Solution Construction Paradigms

Control flow paradigms categorize algorithms based on their repetitive structure.

- loops
- recursion

Solution construction paradigms categorize algorithms based on how solutions are built.

- *decomposition* – solve the problem by breaking it into smaller problems and combining the subproblem solutions
- *series of choices* – build the solution incrementally by making a series of decisions

Series of Choices – Approaches and Flavors

- as with iterative algorithms, there are two main approaches to the series of choices

- **process input**, where the choice or decision is about what to do with each input element in turn
- **produce output**, where the choice or decision is about what what the next output element is

- there are also several common types of tasks which lead to slightly different flavors for the algorithm components

- a **subset**, where the task to is select a subset of the input items subject to some subset membership constraint
- a **sequence**, where the task is to produce an ordering of all or a subset of the input items
- a **labelling**, where the task is to assign labels to the input items

  - these can overlap – pick what best aligns with the primary task
  - not an exhaustive list

Series of Choices – Paradigms

- how many alternatives need to be considered for each decision results in fundamentally different algorithmic paradigms

- If only **one** alternative needs to be considered, the formulation can be iterative. The key focus for the algorithm is determining how to pick that right alternative for each decision, and showing that the series of choices made leads to a correct solution. Greedy algorithms are of this type and will be considered in chapter 7.
- If **more than one** alternative needs to be considered, the formulation is typically recursive. The key focus for the algorithm is how to avoid an exponential blowup in running time. Backtracking, branch-and-bound, and dynamic programming algorithms are of this type and will be considered in chapters 8 to 10.

Greedy Algorithms

- iterative series of choices
  - main steps: repeat make the next choice until done
- always make a *local* decision with no regrets
  - each choice is made without consideration of future possibilities
  - can't reconsider later – must pick the right alternative on the first try
- often, but not exclusively, refers to optimization problems
  - find the best solution among (generally) many legal solutions
  - for non-optimization problems, goal is to find a legal solution among (generally) many invalid (non-)solutions
- don't work for everything
  - some problems aren't solvable with only locally good decisions
- a correctness proof is essential!
  - finding counterexamples is an important technique for identifying incorrect greedy choices
  - often use proof by contradiction for maintaining the invariant

Proof Techniques – Incorrectness

One counterexample is all that is needed to prove an algorithm incorrect.

Properties of a good counterexample –

- simple, which often means small
- verifiable – need to be able to compute the algorithm's output and give a better answer

Strategies –

- think exhaustively – can often enumerate all possible inputs of a small size
- hunt for weakness – look for a case where the algorithm's choice is the wrong thing to do
- try inputs with duplicates or ties, as that neutralizes the algorithm's choice
- seek extremes rather than uniformity

Counterexamples

Find a counterexample to prove the following statement false:

```
a + b >= min(a,b)
```

- think exhaustively
  - can't try all possible pairs of numbers or even all single numbers, but can consider all of the different cases
- cases: > 0, 0, < 0
- both numbers negative e.g. -5, -2
  - -5 + -2 = -7 ≥ min(-5,-2) = -5 → false

1-2. *[3]* Show that $a \times b$ can be less than $\min(a, b)$.

Counterexamples

1-3. *[5]* Design/draw a road network with two points a and b such that the fastest route between a and b is not the shortest route.

- hunt for weakness
  - set up the framework for an example where the undesired situation happens
    → two routes, the faster route (time) is longer (distance)

1-4. *[5]* Design/draw a road network with two points a and b such that the shortest route between a and b is not the route with the fewest turns.

Counterexamples

1-5. *[4]* The subset sum problem is as follows: given a set of integers $S = \{s_1, s_2, \ldots, s_n\}$, and a target number T, find a subset of S that adds up exactly to T. For example, there exists a subset within $S = \{1, 2, 5, 9, 10\}$ that adds up to $T = 22$ but not $T = 23$.

Find counterexamples to each of the following algorithms for the subset sum problem. That is, give an S and T where the algorithm does not find a solution, even though a solution exists.

(a) Pick elements of S in left to right order if they fit,

(b) Pick elements of S from smallest to largest, that is,

(c) Pick elements of S from largest to smallest.

- hunt for weakness
  - set up the framework for an example where the undesired situation happens
    → start with a small set (three things) with T = sum of all elements, then add an element that would get picked instead but does not exactly replace one or more of the original choices

Proof Techniques – Contradiction

- assume that what you want to prove is false
- develop logical consequences from this assumption, until you get to one that is demonstrably false
- since there were no flaws in the deduction, the assumption that what you want to prove is false must have been faulty and thus what you want to prove is true

Proof by Contradiction – Example

- claim: sqrt(2) is irrational
- proof by contradiction
  - assume that sqrt(2) is rational
  - then sqrt(2) = a/b where a, b are integers with no common factors ($b \neq 0$)
  - square both sides: $\text{sqrt}(2)^2 = (a/b)^2 \rightarrow 2 = a^2/b^2 \rightarrow a^2 = 2b^2$
  - thus a^2 is even
  - thus a is even
  - thus $a = 2k$ for some integer k
  - substitute and simplify: $a^2 = 2b^2 \rightarrow (2k)^2 = 2b^2 \rightarrow 4k^2 = 2b^2 \rightarrow 2k = b^2$
  - thus b is also even
  - but if a and b are both even, they have a common factor (2), violating the assumption that they didn't and thus the assumption that sqrt(2) is rational
  - so, sqrt(2) is irrational

Proof Techniques – Contradiction

- for the maintain the invariant step –
  - want to show: if the invariant holds after k iterations, it still holds after k+1 iterations
  - strategy – proof by contradiction
    - assume the invariant holds after k iterations
    - assume this is where things go wrong – the invariant *doesn't* hold after k+1 iterations
    - deduce that something you know is true isn't (contradiction)

Invariants by Contradiction – Example

```
max ← A[0]
for i = 1 to n-1 do
  if A[i] > max then
    max ← A[i]
return max
```

loop invariant –
after i iterations (i.e. at the start of iteration i+1), `max` is the largest element in `A[0..i]`

maintain the invariant –

- assume that after iteration k (at the start of iteration k+1) `max` is the largest element in `A[0..k]`
- assume, for contradiction, that after iteration k+1 (i.e. at the start of iteration k+2) `max` is *not* the largest element in `A[0..k+1]`
- ...what happened?
  - either the largest element in `A[0..k+1]` is in `A[0..k]` or it is `A[k+1]`
  - it cannot be in `A[0..k]` because the loop invariant gives us that `max` is the largest in `A[0..k]` and the loop doesn't update `max` if `A[k+1]` is smaller
  - it cannot be `A[k+1]` because the loop would have updated `max`
- ...so the assumption that this is where the algorithm fails to maintain the invariant must be false and the invariant holds