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
  - divide-and-conquer
  - iterative
  - series of choices – greedy, recursive backtracking, dynamic programming

Algorithmic Paradigms

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

Recursive algorithms have friends solve subproblems.

- construct a complete solution out of complete solutions for smaller subproblems
  - induction lets you demonstrate that the solution for the bigger problem is correct
- base case defines when you stop
  - making progress ensures that you will get there (recursion will terminate)
    - in terms of problem size

16 Steps to Recursive Success

establishing the problem

1. specifications
2. examples
3. size

brainstorming ideas

4. targets
5. tactics
6. approaches

defining the algorithm

7. generalize / define subproblems
8. base case(s)
9. main case
10. top level
    - initial subproblem
    - setup
    - wrapup
11. special cases
12. algorithm

showing correctness

13. termination
    - making progress
    - reaching the end
14. correctness
    - establish the base case(s)
    - show the main case
    - final answer

determining efficiency

15. implementation
16. time and space

Recursive Patterns

Characterized by the number and size of subproblems –

- 1 friend – can often easily be written as iterative instead
  - *constant amount* – subproblem is smaller by a fixed number of elements (typically 1)
    - e.g. $a^n = a\ a^{n-1}$ or $n! = n\ (n-1)!$
  - *constant factor* – subproblem is a fixed fraction of the size (typically ½) – "decrease and conquer"
    - e.g. binary search
    - e.g. $a^n = (a^{n/2})^2$ if n is even, a $(a^{(n-1)/2})^2$ if n is odd
  - *variable factor* – subproblem is smaller, but the size of the reduction varies
    - e.g. gcd(m,n) = gcd(n,m mod n)

Recursive Patterns

Characterized by the number and size of subproblems –

- 2+ friends
  - *divide-and-conquer* – split into $b \geq 2$ subproblems of size n/b (b is typically 2)
    - *process input* – split input in straightforward way, then do work combining subproblem solutions
      - e.g. mergesort
    - *produce output* – do work creating the subproblem instances, then just add a piece to the subproblem solutions
      - e.g. quicksort
    - *narrowing the search space* – each friend searches a different part of the search space
  - *case analysis* – each friend considers a different choice
    - e.g. depth first search

Solving Recurrence Relations

Recursive algorithms tend to lead to recurrence relations in one of two forms:

- split off b elements
  - $T(n) = a\,T(n-b) + f(n)$ where $f(n) = 0$ or $\Theta(n^c \log^d n)$
- divide into subproblems of size n/b
  - $T(n) = a\,T(n/b) + f(n)$ where $\Theta(n^c \log^d n)$

Solving Recurrence Relations

$T(n) = a\,T(n-b) + f(n)$ where $f(n) = \Theta(n^c \log^d n)$

Cases are based on the number of subproblems and f(n).

a	f(n)	behavior	solution
> 1	any	base case dominates (too many leaves)	$T(n) = \Theta(a^{n/b})$
1	≥ 1	all levels are important	$T(n) = \Theta(n\,f(n))$

Solving Recurrence Relations

$T(n) = a\,T(n/b) + f(n)$ where $f(n) = \Theta(n^c \log^d n)$

Cases are based on the relationship between the number of subproblems, the problem size, and f(n).

(log a)/(log b) vs c	d	behavior	solution
<	any	top level dominates – more work splitting/combining than in subproblems (root too expensive)	$T(n) = \Theta(f(n))$
=	> -1	all levels are important – log n steps to get to base case, and roughly same amount of work in each level	$T(n) = \Theta(f(n) \log n)$
=	< -1	base cases dominate – so many subproblems that taking care of all the base cases is more work than splitting/combining (too many leaves)	$T(n) = \Theta(n^{(\log a)/(\log b)})$
>	any		

Divide-and-Conquer

The goal in developing a divide-and-conquer algorithm is often to improve on a polynomial brute-force solution.

- targets should identify the brute force solution and its running time
  - this should be pretty straightforward
  - if not, then divide-and-conquer is being used to try to find a solution in the first place

16 Steps to Divide-and-Conquer Success

establishing the problem

1. specifications
2. examples
3. size

brainstorming ideas

4. targets
   - *identify brute force algorithm / running time*
5. tactics
6. approaches
   - *process input*
   - *produce output*
   - *narrow the search space*

defining the algorithm

7. generalize / define subproblems
8. base case(s)
9. main case
10. top level
    - initial subproblem
    - setup
    - wrapup
11. special cases
12. algorithm

showing correctness

13. termination
    - making progress
    - reaching the end
14. correctness
    - establish the base case(s)
    - show the main case
    - final answer

determining efficiency

15. implementation
16. time and space

Given the price of a stock over an n-day period, determine the best time to have bought and sold 1000 shares of that stock. (Buy and sell once, on different days.)

What is the smallest size problem?

- ○ 0 days
- ○ 1 day
- ○ 2 days
- ○ 3 days
- ○ 4 or more days

Can a ~~process output~~ approach be used here?
produce output

- ○ True
- ○ False

- **Process input**, where the input is divided in half in a straightforward way (such as "first half" and "second half"); the work in the main case is primarily in combining the results from the friends to produce the solution
- **Produce output**, where each friend produces some of the output (typically one friend produces the first part of the output and the other friend produces the second part); the work in the main case is primarily in splitting the input

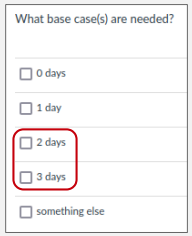
Given the price of a stock over an n-day period, determine the best time to have bought and sold 1000 shares of that stock. (Buy and sell once, on different days.)

Generalize / define subproblems:

- ☐ have one friend find the lowest price day in a region of the array and another friend find the highest price day in a region of the array
- ☐ friend finds the lowest and highest prices in a region of the array
- ☐ friend finds the best buy/sell dates in a region of the array
- ☐ have one friend find the lowest price day in the whole array and another friend find the highest price day in the whole array

7. *Generalize / define subproblems.*
Friends get smaller versions of the original problem, which often takes the form of a generalized version of the original problem. (For example, doing the original task on a portion of the original input is a generalized version of the original problem — the specific task is to work with all of the input, while the generalized version works with any portion of the input, including all of it.) Define the generalized problem, its input, and its output along with pre- and postconditions. Make sure that everything the friend needs or hands back should be covered by the input(s) and output(s) — avoid global variables and global effects.

Given the price of a stock over an n-day period, determine the best time to have bought and sold 1000 shares of that stock. (Buy and sell once, on different days.)



8. *Base case(s).*
   Address how to solve the smallest problem(s). This is often trivial, or is solved via brute force.