Problem Set 1-4

Data Structures and Algorithms

November 1, 2022

PS1-1(9.5)

Formally prove that the INSERTIONSORT algorithm always correctly sorts the input.

PS1-2(9.5)

Build upon the observation

$$GCD(x, y) = GCD(y, x \mod y)$$

to derive an algorithm that can efficiently compute $GCD(x, y)$. You must prove the correctness of your algorithm.

PS1-3(5.1)

1. Prove or give a counterexample: for every positive constant c and every function f from non-negative integers to non-negative reals, $f(cn) \in \Theta(f(n))$.
   Counterexample: let $f(n) = 2^n$, when $c > 1$

$$\lim_{n\to\infty} \frac{f(cn)}{f(n)} = 2^{(c-1)n} \to \infty$$

2. Prove or give a counterexample: for every function f from non-negative integers to non-negative reals, $o(f) = O(f) - \Theta(f)$. Here '-' denotes the set minus operation.
   Counterexample: let

$$g(n) = \begin{cases} f(n) & n \text{ is odd} \\ f(n) \cdot e^{-n} & n \text{ is even} \end{cases}$$

   It's easy to prove that $g \in O(f), g \notin \Theta(f)$ but $g \notin o(f)$

PS1-5(8.9)

Implement a FIFO queue using two stacks.

Suppose there are two stacks S_1 and S_2. S_1 stores the elements, while S_2 is used for dequeeue operation.

- $ENQUEUE(x)$: directly push element x into stack S_1;
- $DEQUEUE()$: pop all the elements in S_1 and push them into S_2. Then, the element at the top of S_2 is the element we have to dequeue. Pop all the elements in S_1 and push them into S_2 (for further operations).

PS1-6(8.2)

Design a MINSTACK data structure that can store comparable elements and supports stack operations PUSH(x), POP() and MIN() which returns the minimum value currently stored in the data structure.

Maintain two stacks S_1 and S_2. S_1 stores the original elements. S_2 stores prefix minimum of S_1.

- $PUSH(x)$: push x into stack S_1, push $\min(x, S_2.top())$ into stack S_2;
- $POP()$: let $y = S_1.top()$ pop S_1, pop S_2, return y;
- $MIN()$: return $S_2.top()$.

A data structure supports INSERT(x) and Median() returns the median.

PS2-1(8.1)

1. Implement MERGESORT.
2. Implement MERGESORT with a linked list as input.

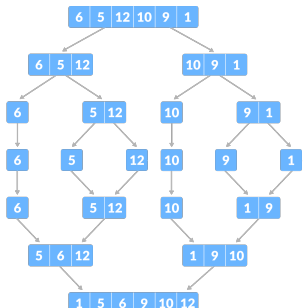

Figure: Merge sort

PS2-1(8.1)

1. Implement MERGESORT.
2. Implement MERGESORT with a linked list as input.

   For the time complexity, $T(n) = 2T(n/2) + cn = \Theta(n \lg n)$.
   For the space complexity, we use $O(n)$ space, $temp[1...n]$ to store elements during the mergence, and $O(logn)$ space used by recursive stack, So we use $O(n)$ space beside input.

PS2-3(7.6)

1. Recursively convert the decimal integer 10^n into binary.

   PWR2BIN(n):
   If n=1 then return 1010_2
   else $z \leftarrow PWR2BIN(n/2)$ and return $FASTMULTIPLY(z, z)$

   Suppose the running time of the algorithm is $T(n)$. Then we have

$$T(n) = \begin{cases} O(1) & , \quad n = 1 \\ T(n/2) + O(n^{\log_2 3}) & , \quad o.w. \end{cases}$$

   By master theorem, $a = 1$, $b = 2$, $f(n) = \Omega(n^{\log_2 3})$, we have $T(n) = O(n^{\log_2 3})$.

PS2-3(7.6)

2. Recursively convert any decimal integer x with n digits (where n is a power of 2) into binary.

   DEC2BIN(x):
   if n=1 then return binary[x]
   else Split x into two decimal numbers x_L, x_R with $n/2$ digits each return $FASTMULTIPLY(DEC2BIN(x_L), POW2BIN(n/2)) + DEC2BIN(x_R)$.

   Suppose the running time of the algorithm is $T(n)$. Then we have

$$T(n) = \begin{cases} O(1) & , \quad n = 1 \\ 2 \cdot T(n/2) + O(n^{\log_2 3}) & , \quad o.w. \end{cases}$$

   By master theorem, $a = 2$, $b = 2$, $f(n) = \Omega(n^{\log_2 3})$, we have $T(n) = O(n^{\log_2 3})$.

PS2-4(6.6)

1. Devise a variant of Karatsuba's algorithm that squares any n-digit number in $O(n^{\log_2 3})$ time, by reducing to squaring three $\lceil \frac{n}{2} \rceil$ -digit numbers.
   Let x be an n-digits number. Take $m = \lceil \frac{n}{2} \rceil$, $x = a \times 10^m + b$ $(b < 10^m)$, then

$$\begin{aligned} x^2 &= a^2 \times 10^{2m} + b^2 + 2ab \times 10^m \\ &= a^2 \times 10^{2m} + b^2 + \left(a^2 + b^2 - (a-b)^2\right) \times 10^m \end{aligned}$$

   We only need to square three $\frac{n}{2}$-bit squares $a^2, b^2, (a-b)^2$, with additional addition, multiplication in $O(n^{\log_2 3})$. So time complexity

$$T(n) = 3T(\frac{n}{2}) + O(n^{\log_2 3}) = O(n^{\log_2 3}).$$

PS2-4(6.6)

2. Is it asymptotically faster to square an n-bit integer than to multiply two n-bit integers? No.
   - On the one hand, if we have any square algorithm, we can compute multiplication in the following way
   $$xy = \frac{(x+y)^2 - (x-y)^2}{4}.$$
   Notice multiplication needs $\Omega(n)$ time, so cost of addition and division here doesn't count. Therefore, we conclude that square cannot be asymptotically faster than multiplication.(or **square is harder than multiplication.**)
   - On the other hand, if we have any multiplication algorithm, we can directly use is to calculate $x \times x = x^2$. Therefore, **multiplication is harder than square.**
   - In summary, square is as hard as multiplication.

PS2-5(11.9)

1. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

   Consider the following procedure:

   - For each $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$, test chip $2i$ and $2i+1$, and add chip $2i$ to set C if both reports good.
   - If $n \equiv 1 \mod 4$, add chip n to set C.

   It can be verified that

   - In set C, the number of good chips is strictly larger than bad chips;
   - $|C| \leq \lceil \frac{n}{2} \rceil$.

PS2-5(11.9)

3. Show that the good chips can be identified with $O(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

   Recursively apply procedure in (a). Hence,

   $$T(n) = T(n/2) + O(n),$$

   which implies $T(n) = O(n)$.

PS2-5(11.9)

4. Prove that if more than n/2 chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor

   We assume that there are b bad chips $B_1, B_2, \ldots, B_b$, and g good chips $G_1, G_2, \ldots, G_g$. It is impossible to distinguish B_1 and G_1 if
   - For each $1 \leq i \leq g$, chip B_i will report chip X good if and only if $X = B_j$ for some $1 \leq j \leq g$;
   - For each $g + 1 \leq i \leq b$, chip B_i will report chip X bad in any circumstances.

PS3-1(12.3)

Give asymptotic upper and lower bounds for T(n)

- (a)(b)(c)(d): directly apply master theorem.
- (e): $T(n) = \Omega(n)$, $T(n) = O(\sum_{i=0}^{\log n} \left(\frac{7}{8}^i\right) \cdot n) = O(n)$
- (f): $T(n) = \log n! = \Theta(n \log n)$
- (g): Let $n = 2^k$, $g(k) = \frac{T(2^k)}{2^k}$, then we have $g(k) = g(\frac{k}{2}) + 1$, apply master theorem.

PS3-2(8.6)

1. Give an asymptotic tight bound for the recurrence $T(n) = T(d) + T(n-d) + cn$, where constants $c, d \in \mathbb{N}^+$. $O(n^2)$
2. Give an asymptotic tight bound for the recurrence $T(n) = T(dn) + T((1-d)n) + cn$, where $d \in (0,1)$ is a constant and $c > 0$ is also a constant. $O(n \log n)$

PS3-3(OJ)

Find the smallest element in an n-element bitonic array in $O(\log n)$ time.

Use a variant of binary search.

PS3-4(OJ)

Find whether the majority element exists. If so ,find the majority element.

- An $O(n \log n)$ algorithm: divide and conquer.
- An $O(n)$ algorithm: Boyer Moore Voting Algorithm with an additional check.

PS3-5(8.1)

1. Prove that in a binary heap containing n nodes, there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h. Prove by induction.
2.) Implement the HEAPUPDATE(i, val) operation in a binary max-heap containing n nodes, which changes the value of the element at index i to val. You should check both if can be swap up and down.

PS3-6(8.6)

Devise an $O(nlgk)$-time algorithm to merge these k sorted lists into one sorted list.

Using heap or divide and conquer.

PS3-7(1.7)

Prove that when all elements are distinct, the running time of HEAPSORT is $\Omega(n \log n)$.
Proof: Consider a perfect tree T contains $n = 2^k - 1$ elements of $\{1, 2, ..., 2^k - 1\}$. Define

$$L = \{1, ..., 2^{k-1}\}, \quad R = \{2^{k-1}, 2^k - 1\}$$

.

1. At least a half of the number in leaves are in L. Denote them by L'.
2. After the first half popping operation, suppose $D(i)$ denote the distance between and number i and the root. Obviously the complexity is at least

$$\sum_{i \in L'} D(i)$$

3. Since there are at most 2^i nodes with distance i to the root, and $|L'|$ is at least $\frac{n}{4}$, the amount is at least

$$\sum_{i \leq \log \frac{n}{8}} i \cdot 2^i = \Omega(n \log n)$$

PS4-1(8)

The Closest Pair Problem.

1. Correctness: prove by induction.
2. Time complexity: $T(n) = 2T(\frac{n}{2}) + n \log n$, apply master theorem, $T(n) = n \log^2 n$.
3. How to reduce to $O(n \log n)$: Presort according to y-coordinate.
   $T(n) = 2T(\frac{n}{2}) + n \log n$, we have $T(n) = n \log n$

PS4-2(8.7)

Given a list I of n intervals, specified as (x_i, y_i) pairs, return a list where the overlapping intervals are merged.

Sorted by the first index and merge accordingly.

PS4-3(8.7)

1. Describe an algorithm to sort an arbitrary stack of n pancakes using O(n) flips.
   We can move the i-th pancake to the j-th position without affecting the order of the $(j+1)$-th to n-th pancakes using constant times of flip.
2. Describe a stack of n pancakes that requires $\Omega(n)$ flips to sort.

$$P = [1, 3, 5, 7, ..., 2, 4, 6, 8, ...].$$

PS4-4(8.4)

Strange Sort

- Running time: apply master theorem, $T(n) = \Theta\left(n^{\log_{3/2} 3}\right)$.
- Correctness: prove by induction.
- Slight modification: prove by contradiction .
- Number of swap: bounded by the number of inversions, which is bounded by $\binom{n}{2}$.

PS4-5(8.8)

Improvement for quick sort.

1. Exact formula for p_i':$p_i' = \frac{(i-1)(n-i)}{\binom{n}{3}}$
2. Let $m = \lfloor \frac{n+1}{2} \rfloor$, Compare p_m with ordinary implementation: $\lim_{n\to\infty} \frac{p_m}{p_m'} = \frac{3}{2}$.
3. Define a 'good' split to mean choosing the pivot as $x = A'[i]$, where $n/3 \leq i \leq 2n/3$: $P \approx \frac{13}{27}$, $P' = \frac{1}{3}$.
4. Does the median-of-3 method reduce the (asymptotic) best-case or worst-case running time of randomized quicksort? No, but more stable.

PS4-6(OJ)

Devise an efficient divide-and-conquer algorithm to determine how many pairs of these line segments intersect.

Sorted by the p-coordinate, then the number of pairs of these line segments is the number of inversions in the q array.
Use mergesort to count the number of inversions.

The End