CSE 317: Design and Analysis of Algorithms

Shahid Hussain

Week 3: September 2, 4: Fall 2024

Recursion and Induction

Integer Exponentiation

- Given $x \in \mathbb{R}$, $x \neq 0$, and $n \in \mathbb{N}$, compute x^n
- Straight forward: $x^n = x \cdot x \cdot x \cdots x$ (n times)
- Time complexity: $\Theta(n)$ (exponential in size of n)
- Can we do better?
- Yes, using recursion
- Suppose $m = \lfloor n/2 \rfloor$, and we know how to compute x^m
- Then: x^n can be computed as following:

$$x^n = \begin{cases} (x^m)^2 & \text{if } n \text{ is even} \\ x \cdot (x^m)^2 & \text{if } n \text{ is odd} \end{cases}$$

- Time complexity: $\Theta(\log n)$

Generating Permutations

- For a sequence $1, 2, 3$ there are $3! = 6$ permutations

$$\begin{array}{ccc} 1, & 2, & 3 \\ 1, & 3, & 2 \\ 2, & 1, & 3 \\ 2, & 3, & 1 \\ 3, & 1, & 2 \\ 3, & 2, & 1 \end{array}$$

- So to generate all permutations the time complexity is dominated by actually outputting the permutations

Generating Permutations

- To generate all permutations of n elements, we can:
- Fix the first element, and generate all permutations of the remaining $n-1$ elements and then insert the first element at all possible positions
- Let $f(n)$ denote the time complexity of generating all permutations of n elements (excluding the time to output the permutations)
- Then $f(n) = nf(n-1) + n$ with $f(1) = 0$

Generating Permutations: Solving the Recurrence

- Let $f(n) = n!h(n)$ for some function $h(n)$ with $h(1) = 0$
- Then $n!h(n) = n(n-1)!h(n-1) + n$

$$\begin{aligned} h(n) &= \frac{n(n-1)!}{n!}h(n-1) + \frac{n}{n!} \\ &= h(n-1) + \frac{1}{(n-1)!} \\ &= h(1) + \sum_{j=2}^{n} \frac{1}{(j-1)!} = \sum_{j=1}^{n-1} \frac{1}{j!} \\ &< \sum_{j=1}^{\infty} \frac{1}{j!} = e - 1 \end{aligned}$$

- So $f(n) = O(n!)$
- Time complexity to output: $O(nn!)$

Evaluating Polynomials

- Let $P_n(x) = \sum_{j=0}^{n} a_j x^j$ be a polynomial of degree n in x
- Suppose we are given the sequence $a_0, a_1, \ldots, a_n$ and x and we want to compute $P_n(x)$
- This will require n exponentiations, n multiplications, and n additions (inefficient)
- We can do better using recursion

Evaluating Polynomials

- We observe that:

$$\begin{aligned}
P_n(x) &= a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 \\
&= (a_n x^{n-1} + a_{n-1}x^{n-2} + \cdots + a_1)x + a_0 \\
&= ((a_n x^{n-2} + a_{n-1}x^{n-3} + \cdots + a_2)x + a_1)x + a_0 \\
&\vdots \\
&= (\cdots((a_n x + a_{n-1})x + a_{n-2})x + \cdots + a_1)x + a_0 \\
P_n(x) &= xP_{n-1}(x) + a_0
\end{aligned}$$

- It translates into following algorithm

Evaluating Polynomials

Algorithm: EVAL-POLY$(P, a_0, a_1, \ldots, a_n, x)$

Output: $P_n(x) = \sum_{j=0}^{n} a_j x^j$

1. $p = a_n$
2. **for** $j = 1$ **to** n
3. $\quad p = xp + a_{n-j}$
4. **return** p

- Clearly the time complexity is $\Theta(n)$ (number of multiplication is linear in n)

Finding Majority Element

- Given a sequence of A of n elements
- An element in A is called *majority* if it appears more than $\lfloor n/2 \rfloor$ times in A
- For example, in the sequence $A = \langle 1, 2, 2, 2, 3, 2, 2, 1, 2 \rangle$, the element 2 is the majority element
- A straight forward (brute force) algorithm will require $O(n^2)$ comparisons
- We can do better by first sorting the sequence and then counting the number of occurrences of each element
- Time complexity: $O(n \log n)$
- We can do even better using recursion

Findning Majority Element

Observation

In a given sequence, if a majority element exists, removing any two distinct elements from the sequence does not change the majority element.

- We can use this observation to find the majority element
- By traversing the sequence and removing two distinct elements

Finding Majority Element

Algorithm: MAJORITY-ELEMENT

Input: A sequence $A = \langle a_1, \ldots, a_n \rangle$ of $n > 0$ elements

Output: The majority element in A if it exists, otherwise NONE

1. $x = \text{CANDIDATE}(1)$
2. $c = 0$
3. **for** $j = 1$ **to** n
4.   **if** $a_j = x$ **then** $c = c + 1$
5.   **else** $c = c - 1$
6. **if** $c > \lfloor n/2 \rfloor$ **then return** x
7. **else return** NONE

Finding Majority Element

Algorithm: CANDIDATE

Input: An index $m < n$

Output: A possible *candidate* element in A

1. $j = m$, $x = a_m$, $c = 1$
2. **while** $j < n$ **and** $c > 0$
3. $\quad j = j + 1$
4. $\quad$ **if** $a_j = x$ **then** $c = c + 1$
5. $\quad$ **else** $c = c - 1$
6. **if** $j = n$ **then return** x
7. **else return** CANDIDATE$(j + 1)$