        return 0;
    }
}
```



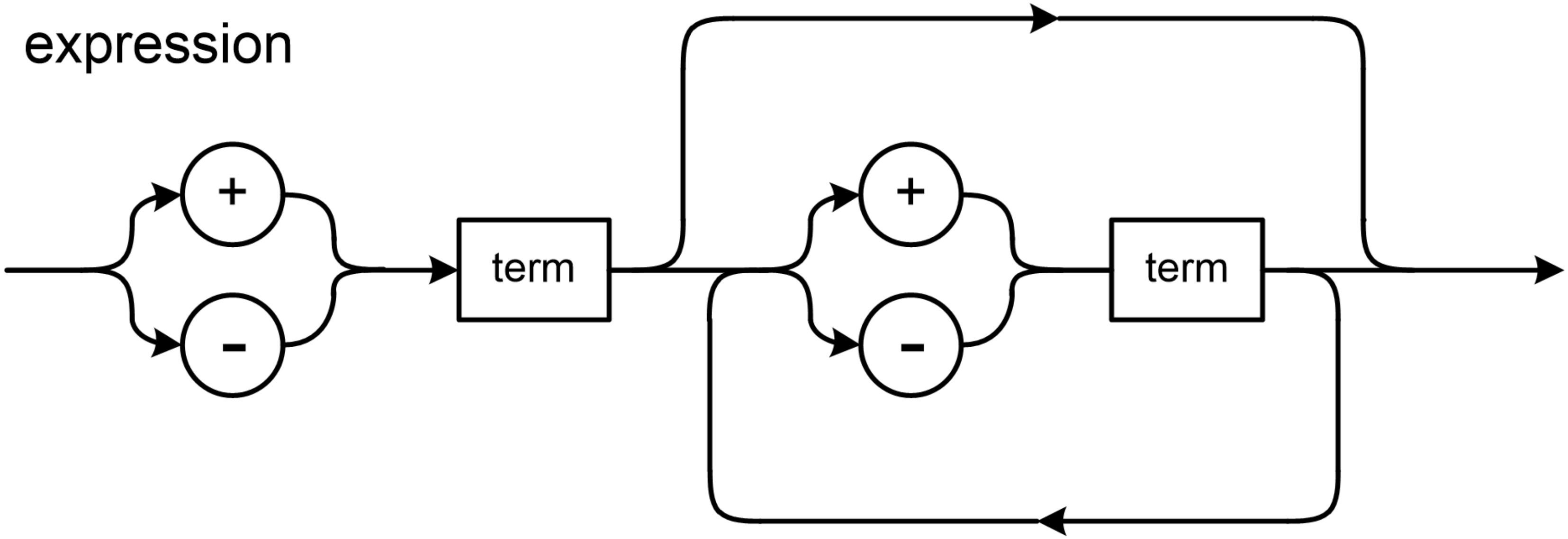
Assignment #10 Sample Solution, *cont'd*

```
double Calculator::number() const throw(string)
{
    double value;
    cin >> value;   // let >> read and evaluate the number
    return value;
}
```

Assignment #10 Extra Credit

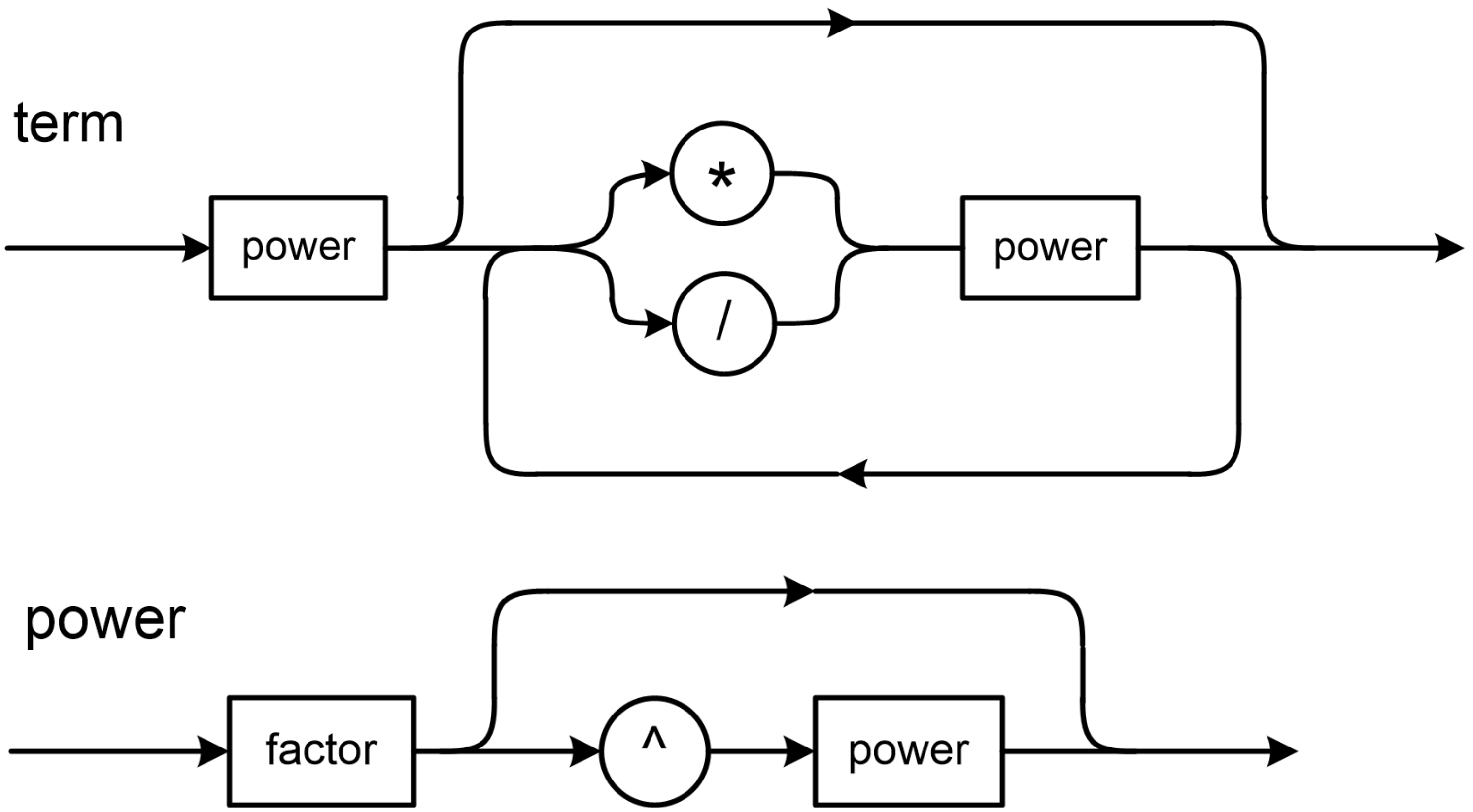
- A leading + or – sign.

expression



Assignment #10 Extra Credit, *cont'd*

- Exponentiation



Introduction to Algorithm Analysis

- To analyze an algorithm, we measure it.
- A convenient measure must be:
  - A resource we care about (elapsed time, memory usage, etc.).
  - Quantitative, to make comparisons possible.
  - Easy to compute.
  - A good predictor of the "goodness" of the algorithm.

In this class, we will be concerned mostly with elapsed time.

Introduction to Algorithm Analysis, *cont'd*

- Our concern generally is <u>not how long</u> a particular run of an algorithm will take, but <u>how well the algorithm scales</u>.

- How does the run time increase as the amount of input increases?
  - Example: How does the reading time of a book increase as the number of pages increases?
  - Example: How does the run time of a particular sort algorithm increase as the number of items to be sorted increases?

Example: Reading Books

- Algorithm: Read a book.
- Measure: Length of time to read a book.

- Given a set of books to read, can we predict how long it will take to read each one, without actually reading it?

- Possible ways to compute reading time:
  - weight of the book
  - physical size (width, height, thickness) of the book
  - total number of words
  - total number of pages

Introduction to Algorithm Analysis, *cont'd*

- When we compare two algorithms, we want to compare how well they scale.
  - How do their elapsed run times grow as the size of the input grows?
  - How do their growth rates compare?
- Can we do this comparison without actually running the algorithms?
  - Some algorithms may be too expensive to run.

How Well Does an Algorithm Scale?

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Figure 2.1 Typical growth rates

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012
ISBN 978-0-13-257627-7

How Well Does an Algorithm Scale? *cont'd*

	Algorithm Time			
Input Size	1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

Figure 2.2 Running times of several algorithms for maximum subsequence sum (in seconds)

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012
ISBN 978-0-13-257627-7

How Well Does an Algorithm Scale? *cont'd*

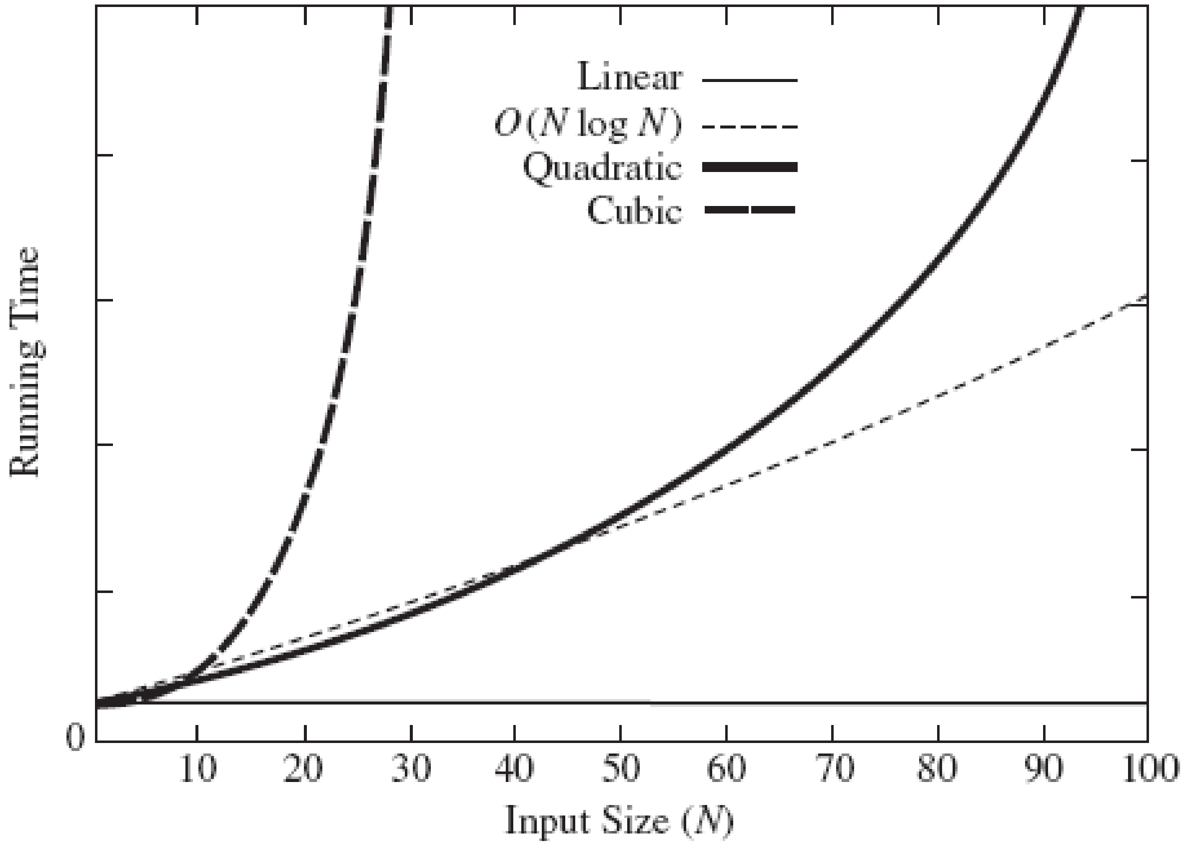


Figure 2.3 Plot (N vs. time) of various algorithms

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012
ISBN 978-0-13-257627-7

How Well Does an Algorithm Scale? *cont'd*

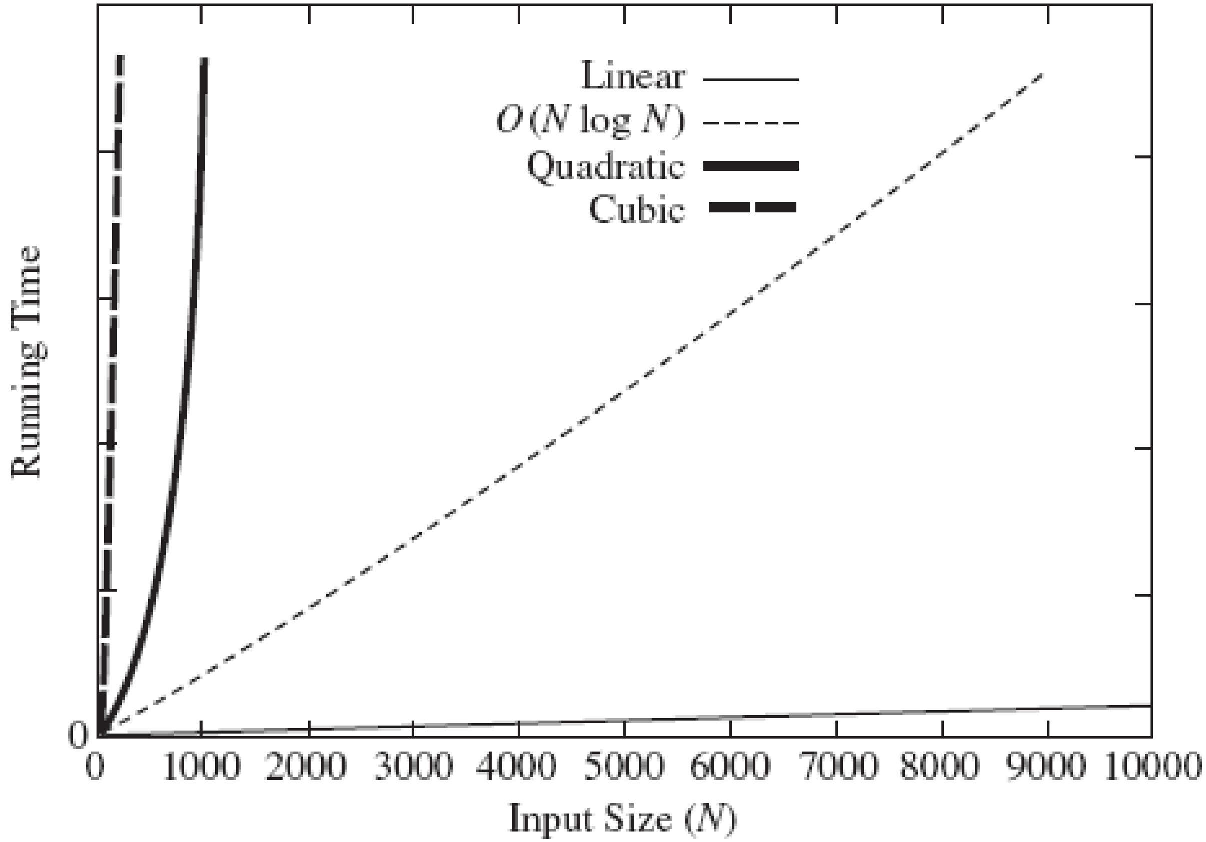
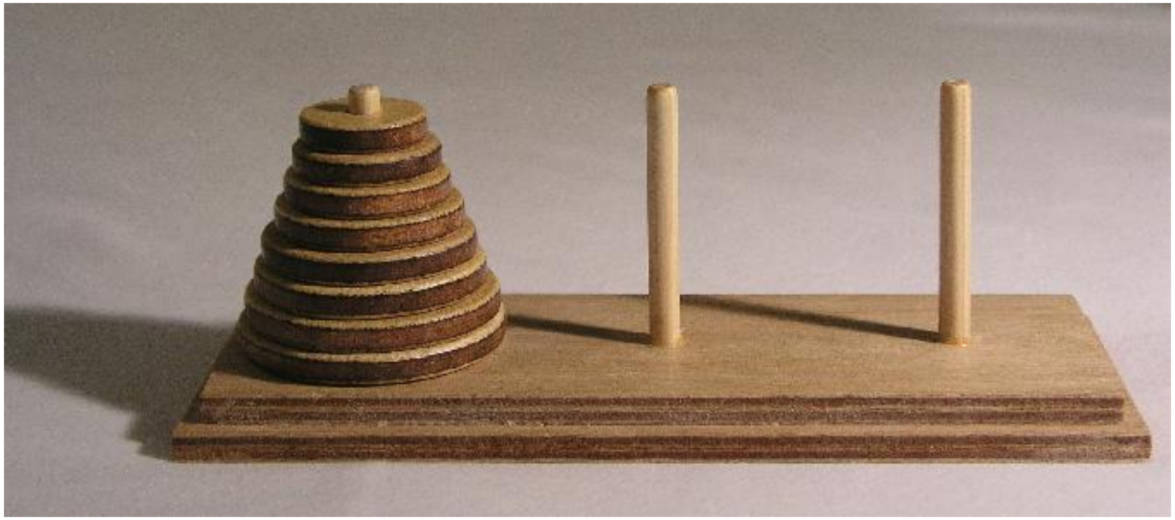


Figure 2.4 Plot (N vs. time) of various algorithms

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012
ISBN 978-0-13-257627-7

Towers of Hanoi



- **Goal:** Move the stack of disks from the source pin to the destination pin.
  - You can move only one disk at a time.
  - You cannot put a larger disk on top of a smaller disk.
  - Use the third pin for temporary disk storage.

Towers of Hanoi, *cont'd*

- Label the pins A, B, and C. Initial roles:
  - A: source
  - B: destination
  - C: temporary

During recursive calls, the pins will assume different roles.

- Base case: n = 1 disk
  - Move disk from A to B *(source → destination)*

- Simpler but similar case: n-1 disks
  - Solve for n-1 disks: A to C *(source → temp)*
  - Move 1 disk from A to B *(source → destination)*
  - Solve for n-1 disks: C to B *(temp → destination)*

Towers of Hanoi: Analysis

- How can we measure how long it will take to solve the puzzle for n disks?
- What's a good predictor?
  - The number times we move a disk from one pin to another.
  - Therefore, let's count the number of moves.

Towers of Hanoi: Analysis, *cont'd*

- Solve n disks
  - Solve for n-1 disks *(source → temp)*
  - Move 1 disk *(source → destination)*
  - Solve for n-1 disks *(temp → destination)*

- What is the pattern in the number of moves as n increases?
  - Let $f(n)$ be the number of moves for n disks.

$$f(n) = \begin{cases} 1 & n = 1 \\ 2f(n\text{-}1) + 1 & n > 1 \end{cases}$$

Towers of Hanoi: Analysis

$$f(n) = \begin{cases} 1 & n = 1 \\ 2f(n\text{-}1) + 1 & n > 1 \end{cases}$$

- This is a recurrence relation.
  - f shows up in its own definition: $f(n) = 2f(n\text{-}1) + 1$
  - The mathematical analogy of recursion.

- Can we find the definition of function f ?
  - Observation:
    Since $f(n) = 2f(n\text{-}1) + 1$, we know that $f(n) > 2f(n\text{-}1)$.
  - Therefore, if we increase the number of disks from n to n+1, the number of moves will at least double.

Towers of Hanoi: Count Moves

Hanoi2.cpp

```
int main()
{
    cout << "Disks Moves" << endl;

    for (int n = 1; n <= 10; n++) {
        int count = 0;
        solve(n, Pin::A, Pin::B, Pin::C, count);
        cout << setw(5) << n << setw(5) << count << endl;
    }
}

void move(Pin from, Pin to, int& count)
{
    count++;
}
```

Don’t print. Just count moves.

Towers of Hanoi: Analysis

Disks	Moves
1	1
2	3
3	7
4	15
5	31
6	63
7	127
8	255
9	511
10	1023

- What's the pattern?

$$f(n) = 2^n - 1$$

- Can we prove this?
  - Just because this formula holds for the first 10 values of n, does it hold for all values of $n \geq 1$?

Proof by Induction: Base Case

Prove that <u>if</u>:

$$f(n) = \begin{cases} 1 & n = 1 \\ 2f(n-1) + 1 & n > 1 \end{cases}$$

<u>then</u>:

$f(n) = 2^n - 1$
for all $n \geq 1$

- Let $n = 1$.
- Then $f(1) = 2^1 - 1 = 1$ is true.

Proof by Induction: Inductive Step

Prove that <u>if</u>:

$$f(n) = \begin{cases} 1 & n = 1 \\ 2f(n\text{-}1) + 1 & n > 1 \end{cases}$$

<u>then</u>:

$f(n) = 2^n - 1$
for all $n \geq 1$

- Let $n > 1$.
- **Inductive hypothesis:**
  Assume that $f(k) = 2^k - 1$ is true for all $k < n$, where $n > 1$
- Since $n\text{-}1 < n$, then by our hypothesis: $\boxed{f(n\text{-}1)} = \boxed{2^{n-1} - 1}$.
- From the recurrence relation:
  $f(n) = 2\boxed{f(n\text{-}1)} + 1 = 2\boxed{(2^{n-1} - 1)} + 1 = 2^n - 1$.
- So $f(n) = 2^n - 1$ for all $n > 1$.
- Therefore, if $f(k) = 2^k - 1$ is true for all $k < n$,
  it must also be true for n as well.

Proof by Induction: What Happened?

Prove that <u>if</u>:

$$f(n) = \begin{cases} 1 & n = 1 \\ 2f(n-1) + 1 & n > 1 \end{cases}$$

<u>then</u>:

$f(n) = 2^n - 1$
for all $n \geq 1$

- First we proved it for $n = 1$ (the base case).
- Then we proved that <u>if</u> it's true for all $k < n$, where $n > 1$ (the induction hypothesis) then it must also be true for n.
- Suppose $n = 2$. Since we know it's true for $n = 1$ (the base case), it must be true for $n = 2$ (from above).
- Suppose $n = 3$. Since we know it's true for $n = 2$ (from above), it must be true for $n = 3$.
- Etc.!

Another Proof By Induction Example

Observe that:

$1 = 1^2$
$1 + 3 = 4 = 2^2$
$1 + 3 + 5 = 9 = 3^2$
$1 + 3 + 5 + 7 = 16 = 4^2$
… ?

Prove:

$1 + 3 + 5 + 7 + \ldots + (2n - 1) = n^2$ for all $n > 0$

Base case:
Let $n = 1$. Then $1 = 1^2$ is obviously true.

- **Induction hypothesis:**
  Assume that $1 + 3 + 5 + 7 + \ldots + (2k - 1) = k^2$ for some $k > 0$
- **Then show that:** $1 + 3 + 5 + 7 + \ldots + (2(k + 1) - 1) = (k + 1)^2$
- $1 + 3 + 5 + 7 + \ldots\ldots\ldots\ldots\ldots + (2(k + 1) - 1) =$
- $1 + 3 + 5 + 7 + \ldots + (2k - 1) + (2(k + 1) - 1) =$
- $k^2 + (2(k + 1) - 1) = k^2 + 2k + 1$
- $= (k + 1)^2$

True for $n = 1$.
If true for $n = k$
then true for $n = k+1$.

Algorithm Analysis

- An algorithm is a set of operations to perform in order to solve a problem.
- We want to know how an algorithm scales as its input size grows.
- If $T(N)$ is the running time of an algorithm with N input values, then how does $T(N)$ change as N increases?

Big-Oh and its Cousins

- Let $T(N)$ be the running time of an algorithm with N input values.

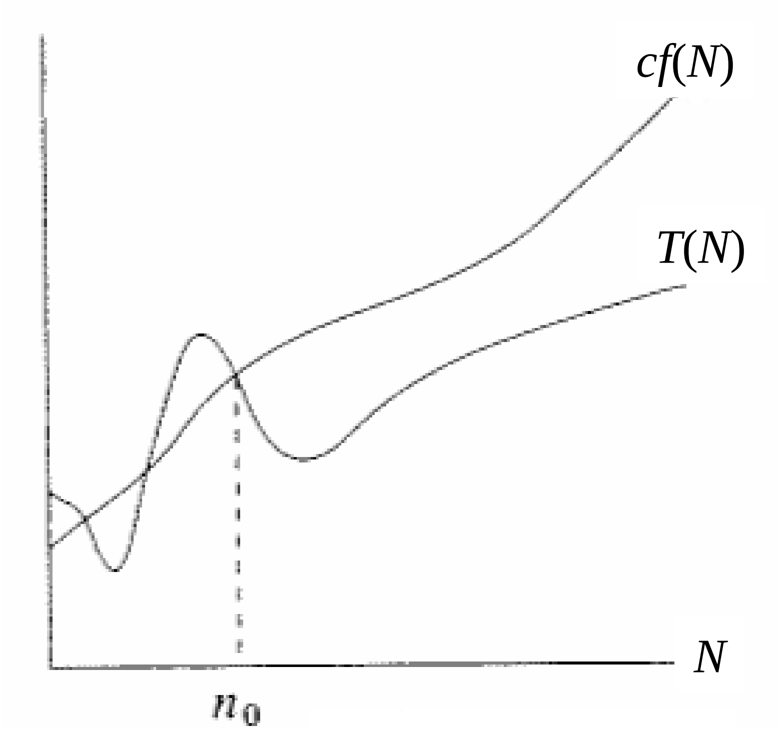
- Big-Oh
  - $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.
  - In other words, when N is sufficiently large, function $f(N)$ is an upper bound for time function $T(N)$.
    - We don't care about small values of N.
  - $T(N)$ will grow no faster than $f(N)$ as N increases.

Big-Oh and its Cousins, *cont'd*

- Let $T(N)$ be the running time of an algorithm with N input values.

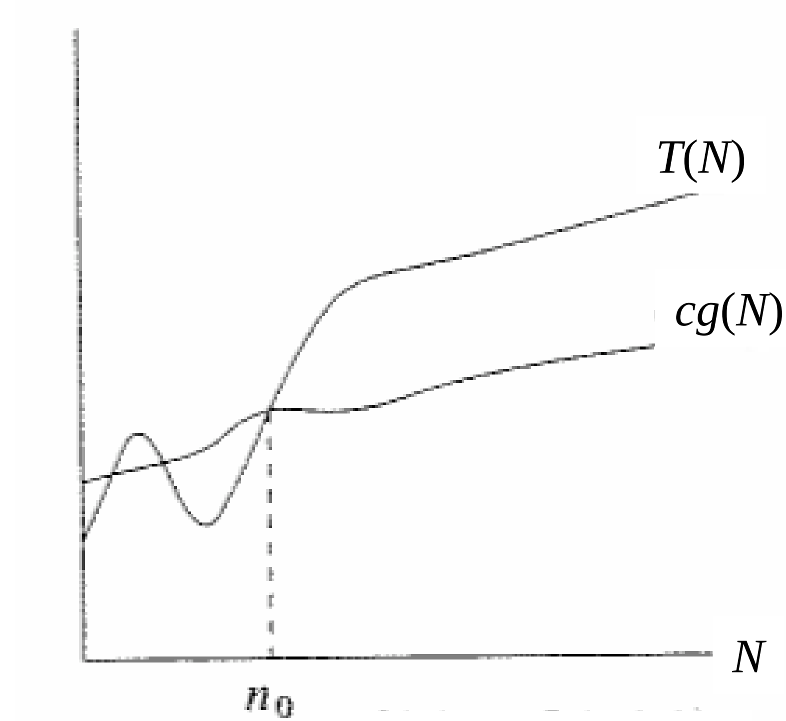
- Omega
  - $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.
  - In other words, when N is sufficiently large, function $g(N)$ is <u>lower bound</u> for time function $T(N)$.
    - We don't care about small values of N.
  - $T(N)$ will grow <u>at least as fast</u> as $g(N)$ as N increases.

Big-Oh and its Cousins, *cont'd*



$$T(N) = O(f(N))$$

Upper bound



$$T(N) = \Omega(g(N))$$

Lower bound

Big-Oh and its Cousins, *cont'd*

- Let $T(N)$ be the running time of an algorithm with N input values.

- Theta
  - $T(N) = \Theta(h(N))$ if and only if:
    - $T(N) = O(h(N))$ and
    - $T(N) = \Omega(h(N))$
  - In other words, the rate of growth of $T(N)$ equals the rate of growth of $h(N)$.

Big-Oh and its Cousins, *cont'd*

- Let $T(N)$ be the running time of an algorithm with N input values.

- Little-Oh
  - $T(N) = o(p(N))$ if there are positive constants c and n_0 such that $T(N) < cp(N)$ when $N \geq n_0$.
  - $p(N)$ is similar to the upper bound function $f(N)$ but instead of $T(N) \leq cf(N)$ we have $T(N) < cp(N)$.

Big-Oh and its Cousins, *cont'd*

- If $T_1(N) = O(f_1(N))$ and $T_2(N) = O(f_2(N))$ then
  - $T_1(N) + T_2(N) = O(f_1(N) + f_2(N))$ or $O(\max(f_1(N), f_2(N)))$
  - $T_1(N) \text{ x } T_2(N) = O(f_1(N) \text{ x } f_2(N))$
- If $T(N)$ is a polynomial of degree k, then
  - $T(N) = \Theta(N^k)$
- If $T(N) = \log^k N$ for any constant k, then
  - $T(N) = O(N)$
  - Logarithms grow slowly!

Quiz

- Canvas: Quiz 7 – 2017 Apr 13
  - 20 minutes

Break

Towers of Hanoi: Rate of Growth

- We decided that a good predictor of $T(n)$ for solving the Towers of Hanoi problem was $f(n)$.
  - n is the number of disks
  - $f(n) = 2^n-1$ is the number of disk moves
- Therefore,

$$T(n) = \Theta(2^n)$$

Compare Growth Rates

- If we want to compare the growth rates of two functions $f(N)$ and $g(N)$, compute

$$\lim_{N \to \infty} f(N) / g(N)$$

  - The limit is 0: $f(N) = o(g(N))$
    $g(N)$ is an upper bound for $f(N)$.
  - The limit is a constant $c \neq 0$: $f(N) = \Theta(g(N))$
    $f(N)$ *and* $g(N)$ have the same growth rate.
  - The limit is ∞: $g(N) = o(f(N))$
    $f(N)$ is an upper bound for $g(N)$.

General Rules for Computing Running Time

- Consecutive statements
  - Add the running times of the statements.
  - Generally, only consider the statement with the maximum running time.
- Branching statement
  - The running time of the entire statement is at most the maximum running time of its branches.

Computing Running Time, *cont'd*

- Loop
  - The running time of a loop is at most
    the number of iterations times
    the running time of the statements in the loop.

- Nested loops
  - Compute the running time of the statements in the innermost loop, then multiply by the product of the numbers of iterations of all the loops.

Scalability of Different Algorithms

- Problem: Compute the n^{th} Fibonacci number.
- Two algorithms to solve this problem:
  - Start with 1, 1, and repeatedly add the previous two values.
    - LinearGrowthRate: $T(N) = O(N)$
  - Use recursion: $fib(n) = fib(n\text{-}2) + fib(n\text{-}1)$
    - ExponentialGrowthRate: $T(N) = \Omega(1.5^N)$

Why is the growth rate exponential?

Scalability of Different Algorithms, *cont'd*

- One set of results for the Fibonacci problem.
  - Times in milliseconds.

n	Linear	Exponential
5	0	0
10	0	1
15	0	0
20	0	2
25	0	3
30	0	4
35	0	45
40	0	504
45	0	5358
50	0	59267

Hash Tables

- Consider an array or a vector.
  - To access a value, you use an integer index.
- The array "maps" the index to a data value stored in the array.
  - The mapping function is very efficient.
  - As long as the index value is within range, there is a strict one-to-one correspondence between an index value and a stored data value.
- We can consider the index value to be the "key" to the corresponding data value.

Hash Tables, *cont'd*

- A hash table also stores data values.
  - Use a key to obtain the corresponding data value.
- The key does not have to be an integer value.
  - For example, the key could be a string.
- There might not be a one-to-one correspondence between keys and data values.
- The mapping function might not be trivial.

Hash Tables, *cont'd*

- We can implement a hash table as an <u>array of cells</u>.
  - Refer to its size as *TableSize*.

- If the hash table's mapping function maps a key value into an integer value in the range 0 to *TableSize* – 1, then we can use this integer value as the index into the underlying array.

Hash Tables, *cont'd*

- Suppose we're storing employee data records into a hash table.
- We use an employee's name as the key.

Hash Tables, *cont'd*

- Suppose that the name
  - *john* hashes (maps) to 3
  - *phil* hashes to 4
  - *dave* hashes to 6
  - *mary* hashes to 7
- This is an ideal situation because each employee record ended up in a different table cell.

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Figure 5.1 An ideal hash table

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012
ISBN 0-13-257627-9

Hash Function

- We need an ideal hash function to map each data record into a distinct table cell.
  - It can be very difficult to find such a hash function.

A Simple Hash Function

- Suppose our keys are words and the table size is 10,007 (a prime number).

```
int hash(const string& word, int table_size)
{
    int hashVal = 0;
    for (char ch : word) hashVal += ch;
    return hashVal%table_size;
}
```

- Problem: This hash function does not distribute the keys well if the table is large.
  - The maximum ASCII character value is 127.
  - If a typical word is 8 characters long, the hash function generally has values from 0 to 1,016.

Another Simple Hash Function

```
int hash(const string& word, int table_size)
{
    return (key[0] + 27*key[1] + 729*key[2])%table_size;
}
```

- We use only the first three letters of each word.
  - 27 letters in the alphabet + space
  - $729 = 27^2$
- Good distribution into a table of 10,007 if the first three letters are random.
  - But the English language is not random and many words will start with the same three letters.

A Better Hash Function

```
int hash(const string& word, int table_size)
{
    unsigned int hashVal = 0;
    for (char ch : word) hashVal = 37*hashVal + ch;
    return hashVal%table_size;
}
```

- Calculates a polynomial function
  by nested multiplication (Horner's rule).
- Easy and fast to calculate.
- Distributes the keys well into a large table.

Collisions

- The more data we put into a hash table, the more "collisions" occur.
- A collision is when two or more data records are mapped to the same table cell.
- How can a hash table handle collisions?

Keys for Successful Hashing

- Good hash function
- Good collision resolution
- Size of the underlying array a <u>prime number</u>

Collision Resolution

- Separate chaining
- Open addressing
  - Linear probing
  - Quadratic probing

Collision Resolution: Separate Chaining

- Each cell in a hash table is a pointer to a linked list of all the data records that hash to that entry.
- To retrieve a data record, we first hash to the cell.

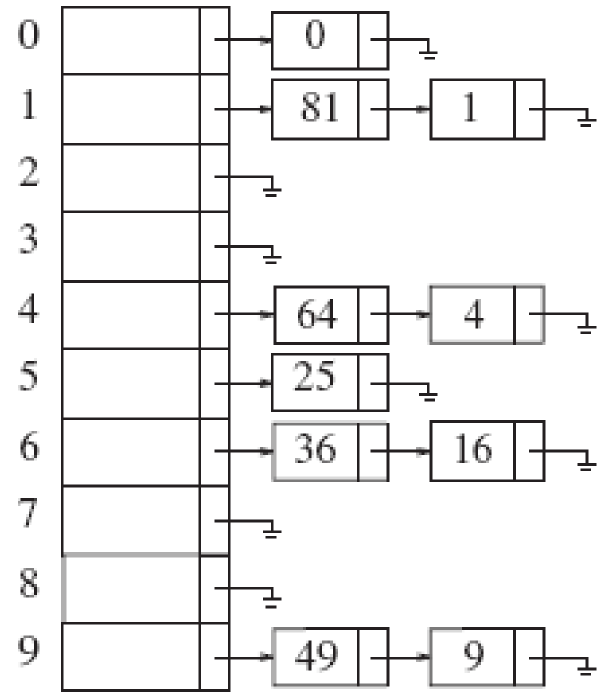


Figure 5.5 A separate chaining hash table

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012
ISBN 0-13-257627-9

Collision Resolution: Separate Chaining, *cont'd*

- Then we search the associated linked list for the data record.
- We can sort the linked lists to improve search performance.

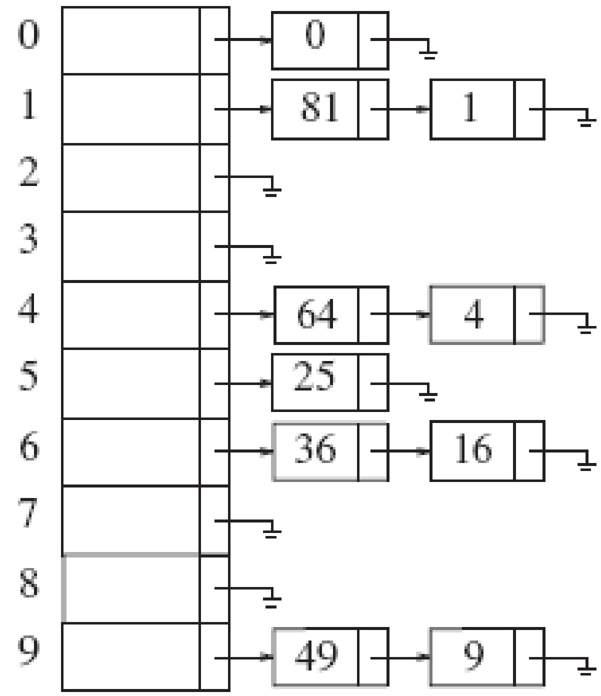


Figure 5.5 A separate chaining hash table

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012
ISBN 0-13-257627-9

Collision Resolution: Open Addressing

- Does not use linked lists.
- All the data resides in the table.
- When a collision occurs,
  try a different table cell.
- We will consider two types of open addressing:
  - linear probing
  - quadratic probing

Collision Resolution: Linear Probing

- Try in succession $h_0(x), h_1(x), h_2(x), \ldots$
- $h_i(x) = (hash(x) + f(i)) \bmod TableSize$, with $f(0) = 0$
  - $hash(x)$ produces the home cell.
- Function f is the collision resolution strategy.
- With linear probing, f is a linear function of i, typically, $f(i) = i$

Collision Resolution: Linear Probing, *cont'd*

- Insertion
  - If a cell is filled, look for the next empty cell.
- Search
  - Start searching at the home cell, keep looking at the next cell until you find the matching key is found.
  - If you encounter an <u>empty cell</u>, there is no key match.
- Deletion
  - Empty cells will prematurely terminate a search.
  - Leave deleted items in the hash table but <u>mark them as deleted</u>.

Collision Resolution: Linear Probing, *cont'd*

- Suppose *TableSize* is 10, the keys are integer values, and the hash function is the key value modulo 10.
  - We want to insert keys 89, 18, 49, 58, and 69.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.11 Hash table with linear probing, after each insertion

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012
ISBN 0-13-257627-9

Collision Resolution: Quadratic Probing

- Linear probing causes primary clustering.
- Try quadratic probing instead: $f(i) = i^2$.

49 collides with 89: the next empty cell is 1 away.

58 collides with 18: the next cell is filled. Try $2^2 = 4$ cells away from the home cell.

Same for 69.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure 5.13 Hash table with quadratic probing, after each insertion

Data Structures and Algorithms in Java, 3rd ed.
by Mark Allen Weiss
Pearson Education, Inc., 2012
ISBN 0-13-257627-9

Collision Resolution: Quadratic Probing, *cont'd*

- Try quadratic probing instead: $f(i) = i^2$.

- i^2 is easy to compute, for i = 0, 1, 2, ...
  - Remember that we proved that

$1 = 1^2$
$1 + 3 = 4 = 2^2$
$1 + 3 + 5 = 9 = 3^2$
$1 + 3 + 5 + 7 = 16 = 4^2$
…

Load Factor

- The load factor λ of a hash table is the ratio of the number of elements in the table to the table size.
  - λ is much more important than table size.
- For probing collision resolution strategies, it is important to keep λ under 0.5.
  - Don't let the table become more than half full.
- If quadratic probing is used and the table size is a prime number, then a new element can always be inserted if the table is at most half full.

Assignment #11

- Read a copy of the U.S. Constitution and its amendments, and build a concordance table.
  - Concordance: An alphabetical list of words in a text, each word with the number of times it appears.
- Maintain the concordance in both a <u>sorted STL vector</u> and in an <u>STL map</u> (hash table).
- Compare the timings of the vector and the map:
  - Insertion of words
  - Searching for words