Course Introduction

- Why Algorithms
  - Essential for design and development
  - Impact
  - Problem solving skills
- Content: D*esign and analysis of algorithms*
  - Brute force, exhaustive search, and graph algorithms
  - Decrease, Transform, Divide and Conquer
  - Greedy algorithm design paradigm
  - Dynamic programming algorithm design paradigm
  - Iterative improvement
  - NP-complete problems
  - Approximation Algorithms

Goals of the Course

- Develop basic understanding of design approaches
  - Brute Force, Greedy, Divide and Conquer, Dynamic Programming, etc.
- Master basic algorithms for important problems in computing for a range of problem domains
- Problem solving skills. Be able to:
  - Apply the design approaches in unfamiliar contexts
  - Use small examples, find counterexamples
  - Think both top-down and bottom-up
- Analysis of time complexity of algorithms
  - Asymptotic analysis: $O()$, $\Omega()$, $\Theta()$
- Proving correctness of algorithms
- P, NP, NP complete and their implications for algorithm design

Analysis of algorithms

- Issues:
  - **correctness**
  - **time efficiency**
  - space efficiency
  - **optimality**
- Approaches to efficiency:
  - **theoretical analysis**
  - empirical analysis

- Can we simplify theoretical analysis so that it is a usable tool (most of the time)? Yes! Asymptotic Analysis!

Theoretical analysis of time efficiency

- Time efficiency: the number of instructions as a **function of input size** using the random access machine model or "basic operation" approach.

- Input size: amount of memory necessary to store the problem's basic data or length of the data to be read to define the problem

- Random access machine model or "basic operation" approach.
  - Goal: to have a metric that is independent of the hardware and software technology.
  - Random access machine model: count instructions of an idealized computer
  - "basic operation" approach: count specific type of instructions that are the determining factor in how long a computation will take

An algorithm's efficiency may depend on specific input

C(n): the number of instructions for an input of size n

- Worst case: C_{worst} (n) – max for inputs of size n
- Best case: C_{best}(n) – min for inputs of size n
- Average case: C_{avg}(n) – "average" for inputs of size n

  (The expected value of the number of instructions given some assumption about the probability distribution of all possible inputs or behavior of a randomized algorithm.)

Analysis of Efficiency: Asymptotic Efficiency

- Pay little attention to smaller factors
  - Small instruction time differences
  - Constant factors - note: These may be important, especially when different algorithms have the same order of growth
  - Lower order terms
- Asymptotic efficiency – tells us what happens for large input sizes which is what we care about since when the input is small performance is generally not an issue
- Generally looking for a worst case guarantee
  - Usually make the least assumptions
  - Easiest to analyze in most cases

General Plan for Analysis of algorithms

- Decide on parameter *measure for input size*
- Identify algorithm's *basic operation*
- Does the number of times the basic operation execute depend on the input? Determine which of *worst*, *average*, and *best* cases needs to be used to analyze the efficiency of the algorithm
- Set up a **formula** for the number of times the basic operation is executed, generally
  - a **summation** for non- recursive algorithms
  - a **recurrence relation** with an appropriate initial condition for recursive algorithms
- If possible, develop a closed form solution.
  - Simplify the sum using standard formulas and rules (see Appendix A)
  - Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method. (see Appendix B)

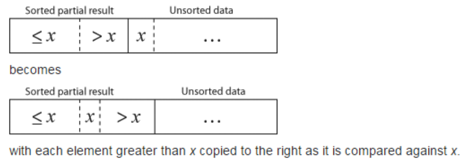
Analysis of Selection Sort
(Replace inner loop with something more efficient??)

ALGORITHM *SelectionSort*$(A[0..n-1])$
  //Sorts a given array by selection sort
  //Input: An array $A[0..n-1]$ of orderable elements
  //Output: Array $A[0..n-1]$ sorted in ascending order
  for $i \leftarrow 0$ **to** $n-2$ **do**
    $min \leftarrow i$
    for $j \leftarrow i+1$ **to** $n-1$ **do**
      if $A[j] < A[min]$ $min \leftarrow j$
    swap $A[i]$ and $A[min]$

- Analysis
  - Input size – number of entries to be sorted
  - Basic operation – comparison
  - Best case? Worst case?
  - Closed form solution

Insertion Sort: sort in non-decreasing order

```
for i = 1 to n-1
   j ← i
   while j > 0 and A[j-1] > A[j]
      swap A[j] and A[j-1]
      j ← j – 1
   end while
end for
```



Correctness: loop invariant

- Analysis
  - Input size – number of entries to be sorted
  - Basic operation – comparison
  - Best case? Worst case?
  - Closed form solution

Types of formulas for counting operations

- Exact formula

  e.g., $C(n) = n(n-1)/2$

- Formula indicating order of growth with specific multiplicative constant

  e.g., $C(n) \approx 0.5\ n^2$

- Formula indicating order of growth with unknown multiplicative constant: Asymptotic Complexity

  e.g., $C(n) \approx cn^2$ $O(g(n))$, $\theta(g(n))$, $\Omega(g(n))$

Asymptotic Order of Growth

- Idea: Order of growth within a constant multiple as $n \to \infty$
- Simple question this can help us answer:
  - How much faster will algorithm run on computer that is twice as fast?

  vs.
  - How much longer does it take to solve problem of double input size?

Why order of growth matters!

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Algorithms and their specification

- An algorithm is a finite sequence of unambiguous instructions for solving a problem in a finite amount of time.
- Pseudo Code: Program = Algorithm + Data Structure
  - if – then – else – case/switch
  - for ( ) – while( )
  - call – return
  - Data structure's - e.g. Arrays
  - Comments and white space to show flow of control
- Goal of Pseudo Code : Rigorous but without overhead of programming language syntax – Communicate to human exactly what must be done

Lab 1, Week 1

- Interval Scheduling: Find the largest compatible subset of jobs to run on a processor
  - Greedy Algorithm: Earliest Finish Time First
  - The fact this produces an optimal solution can be proved using a "stay ahead" argument. We will cover this in more detail when we cover Greedy algorithms
- Euclid's algorithm:
  - This is an example of a reduce and conquer algorithm where the size of the problem is reduced at each stage of the algorithm
  - To determine the optimal strategy of Euclid's game, you must *guess* at the pattern that you see from examining small cases. The solution will be posted on PolyLearn

Asymptotic Complexity Classes: O(g(n), Ω(g(n), θ(g(n)

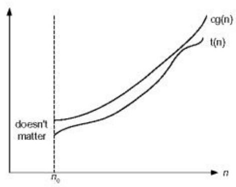


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

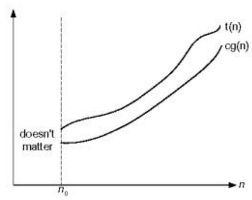


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

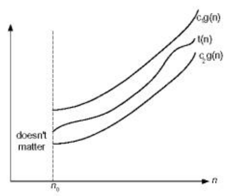


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Formal Definitions: Asymptotic Order of Growth

- Upper bounds. T(n) is O(f(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.
- Lower bounds. T(n) is Ω(f(n)) if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.
- Tight bounds. T(n) is Θ(f(n)) if T(n) is both O(f(n)) and Ω(f(n)).
- Ex: $T(n) = 32n^2 + 17n + 32$.
  - T(n) is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
  - T(n) is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.

Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n)) \leftrightarrow g(n) \in \Omega(f(n))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$ then $f(n) \in O(h(n))$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

Orders of growth of some important functions

- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is
- All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$
- Exponential functions a^n have different orders of growth for different a's, e.g. 3^n grows faster then 2^n
- order $\log n$ < order n^α $(\alpha>0)$ < order a^n < order $n!$ < order n^n

Notation

- Slight abuse of notation. $T(n) = O(f(n))$.
  - Asymmetric:
    - » $f(n) = 5n^3$; $g(n) = 3n^2$
    - » $f(n) \in O(n^3)$ and $g(n) \in O(n^3)$
    - » avoid writing $f(n) = O(n^3)$ and $g(n) = O(n^3)$ since this would imply that from an order of growth perspective that $f(n) = g(n)$
    - » but $f(n) \neq g(n)$ from an order of growth viewpoint.
  - Thus we use the notation: $T(n) \in O(f(n))$.
- Meaningless statement. Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.
  - Statement doesn't "type-check."
  - Use Ω for lower bounds.

Important Orders of Growth: $\theta(n)$

- Linear : $\theta(n)$ merging 2 lists
- Log-linear : $\theta(n \log n)$ Merge sort
- Quadratic : $\theta(n^2)$ counting pairs
- Cubic : $\theta(n^3)$ counting triples, matrix multiply
- Exponential : $\theta(2^n)$ subsets
- Factorial : $\theta(n!)$ permutations

Analyzing Algorithms

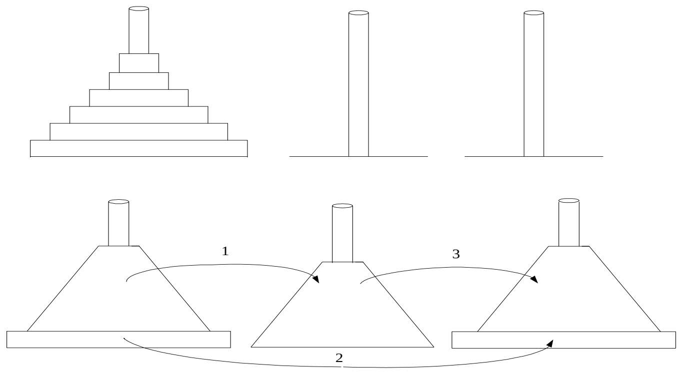
1. Decide on a parameter indicating an input's size.
2. What will be counted: basic operation / instructions
3. Will the number of instructions executed vary on different inputs of the same size? (If so, the worst, average, and best cases must be investigated separately.)
4. Set up a summation (iteration) or recurrence relation (recursion) expressing the number of instructions executed
5. Solve the recurrence or simplify the summation to get a closed form solution:
   - standard formulas and rules for summations
   - solve recurrence relations

Useful summation formulas and rules

- $1+1+\cdots+1 = n \in \Theta(n)$
- $1+2+\cdots+n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$
- $1 + a + \cdots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

  In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

The Tower of Hanoi Puzzle



Recurrence for number of moves:

Solving recurrence for number of moves

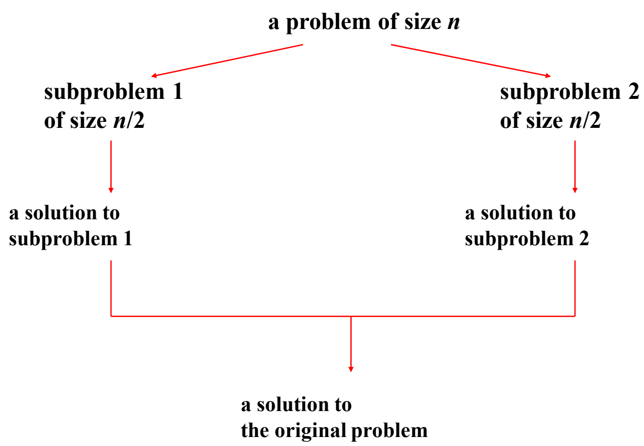
- $M(n) = 2M(n-1) + 1$, $M(1) = 1$

Tree of calls for the Tower of Hanoi Puzzle

Divide-and-Conquer

- The most-well known algorithm design strategy:
- Divide instance of problem into two or more smaller instances
- Solve smaller instances recursively (can implement iteratively)
- Obtain solution to original (larger) instance by combining these solutions

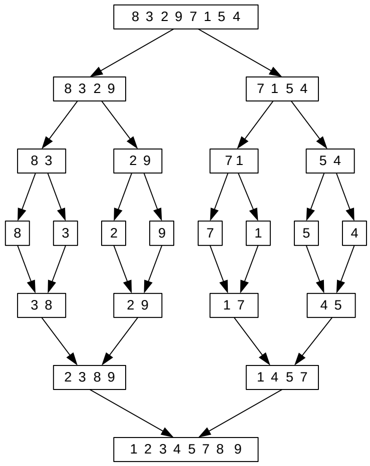
Divide-and-Conquer Technique



Mergesort

- Split array A[0..n-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one of the arrays:
    - » compare the first elements in the remaining unprocessed portions of the arrays
    - » copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other array into A.

Mergesort Example



Pseudocode of Mergesort

ALGORITHM *Mergesort*$(A[0..n-1])$
  //Sorts array $A[0..n-1]$ by recursive mergesort
  //Input: An array $A[0..n-1]$ of orderable elements
  //Output: Array $A[0..n-1]$ sorted in nondecreasing order
  if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

Pseudocode of Merge

ALGORITHM $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$

```
//Merges two sorted arrays into one sorted array
//Input: Arrays B[0..p − 1] and C[0..q − 1] both sorted
//Output: Sorted array A[0..p + q − 1] of the elements of B and C
i ← 0; j ← 0; k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]; i ← i + 1
    else A[k] ← C[j]; j ← j + 1
    k ← k + 1
if i = p
    copy C[j..q − 1] to A[k..p + q − 1]
else copy B[i..p − 1] to A[k..p + q − 1]
```


Analysis of Mergesort: Recursion Tree Method, Back Substitution

Analysis of Mergesort

- All cases have same efficiency: Θ(n log n)
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting:
  - $\lceil \log 2\ n! \rceil \approx$ n log2 n - 1.44n
- Space requirement: Θ(n) (not in-place)
- Can be implemented without recursion (bottom-up)

Brute Force

A straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved

Simple Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing n!
3. Multiplying two matrices
4. Searching for a given value in a list
5. Naïve sorting algorithms
   e.g. Selection Sort

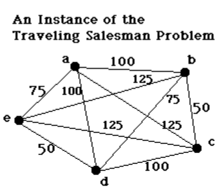
Exhaustive Search

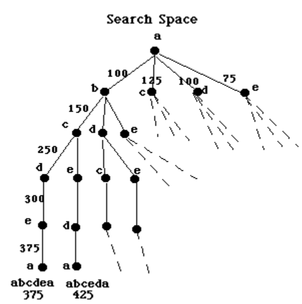
- A brute force solution to a problem involving search for an element with a special property.

- Method:
  - Generate a list of all potential solutions to the problem in a systematic manner.
  - Evaluate potential solutions one by one disqualifying infeasible ones and, for an optimization problem, keeping track of the best one found so far.
  - When search ends, announce the solution(s) found.

Example 1: Traveling Salesman Problem

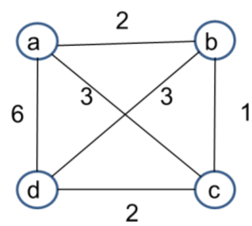
- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city





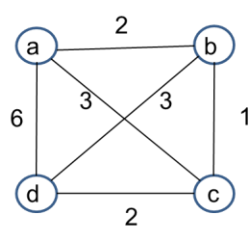
Example 1: Traveling Salesman Problem

- Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city
- Alternatively: Find shortest *Hamiltonian circuit* in a weighted connected graph
- Example:



Example 1: Traveling Salesman Problem

- Nearest Neighbor (greedy) does not guarantee an optimal solution!



TSP by Exhaustive Search

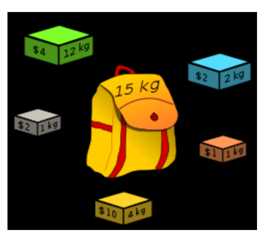
Tour	Cost
a→b→c→d→a	2+3+7+5 = 17
a→b→d→c→a	2+4+7+8 = 21
a→c→b→d→a	8+3+4+5 = 20
a→c→d→b→a	8+7+4+2 = 21
a→d→b→c→a	5+4+3+8 = 20
a→d→c→b→a	5+7+3+2 = 17

How many **distinct** tours? 5 cities? N cities?

Example 2: 0-1 Knapsack Problem

- Given n items:
  - weights: w1 w2 … wn
  - values: v1 v2 … vn
  - a knapsack of capacity W
- Find most valuable subset of the items that fit into the knapsack
- Example: Knapsack capacity W=16

item	weight	value
1	2	$20
2	5	$30
3	10	$50
4	5	$10



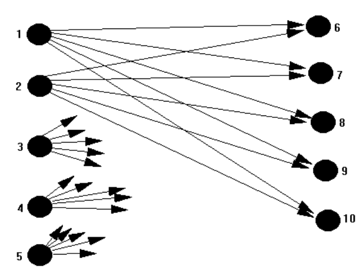
Knapsack Problem by Exhaustive Search

Subset	Total weight	Total value
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

Example 3: The Assignment Problem

There are n people who need to be assigned to n jobs, one person per job. The cost of assigning person i to job j is C[i,j]. Find an assignment that minimizes the total cost.

	Job 0	Job 1	Job 2	Job 3
Person 0	9	2	7	8
Person 1	6	4	3	7
Person 2	5	8	1	8
Person 3	7	6	9	4



How many assignments are there?

Assignment Problem by Exhaustive Search

	9	2	7	8
C =	6	4	3	7
	5	8	1	8
	7	6	9	4

Assignment (col.#s)	Total Cost
1, 2, 3, 4	9+4+1+4=18
1, 2, 4, 3	9+4+8+9=30
1, 3, 2, 4	9+3+8+4=24
1, 3, 4, 2	9+3+8+6=26
1, 4, 2, 3	9+7+8+9=33
1, 4, 3, 2	9+7+1+6=23
	etc.

Final Comments on Exhaustive Search

- Exhaustive-search algorithms run in a realistic amount of time only on very small instances
- In some cases, there are much better alternatives!
  - Euler circuits
  - shortest paths
  - minimum spanning tree
  - assignment problem
- In many cases, exhaustive search or its variation is the only known way to get exact solution