HW 2

Charles Liu
Advanced Data Structures - Dr. Subu Kandaswamy
February 25th 2025

---

1. (5 points) Write an algorithmic pseudocode that reverses a singly linked list in one pass of the list. The function accepts a pointer to the beginning of the list, and returns a pointer to the beginning of the reversed list. The function must not create any new nodes. You should assume that each node has only a next pointer that points to the next node in the list.

```
Node* reverseList(Node* head) {
    pCur = pHead;
    pPrev = nullptr;
    while(pCur != nullptr) {
        pTemp = pCur->pNext;
        pCur->pNext = pPrev;
        pPrev = pCur;
        pCur = pTemp;
    }
    return pPrev;
}
```

---

2. (5 points) Write an algorithmic pseudocode that detects if there is a "cycle" in a singly linked list. A cycle is defined as a sequence of links that, once entered, does not end. Sequences can be one or more in length. For example, consider a linked list that has the following sequence starting from the head of the list:
   $A \to B \to C \to D \to B \to \ldots$

   This has a cycle of sequence length 3. As another example, consider the following linked list:
   $A \to B \to C \to C \to \ldots$

This has a cycle of sequence length 1.

Your algorithm must terminate and it should return a boolean— `true` if there is a cycle, and `false` otherwise.

This is a classic *Floyd's Cycle Finding/Hare-Tortoise Algorithm*.

```
//get next pointer safely
Node* getNext(Node* pCur) {
    if (pCur == nullptr) {
        return nullptr;
    }
    else {
        return pCur->pNext;
    }
}

bool detectCycle(Node* pHead) {
    Node* fastPtr;
    Node* slowPtr;
    while (true) {
        fastPtr = getNext(getNext(fastPtr));
        slowPtr = getNext(slowPtr);
        if (fastPtr == nullptr) {
            //no cycle
            return false;
        }
        if (slowPtr == fastPtr) {
            //found cycle
            return true;
        }
    }
}
```

---

3. (3 points) For the following function $f(n) = \frac{n^3}{100} + 10n^2 + n + 3$, select all choices that apply:

Green = selected

- [x] ~~(a)~~ $f(n) = O(n^3)$

- Since we can take $c = 1, n_0 = 100$ where $f(n) \le cn^3$ for all $n \ge n_0$, this works.
  - [x] ~~(b)~~ $f(n) = \Omega(n^2)$
- Since we can take $c = 1, n_0 = 0$ where $f(n) \ge cn^2$ for all $n \ge n_0$, this works.
  - [ ] (c) $f(n) = \Theta(n^2)$
  - [x] ~~(d)~~ $f(n) = \Theta(n^3)$
- Since we can take $c_1 = 0.0001, c_2 = 1, n_0 = 100$ where $c_1 n^3 \le f(n) \le c_2 n^3$ for all $n \ge n_0$, this works.
  - [ ] (e) $f(n) = O(n^2)$
  - [x] ~~(f)~~ $f(n) = \omega(n^2)$
- Since $\lim_{n\to\infty} \frac{\frac{n^3}{100}+10n^2+n+3}{n^2} = \lim_{n\to\infty} \frac{n}{100} + 10 + \frac{1}{n} + \frac{3}{n^2} = \infty$, this works.

---

4. (6 points) Sort the following set of functions in increasing (equivalently, non-decreasing) order of their respective growth rates:

   $N, \sqrt{N}, N^{1.5}, N^2, N\log N, N\log\log N, N\log^2 N, N\log(N^2), 2/N, 2^N, 2^{N/2}, 37, N^2\log N, N^3$

   If there are functions that grow at the same/comparable rate, indicate them inside curly braces. There is no need to show proofs or give an explanation.

$2/N, 37, \sqrt{N}, \{N\log N, N\log(N^2)\}, \{N\log\log N, N\log^2 N\}, N^{1.5}, N^2, N^2\log N, N^3, \{2^{N/2}, 2^N\}$

---

5. (8 points) Using the definitions for asymptotic notations:

   i) Prove that $5\lg n = o(n\lg n)$
   ii) Prove that $2\lg n = O(\sqrt{n})$

   Note that for your proofs, you will have to show the existence of the two constants c and n_0 as per the respective asymptotic (O or o) notation's definition. If needed, it is okay if you end up using a spreadsheet calculator to compare the two functions, in order to find a combination of c and n_0 that works, like I did during the lecture time at least for one example.

   Also, recall from the lectures that $lg(n)$ is same as $\log_2(n)$.

i)

Theorem. $5 \lg n = o(n \lg n)$

Proof:

We can solve that

$$\lim_{n \to \infty} \frac{5 \log_2 n}{n \log_2 n} = \lim_{n \to \infty} \frac{5}{n} = 0$$

Thus, $n \log_2 n$ grows faster than $5 \log_2 n$, so we can find positive constants $c, n_0 \in \mathbb{R}$ where at $n \geq n_0$,

$$5 \lg n < cn \, \mathrm{lg} n$$

As such,

$$5 \lg n = o(n \lg n)$$

ii)

Theorem. $2 \lg n = O(\sqrt{n})$

Proof:

We want to show that $2 \lg n \leq c\sqrt{n}$ for $n \geq n_0$ where $c, n_0 \in \mathbb{R}$ are positive constants.
Let $c = 1$. We want to find n_0 where $n \geq n_0$ means $2 \lg n \leq \sqrt{n}$.
We can solve that $2 \lg n_0 = \sqrt{n_0}$ means $n_0 = 256$.
Taking $c = 1$ and $n_0 = 256$, we can always guarantee that $2 \lg n \leq c\sqrt{n}$ for $n \geq n_0$.
Thus, since there exists c, n_0, $2 \lg n = O(\sqrt{n})$

---

6. (8 points) Consider this sorting algorithm to sort n integers in array $A[1...n]$. First, find the smallest element and swap it with $A[1]$. Next, find the second smallest element and swap it with $A[2]$. Repeat the process until all n elements are sorted. What are the worst-case and best-case running time complexities for this algorithm? You can express your answer in terms of big-Oh ($O(.)$) or Theta ($\Theta(.)$) notation. The more precise you get the better it is. Therefore Theta is preferable.

Best and worst-case:

$\Theta(n^2)$

In either case, for every one of the n indices in the array, the algorithm needs to sift $n - k$ elements where k is the number of elements sorted. Since k goes from 0 to n while sorting, the total time complexity is

$$\sum_{k=0}^{n}(n-k) = \frac{(n+1)(n)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

.

Taking the dominant term, we get that the complexity of the algorithm is

$$\Theta(n^2)$$

Note:

If the algorithm is designed better (where it scans for sortedness so it doesn't resort already sorted elements), the time complexity of the best-case would be $\Theta(n)$ as it only needs to scan every element once.