IA Algorithms Revision Notes

Leran Cai

May 7, 2019

1 Smth to say

This is created for all my supervisees. I really hope every one of you can achieve First in your exam. At least the algorithm problems should not be the hard part for all of you.

I put some extra exercises in the end as promised. I tried my best to group up similar problems together so you can not only learn from one single problem but also you may be able to learn how to ask follow-up questions like what I did in our supervisions. We never just learn from one specific problem. What we really need to learn is how to create and solve new questions when having seen one.

2 Top tips

I think there are some key things that you should always remember when answering questions in your exam. I should have mentioned most of them in our supervisions.

1. Never waste time talking useless details. Remember we've seen those modal answers and only key points can get marks. Get full marks for algorithms problems and save time for other problems.

2. Remember to draw graphs/tables/... to help you present your answers. Good handwriting always helps.

3. Choose problems wisely. Don't write down your answers immediately. Read the entire question, see if you can get most of it, then decide.

4. If you get stuck on some questions, skip them, save some space for them on your answer sheet, then go back to it.

5. Remember that all the subproblems are connected. One problem is essentially one big topic. You might want to find some hints from other subproblems if you get stuck. For example, you may guess how to design your algorithms based on the time complexity.

6. In your exam, everything should be connected to what you've learned. Do not improvise too much.

7. You get marks for your short maths proofs and neat algorithms (formula and pseudocode), not a one-page long essay.

3 Sorting Algorithms

3.1 Summary of the running times

Algorithm	Worst-case running time	Average-case/expected running time
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	-
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$

Remark 1 *Don't try to memorize these running times. Try to understand how to derive them. Figure out the worst-case running time and the average-case running time by just looking at the name of the sorting algorithm.*

3.2 Lower bound of a comparison sort

If we use a comparison sort, sorting takes $\Omega(n \lg n)$. The proof can be found in Section 8.1 in CLRS3 (decision tree model). Intuitively speaking, to sort an array, every two elements are compared either directly or indirectly. You cannot have all the information by $O(n)$ comparisons unless you know something before you compare them.

Remark 2 *So this tells us that when you are expected to perform a $O(n)$ sorting algorithm, you have to collect more information.*

3.3 Heap and HeapSort

Don't forget we still have this nice sorting algorithm. It uses a data structure, **max-heap/min-heap**. Or in Java/C++ we use priority queue.

4 Things we need to learn more than just sorting algorithms

I think it's worth writing some comments in a separate section because we may want to conclude some patterns and ideas to help us solve more problems.

The divide-and-conquer approach Many useful algorithms are **recursive** in structure. To solve a problem, we usually call the algorithm recursively. These algorithms typically follow a **divide-and-conquer** approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

Remark 3 *Usually when you see divide-and-conquer, we divide them into two parts. It's natural to think of dividing them into more than two parts. Think about those Tripos questions we've seen.*

Solve a recurrence relation You may need to solve recurrence relations like $f(n) = 2f(n/2) + kn$ (from the lecture notes) and $f(n) = 2f(\sqrt{n}) + c$. Typically, we first substitute n by 2^m and then try to simplify the formula. In the end we substitute m by $\lg n$ and 2^m by n to solve the recurrence.

Remark 4 *Think about the relationship between recursion and dynamic programming.*

Linear time sorting algorithms Sometimes it's easy for you to neglect them. They did appear in many past exam questions. Whenever you see a question with some extra assumptions (e.g. the inputs are drawn from a distribution) and your task is to sort them, keep these algorithms in mind.

5 Algorithm Design

We mainly learned dynamic programming and greedy strategy in this course.

5.1 Dynamic programming

Definition 5 ***Optimization problems*** *can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such* a *solution an* ***optimal solution*** *to the problem, as opposed to* the *optimal solution.*

Definition 6 ***Optimal substructure****: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.*

The dynamic-programming method works as follows:

1. Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly. We arrange for each subproblem to be solved only once.
2. If we need to refer to a subproblem's solution again, we just look it up rather than recompute it.
3. Dynamic programming requires additional memory to save computation time; thus it serves an example of a **time-memory trade-off**.
4. The savings may be dramatic, e.g. from exponential to polynomial.
5. A dynamic-programming approach runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.

Top-down with memoization In this approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem. The procedure first checks whether it has previously solved the subproblem. If so, it returns the saved value, otherwise, it computes the value in the usual manner. We say the recursive procedure has been **memoized**.

Bottom-up method This approach depends on some natural notion of the size of a subproblem, such that solving any particular subproblem depends only on solving smaller subproblems. We sort the subproblems by size and solve them in size order, smallest first. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon.

Remark 7 *The two methods yield algorithms with the same asymptotic running time except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much* ***better constant factors, since it has less overhead for procedure calls.***

Remark 8 *In your* ***exam****, if you know a problem can be solved by dynamic programming, always present the recursive function (if there is one) at the very beginning of your answer. Then explain how you would use this recursive function to solve the problem by solving subproblems first.*

Don't be stubborn, be flexible. There is no point being stick to just top-down or just bottom-up method. Choose whichever is easier to think. Usually top-down is easier to help you construct the recursive function. Then try to implement the bottom-up solution because usually it has less overhead and may be a more decent solution.

Difference from the divide-and-conquer approach Dynamic programming is related to the strategy of divide-and-conquer in that it breaks up the original problem recursively into smaller problems that are easier to solve. The essential difference is that the subproblems may **overlap** here.

5.2 Greedy algorithms

Usually greedy algorithms start by performing whatever operation contributes as much as any single step can towards the final goal. The next step will be the best step that can be taken from the new position and so on. The **problem** is that sometimes greed can get you stuck in a **local maximum**, so it's always a prudent idea to **develop a correctness proof**.

However, many interesting problems require you to be both greedy and clever. Sometimes naive approaches would fail you. Or it might take you some time to figure out how to be greedy. Think about the "activity selection problem".

Greedy algorithms often connect with optimization problems. The question is how can you prove the correctness of your algorithm? How can you prove that no other solutions can beat you? Well, speaking of proving something, common strategies might work, like prove by **induction** and **contradiction**. Of course there are also combinatorial ways.

6 Data Structure

6.1 Elementary data structures

stack In a stack, the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out, or LIFO, policy.

queue In a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy.

6.2 Binary Search Tree

Tree traversal An **inorder** tree walk prints the key of the root of a subtree between printing the values in its left subtree and printing those in its right subtree. Similarly, a **preorder** tree walk prints the root before the values in either subtree, and a **postorder** tree walk prints the root after the values in its subtrees.

Running times

Operation	Running time
Search	$O(h)$
Insert	$O(h)$
Delete	$O(h)$

Remark 9 *Note that here I skipped Red-Black trees, 2-3-4 trees and B-trees since for those data structures just read through the lecture notes. In this revision notes, I would only point out some key points that you should be very familiar with.*

We've seen one Red-Black tree problem in our past supervision. Remember how you convert them into a 2-3-4 tree?

There are 3 cases when inserting a new node in Red-Black tree, can you remember what they are?

6.3 Hash tables

Chaining. The locations in the array hold linear lists that collect all the keys that hash to that particular value.

Open addressing. Use the hash value $h(n)$ as a first preference for where to store the given key in the array. On adding a new key, if that location is empty then use it; otherwise, a succession of other probes are made of the hash table according to some rule until either the key is found to be present or an empty slot for it is located.

Linear probing $h(k) + j \mod m$.

Quadratic probing $h(k) + cj + dj^2 \mod m$.

Double hashing $h_1(k) + j \cdot h_2(k) \mod m$.

7 Graphs

7.1 Running times

Algorithm	Running time
DFS	$\Theta(V+E)$
Topological Sort	$\Theta(V+E)$
BFS	$\Theta(V+E)$
Dijkstra	$O(E+V\log V)$
Bellman-Ford	$O(VE)$
Johnson's algorithm	$O(V^2\log V+VE)$
All pairs shortest paths with matrices	$O(V^3\log V)$
Prim	$O(E+V\log V)$
Kruskal	$O(E\log V)$
Ford-Fulkerson	$O(Ef^*)$

7.2 Dijkstra

Dijkstra's algorithm can be used to solve the single source shortest path problem. It is used for graphs where all edges have non-negative weights. However, if we allow the priority queue to pop the shortest distance to the same vertex more than once, we can apply this modified version of Dijkstra's algorithm to graphs with negative edges but no negative cycles.

7.3 Bellman-Ford

If the graph contains no negative cycles reachable from the source then for every pair of vertices compute the weight of the minimal-weight path; otherwise detect that there is a negative cycle reachable from the source.

8 Advanced Data Structures

8.1 Different ways to implement priority queues

	Binary Heap	Binomial Heap	Fibonacci Heap
peekmin	$O(1)$	$O(\log n)$	$O(1)$
popmin	$O(\log n)$	$O(\log n)$	$O(\log n)$
push	$O(\log n)$	$O(\log n)$	$O(1)$
decreasekey	$O(\log n)$	$O(\log n)$	$O(1)$
delete	$O(\log n)$	$O(\log n)$	$O(\log n)$
merge	$O(n)$	$O(\log n)$	$O(1)$

8.2 Amortized analysis

Accounting method Each aggregate operation is assigned a payment. The payment is intended to cover the cost of elementary operations needed to complete this particular operation, with some of the payment left over, placed in a pool to be used later.

The difficulty with problems that require amortized analysis is that, in general, some of the operations will require greater than constant cost. This means that no constant payment will be

enough to cover the worst case cost of an operation, in and of itself. With proper selection of payment, however, this is no longer a difficulty; the expensive operations will only occur when there is sufficient payment in the pool to cover their costs.

Potential method The amortized cost is $c + \Phi(S') - \Phi(S)$. $T_{amortized} = T_{true} - \Phi(S_0) + \Phi(S_k)$.

Remark 10 *If this is in your exam, then it might be tricky to find the correct potential function. To solve the problem, you might want to first memorise the definition of it in your lecture notes and the examples as well. Then see if you can borrow some ideas. If not, the only thing you can do is to guess. Be brave and try to find something then justify it.*

8.3 Lower bound

Theorem 11 *In the Fibonacci heap, if a node has d children, then the total number of nodes in the subtree rooted at that node is at least F_{d+2}, the $(d+2)$th Fibonacci number.*

Note that another useful lower bound is $F_{d+2} \geq \phi^d$ where $\phi = (1 + \sqrt{5})/2$ is the golden ratio.

9 Geometrical algorithms

9.1 Jarvis's march

It takes $O(nh)$ where n is the number of points and h is the number of points in the convex hull.

9.2 Graham's scan

The initial sort takes $O(n \log n)$. The loop is $O(n)$. Note that when implementing Graham's scan, floating point precision means that Graham's scan is useless when the angles are very small because the numbers are so small that the side-check would just give random outputs.

10 Exercises

This section includes some of my favorite problems. You may have seen them/similar problems in our supervisions.

1. Implement an in-place merge-sort.
2. Find the median in an array **without sorting them first**.
3. Find the K-th largest element in an array **without sorting them first**.
4. Given a 2d grid map of 1s (land) and 0s (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.
5. Given a 2D board and a word, find if the word exists in the grid. The word can be constructed from letters of sequentially adjacent cell, where adjacent cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

6. So you have a start word and an end word. They have the same length l. You also have a dictionary with words of length l. Starting from the start word, you can only change one letter of the word at one step and you can only change it to a word in your dictionary. If there exists a way to change the start word to the end word, output the least number of steps you need. If not, output -1.

7. Activity selection: You have a list of activities with starting times and ending times. Find the maximum number of activities you can attend. $\{(s_1, e_1), (s_2, e_2)...(s_n, e_n)\}$.

8. Merge intervals: You have a list of closed intervals: $\{[s_1, e_1], [s_2, e_2]...[s_n, e_n]\}$. If two intervals overlap, then you merge them. For example, $[1, 3], [2, 4]$ can be merged to $[1, 4]$, but $[1, 2], [3, 4]$ cannot. How many intervals in the end do you get?

9. Meeting rooms: You have a list of meetings with starting times and ending times. What is the minimum number of rooms you need to book in order to make sure that all meetings can have a room to use.

10. Solve the longest common subsequence problem.

11. Solve the longest common substring problem.

12. Knapsack problem: You have a list of items with weight w_i and value v_i and a bag that can have items of weight at most W. What is the maximum value you can achieve if you can only pick one item of each kind? What if you can have infinite supply of each item?

13. You have a word and a dictionary. If a word can be split into words in the dictionary, return true, otherwise false. For example, your word is "otherwise" and your dictionary is $\{$other, wise$\}$, you return true. If your dictionary is $\{$other$\}$, return false.

14. Use two stacks to implement a queue.

15. Implement a special stack which can also tell you the maximum element in the stack.

16. Dijkstra is a single source shortest path algorithm. Give me a single target shortest path algorithm.

17. In bipartite graph, find the maximum matching.