Revision notes - MA3252

Ma Hongqiang

February 21, 2018

Contents

1 Preliminaries **2**

2 Brute Force **4**

3 Divide and Conquer **6**

4 Dynamic Programming **11**

5 Greedy Algorithm **14**

Remark:
In this note, I will save myself the trouble of typing out pseudocode. It is because

- Pseudocode is easily accessible from lecture notes and easily reconstructible from *ideas* given.
- I still suck in using `algorithmic` package.

I regret this decision of mine has caused inconvenience of you.

1 Preliminaries

Definition 1.1 (Algorithm)**.**
An algorithm is a sequence of **unambiguous** instructions for solving a problem, i.e., for obtaining a required **output** for any legitimate **input** in a **finite** amount of time.

Algorithms can be presented in the language of pseudocode. In CS3230, proven of correctness of algorithms is required.
Information about learnt data structure will not appear here. A remark for trees is that the root is at level 0 if level/depth of a node is considered and leaves are at level 0 if height of the node is considered.
There are commonly 4 ways to analyse algorithms, namely

- Induction
- Recurrence Relations
- Mathematical Tools, and
- Invariants and Loop Invariants

Information about mathematical preliminaries will not appear here, except for the following useful property:

$$\frac{b^{n+1} - a^{n+1}}{b - a} = \sum_{i=0}^{n} a^i b^{n-i} \leq (n+1)b^n \quad \text{if } 0 \leq a < b$$

1.1 Asymptotic Notations

In analysis of algorithms, lower/upper bounds say $f(n)$ considers every input size $n \in \mathbb{N}$. It may not be desirable so most usually **asymptotic** lower/upper bounds are of interest.
In this subsection, we suppose that f, g are functions obeying $f, g : \mathbb{N} \to \mathbb{N}$.

Definition 1.2 (Asymptotic Upper Bound O)**.**
We denote $f \in O(g)$ *if and only if* there exists constants $c > 0$ and n_0 such that for all $n \geq n_0$, $f(n) \leq cg(n)$.

Remark: c does not depend on n. To show that $f \in O(g)$,

1. Choose suitable constants c and n_0.
2. Show that $f(n) \leq cg(n) \forall n > n_0$.
3. QED

Similarly we have the following two definition for asymptotic lower and tight boudns.

Definition 1.3 (Asymptotic Lower Bound Ω)**.**
We denote $f \in \Omega(g)$ *if and only if* there exists constants $c > 0$ and n_0 such that for all $n \geq n_0$, $f(n) \geq cg(n)$.

Definition 1.4 (Asymptotic Tight Bound Θ)**.**
We denote $f \in \Theta(g)$ if and only if $f \in O(g)$ and $f \in \Omega(g)$.

Remark: By rights, the definition should contain only the *if* direction. However, the following theorem shows that the we can partition the relationship between f and g into three mutually exclusive cases, which then proves the only if cases.

Theorem 1.1.
Suppose $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$.

(1) If $c = 0$, then $f \in O(g)$ but $f \notin \Omega(g)$.

(2) If $c = \infty$, then $f \in \Omega(g)$ but $f \notin O(g)$.

(3) If $o < c < \infty$, then $f(n) \in \Theta(g)$.

With this theorem, we can employ limit theorem to prove the bounds.
In the previous definitions, we only need **one** pair of permissible (c, n_0) pair. The next two definitions requires a permissible n_0 for **all** values of c.

Definition 1.5 (Small o and ω Notation)**.**
We denote $f \in o(g)$ if for all $c > 0$, there exists a x_0 such that for all $x > x_0$, $f(x) \leq cg(x)$.
We denote $f \in \omega(g)$ if for all $c > 0$, there exists a x_0 such that for all $x > x_0$, $f(x) \geq cg(x)$.

1.2 Recurrence Relations

Simple recurrence relations can be solved in methods covered in `CS1231.pdf`.
For difficult recurrence relations, **induction** is the more permissible method by making an intelligent guess and proving it.
The following master theorem solves bounds for recurrences in the form of

$$T(n) \leq aT(\frac{n}{b}) + f(n) \quad \text{where } f(n) = O(n^k)$$

Here, a, b, k are constants directly read off from the recurrence.

Theorem 1.2 (Main Recurrence Theorem)**.**
For recurrence theorems of the above form, we have

$$T(n) = \begin{cases} O(n^k) & \text{if } a < b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^{\log_b a}) & \text{if } a > b^k \end{cases}$$

The theorem is also true for lower bounds and tight bounds, by changing to the corresponding bounds notations.

2 Brute Force

Brute force is a problem solving paradigm that directly tackles the problems without worrying about costs.

2.1 Brute Force Sorting

Sorting by brute force include selection sort, bubble sort and insertion sort.
In this section, sorting are performed on 1-indexed array $A[1..n]$, and we sort in increasing order.

2.1.1 Selection Sort

Idea:

- In each iteration, put the ith smallest element in $A[i]$, by swapping the said element with the out-of-order element in $A[i]$.
- Since before the ith iteration, the first $i-1$ elements are the $(i-1)$th smallest elements in the correct order, to find the ith smallest, just traverse $A[i..n]$ and choose the smallest element in it.

Analysis:

- Selection sort runs in $\Theta(n^2)$ for best, worst and average cases.
- Selection sort is in place.
- Selection sort is not stable.

2.1.2 Bubble Sort

Idea:

- In each iteration, swap pairwise elements which are out of order, from beginning to end.
- After ith iteration, the last i elements are sorted.
- In kth iteration, the largest element unsorted(i.e., in $A[1..n-k+1]$) will be eventually swapped to $A[n-k+1]$.

Analysis:

- Bubble sort runs in $\Theta(n^2)$ for best, worst and average cases.
- In place.
- Stable.

2.1.3 Insertion Sort

Idea:

- In ith iteration, choose $A[i]$ and compare with all elements in $A[1..i-1]$ from $A[i-1]$ to $A[1]$ and insert into the correct place.
- After i iterations, $A[1..i]$ is sorted.

Analysis:

- Best case: $O(n)$. Worst case: $O(n^2)$.
- In place
- Stable

2.2 Sequential Search

Sequential search is done by examining all the elements in an array in sequence in looking for an element.
It has best case $O(1)$ and worst case $O(n)$.

2.3 String Matching

In String matching, a pattern described by $P[1..m]$ is to be matched in a string described by $A[1..n]$.
Brute force matching requires examination of m characters for the first $n-m$ characters.
It has best case $O(m)$ and worst case $\Theta(mn)$.
It has (surprisingly good) average case $\Theta(n)$.

2.4 All Pair Closest Pair

In closest Pair problem, there is a set of n points in m-dimensional space.
Brute force method requires computation of all distances, which runs in $\Theta(n^2m)$.

2.5 Convex Hull

In convex hull problem, it is required to find the smallest convex hull from a given set of $n \geq 3$ points.
Brute force method requires to union all the lines formed by a pair of points which ensures all other points are on the same side. It runs in $\Theta(n^3)$.

2.6 Exhaustive Search

Exhaustive search enumerates all the possible cases and consider all of them.
Some famous problems under exhaustive search method include Travelling Salesman, Knap Sack and more.

3 Divide and Conquer

Divide and conquer is another problem-solving paradigm which

1. Divide the problem into parts
2. Solve each part
3. Combine the solutions

Complexity of Divide and Conquer is usually of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

3.1 Merge Sort

Idea:

1. Divide the array into two parts of minimal size difference
2. Apply merge sort on each of the part (recursively)
3. Merge the two sorted parts.

During merge of $(A \mid B)$, to ensure stability, the condition used should be of the form $A[i] \leq B[j]$. **Analysis**:

- Worst case time complexity: $\Theta(n \log n)$. Average case also $\Theta(n \log n)$.
- Stable
- Not in-place. Merge sort uses extra space for merging: $O(n)$.

3.2 Quick Sort

Idea:

1. Choose the first element in the array as pivot element
2. Partition the arrays into two parts: (1) numbers $<$ pivot **and** (2) numbers $\geq$ pivot.
3. Quick sort the two partitions (recursively).

The implementation of `partition` is quite intricate. **Analysis**:

- Worst case complexity is $O(n^2)$, and it happens when the array is already sorted.
- Best case: $O(n \log n)$, and it happens when each round of partition divides the two parts into if possible, equal or nearly equal (difference ≤ 1) size.

3.3 Randomised Quick Sort

Randomised quick sort is an improved variant from the quick sort, where the pivot element is now chosen randomly and swapped with the first element, before we apply the same `partition` routine. **Analysis**:

- Worst-case runtime is still $O(n^2)$.
- Average-case runtime is now $\Theta(n \log n)$.
- In place. Space efficiency is $O(n)$.
- Not stable, due to the swapping protocol. Example: $3, 3, 2$ with first pivot 2.

In fact, by the method of decision tree, we can show that any comparison-based sorting algorithm is $\Omega(n \log n)$.

3.4 Binary Search

Binary search takes in a sorted array. Therefore, only one half of the array will be relevant to continuation for every iteration.
Idea:

1. Find the element at the middle of the array.
2. Compare with the key K to decide which subarray to continue.
3. Binary search on the subarray.

Analysis:

- Time complexity $O(\log n)$.

3.5 Tiling Problem

Theorem 3.1.
Let $n = 2^k$ for some $k \in \mathbb{N}$. A $n \times n$ square board with one of the 1×1 square missing can be tiled by tromino.

Theorem 3.2.
For $n > 11$, any $n \times n$ board with one square missing can be tiled with trominoes if

- n is odd, and
- n is not a multiple of 3

3.6 Binary Tree

Definition 3.1 (Binary Tree)**.**
Binary tree is either empty or consists of root, left binary subtree and right binary subtree.

Definition 3.2 (Full Binary Tree)**.**
A binary tree is a **full** binary tree if there are either no or two children for each node.

Theorem 3.3. *For a full binary tree, number of leaves l and number of internal nodes i follows the relation of*

$$i = l - 1$$

3.6.1 Height of Binary Tree

Base case: Height of single node is 0.
Height of a tree is -1 if the tree is `null` and is max{height(left subtree), height(right subtree)}+ 1 otherwise.
Analysis: Total complexity $\Theta(n)$.

3.6.2 Traversal

There are three types of traversal:

- Preorder, where node is traversed before its children
  `if(T` $\neq$ `null) {visit(T); preorder(left(T)); preorder(right(T))}`
- Inorder, where node is traversed in between its children
  `if(T` $\neq$ `null) {inorder(left(T)); visit(T); inorder(right(T));}`
- Postorder, where node is traversed after its children
  `if(T` $\neq$ `null) {postorder(left(T)); postorder(right(T)); visit(T);}`

3.7 Multiplication of Large Numbers

Suppose x, y are two base-10 number of n digit long, and n is a power of 2. Then we can perform $x \times y$ fast by

- Write $x = x_1 \times 10^{\frac{n}{2}} + x_0$, i.e., x_1 is the first half of x, where x_0 is the second half. Write y in the similar fashion $y_1 \times 10^{\frac{n}{2}} + y_0$.
- Then $z = x \times y = (x_1 \times 10^{\frac{n}{2}} + x_0)(y_1 \times 10^{\frac{n}{2}} + y_0) := z_2 \times 10^n + z_1 \times 10^{\frac{n}{2}} + z_0$.
- Here $z_2 = x_1 \times y_1$, $z_0 = x_0 \times y_0$.
- The trick to obtain z_1 is $z + 1 = (x_1 + x_0) \times (y_1 + y_0) - (z_2 + z_0)$.

Analysis:

- $M(n) = 3M(\frac{n}{2})$, for $n > 1$ and $M(1) = 1$.

- $A(n) = 3A(\frac{n}{2}) + cn$, for $n > 1$ and $A(1) = 1$.
- $M(n) \in \Theta(n^{\log_2 3})$ and $A(n) \in \Theta(n^{\log_2 3})$.

This is faster than the naive $\Theta(n^2)$ algorithm learnt in primary school.

3.8 Multiplication of Matrix

The idea used for large number multiplication is also applicable to matrix multiplication. Note that the naive multiplication by row and column expansion is of $O(n^3)$ time complexity.

Theorem 3.4 (Strassen's Algorithm)**.**
Suppose A, B are two matrix of dimension $n \times n$ and n is a power of 2. We divide A, B into four submatrices:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

and conquer the multiplication by calculating

$$\begin{aligned} q_1 &= (a_{11} + a_{22} \times (b_{11} + b_{22})) \\ q_2 &= (a_{21} + a_{22}) \times b_1 1 \\ q_3 &= a_{11} \times (b_{12} - b_{22}) \\ q_4 &= a_{22} \times (b_{21} - b_{11}) \\ q_5 &= (a_{11} + a_{12}) \times b_{22} \\ q_6 &= (a_{21} - a_{11}) \times (b_{11} + b_{12}) \\ q_7 &= (a_{12} - a_{22}) \times (b_{21} + b_{22}) \end{aligned}$$

which gives

$$AB = \begin{pmatrix} q_1 + q_4 - q_5 + q_7 & q_3 + q_5 \\ q_2 + q_4 & q_1 + q_3 - q_2 + q_6 \end{pmatrix}$$

Analysis:

- $M(n) = 7M(\frac{n}{2})$, $M(1) = 1$, so $M(n) \in \Theta(n^{\log_2 7})$.
- $A(n) = 7A(\frac{n}{2}) + O(n^2)$, $A(1) = 0$, so $A(n) \in \Theta(n^{\log_2 7})$.

3.9 Finding Closest Pair of Points on a Plane

Given n points on a plane where a, b are non-negative rational numbers, one closest pair of points can be chosen using the following algorithm:

1. Find x_c, the median of the x-coordinates of the points.
2. Divide the points into two groups of (nearly) equal size based on them having x-coordinate $\leq x_c$ or $\geq x_c$. The way to divide points with $x = x_c$ needs to be consistent throughout all divisions.

3. Find closest pair among each of the two groups inductively.

4. Let the closest pair have distance δ.

5. Consider all points whic have x-coordinate between $x_c - \delta$ and $x_c + \delta$ and sort them according to the y-coordinate

6. For each point, find the distance between it and the next 7 points in the list as formed in step (5).

7. Report the shortest distance among all the distance found above, and δ.

The proof of correctness relies on the partition of $[x-\delta, x+\delta] \times [y, y+\delta]$ into 8 $[0, \frac{\delta}{2}] \times [0, \frac{\delta}{2}]$ subregions, and we cannot have duplicate points within each region due to the minimum distance δ constraint. **Analysis**: If sorting of all the points is done beforehand, we have $T(n) \leq 2T(\frac{n}{2}) + cn$ which in turn gives $T(n) \in O(n \log n)$.

4 Dynamic Programming

The basic idea of dynamic programming is memoisation. Usually relation of $f(n)$ with other values can be described nicely in a recursion.

4.1 Fibonacci Numbers

$$F(n) = F(n-1) + F(n-2), \;\; F(1) = F(2) = 1$$

We can solve $F(n)$ using $O(1)$ space and $O(n)$ time using iteration with two shifting temporary variables.

4.2 Binomial Coefficients

$$C(n,k) = C(n-1,k-1) + C(n-1,k), \text{ for } n > k > 0$$

Base case is $C(n,0) = C(n,n) = 1$.
Analysis: Both time and space complexity is $\Theta(nk)$.

4.3 Transitive Closure

Transitive closure can be used to determine if there is a non-trivial directed path from node i to node j.

Theorem 4.1 (Warshall's Algorithm)**.**
Suppose A is the adjacency matrix of the directed graph, let $R^k(i,j)$ denotes whether there is a path from i to j which has intermediate vertices only among $\{1,2,\ldots,k\}$. We have

- $R^0(i,j) = A$.
- $R^k(i,j) = 1 \Rightarrow R^{k+1}(i,j) = 1$.
- Also, $R^k(i,k+1) = 1$ and $R^k(k+1,j) = 1$ implies $R^{k+1}(i,j) = 1$.

Therefore, R^0 up to R^{n-1} can be computed.
Analysis: Time complexity $O(n^3)$.

4.4 All Pair Shortest Path

We here assume that there is no negative circuits in teh graph. Otherwise, the all pair shortest path will be undefined due to $-\infty$ cost value for some of the path.

Theorem 4.2 (Floyd's Algorithm)**.**
We define $D_k[i,j]$ as the length of the shortest path from i to j where the intermediate vertices are all $\leq k$.
Clearly, $D_0[i,j] = A[i,j]$. Here we assume $A[i,j] = \infty$ if there is no path from i to j and $A[i,i] = 0$ for all i.
The idea is to loop through k, then i and j, and if $D[i,j] > D[i,k] + D[k,j]$,

$$D[i,j] \leftarrow D[i,k] + D[k,j] \text{ if } D[i,j] > D[i,k] + D[k,j]$$
$$\text{next}[i,j] \leftarrow \text{next}[i,k]$$

4.5 0/1 Knapsack Problem

It is a simpliciation of the general knapsack problem, where the weights $W_1, \ldots, W_n$ of objects $O_1, \ldots, O_n$ all have integral values. Further assume objects have values $V_1, \ldots V_n$. The maximum capacity is C.
We hope to find a set $S \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in S} W_i \leq C$ and $\sum_{i \in S} V_i$ is maximised.
The base case is $F(C, 0) = 0$. If $W_j \leq C$ then, $F(C, j) = \max(F(C, j-1), F(C - W_j, j - 1) + V_j)$; else, $F(C, j) = F(C, j-1)$.
We fill the DP table by looping through $s = 0$ to C then $j = 1$ to n.
We maintain whether an item is used in Used(s, j), and its value equals `!F(s,j)==F(s,j-1)`.
We can backtrack the item picked by looping down $j = n$ to 1 of Used(`left`, j) and update `left=left-`W_j if Used(`left`, j) is `true`.

4.6 Coin Changing using DP

Given some denominations $d[1] > d[2] > \cdots > d[n] = 1$, we want to find the minimal number of coins needed to make change for certain amount S.
Let $C[i, j]$ denote the number of coins needed to obtain value j, when one only uses coins $d[i], \ldots, d[n]$. Clearly, $C[n, j] = j$, for $0 \leq j \leq S$.
We also have $C[i-1, j] = \min(1 + C[i-1, j - d[i-1]], C[i, j])$. We set used$[i-1, j]$ to be `true` if the former term is fewer.
The algorithm runs in $\Theta(S \times n)$.
To get the number of each type of coins used, we look at denominator $d[1]$ first, and recurse up to $d[n]$.

4.7 Matrix Multiplication

Suppose we have matrices M_j with r_j rows and c_j columns, for $1 \leq j \leq n$, where $c_j = r_{j+1}$m for $1 \leq j < n$. We want to compute $M_1 \times \cdots \times M_n$ using the least number of multiplication.
Let $F(i, j)$ denote the minimum number of operations needed to compute $M_i \times \cdots \times M_j$, then $F(i, j)$ is minimal over $k (i \leq k < j)$ of $F(i, k) + F(k+1, j) + \underbrace{r_i c_k c_j}_{\text{cost of multiplication the two components}}$.
The base case is $F(i, i) = 0$ for $1 \leq i \leq n$. We calculate $F(i, j)$ in the skew diagonal fashion.
The time complexity is $O(n^3)$.

4.8 Longest Common Subsequence

Definition 4.1 (Subsequence)**.**
A subsequence of a sequence $a[1], a[2], \ldots, a[n]$ is a sequence of the form $a[i_1], a[i_2], \ldots, a[i_k]$ such that $1 \leq i_1 < i_2 < \cdots < i_k \leq n$.

In the problem of Longest Common Subsequence, we have two string $x[1..n]$ and $y[1..m]$. We hope to find the longest common subsequence of the above two sequences.
The idea is

- If $x[n] = y[m]$, the longest common subsequence is (the longest common subsequence of $x[1..n-1]$ and $y[1..m-1]$)$\oplus x[n]$.

- If not, then the longest common subsequence is the longer of
  - the longest common subsequence of $x[1..n]$ and $y[1..m-1]$ or
  - the longest common subsequence of $x[1..n-1]$ and $y[1..m]$.

Let $F(i,j)$ be the length of longest common subsequence of $x[1..i]$ and $y[1..j]$. The base case is $F[i,j]=0$ for either $i=0$ or $j=0$. The recursive case is described above.
The time complexity for this algorithm is $\Theta(mn)$.
The longest common subsequence can be recovered by the information of F.

5 Greedy Algorithm