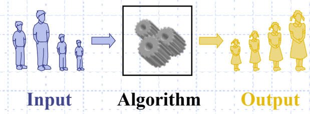
4. Algorithm Analysis

- Analysis of Algorithms and Data Structures
- Calculate the execution costs for non-recursive algorithms
- The big-Oh notation.
- Reasonable vs. Unreasonable Algorithms

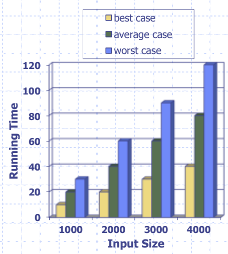
Analysis of Algorithms

- O() Analysis of Algorithms and Data Structures
- Reasonable vs. Unreasonable Algorithms
- Using O() Analysis in Design



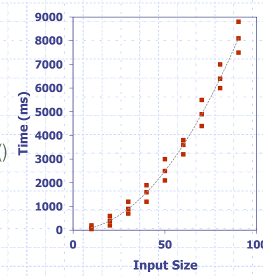
Running Time

- The running time of an algorithm varies with the input and typically grows with the input size
- Average case difficult to determine
- We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a function like ftime() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult.
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used.

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

- High-level description of an algorithm
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues
- A language that is made up for expressing algorithms.
- Looks like English combined with C, Pascal, whatever suites you…

Example: find the max integer in an array

```
function arrayMax(A, n)
  Input: int A[n]
  Output: maximum element of A

  int currentMax = A[0];
  for (i = 1; i < n; i++){
    if (A[i] > currentMax) {
      currentMax = A[i]
    }
  }
  return currentMax
```


Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** …
  - **do** …
  - **for** …
  - Indentation and braces
- Function declaration

  Function *fname* (*arg* [, *arg*…])
  Input:…
  Output:…
  body
- function call

  fname(*arg* [, *arg*…])
- Return value

  return *expression*
- Expressions(C-like)
- Or

  n^2 Superscripts and other mathematical formatting allowed

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent of the programming language
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Comparison
  - Calling a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
function arrayMax(A, n)
  currentMax = A[0];
  for (i = 1; i < = n-1; i++){
      if (A[i] > currentMax) {
          currentMax = A[i];
      }
  }
  return currentMax;
```


Analysis of nonrecursive algorithms

- Identify the input size n
- Identify the primitive operation
- Set up a sum for the number of the times the primitive operation is executed
- Simplify the sum to generate a function of n

Example: Set up a sum

Primitive operation: comparison

The sum: $\sum_{i=1}^{n-1} 1$, lower limit: initial loop condition

upper limit: terminating condition

one comparison each iteration

```
function arrayMax(A, n)
  currentMax = A[0];
  for (i = 1; i < = n-1; i++){
      if (A[i] > currentMax) {
          currentMax = A[i];
      }
  }
  return currentMax;
```

Example: Simplify a sum

- $\sum_{i=l}^{u} 1 = 1 + 1 + 1 + \cdots + 1 = u - l + 1$
- $\sum_{i=1}^{n-1} 1 = n - 1 - 1 + 1 = n - 1$
- We thus have the number of comparison: n-1

```
function arrayMax(A, n)
    currentMax = A[0];
    for (i = 1; i < = n-1; i++){
        if (A[i] > currentMax) {
            currentMax = A[i];
        }
    }
    return currentMax;
```


Useful summation formulas and rules

$\Sigma_{l \le i \le u} 1 = 1+1+\ldots+1 = u - l + 1$

In particular, $\Sigma_{1 \le I \le u} 1 = n - 1 + 1 = n$

$\Sigma_{1 \le i \le n}\, i = 1+2+\ldots+n = n(n+1)/2 \approx n^2/2$

$\Sigma_{1 \le i \le n}\, i^2 = 1^2+2^2+\ldots+n^2 = n(n+1)(2n+1)/6 \approx n^3/3$

$\Sigma_{0 \le i \le n}\, a^i = 1 + a + \ldots + a^n = (a^{n+1} - 1)/(a - 1)$ for $a \ne 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \ldots + 2^n = 2^{n+1} - 1$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \quad \Sigma c a_i = c\Sigma a_i \quad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

Example:

- Primitive operation: comparison
- Set up the sum: $f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

This is a nested sum. The outer (left) sum is for the outer loop, the inner (right) sum is for the inner loop. They should be simplified from inner to outer.

```
function uniqueElement(A, n)
    for (i = 0; i <= n-2; i++){
        for (j = i+1; j <= n-1; j++){
            if (A[i] == A[j]) {
                return false;
        }
    }
    return true;
```


Example: simplify the sum

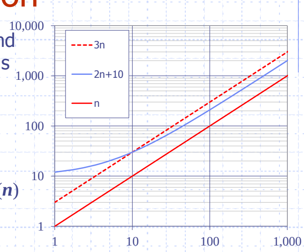
$$f(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$
$$= \sum_{i=0}^{n-2} (n - 1 - i - 1 + 1) = \sum_{i=0}^{n-2} (n - 1 - i)$$
$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i$$
$$= (n-1) \sum_{i=0}^{n-2} 1 - (n-2)(n-1)/2$$
$$= (n-1)^2 - (n-2)(n-1)/2 = (n-1)n/2 \approx n^2/2$$

```
function uniqueElement(A, n)
    for (i = 0; i <= n-2; i++){
        for (j = i+1; j <= n-1; j++){
            if (A[i] == A[j]) {
                return false;
        }
    }
    return true;
```


Big-Oh Notation

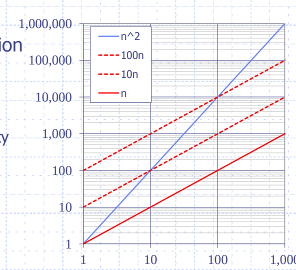
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \le cg(n)$ for $n \ge n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big-Oh Notation (cont.)

- Example: the function n^2 is not $O(n)$
  - $n^2 \le cn$
  - $n \le c$
  - The above inequality cannot be satisfied since c must be a constant



Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Big-Oh Algorithm Analysis

- The analysis of an algorithm determines the running time in big-Oh notation
- To perform the analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determine that algorithm ***arrayMax*** executes at most $7n - 1$ primitive operations
  - We say that algorithm ***arrayMax*** "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

Traversals

- Traversals involve visiting every node in a collection of size *n*.
- Because we must visit every node, a traversal must be O(*n*) for any data structure.
  - If we visit less than *n* elements, then it is not a traversal.
  - If we have to process every node during traversal, then O(process)*O(*n*)

Searching for an Element

Searching involves determining if an element is a member of the collection.

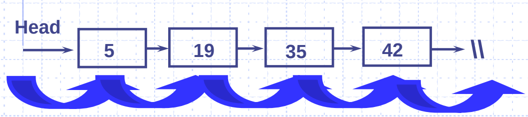
- Simple/Linear Search:
  - If there is no ordering in the data structure
  - If the ordering is not applicable
- Binary Search:
  - If the data is ordered or sorted
  - Requires non-linear access to the elements

Simple Search

- Worst case: the element to be found is the n^{th} element examined, or an unsuccessful search
- Simple search must be used for:
  - Sorted or unsorted linked lists
  - Unsorted array
  - Binary tree (to be discussed)

Example: Linked List

- Let's determine if the value 83 is in the collection:



83 Not Found!

Big-O of Simple Search

- The algorithm has to examine every element in the collection
  - To return a false
  - If the element to be found is the n^{th} element
- Thus, simple search is O(n).

Binary Search

- We may perform binary search on
  - Sorted arrays
  - Binary search trees
- Tosses out ½ the elements at each comparison.

Binary Search Example

7	12	42	59	71	86	104	212

Looking for 89

Binary Search Example

~~7~~	~~12~~	~~42~~	~~59~~	71	86	104	212

Looking for 89

Binary Search Example

~~7~~	~~12~~	~~42~~	~~59~~	~~71~~	~~86~~	104	212

Looking for 89

Binary Search Example

~~7~~	~~12~~	~~42~~	~~59~~	~~71~~	~~86~~	~~104~~	~~212~~

89 not found – 3 comparisons

3 = Log(8)

Binary Search Big-O

- An element can be found by comparing and cutting the work in half.
  - We cut work in ½ each time
  - How many times can we cut in half?
  - $\text{Log}_2 n$
- Thus binary search is O(Log *n*).

Recall

$$\log_2 n = k \bullet \log_{10} n$$
$$k = 0.30103...$$

So: $O(\lg n) = O(\log n)$

In general:

$O(C*f(n)) = O(f(n))$

if C is a constant

Insertion

- Inserting an element requires two steps:
  - Find the right location
  - Perform the instructions to insert
- If the data structure in question is unsorted, then it is O(1)
  - Simply insert to the front in the case of a linked list
  - Simply insert to end in the case of an array
  - There is no work to find the right spot and only constant work to actually insert.

Insert into a Sorted Linked List

Finding the right spot is O(*n*)
- Recurse/iterate until found

Performing the insertion is O(1)

Total work is O(*n* + 1) = O(*n*)

Inserting into a Sorted Array

Finding the right spot is O(Log *n*)
- Binary search on the element to insert

Performing the insertion
- Shuffle the existing elements to make room for the new item

Shuffling Elements

Note – we must have at least one empty cell

5	12	35	77	101	

↑

Insert 29

Big-O of Shuffle

Worst case: inserting the smallest number

5	12	35	77	101	

Would require moving N elements…
Thus shuffle is O(n)

Big-O of Inserting into Sorted Array

Finding the right spot is O(Log n)

Performing the insertion (shuffle) is O(n)

Sequential steps, so add:
Total work is O(Log n + n) = O(n)

Two Sorting Algorithms

- Bubble-sort O(n^2)
  - Brute-force method of sorting
  - Loop inside of a loop
- Merge-sort O(n log$n$)
  - Divide and conquer approach
  - Recursively call, splitting in half
  - Merge sorted halves together

Bubble-sort Review

Bubble-sort works by comparing and swapping values in a list

1	2	3	4	5	6
77	42	35	12	101	5

Bubble-sort Review

Bubble-sort works by comparing and swapping values in a list

1	2	3	4	5	6
42	35	12	77	5	$\boxed{101}$

Largest value correctly placed


```
void bubbleSort(int a[], int n){
int temp;
  for (int i = 1; i < n; i++){
     for (int j = 0; j < n -1; j++){
          if (a[j] > a[j+1]){
               temp = a[j];
               a[j] = a[j+1];
               a[j+1] = temp;
          }
     }
  }
```

N-1

to_do

Analysis of Bubblesort

- Step 1. ?
- Step 2. ?
- Step 3. ?
- Step 4. ?

Bubblesort Complexity

The complexity is:

O(?)

O(n^2) Runtime Example

Assume you are sorting 250,000,000 items:

n = 250,000,000

$n^2 = 6.25 \times 10^{16}$

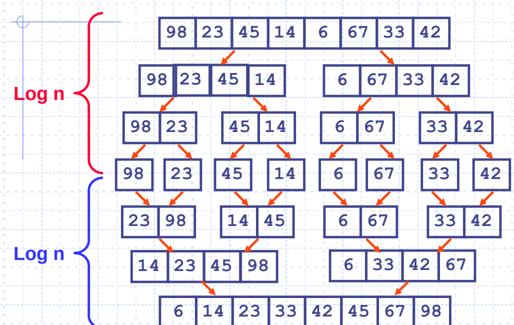
If you can do one operation per nanosecond (10^{-9} sec)

It will take 6.25×10^7 seconds

So $\dfrac{6.25 \times 10^7}{60 \times 60 \times 24 \times 365}$

= 1.98 years

Mergesort



Analysis of Mergesort

Phase I

- Divide the list of n numbers into two lists of $n/2$ numbers
- Divide those lists in half until each list is size 1

Log n steps for this stage.

Phase II

- Build sorted lists from the decomposed lists
- Merge pairs of lists, doubling the size of the sorted lists each time

Log n steps for this stage.

Mergesort Complexity

Each of the n numerical values is compared or copied during each pass

- The total work for each pass is O(n).
- There are a total of Log n passes

Therefore the complexity is:

$$O(\underbrace{\text{Log } n}_{\text{Break apart}} + \underbrace{n * \text{Log } n}_{\text{Merging}}) = O(n * \text{Log } n)$$

$O(n \text{ Log} n)$ Runtime Example

```
Assume same 250,000,000 items
n*Log(n) = 250,000,000 x 8.3
         = 2, 099, 485, 002

With the same processor as before

             2 seconds
```


Reasonable vs. Unreasonable

Reasonable algorithms have polynomial factors
- O (Log n)
- O (n)
- O (n^K) where K is a constant

Unreasonable algorithms have exponential factors
- O (2^n)
- O (n !)
- O (n^n)

Algorithmic Performance Thus Far

- Some examples thus far:
  - O(1) Insert to front of linked list
  - O(n) Simple/Linear Search
  - O(n Log n) MergeSort
  - O(n^2) BubbleSort
- But it could get worse:
  - O(n^5), O(n^{2000}), etc.

An O(n^5) Example

For n = 256
$n^5 = 256^5$ = 1,100,000,000,000

If we had a computer that could execute a million instructions per second…

- 1,100,000 seconds = 12.7 days to complete

But it could get worse…

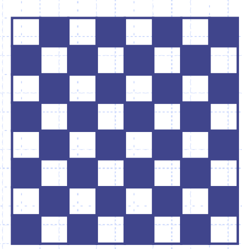
The Power of Exponents

It is hard to understand the basic principles behind exponential growth. Perhaps it is easier to understand in terms of doubling time. In exponential growth, each time a value doubles the new value is greater than all previous values combined.

Consider the story of the peasant that did a great favor for a king. The king asked how he could repay the peasant. In response, the peasant asked the king to place two pieces of grain on a square of a chess board, and double the amount of grain on each following square (2 on the first, 4 on the second, 8 on the third, 16 on the fourth, and so on). "Sure," says the king thinking that would not require much grain. However, the king does not understand exponential growth.

The Power of Exponents

A rich king and a wise peasant…



The King has to Pay

Square(n)	Pieces of Grain
1	2
2	4
3	8
4	16
...	
63	9,223,000,000,000,000,000
64	18,450,000,000,000,000,000

2^n

How Bad is 2^n ?

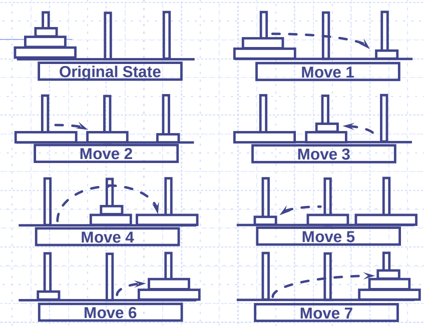
- Imagine being able to grow a billion (1,000,000,000) pieces of grain a second…

?

- It would take
  - 585 years to grow enough grain just for the 64th chess board square
  - Over a thousand years to fulfill the peasant's request!

So the King cut off the peasant's head.

Towers of Hanoi: Solution



Towers of Hanoi - Complexity

For 3 rings we have 7 operations.

In general, the cost is

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

Each time we increment n, we double the amount of work.

This grows incredibly fast!

Towers of Hanoi (2^n) Runtime

For $n = 64$
$2^n = 2^{64} = 18{,}450{,}000{,}000{,}000{,}000{,}000$

If we had a computer that could execute a million instructions per second…

- It would take 584,000 years to complete

Where Does this Leave Us?

- Clearly algorithms have varying runtimes.
- We'd like a way to categorize them:
  - Reasonable, so it may be useful
  - Unreasonable, so why bother running

Performance Categories of Algorithms

Polynomial	Sub-linear	O(Log n)
	Linear	O(n)
	Nearly linear	O(n Log n)
	Quadratic	O(n^2)
	Exponential	O(2^n)
		O($n!$)
		O(n^n)

Reasonable vs. Unreasonable

Reasonable algorithms have polynomial factors
- O (Log n)
- O (n)
- O (n^K) where K is a constant

Unreasonable algorithms have exponential factors
- O (2^n)
- O ($n!$)
- O (n^n)

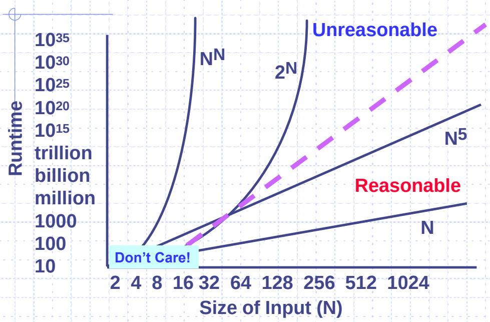
Reasonable vs. Unreasonable

Reasonable algorithms
- May be usable depending upon the input size

Unreasonable algorithms
- Are impractical and useful to theorists
- Demonstrate need for approximate solutions

Remember we're dealing with large n (input size)

Two Categories of Algorithms



Properties of the O notation

- Constant factors may be ignored
  - $\forall\ k > 0,\ kf$ is $O(f)$
- Fastest growing term dominates a sum
  - f is $O(g)$ and h is $O(r)$ then $f + h$ is $O(max(g, r))$

  eg $an^4 + \log n$ is $O(max(n^4, \log n)) \rightarrow O(n^4)$
- Polynomial's growth rate is determined by leading term
  - If f is a polynomial of degree d, then f is $O(n^d)$

  eg $10n^4 + 5n^6 + n^2$ is $O(n^6)$

Properties of the O notation

- f is $O(g)$ is transitive
  - If f is $O(g)$ and g is $O(h)$ then f is $O(h)$
- Product of upper bounds is upper bound for the product
  - If f is $O(g)$ and h is $O(r)$ then $f*h$ is $O(g*r)$
- All logarithms grow at the same rate
  - $\log_b n$ is $O(\log_d n)$ $\forall$ $b, d > 1$

Simple Examples:

- Simple statement sequence
  s_1; s_2; .... ; s_k
  - $O(1)$ as long as k is constant
- Simple loops
  for(i=0;i<n;i++) { s; }
  where s is $O(1)$
  - Time complexity is $n * O(1)$ or $O(n)$
- Nested loops
  for(i=0;i<n;i++)
  for(j=0;j<n;j++) { s; }
  - Complexity is $O(n^2)$