Algorithmic Paradigms Wrapup

Key points –

- how to apply the *n*-step algorithm development processes for iterative, greedy, divide-and-conquer, recursive backtracking, dynamic programming
  - especially how the common patterns (process input, produce output, narrowing the search space) apply to and provide structure and direction for the main steps / general case and correctness arguments (loop invariant, maintaining the invariant)

Algorithmic Paradigms Wrapup

Key points –

- what a rigorous argument of correctness looks like, and the ability to assess your own solution
  - be able to identify and construct an appropriate loop invariant
  - understand the form of the establish and maintain steps
    - iterative – explain why the process results in the loop invariant holding
    - recursive – explain why the process results in the correct subproblem solution
  - understand the form of the final result step
    - iterative – explain why the loop invariant being true when the loop exits means that the final result of the algorithm is correct
    - recursive – explain why the correct answer for the initial subproblem means that the final result of the algorithm is correct
  - detect and avoid the trap of claiming that the answer is correct because the process is right
    - instead, must explain why the process results in the correct answer

Algorithmic Paradigms Wrapup

Key points –

- the role and use of examples and counterexamples in figuring out the algorithm and arguing correctness

Algorithmic Paradigms Wrapup

Key points –

- key building blocks
  - sorting is O(n log n) – what algorithms achieve that, applicability/tradeoffs
  - search – sequential search vs binary search (running time, applicability)
  - core ADTs, main implementations of those ADTs, and their running times – how quickly can common operations be done, applicability/tradeoffs
- where it is productive to look for improvements
  - in data structures
  - in algorithms

Improvement Techniques

Algorithmic strategies –

- try a different paradigm
- improvements within a paradigm
- heuristics and randomization

Implementation strategies –

- better data structures

Improvement Techniques

Try a different paradigm –

- divide-and-conquer typically looks to improve a polynomial-time brute force
  - applies when the input can be split and processed as independent subproblems
- when building the solution is formulated as a series of choices...
  - recursive backtracking is exponential
  - dynamic programming is often polynomial or pseudopolynomial
  - greedy is typically polynomial

  ...but whether dynamic programming is effective or greedy can be used depends on the nature of the problem

Improvement Techniques

Improvements within a paradigm –

- greedy
  - speed up making the next greedy choice
    - when the choice is about which input item to process or select for the output next, sort first – O(n log n) to sort (before the loop), but then O(1) for each choice
    - PQ/heap for repeated “find best” in a dynamic environment – O(n) to build heap from all elements, O(log n) for each choice
  - try a different pattern
    - e.g. process input vs produce output

Improvement Techniques

Improvements within a paradigm –

- divide-and-conquer
  - address the split/combine step
    - if the combine step involves computation, can the friends return that instead?
  - address the number of problems and the size of the problems
    - pass fewer total elements to friends
      - adopt a narrowing the search space approach where elements are eliminated and not handed to the friends
    - more smaller problems
      - e.g. bucket sort, shuffle sort, counting sort, radix sort

Improvement Techniques

Improvements within a paradigm –

- recursive backtracking
  - reduce the branching factor
  - reduce the solution length (number of decisions)
  - pruning
  - branch-and-bound
    - tighter bound and initial solution estimates
    - search more promising branches first – best first search + A*

Improvement Techniques

Improvements within a paradigm –

- dynamic programming
  - reduce the number of different subproblems
    - need (substantially) fewer states than partial solutions
    - reducing the number of states often requires some way of ordering the decisions
  - reduce how much work is done per subproblem
    - reduce the number of next choices for a decision
    - representation is a factor – how quickly a subproblem solution can be looked up (which is really how quickly a subproblem can be turned into an array index)

Strategies for Improvement

Heuristic and randomized algorithms and data structures generally perform well but can have poor worst-case performance.

- *bucketing* performs well when the distribution of data is roughly uniform
  - e.g. hashtables – generally O(1)
  - e.g. bucket sort – generally Θ(n) when $k = \Theta(n)$
    - distribute elements into k buckets based on key ranges
    - sort each bucket e.g. with insertion sort
    - $O(n+n^2/k+k)$ on average, $O(n^2)$ worst case with insertion sort
      - if input distribution is not uniform but is known or can be estimated, can choose buckets with constant density to maintain O(n) average performance

Strategies for Improvement

- *randomization* "fixes" worst cases by making them unlikely, and often result in much simpler algorithms
  - e.g. randomized quicksort – sorted/reverse sorted are no longer the worst cases
  - e.g. splay trees
- *Las Vegas* algorithms guarantee correctness and are usually efficient
  - random selection methods
    - e.g. randomized quicksort
    - e.g. randomized hashing – randomly pick hash function from a collection
- *Monte Carlo* algorithms guarantee efficiency and are usually correct
  - random sampling methods
    - e.g. to approximate median, find median of a small random sample of elements

Strategies for Improvement

Improve the data structure –

- e.g. $O(n^2)$ selection sort becomes O(n log n) heapsort
  - speed up “repeatedly find smallest” step
- e.g. $O(n^2)$ insertion sort becomes O(n log n) with balanced BST
  - speed up “insert into sorted collection” step

Implementation design strategy –

- start with an ADT
  - e.g. instead of “use a hashtable” or “put everything into a balanced BST”, identify “this is a lookup problem”
- then consider how to support those operations
  - is there a standard ADT and implementation that is efficient for everything needed?
  - if not, design your own

Beyond Big-Oh

- big-Oh isn't enough to distinguish between algorithms of the same complexity
  - then implementation and system details (e.g. cache performance, memory size) become important
    - → implement and test!
- for very large datasets, constant factors are important even for low complexity (O(n), O(n log n)) algorithms
  - e.g. external sorting