

Week 2 Lesson 1

Introduction to algorithms

Agenda

- Algorithm definition
- Introducing some real life problems
- Algorithm strategies

What is an Algorithm

- Algorithms are the ideas behind computer programs.
- An algorithm is the thing which stays the same whether the program is in C running on a core i7 in New York or is in Java running on a Macintosh in Kathmandu!
- To be interesting, an algorithm has to solve a general, specified problem.
 - An algorithmic problem is specified by describing the set of instances it must work on and what desired properties the output must have.

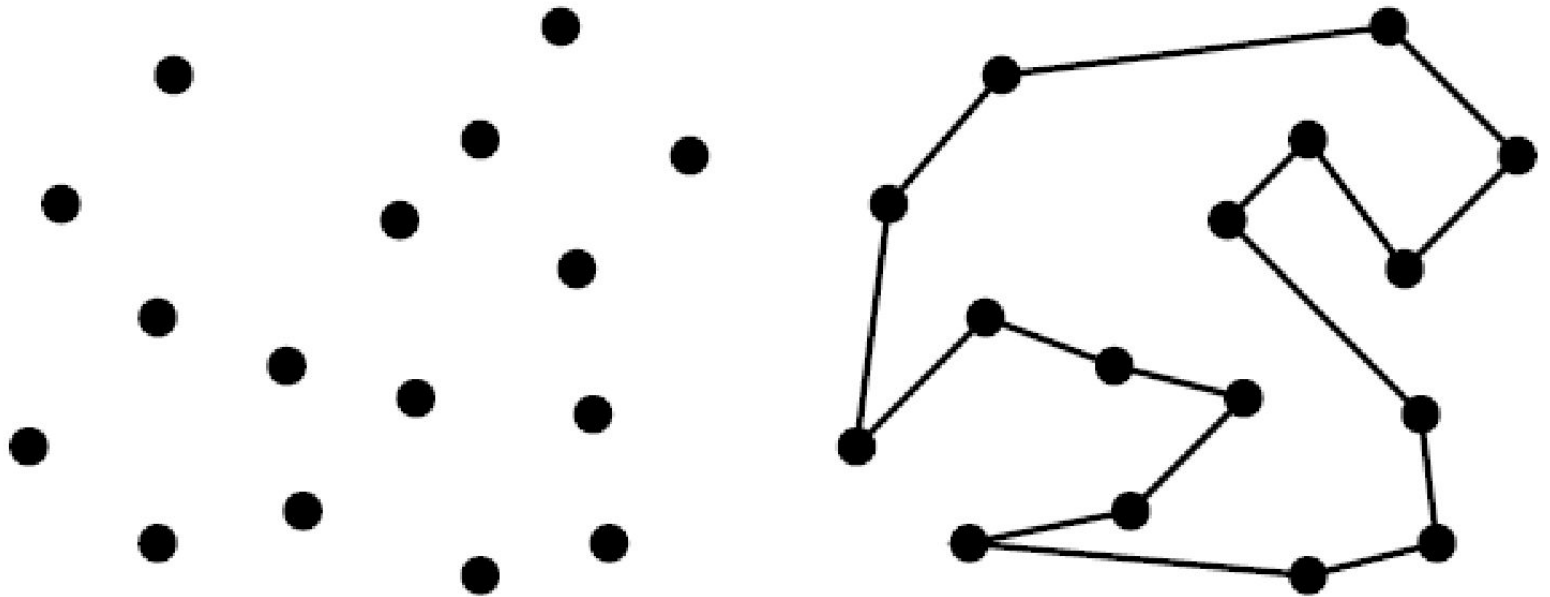
Example: Sorting

- Input: A sequence of N numbers $a_1 :: a_n$
- Output: the permutation (reordering) of the input sequence such as $a_1 < a_2 < \dots < a_n$.
- We seek algorithms which are *correct* and *efficient*.

Robot Tour Optimization

Suppose you have a robot arm equipped with a tool, say a soldering iron. To enable the robot arm to do a soldering job, we must construct an ordering of the contact points, so the robot visits (and solders) the points in order. We seek the order which minimizes the testing time (i.e. travel distance) it takes to assemble the circuit board.

Find the Shortest Robot Tour



Nearest Neighbor Tour

- A popular solution starts at some point p_0 and then walks to its nearest neighbor p_1 first, then repeats from p_1 , etc. until done.
- **Nearest Neighbor Tour is Wrong!**
- **A Correct Algorithm: Exhaustive Search**
- We could try all possible orderings of the points, then select the one which minimizes the total length

Efficiency: Why Not Use a Supercomputer?

- **Exhaustive Search is Slow!**
- Because it tries all $n!$ permutations, it is much too slow to use when there are more than 10-20 points.
- A faster algorithm running on a slower computer will *always* win for sufficiently large instances
- Usually, problems don't have to get that large before the faster algorithm wins

Selecting the Right Jobs

- A movie star wants to select the maximum number of starring roles such that no two jobs require his presence at the same time

Tarjan of the Jungle

The Four Volume Problem

The President's Algorist

Steiner's Tree

Process Terminated

Halting State

Programming Challenges

"Discrete" Mathematics

Calculated Bets

Earliest Job First

- Start working as soon as there is work available:

EarliestJobFirst(I)

Accept the earliest starting job j from I which does not overlap any previously accepted job, and repeat until no more such jobs remain.

Earliest Job First is Wrong!

- The first job might be so long (War and Peace) that it prevents us from taking any other job.



Shortest Job First

- Always take the shortest possible job, so you spend the least time working (and thus unavailable).

ShortestJobFirst(I)

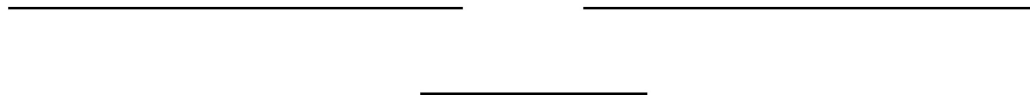
While (I) do

Accept the shortest possible job j from I .

Delete j , and intervals which intersect j from I .

Shortest Job First is Wrong!

- Taking the shortest job can prevent us from taking two longer jobs which barely overlap it.



First Job to Complete

- Take the job with the earliest completion date:

OptimalScheduling(I)

While (I) do

Accept job j with the earliest completion date.

Delete j , and whatever intersects j from I .

Algorithm Strategies

General Concepts

- **Algorithm strategy**
 - Approach to solving a problem
 - May combine several approaches
- **Algorithm structure**
 - Iterative \Rightarrow execute action in loop
 - Recursive \Rightarrow reapply action to subproblem(s)
- **Problem type**
 - Satisfying \Rightarrow find any satisfactory solution
 - Optimization \Rightarrow find **best** solutions (vs. cost metric)

Some Algorithm Strategies

- **Divide and conquer algorithms**
- **Dynamic programming algorithms**
- **Greedy algorithms**
- **Backtracking algorithms**
- **Branch and bound algorithms**
- **Heuristic algorithms**

Divide and Conquer

- **Based on dividing problem into subproblems**
- **Approach**
 1. **Divide problem into smaller subproblems**
 - **Subproblems must be of same type**
 - **Subproblems do not need to overlap**
 2. **Solve each subproblem recursively**
 3. **Combine solutions to solve original problem**
- **Usually contains two or more recursive calls**

Divide and Conquer – Examples

- **Binary Search**
- **Quicksort**
 - Partition array into two parts around pivot
 - Recursively quicksort each part of array
 - Concatenate solutions
- **Mergesort**
 - Partition array into two parts
 - Recursively mergesort each half
 - Merge two sorted arrays into single sorted array
- **Counting Inversion**

Dynamic Programming Algorithm

- **Based on remembering past results**
- **Approach**
 1. **Divide problem into smaller subproblems**
 - Subproblems must be of same type
 - Subproblems must **overlap**
 2. **Solve each subproblem recursively**
 - May simply look up solution
 3. **Combine solutions into to solve original problem**
 4. **Store solution to problem**
- **Generally applied to optimization problems**

Fibonacci Algorithm

- **Fibonacci numbers**
 - fibonacci(0) = 1
 - fibonacci(1) = 1
 - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
- **Recursive algorithm to calculate fibonacci(n)**
 - If n is 0 or 1, return 1
 - Else compute fibonacci(n-1) and fibonacci(n-2)
 - Return their sum
- **Simple algorithm \Rightarrow exponential time $O(2^n)$**

Dynamic Programming – Example

- **Dynamic programming version of fibonacci(n)**
 - **If n is 0 or 1, return 1**
 - **Else solve fibonacci(n-1) and fibonacci(n-2)**
 - **Look up value if previously computed**
 - **Else recursively compute**
 - **Find their sum and store**
 - **Return result**
- **Dynamic programming algorithm $\Rightarrow O(n)$ time**
 - **Since solving fibonacci(n-2) is just looking up value**

Dynamic Programming - Example

- **0-1 Knapsack**
- **Longest Common Subsequence**
- **Longest Increasing Sequence**
- **Sum of Subset**
- **Warshall's All pairs shortest path**
- **Bellman Ford's Single Source Shortest Path**
- **Matrix Chain Multiplication**

Greedy Algorithm

- Based on trying best current (local) choice
- Approach
 - At each step of algorithm choose best local solution
- Avoid backtracking, exponential time $O(2^n)$
- Hope local optimum lead to **global** optimum

Greedy Algorithm – Example

Kruskal's Minimal Spanning Tree Algorithm

sort edges by weight (from least to most)


tree = \emptyset

for each edge (X,Y) **in order**

if it does not create a cycle

add (X,Y) to tree

stop when tree has N-1 edges



**Picks best
local solution
at each step**

Greedy Algorithm - Example

- **Dijkstra's Single Source Shortest Path**
- **Minimum Spanning Tree – Prim & Kruskal**
- **Fractional Knapsack Problem**
- **Huffman Coding**

- **Thanks for patience hearing!**