COMP3821

Extension Algorithms and Programming Techniques

Charlie Bradford

May 10, 2018

1 Introduction

2 Algorithm Analysis

2.1 Asymptotic Behaviour

$f(n) = O(g(n))$ $\exists n_0, c \in \mathbb{R}(\forall n > n_0(0 \leq f(n) \leq cg(n)))$ i.e. g(n) is the worst case.

$f(n) = \Omega(g(n))$ $\exists n_0, c \in \mathbb{R}(\forall n > n_0(0 \leq cg(n) \leq f(n)))$ i.e. g(n) is the best case.

$f(n) = \Theta(g(n))$ $f(n) = O(g(n)) \wedge f(n) = \Omega(g(n))$ i.e. f(n) has the same best and worst case growth.

We also have the notation $f(n) = o(g(n))$. This is used when $g(n)$ is not asymptotically tight. For example $n \neq O(n^2)$ but $n = o(n^2)$. $f(n) = \omega(g(n))$ is used similarly.

2.2 The Master Theorem

Let:
$a, b \in \mathbb{Z}, a \geq 1 \wedge b > 1$
$f(n) \geq 0 \wedge f'(n) \geq 0$
$T(n) = aT(\frac{n}{b}) + f(n)$
Then:
$\exists \epsilon \geq 0(f(n) = O(n^{log_b a - \epsilon})) \Rightarrow T(n) = \Theta(n^{log_b a})$
$\exists \epsilon \geq 0(f(n) = \Theta(n^{log_b a})) \Rightarrow T(n) = \Theta(n^{log_b a} log_2 n)$
$\exists \epsilon \geq 0(f(n) = \Omega(n^{log_b a + \epsilon})) \wedge \exists c < 1(af(\frac{n}{b}) \leq cf(n)) \Rightarrow T(n) = \Theta(f(n))$

2.3 Fast Integer Multiplication

If we are multiplying two n-digit numbers, A and B, we first split them into halves

$$A = A_1 10^{\frac{n}{2}} + A_0$$

$$B = B_1 10^{\frac{n}{2}} + B_0$$

Then we can calculate AB:

$AB = A_1 B_1 10^n + (A_0 B_1 + A_1 B_0) 10^{\frac{n}{2}} + A_0 B_0$

Here we have halved the size of each multiplication (reducing complexity by fourfold)

but quadrupling the number of multiplications

$= A_1 B_1 10^n + ((A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0) 10^{\frac{n}{2}} + A_0 B_0$

Now we have reduced the number of multiplications, leading to a lower complexity than $O(n^2)$

Now we have $T(n) = 3T(\frac{n}{2}) + cn$. $f(n) = cn$ as addition is linear. $log_2 3 \sim 1.6$ so $cn = O(n^{log_2 3})$. Therefore the first case of the master theorem applies and $T(n) = \Theta(n^{log_2 3})$.

Theoretically we could keep making the algorithm faster by splitting the number into more bits, but the constant factors start to get so large that the performance is to slow with reasonable n.

2.4 Problems

Put examples.

3 Divide-And-Conquer Method

3.1 Weighing Coins

Problem: we have nine coins and one is lighter than the others. How do we find the lighter by using a scale only twice times. Solution: Weigh three of the coins against three more. Select the lighter three, if they are the same then take the coins that were not eighed. Weigh one coin agaist the other, now you have the lighter coin.

3.2 Multiplying Polynomials

A polynomial of degree n is uniquely determined by the coefficients A_i. As in $A_0 + A_1x + A_2x^2 + ... + A_nx^n$. Thus we can determine the values of the coefficents using only n+1 different valuex of x.

$$P_A(x) \leftrightarrow \{(x_0, P_A(x_0)), (x_1, P_A(x_1), ..., (x_n, P_A(x_n))\}$$

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_{2n} \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & x_{2n+1} & x_{2n+1}^2 & \dots & x_{2n+1}^{2n} \\ 1 & x_{2n+1}^2 & x_{2n+1}^{2*2} & \dots & x_{2n+1}^{2*2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{2n+1}^{2n} & x_{2n+1}^{2n*2} & \dots & x_{2n+1}^{2n*2n} \end{bmatrix} = \begin{bmatrix} P_a(1) \\ P_a(x_{2n}) \\ P_a(x_{2n}^2) \\ \vdots \\ P_a(x_{2n}^{2n-1}) \end{bmatrix}$$

$$\begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_{2n} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & x_{2n+1} & x_{2n+1}^2 & \dots & x_{2n+1}^{2n} \\ 1 & x_{2n+1}^2 & x_{2n+1}^{2*2} & \dots & x_{2n+1}^{2*2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{2n+1}^{2n} & x_{2n+1}^{2n*2} & \dots & x_{2n+1}^{2n*2n} \end{bmatrix}^{-1} \begin{bmatrix} P_a(1) \\ P_a(x_{2n}) \\ P_a(x_{2n}^2) \\ \vdots \\ P_a(x_{2n}^{2n-1}) \end{bmatrix}$$

But with polynomials that have degree in the thousands, or even millions, this breaks down as not all x_i can be 1. Except it can. See below.

3.3 Fast Fourier Transform

3.4 Discrete Fourier Transform

4 The Greedy Method

A greedy algorithm creates a solution through a series of steps, choosing each movement at each step without considering the whole problem.

4.1 Activity Selection

4.2 Djikstra's Algorithm

Djikstra's algorithm finds the shortest distance between two nodes in a graph. Starting from the origin node s in a graph $G(V, E)$, algorithm maintains a set S of vertices u such that the minimum distance between s and u, $d(u)$, is known. Then for each node $v \in V - S$, we find the shortest path through S to u, where there is a single edge connecting v and u. Then, for all such nodes, we consider $d'(v) = \forall u \in S.e = (u, v), min(d(u) + l_e)$. Then once we have the pair (u, v) for which $d'(v)$ is minimal, we add v to S.

4.3 Machining Problem

Items have to be machined and then polished. One machine does the machining, and a second does the polishing. N items I, for each item I_k you know the machining time M_k and the polishing time P_k. How do you schedule the machining and polishing so that so that the entire process takes as little time as possible?

Answer: In increasing order of P_k.

Explanation: The machining machine can be run constantly so $\sum_{k=1}^{n} M_k$ is constant. So by scheduling items with the lowest P_k first we get the over as quickly as possible to free time for later items.

4.4 Discrete Knapsack Problem

4.5 Huffman Codes

4.6 Set Cover Approximation

5 Dynamic Programming

6 Network Flow Algorithms

A flow network is a directed graph, with a 'source' and a 'sink.' Network flow graphs are weighted, with each edge having a maximum capacity. A flow is a real non-negative function that maps each edge to a value $f : E \Rightarrow \mathbb{R}$. Each flow must satisfy

Capacity Constraint $\forall e(u,\ v) \in E.\ f(u,\ v) \leq e(u,\ v)$

Flow Conservation $\forall v \in V - \{s,\ t\}$.

$$\sum_{(u,v)\in E} f(u,\ v) = \sum_{(v,w)\in E} f(v,w)$$

When we create a flow in a graph we must reduce the capacity in the direction of the flow and create a virtual flow in the opposite direction. This represents the potential to reduce flow through the original edge.

6.1 Ford-Fulkerson Algorithm

- Keep adding flows until there is no residual flow
- Flows that can be added are called augmenting flows
- When there is now residual flows, you have achieved maximal flow
- We know that this is a valid tactict because of the notion of a minimal cut
- Slow - Exponential

6.2 Minimal Cut

- A cut is a partitioning of the flow graph with the source on one side and the sink on the other
- The maximum flow across a cut is the sum of all flows towards the sink less the sum of all flows towards the source
- The cut with the lowest maximum flow is the bottleneck of the flow graph
- The maximal flow of the graph is equal to the flow across this bottleneck
- When there is no residual flow across the graph then the bottleneck has been completely filled and we have found the max flow

6.3 Edmonds-Karp Algorithm

- Same as Ford-Fulkerson but the flow that uses the minimum number of edges is selected first
- Still pretty slow - $O(|V||E|^2)$

6.4 Multiple Sources & Sinks

- Add super sources and sinks
- These connect to all sources and sinks and edges can ahve infinite capacity
- Some edges can have secondary weights such as cost of transport
- There are algorithms to find the cheapest of all max flows.

6.5 Other problems

6.5.1 Maximal Matching of Bipartite Graphs

- Add super source connecting to one side
- Add super sink connecting to the other
- Set all edge capacitites to one
- Find maximal flow
- The maximal matching graph is the subgraph of all edges with flow

6.5.2 Movie Rental

Assume you have a movie rental agency. At the moment you have k movies in stock, with m_i copies of movie i. Each of n customers can rent out at most 5 movies at a time. The customers fave sent you their preferences which a list of movies they would like to see. Your goal is to dispatch the largest possible number of movies.

- Form a bipartite graph with customers on source side and movies on sink side
- Super source with capacity 5 for each customer
- Each customer linked to preferred movies with linked one (can be secondarily weighted)
- Find maximal flow, graph shows who gets what

6.5.3 Computer Network

On one side of a graph you have several servers and on the other you have end users, there are also many router nodes in between. You need to shut down flow from servers to hosts, which links need to be shut down. Shutting down a link incurs a cost so you need to shutdown the fewest possible.

- Create super source connecting to all the servers and a super sink connecting to all the end users
- Give all link a weight of one
- Find the max flow graph
- The verticies accessible from the sink via augmenting paths are one side of the partition, non-accessible verticies are the other side of the partition.

7 String Matching

If there is a need to find a contiguous substring in a much longer string, character comparison algorithms get very complicated if the either string cannot fit in a register. There are several was to resolve this.

7.1 Rabin-Karb

To find the string B in the much long string A we would first map each letter in their alphabet $\mathbb{A}$ to an integer. We now define the primary hashing function $h(x)$ to be

$$\begin{aligned} h(B) &= h(b_1 b_2 b_3 ... b_m) && (1)\\ &= d^{m-1}b_1 + d^{m-2}b_2 + ... + d^1 b_{m-1} + b_m && (2)\\ &= b_m + d(b_{m-1} + d(b_{m-2} + ... + d(b_2 + db_1)...)) && (3)\\ & && (4) \end{aligned}$$

Now we pick some large prime number p such that $(d-1)p$ fits in a single register. Then we define the final hashing function to be $H(x) = h(x) \textbf{ mod } p$.

Now, taking A_s to be the substring of A starting as character s and of the same length as B, we compute $H(A_s)$. In order to save time and complexity we need only computer $H(A_1)$ then we can iterate through the relation $H(A_{s+1}) = (d * H(A_s) - (d^m \textbf{ mod} p)a_s + a_{s+m}) \textbf{ mod } p$. Now we need only do a character by character matching if $H(A_s) = H(B)$.

8 Linear Programming

9 Intractable Problems and Approximation Algorithms

10 Randomisation

10.1 Random Select

10.2 Linear Time for Order

10.3 Hash Functions

10.4 Skip Lists

- Like a doubly linked list but certain nodes have different heights (up to max $log_2 n$ for n-element list)
- Searching for k:
  1. Start at head
  2. Move to the node pointed to at the highest level
  3. If the value is smaller than k go to 2.
  4. If the value is equal to k return
  5. Move the the node pointed to at the next highest level, go to 3.
  6. Expected time $O(log_2 n)$
- Insertion node with value k:
  - Find the correct location for k
  - The height, i, for k's node is $\frac{1}{2^i}$
  - The node is linked from the bottom up
  - Expected time if $O(log_2 n)$ for searching and $O(1)$ for inserting
- Deletion:
  - As in a doubly linked list
  - Sort out all pointers from the bottom up