Intro to Algorithms

Describe an algorithm in one word.

Recipe

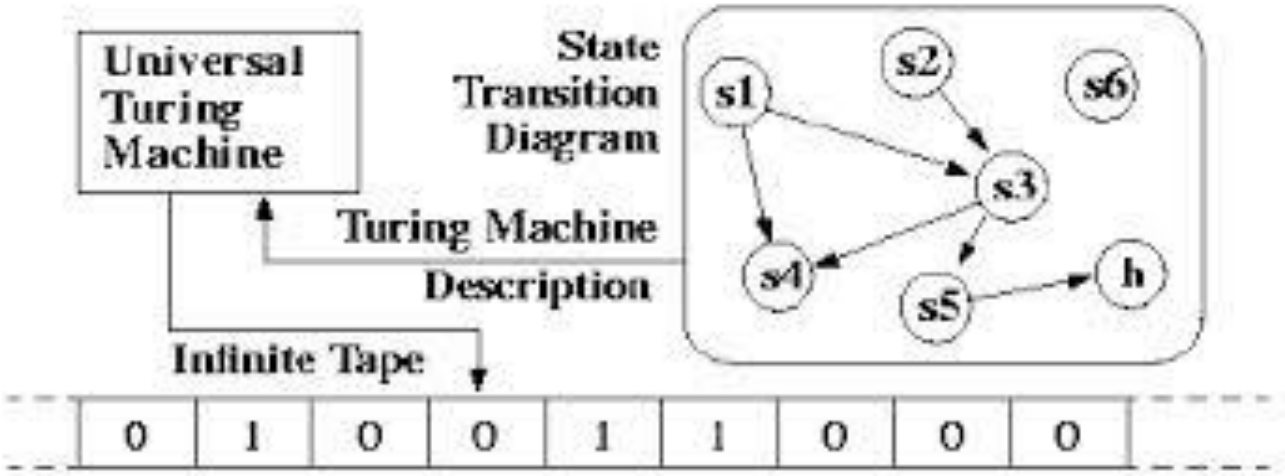
An algorithm is a **recipe** for solving a problem.

More Formal Definition

- An algorithm is a (finite) sequence of non-ambiguous steps for solving a problem.
- The definition of **non-ambiguous** depends on the context and audience. For example if you are describing an algorithm to another human, e.g., a student or professional in CS, a step can be described at a higher level and still be non-ambiguous. For example, sort the list.
- If talking to a computer then the algorithm is communicated via a program and steps need to be performed at a lower level, eventually at the machine language level.
- When we describe an algorithm in this course, most of the time we will use pseudocode, which does not need to be as rigorous as a high-level programming language like C++, Java, Python, just understandable to another human with expertise in programming. The pseudocode conventions we use are given in Appendix D of the textbook *Algorithms: Foundations and Design Strategies.* However, it is ok to deviate somewhat from these conventions as long as it is clear what is meant.

Formal Definition

The formal definition of an algorithm requires the concept of a Turing Machine.



For this course it will suffice to use the less formal definition.

Three Important Issues to Consider when Designing an Algorithm

- **Correctness** – use mathematical techniques like inductions, loop invariants.
- **Efficiency** – is the algorithm the fastest for solving the problem. Need to analyze its computing time.
- **Choice of Data Structure** (ADT) – the choice of data structure can effect the efficiency of algorithm.

Coin Changing

- Consider the problem of returning (correct) change using quarters, dimes, nickels, pennies.

- The everyday algorithm uses the greedy method: choose the most quarters, then the most dimes for remaining change, etc.
- We take for granted that it returns the fewest coins.

Everyday Coin Changing

Make change of C cents using **fewest** coins

Quarters = C / 25;

R = C – 25*Quarters;

Dimes = R / 10;

R = R – 10*Dimes;

Nickels = R / 5;

Pennies = R – 5*Nickels

Same Algorithm without Nickels

Quarters = C / 25;

R = C – 25*Quarters;

Dimes = R / 10;

Pennies = R – 10*Dimes;

PSN. Is this algorithm still correct, i.e., are fewest coins used?

(pause video to think about this)

So why does greedy method work when there are nickels?

- We know empirically that the greedy method for making change works from our everyday experience with receiving and making change.
- How to prove mathematically that it works?
- Apply **proof by contradiction.**

Proof by contradiction

Assume greedy method of making change does not involve the fewest coins. Now consider an optimal solution that makes the same change *C*, but uses the fewest coins.

Let g_{25}, g_{10}, g_5, g_1 be the number of quarters, dimes, nickels, pennies in the greedy solution.

Let p_{25}, p_{10}, p_5, p_1 be the number of quarters, dimes, nickels, pennies in the optimal solution.

Since the greedy and optimal solution make the same change *C* we have

$$C = 25 \times g_{25} + 10 \times g_{10} + 5 \times g_5 + g_1 = 25 \times p_{25} + 10 \times p_{10} + 5 \times p_5 + p_1$$

Then, based on our assumption that the greedy does not involve the fewest coins, we have

$$g_{25} + g_{10} + g_5 + g_1 > p_{25} + p_{10} + p_5 + p_1$$

Trick

The clever idea (trick) in getting a handle on the proof is to make some observations about the optimal solution.

PSN. Obtain upper bounds p_{10}, p_5, p_1

(pause video to think about this)

Hint

Could an optimal solution involve 3 or more dimes?

Could it involve 2 or more nickels?

Could it involve more than 5 pennies?

If it involves a nickel can it involve 2 or more dimes?

We have shown that $p_{10} \leq 2$, $p_5 \leq 1$, $p_1 \leq 4$ and if $p_5 = 1$, then $p_{10} \leq 1$.

First consider the case where there are no nickels in the optimal solution, i.e., $p_5 = 0$. Then the most change the optimal solution can make using only dimes, nickels and pennies, involves 2 dimes and 4 pennies for a total of 24¢.

Now consider the case where there is one nickel, i.e., $p_5 = 1$. Then the most change the optimal solution can make using only dimes, nickels and pennies, involves 1 dimes, 1 nickel and 4 pennies for a total of 19¢.

Number of quarters chosen by greedy and optimal solutions

Assume the optimal and greedy solution differ in the number of quarters chosen, i.e., $g_{25} \neq p_{25}$. By definition of the greedy method it chooses more quarters, i.e., $g_{25} > p_{25}$.

Since the greedy and optimal solution make the same amount of change C, i.e.,

$$C = 25 \times g_{25} + 10 \times g_{10} + 5 \times g_5 + g_1 = 25 \times p_{25} + 10 \times p_{10} + 5 \times p_5 + p_1$$

the optimal solution needs to make up the shortage of at least 25¢ using only dimes, nickels and quarters. But, this is impossible since we showed on the previous slide that the optimal solution can make change of at most 24¢ using only dimes, nickels and pennies. Since we have obtained a contradiction, we can conclude the that greedy and optimal choose the same number of quarters, i.e., $\boldsymbol{g_{25} = p_{25}}$.

We've shown that $g_{25} = p_{25}$

Now consider the remaining change R after using the quarters are used, i.e.,

$$R = C - 25 \times g_{25} = C - 25 \times p_{25} .$$

Using a similar argument, we can show that the greedy and optimal solutions use the same number of dimes. Otherwise, optimal solution is short at least one dime and can make at most 9 cents using 1 nickel and 4 pennies.

Updating remaining change after dimes are used, we can show they involve the same number of nickels. Otherwise, optimal solution is short at least one nickel and can only make at most 4 cents using pennies.

All that is left is pennies and since the greedy and optimal solutions make the same total change and we have shown they use the same number of quarters, dimes and nickels, they are forced to use the same number of pennies.

It follows that greedy and optimal involves exactly the same number of coins, which is a **contradiction** to assumption greedy does not involve the fewest coins.

We have obtained a contradiction to the assumption that the greedy method does not use the fewest number of coins in making change. Therefore, the opposite is true, i.e., the greedy method uses the fewest coins to make change. This completes our proof by contradiction.

General Coin-Changing Problem is hard

We have shown that the greedy method works for US denominations.

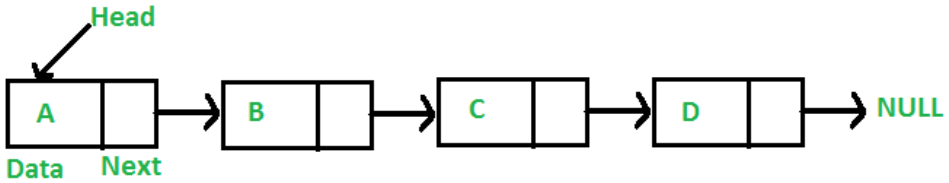
Surprisingly, the problem with general denominations is hard.

It has been shown to be NP-hard. We will discuss NP-complete and NP-hard later in this course.

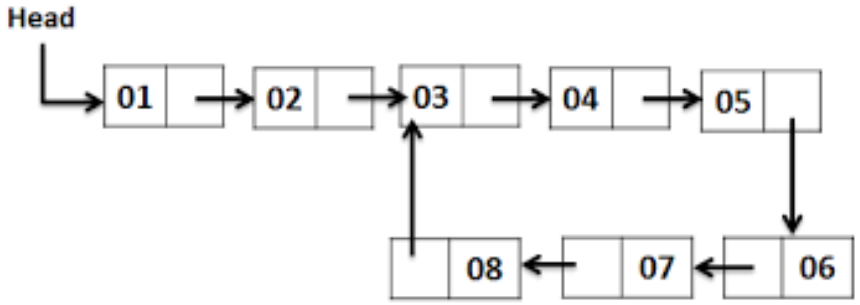
There is no known polynomial time algorithm in the worst case for solving the coin-changing problem for general denominations.

PSN: Broken Linked List

Give an efficient algorithm that uses a minimum of extra space for testing whether a linked list is “broken”, i.e., has a next pointer that points to previous node in the linked list, thereby forming a cycle.



Not broken



Broken

Turing's Halting Problem

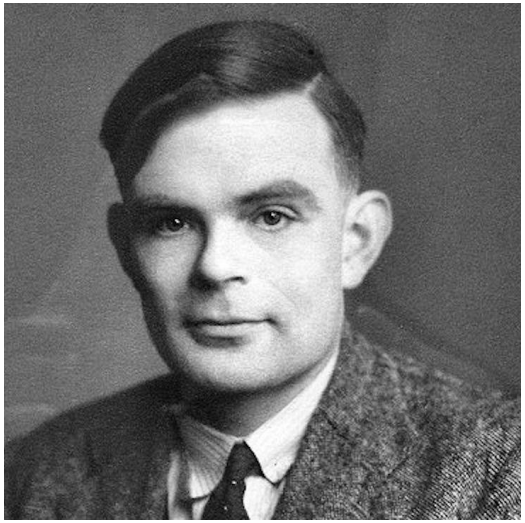
In computability theory, the **halting problem** is the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running,

or continue to run forever.

Alan Turing proved in 1936 that a general algorithm to solve the

halting problem for all possible

program-input pairs cannot exist.



Mathematical Tools

Induction

Recursion

Asymptotic Notation

Probability Theory

Graph Theory

Recursion

- Recursion is one of the most powerful design strategies for algorithms. It is applicable for the solution to those problems whose solution for a given input to a problem can be expressed in terms of solutions to the SAME PROBLEM with smaller or simpler inputs. Typically, code for implementing a recursive solution to a problem involves writing a function or procedure that invokes itself.
- Major design strategies such as Divide-&-Conquer, Dynamic Programming and Backtracking apply recursion.
- Using a **stack**, every algorithm can be written as a non-recursive algorithm.
- A **stack** is what the computer uses to implement recursive functions. More generally, when function *A* calls a function *B*, information such as the values of local variables, the line where call was made, etc., are pushed on the stack. After *B* is completed, this information is popped from the stack and used to continue the execution of *A*.

Major Design Strategies

The Greedy Method

Divide-and-Conquer

Dynamic Programming

Backtracking

Branch-and-Bound

Greedy Method

- The Greedy Method solves optimization problems by making locally optimal choices and hoping that these choices lead to a globally optimal solution.
- We have already seen the Greedy Method in action with the familiar problem of making change.

Divide-and-Conquer

- Divide-and-Conquer is essentially a special case of recursion in which a given problem is divided into (usually) two or more subproblems of the exactly the same type, and the solution to the problem is expressed in terms of the solutions to the subproblems.
- MergeSort is a classical example. Given a list to sort, MergeSort divides the list in half, recursively sorts the first half and second half, then invokes a procedure to merge the two sorted sublists into a sorting of the entire list.
- In this course we use Divide-&-Conquer to design algorithms for polynomial and integer multiplication, matrix multiplication, the celebrated FFT algorithm for computing DFTs.

Dynamic Programming

- Like Divide-&-Conquer Dynamic Programming also applies recursion, but in a bottom up fashion. It is a technique in which solutions to a problem are built up from solutions to subproblems (like Divide-and-Conquer), but where all the smallest subproblems to a given problem are solved before proceeding on to the next smallest subproblems.
- For example, a dynamic programming solution to sorting could be based on a "bottom-up" version of Mergesort, where the two-element sublists
- $L[0:1]$, $L[2:3]$, ..., $L[n-2:n-1]$ are sorted first, then the four-element sublists $L[0:3]$, $L[4:7]$, ..., $L[n-4:n-1]$ are sorted and so forth.
- Dynamic programming is usually applied to situations where the problem is to optimize an objective function, and where the optimal solution to a problem must be built up from optimal solutions to subproblems, i.e., the Principle of Optimality holds. Its efficiency results by eliminating suboptimal subproblems when moving up to larger subproblems.
- In this course we will discuss dynamic programming solutions for finding all-pairs shortest paths, optimal matrix parenthesization, optimal binary search trees, edit distance.

Backtracking and Branch-&-Bound

- Backtracking and Branch-&-Bound refer to strategies to solve problems that have associated state-space trees.
- Backtracking searches the state-space tree using depth-first search, whereas Branch-&-Bound uses other searches, such as breadth-first search.
- These search strategies have wide applicability, and apply to most problems whose solution depends on making a series of decisions.
- The state-space tree for the problem simply models all possible decisions that can be made at each stage.
- The state-space tree is usually quite large, so that examining each node in the tree is not usually feasible.
- However, by using appropriate bounding functions, the searches are often efficient due to cutting off large subtrees of the state-space tree when it is determined that no solution can lie in these subtrees.

Joke (bad)

Why did the engineering students not finish the lecture video?

They were getting a little ANSI.

ANSI
American National Standards Institute

This joke is not only bad, it's nerdish bad!