Developing Algorithms

Strategies –

- realize your problem is another well-known problem in disguise
  - it is searching or sorting
  - there's a data structure for that
  - it is a graph problem
- develop a new algorithm
  - control-flow paradigms: iterative, recursive
  - solution-construction paradigms: decomposition, series of choices
  - design paradigms: divide-and-conquer, greedy, backtracking + branch-and-bound, dynamic programming

How to Design Algorithms

- establish the problem
- identify avenues of attack
- define the algorithm
- show termination and correctness
- determine efficiency

How to Design Algorithms

Establish the problem. Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*
  State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each?
- *Examples.*
  If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

How to Design Algorithms

Identify avenues of attack. Determine the algorithmic approach(es) to try. In real problem solving the appropriate paradigm is not always clear at the start, so several possibilities may need to be considered. In assigned problems the paradigm is often specified or implied, allowing this step to be abbreviated or skipped.

- *Known targets.*
  Identify any applicable time and space requirements for your solution. This might be stated as part of the problem ("find an $O(n \log n)$ algorithm"), come from having an algorithm you are trying to improve on, or stem from the expected input size (e.g. you need to work with very large inputs so you need $O(n \log n)$ or better).
- *Approach.*
  Identify applicable solution-construction approaches: decomposition or series of choices? For each, consider briefly what that approach would look like for this problem — does it even make sense?
- *Paradigms and patterns.*
  Based on the targets and approach(es) identified, identify the applicable paradigm(s) and specific pattern(s) within each paradigm. For each, consider briefly what that paradigm and pattern would look like for this problem — does it even make sense?

How to Design Algorithms

Define the algorithm. Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Algorithm.*
  State the whole algorithm.

  The algorithm should be specified at as high a level as possible but with enough detail to clearly convey the steps and to be able to show correctness and (with the addition of implementation details later) address running time. Can a person familiar with standard data structures and algorithms carry out the algorithm by hand based on what is written on the page?

How to Design Algorithms

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.*
  Explain why the algorithm always terminates i.e. it always eventually produces a solution.
- *Correctness.*
  Explain why the solution produced is the correct solution for every valid input instance.

How to Design Algorithms

Determine efficiency. Evaluate the running time and space requirements of the algorithm.

- *Implementation.*
  Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.
- *Time and space.*
  Assess the running time and space requirements of the algorithm given the implementation identified.
- *Room for improvement.*
  Are the targets met? Is it necessary to do better? If improvements in running time and/or space are needed, identify possible avenues for improvement.

Control Flow Paradigms

Iterative algorithms proceed forward towards the solution one step at a time.

- repetition through loops

Recursive algorithms have friends solve smaller instances of the same problem (subproblems).

- repetition through recursion

How to Design Iterative Algorithms

- establish the problem
- identify avenues of attack
- define the algorithm
- show termination and correctness
- determine efficiency

In a group of people, it is to be expected that some of them may not want to work with each other. Assuming that each person has at most *d* other people that they don't want to work with, divide the people into *d*+1 groups so that everyone is in exactly one group and no one is in a group with someone they don't want to work with.

Establish the problem. Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*
  State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each?
- *Examples.*
  If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

- task
  - given *n* people and, for each person, up to *d* other people they don't want to work with, divide the people into *d*+1 groups so that everyone is in exactly one group and no one is in a group with someone they don't want to work with
- input
  - *n* people
  - for each person, the up to *d* other people they don't want to work with
- output
  - membership of *d*+1 groups

Common Iterative Patterns

Identify avenues of attack.

- *Paradigms and patterns.*

Iterative algorithms can be characterized by the main focus of the loop –

- process input
  - go through the collection of input items one at a time
- produce output
  - produce the collection of output items one at a time
  - build up the output one bit at a time
- narrow the search space
  - repeatedly eliminate items that aren't the one you are looking for
  - repeatedly eliminate solutions that aren't the one you want
  - must be able to eliminate things without looking at them directly

In a group of people, it is to be expected that some of them may not want to work with each other. Assuming that each person has at most *d* other people that they don't want to work with, divide the people into *d*+1 groups so that everyone is in exactly one group and no one is in a group with someone they don't want to work with.

input
- *n* people
- for each person, the up to *d* other people they don't want to work with

output
- membership of *d*+1 groups

Identify avenues of attack.

- *Paradigms and patterns.*

- process input
  - go through the collection of input items one at a time
- produce output
  - produce the collection of output items one at a time
  - build up the output one bit at a time
- narrow the search space
  - repeatedly eliminate items that aren't the one you are looking for
  - repeatedly eliminate solutions that aren't the one you want
  - must be able to eliminate things without looking at them directly

- process input
  - collection of people – for each person, assign them a group
- produce output
  - collection of groups – for each group, determine who is in it
- narrow the search space
  - not applicable – not a search problem

How to Design Iterative Algorithms

Define the algorithm. The core of an iterative algorithm is defining the loop.

- *Main steps.*
  This is the core of the algorithm — the loop body. What's being repeated?
- *Exit condition.*
  When does the loop end?

pattern	loop structure	exit condition
process input	for each input element, process that element and incorporate it into the solution so far	when all of the input elements have been processed
produce output	repeatedly produce the next output element repeatedly produce the next piece of the solution	when all of the output elements have been produced when the solution is complete
narrow the search space	repeatedly eliminate some non-solutions	when the solution has found or there are no solutions left

- *Setup.*
  Whatever must happen before the loop begins.
- *Wrapup.*
  Whatever must happen to get the final answer after the loop ends.
- *Special cases.*
  Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).
- *Algorithm.*
  Assemble the algorithm from the previous steps and state it.

  There shouldn't be new elements here, instead bring together the main steps, exit condition, setup, and wrapup along with any handling needed for special cases and state the whole algorithm.

In a group of people, it is to be expected that some of them may not want to work with each other. Assuming that each person has at most *d* other people that they don't want to work with, divide the people into *d*+1 groups so that everyone is in exactly one group and no one is in a group with someone they don't want to work with.

pattern	loop structure	exit condition
process input	for each input element, process that element and incorporate it into the solution so far	when all of the input elements have been processed

- main steps
  - – incorporate the next input item into the solution to obtain a solution for one more element
  - – for each element, process it
  - – for each person, put them into a group
  - – for each person, put them into a group not containing someone they don't want to work with
  - – produce the next output item
  - – for each group, add people to that group as long as there isn't anyone in the group that they don't want to work with

In a group of people, it is to be expected that some of them may not want to work with each other. Assuming that each person has at most *d* other people that they don't want to work with, divide the people into *d*+1 groups so that everyone is in exactly one group and no one is in a group with someone they don't want to work with.

pattern	loop structure	exit condition
process input	for each input element, process that element and incorporate it into the solution so far	when all of the input elements have been processed

- exit condition – the loop ends when
  - – all of the input items have been processed
  - – when all of the output items have been produced
  - – when every person has been added to a group
  - – when everyone has been assigned to a legal group
  - – when no one else can be added to the current group
  - – when all of the groups are full
  - – when d+1 groups have been formed

In a group of people, it is to be expected that some of them may not want to work with each other. Assuming that each person has at most *d* other people that they don't want to work with, divide the people into *d*+1 groups so that everyone is in exactly one group and no one is in a group with someone they don't want to work with.

- *Setup.*
  Whatever must happen before the loop begins.
- *Wrapup.*
  Whatever must happen to get the final answer after the loop ends.
- *Special cases.*
  Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).
- *Algorithm.*
  Assemble the algorithm from the previous steps and state it.

  There shouldn't be new elements here, instead bring together the main steps, exit condition, setup, and wrapup along with any handling needed for special cases and state the whole algorithm.

- main steps
  - – for each person, put them into a group not containing someone they don't want to work with

How to Design Iterative Algorithms

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.* Show that the loop — and thus the algorithm — always terminates.
  - *Measure of progress.*
    Identify a quantity and the direction of change that leads towards the exit condition.
  - *Making progress.*
    Explain why every iteration of the loop advances the measure of progress towards the exit condition.
  - *The end is reached.*
    Explain why making progress ensures that the exit condition is always reached.

pattern	measure of progress	making progress	termination argument
process input	number of input elements processed	each iteration processes one more element	repeatedly processing one more input element means that eventually all will have been processed
produce output	number of elements in the solution	each iteration produces one more element	repeatedly producing one more output element or one more piece of the solution means that eventually all will have been produced
narrow the search space	size of the current range or (alternatively) the number of solutions still in the current range	each iteration reduces the size of the search space	repeatedly reducing the size of the current range or the number of solutions still in the current range means that eventually there will be no solutions left if the solution hasn't been found

In a group of people, it is to be expected that some of them may not want to work with each other. Assuming that each person has at most d other people that they don't want to work with, divide the people into $d+1$ groups so that everyone is in exactly one group and no one is in a group with someone they don't want to work with.

pattern	measure of progress	making progress	termination argument
process input	number of input elements processed	each iteration processes one more element	repeatedly processing one more input element means that eventually all will have been processed

- exit condition
  - when every person has been added to a group
- measure of progress
  - something to count the number of people added to groups
  - the number of elements considered
  - the number of elements left to consider
  - number of input items processed
  - number of output items processed
  - number of groups that have been filled
  - number of people that have been assigned to a group
  - number of people not yet assigned to a group

How to Design Iterative Algorithms

- *Correctness.* Show that the algorithm is correct.
  - *Loop invariant.*
    State a loop invariant.
  - *Establish the loop invariant.*
    Explain why the loop invariant holds at the beginning of the first and second iterations of the loop.
  - *Maintain the loop invariant.*
    Explain why the loop invariant continues to be true after each iteration — assuming that it holds at the beginning of iteration k, explain why it also holds at the beginning of the next iteration $(k+1)$.
  - *Final answer.*
    Explain why the whole algorithm — setup, loop, wrapup — means that the final result is a correct answer to the problem.

Loop Invariants

- a *loop invariant* is a statement about the program state that is true every time the loop condition is checked
  - it is true before the first iteration
  - it remains true after every iteration
  - it helps explain why the algorithm works
    - showing that the loop invariant is true when the loop exits helps us establish that the overall algorithm is correct
- a good loop invariant is
  - *strong enough* to help prove the final correctness of the algorithm
  - *about correct progress* – it address the portion of the problem that has already been solved

How to Design Iterative Algorithms

- *Correctness.* Show that the algorithm is correct.
  - *Loop invariant.*
    State a loop invariant.
  - *Establish the loop invariant.*
    Explain why the loop invariant holds at the beginning of the first and second iterations of the loop.
  - *Maintain the loop invariant.*
    Explain why the loop invariant continues to be true after each iteration — assuming that it holds at the beginning of iteration k, explain why it also holds at the beginning of the next iteration $(k+1)$.
  - *Final answer.*
    Explain why the whole algorithm — setup, loop, wrapup — means that the final result is a correct answer to the problem.

pattern	loop invariant
process input	have a correct solution for the first k input elements, or (alternatively) haven't gone wrong yet (solution so far is consistent with a solution for the whole problem)
produce output	have produced the first k elements of the correct output
narrow the search space	either the element is within the current search space / set of solutions or it was never present at all, or (alternatively) not all of the solutions (if there are any) have been eliminated

- common forms
  - the first *k* elements satisfy some property
  - a variable stores the correct result for the elements processed so far

In a group of people, it is to be expected that some of them may not want to work with each other. Assuming that each person has at most *d* other people that they don't want to work with, divide the people into *d*+1 groups so that everyone is in exactly one group and no one is in a group with someone they don't want to work with.

pattern	loop invariant
process input	have a correct solution for the first k input elements, or (alternatively) haven't gone wrong yet (solution so far is consistent with a solution for the whole problem)

- loop invariant
  - the first *k* people have been assigned to groups
  - the first *k* people have been assigned to groups so that no one is in a group with someone they don't want to work with
  - there are at most *d*+1 groups
  - the first *k* people have been assigned to groups so that no one is in a group with someone they don't want to work with and there are at most *d*+1 groups
  - have a correct solution for the first *k* input items
  - haven't gone wrong yet
  - have produced the first *k* items of the output
  - the first *k* groups have been assigned people
  - the first *k* groups have been assigned people without anyone being in a group with someone they don't want to work with