COMP3121/9101

Algorithm Design and Analysis

Recitation 3

Divide & Conquer 2

Announcements

- Carefully, go through the recitation problems and solutions.

1 Definitions

- Recurrence relations regularly arise when trying to estimate the time complexity of an algorithm that uses the divide and conquer paradigm. Suppose we have some problem P we wish to solve with input size n. If a divide and conquer algorithm
  - **divides** the problem into a sub-problems, each of size n/b
  - has a **combine** step with time complexity $O(f(n))$

  then its time complexity is the solution to the recurrence

$$T(n) = aT(n/b) + f(n)$$

- In this course we generally assume that simple numerical operations $(+, -, \times, \div)$ can all be computed in constant time, but the time of these operations can be significant when considering large numbers. Recall that an integer of value N has a *size* of $\log_2 N$ bits, and is represented in decimal form by $\log_{10} N$ digits. The time complexity of naively multiplying two n digit numbers is $\Theta(n^2)$.
  - The *Karatsuba Trick* is a divide and conquer algorithm that splits the integers into upper and lower halves. It can be used to multiply two n digit integers in $\Theta(n^{\log_2 3})$ time.

2 The Master Theorem

The *Master Theorem* is a tool that allows us to quickly simplify recurrences of the form

$$T(n) = aT(n/b) + f(n)$$

that fall into one of 3 cases depending on the values of a and b, and the function $f(n)$. We define $c^* = \log_b a$ as the critical exponent, and n^{c^*} as the critical polynomial.

- Case 1: The branching factor outweighs the work done in the combine step. If $f(n) = O(n^{c^*-\epsilon})$ for some $\epsilon > 0$ then $T(n) = \Theta(n^{c^*})$.
- Case 2: The branching factor is roughly equivalent to the amount of work done in the combine step. If $f(n) = \Theta(n^{c^*} \log^k n)$ for some $k \geq 0$, then $T(n) = \Theta(n^{c^*} \log^{k+1} n)$
- Case 3: The branching factor is outweighed by the amount of work done in the combine step. If $f(n) = \Omega(n^{c^*+\epsilon})$ for some $\epsilon > 0$, and for some $k < 1$, and some n_0

$$af(n/b) \leq kf(n)$$

  holds for all $n > n_0$, then $T(n) = \Theta(n^{c^*})$.

Solve the following recurrences using the Master Theorem, or explain why they do not satisfy any of the 3 cases:

(a) $T(n) = 5T(n/4) + 2n$

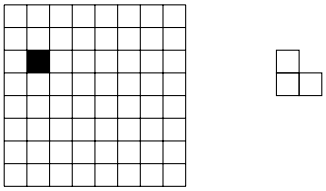
(b) $T(n) = 9T(n/3) + n^2 \log n$

(c) $T(n) = 4T(n/2) + n^3$

(d) **[Hare Only]** $T(n) = T(n/2) + 2n - n\cos n$

3 Divide & Conquer

Let n be a power of two. You are given an $n \times n$ board with one of its cells missing (i.e., the board has a hole). The position of the missing cell can be arbitrary. You are also given a supply of "trominoes", each of which can cover three cells as below.



(a) Design a divide and conquer algorithm to fill the $n \times n$ board (with the missing cell) with trominoes.

- Be sure to discuss what the divide, conquer, and combine steps are.
- How do each of these steps together construct the overall algorithm?
- It might help to look at small values of n, and use previous constructions to build your algorithm.

(b) Justify the correctness of your algorithm.

- What is the base case? Does your algorithm correctly solve the base case?
- If smaller instances are solved correctly, does the "combine" step ensure the correctness for the parent instance?

(c) Analyse the time complexity of your algorithm using the Master Theorem.

4 Polynomials, Convolution & the FFT

- A *polynomial* is a function of the form

$$P_A(x) = A_n x^n + A_{n-1} x^{n-1} + \cdots + A_1 x + A_0$$

The main operations that we can do to polynomials are addition, evaluation at a point x_0, and multiplication. In this coefficient form given above, addition and evaluation can be done in $O(n)$ time, but multiplying two polynomials takes $O(n^2)$ time. How can we make this faster?

- Polynomials of degree n can be uniquely represented by their values at $n+1$ distinct points. In a value-representation, two polynomials can be quickly multiplied in $O(n)$ time, by simply multiplying the values at the $n+1$ points. If we can quickly convert a polynomial to a value representation and back, we can do the multiplication in much faster than $O(n^2)$ time!
- The *Discrete Fourier Transform* converts a polynomial from coefficient representation to a value representation, by evaluate our polynomial at each of the nth roots of unity. Specifically, for a sequence $A = \langle A_0, A_1, A_2, \ldots, A_n \rangle$, the DFT of the sequence is given by

$$\hat{A}_k = P_A(w_n^k)$$

for all $0 \leq k \leq n-1$. This is sometimes denoted as $\mathcal{F}[A]$.

- The *Fast Fourier Transform* (FFT) is an $O(n \log n)$ algorithm for computing the DFT of a sequence of length n. The *Inverse Fast Fourier Transform* (IFFT) is an $O(n \log n)$ algorithm for computing the Inverse DFT of a sequence of length n.
- This means we can multiply two polynomials in $O(n \log n)$ time by:

  - Use the FFT to find the DFT of both sequences of coefficients.
  - Multiply the result element-wise.
  - Use the IFFT to take the inverse DFT of the product.

- For two sequences $A = \langle A_0, A_1, A_2, \ldots, A_n \rangle, B = \langle B_0, B_1, B_2, \ldots, B_n \rangle$ representing coefficients of polynomials $P_A(x) = \sum_{i=0}^{n} A_i x^i$, $P_B(x) = \sum_{i=0}^{n} B_i x^i$ respectively, the coefficient sequence of their product polynomial is called the *convolution* of A and B.

$$(A * B)_k = \sum_{i=0}^{k} A_i B_{k-i}$$

- Although we teach convolutions purely from the perspective of polynomial multiplication, they are one of the most important operations for interacting with the real world. In particular, convolutions are critically important in:
  - Image processing.
  - All time-series data such as audio recordings, health data, financial data.
  - Compression algorithms for real data.
  - Convolutional Neural Networks.

Let S be a sequence of length n containing only the values 0 and 1. For any integer $1 \leq p \leq n-1$, a pair of indices $1 \leq i < j \leq n$ is called a p-inversion if $S[i] = 1, S[n-j] = 0$ and $j - i = p$. Design an algorithm that runs in $O(n \log n)$ time to compute the number of $p - inversions$ for all values of p from 1 to $n-1$.