Informatics II, Spring 2024, Solution Exercise 3

Publication of exercise: March 4, 2024
Publication of solution: March 11, 2024
Exercise classes: March 11 - March 15, 2024

Learning Goal

- Understand the efficiency of algorithms like binary search.
- Learn how to analyze algorithmic complexity and asymptotic complexity of algorithms.
- Learn how to analyze special case and correctness of algorithms.

Task 1: Linear Search and Binary Search [Easy]

Consider an array A with n distinct integers that are sorted in an ascending order and an integer t.

a) The C funtion `linear_search` traverses the integers in A, one after another, from the beginning. If t is found in A `linear_search` returns 1, otherwise 0. Complete the C funtion `linear_search`(int A[], int n, int t) in task1.c file.

```c
int linear_search(int A[], int n, int t) {
    for(int i = 0; i < n; i++) {
        if (A[i] == t) {
            return 1; // found in the array
        }
    }
    return 0; // not found
}
```

b) The C funtion `binary_search`(int A[], int n, int t) that employs binary search to find integer t in A. Reference the following pseudocode for binary search to implement the `binary_search` function. If t is found in A `binary_search` returns 1, otherwise 0. Complete the C funtion `binary_search`(int A[], int n, int t) in task1.c file.

Algo: BinSearch1(A,v)

Input: sequence $A[1..n]$ of length n, value v
Output: either index i such that $v == A[i]$ or `NIL`

l = 1; r = n;
m = $\lfloor$(l+r)/2$\rfloor$;
while $l \leq r \wedge v \neq A[m]$ **do**
  if $v < A[m]$ **then** r = m-1 **else** l = m+1;
  m = $\lfloor$(l+r)/2$\rfloor$;
if $l \leq r$ **then return** m **else return** NIL;

```c
int binary_search(int A[], int n, int t) {
    int l = 0, r = n - 1;
    int mid;
    while (l <= r) {
        mid = (int)((r - l) / 2 + l);
        // printf("%d\n", mid);
        if (A[mid] == t) {
            return 1; // found in the array
        } else if(A[mid] > t) {
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return 0; // not found
}
```

c) What are the asymptotic complexity for the C funtions `linear_search` and `binary_search`.

$O(n)$ for `linear_search`.

$O(\log_2 n)$ for `binary_search`.

d) Compile task1.c file. Run your codes with the following parameters for n and t:

- n = 1000000, t = 1000000
- n = 10000000, t = 10000000
- n = 100000000, t = 100000000.

Report the run time growth for `linear_search` and `binary_search`, respectively.

Run time for linear search grows linearly when n grows.

Run time for binary search grow logarithmically when n grows.

Task 2: Algorithmic Complexity [Easy]

Below is a pseudocode of a function named `whatDoesItDo`, which takes an array `A[1..n]` of n integers and an integer k as inputs.

Algo: `whatDoesItDo`(A, n, k)

```
result = −1000
for i = 1 to n do
    current = 0
    for j = i to n by k do
        current = current + A[j]
    if current > result then
        result = current
return result
```

Note: In the above pseudocode, `for` $j = i$ `to` n `by` k means we do not increase j by 1, but each time, we increase it by k, i.e., `j=j+k`.

a) Perform exact analysis of the running time of the algorithm.

Instruction	# of times executed	Cost
result = -1000	1	c_1
for $i := 1$ **to** n **do**	$n+1$	c_2
$current = 0$	n	c_3
for $j := i$ **to** n **by** k **do**	$\frac{n^2-n}{2k} + 2n$ *	c_4
$current = current + A[j]$	$\frac{n^2-n}{2k} + n$ **	c_5
if $current > result$ **then**	n	c_6
$result = current$	αn***	c_7
return $result$	1	c_8

* $(1 + \frac{(n-1)}{k} + 1) + (1 + \frac{(n-2)}{k} + 1) + (1 + \frac{(n-3)}{k} + 1) + ... + (1 + \frac{(n-i)}{k} + 1) + ... + (1 + \frac{(n-n)}{k} + 1) = \frac{n^2-n}{2k} + 2n$

** $(1 + \frac{(n-1)}{k}) + (1 + \frac{(n-2)}{k}) + (1 + \frac{(n-3)}{k}) + ... + (1 + \frac{(n-i)}{k}) + ... + (1 + \frac{(n-n)}{k}) = \frac{n^2-n}{2k} + n$

*** $\alpha \in [0, 1]$

$T(n) = c_1 + c_2(n+1) + c_3 n + c_4(\frac{n^2-n}{2k} + 2n) + c_5(\frac{n^2-n}{2k} + n) + c_6(n) + c_7(\alpha n) + c_8$

b) Determine the asymptotic complexity of the algorithm?

As n gets larger, the leading term in the above formula is $\frac{n^2}{k}$. Therefore, the asymptotic complexity of the algorithm is $O(\frac{n^2}{k})$.

Task 3: Asymptotic Complexity [Easy]

a) Calculate the asymptotic tight bound for the following functions and rank them by their order of growth (lowest first). Clearly work out the calculation step by step in your solution.

$$
\begin{aligned}
f_1(n) &= (2n+3)! \\
f_2(n) &= 2\log(6^{\log n^2}) + \log(\pi n^2) + n^3 \\
f_3(n) &= 4^{log_2 n} \\
f_4(n) &= 12\sqrt{n} + 10^{223} + \log 5^n \\
f_5(n) &= 10^{\lg 20} n^4 + 8^{229} n^3 + 20^{231} n^2 + 128 n \log n \\
f_6(n) &= \log n^{2n+1} \\
f_7(n) &= \log^2(n) + 50\sqrt{n} + \log(n) \\
f_8(n) &= 14400
\end{aligned}
$$

- $f_1(n) = (2n+3)! \in \Theta((2n+3)!)$
- $f_2(n) = 2\log(6^{\log n^2}) + \log(\pi n^2) + n^3 = 2\log n^2 \log 6 + \log\pi + \log n^2 + n^3 = 4\log 6 \log n + \log\pi + 2\log n + n^3 \in \Theta(n^3)$
- $f_3(n) = 4^{log_2 n} = (2^2)^{log_2 n} = (2^{log_2 n})^2 = n^2 (*) \in \Theta(n^2)$
- $f_4(n) = 12\sqrt{n} + 10^{223} + \log 5^n = 12\sqrt{n} + 10^{223} + n\log 5 \in \Theta(n)$
- $f_5(n) = 10^{\lg 20} n^4 + 8^{229} n^3 + 20^{231} n^2 + 128 n \log n \in \Theta(n^4)$
- $f_6(n) = \log n^{2n+1} = (2n+1)\log n \in \Theta(n \log n)$
- $f_7(n) = \log^2(n) + 50\sqrt{n} + \log(n) \in \Theta(\sqrt{n})$
- $f_8(n) = 14400 \in \Theta(1)$

$$f_8 < f_7 < f_4 < f_6 < f_3 < f_2 < f_5 < f_1$$

(*): hint: $a^{\log_a n} = n$ by the definition.

b) Assume $f_1(n) = O(1)$, $f_2(n) = O(N^2)$, and $f_3(n) = O(N \log N)$. From these complexities it follows that $f_1(n) + f_2(n) + f_3(n) = O(N \log N)$.

Answer: ☐ True ☒ False

Task 4: Special Case and Correctness Analysis [Medium]

Consider the algorithm `algo1`. The input parameters are an array `A[1..n]` with n distinct integers and $k \leq n$.

Algo: algo1(A, n, k)

```
sum = 0;
for i = 1 to k do
    maxi = i;
    for j = i to n do
        if A[j] > A[maxi] then
            maxi = j;
    sum = sum + A[maxi];
    swp = A[i];
    A[i] = A[maxi];
    A[maxi] = swp;
return sum
```

a) Specify the pre/post conditions of the `algo1` algorithm.

The preconditions (inputs) are an array `A[1..n]` and an integer k.

The post conditions(outputs) are the following:

- sum of the biggest k elements of the array `A[1..n]`. Recursively, we can define the output of `algo1` (sum of the biggest k elements of the array `A[1..n]`) in the following way: Let $sum \in \mathbb{N}$ denote the biggest k elements of the array `A[1..n]`, then we have

  $$\forall i \in [1..k] : sum = sum + A[i], \text{where } A[1..k] \text{ are the biggest } k \text{ integers and sorted in a descending order}$$

- Integers of $A[1..k]$ are the biggest k integers in A and sorted in a descending order.

b) For the two `for` loops in the algorithm:

i. Determine if the loop is `up` loop or `down` loop.

The outer loop `for` $i = 1$ `to` k is an `up` loop, as it runs from low (1) to high k.

The inner loop `for` $j = i$ `to` n is an `up` loop as well, as it runs from low (i) to high n.

ii. Determine the invariants of these two loops and verify whether they are hold in three stages: `initialization`, `maintenance` and `termination`.

We start with the inner loop. The invariant of the inner loop is

$$\forall k \in [i..n] : A[maxi] \geq A[k].$$

`A[maxi]` is the largest element in the array `A[i..j]`.

- **Initialization**. When $j = i$, $A[i, j]$ contains only one element $A[i]$. At this moment (before the loop starts), we have $maxi = i$ and $A[maxi]$ is the only, and the biggest element in $A[i...i]$.

- **Maintenance**. When $i < j < n$, $A[maxi] \geq A[k]\ \forall k \in [i..j-1]$. If $A[j] > A[maxi]$, then $maxi$ is assigned to be j. It follows that $A[maxi] \geq A[m], \forall m \in [i...j]$.
- **Termination**. The inner loop terminates when $j = n$. At this time, if $A[j] > A[maxi]$, then $maxi$ is assigned to be n. It follows that $A[maxi] \geq A[m], \forall m \in [i...n]$. In other words, $A[maxi]$ is the biggest element in $A[i...n]$.

Analyzing the invariant of outer loop relies on the invariant of inner loop. The invariant of the outer loop is

$$\forall q \in [1..k], \forall p \in [k+1...n], A[q] \geq A[p].$$

In other words, top k largest numbers that are sorted in descending order in $A[1...k]$.

- **Initialization**. When $i = 1$, inner loop terminates and gives the largest number in $A[1...n]$, and $maxi = i$. The swap operation moves $A[maxi]$ to $A[1]$. It follows that $p \in [2...n], A[1] \geq A[p]$. Hence

  $$\forall q \in [1..i], \forall p \in [i+1...n], A[q] \geq A[p]$$

  holds true when $i = 1$.
- **Maintenance**. When $1 < i < k$, we have top $i-1$ largest numbers that are sorted in descending order in $A[1...i-1]$. We have

  $$\forall q \in [1..i-1], \forall p \in [i...n], A[q] \geq A[p]$$

  Inner loops terminates and guarantees that $A[i]$ is the largest element in $A[i...n]$ and $\forall q \in [1..i-1]$, $A[q] \geq A[i]$. Hence it also holds true that

  $$\forall q \in [1..i], \forall p \in [i+1...n], A[q] \geq A[p]$$

- **Termination**. The loop terminates when $i = k$, the inner loop terminates and guarantees that $A[k]$ is the largest element in $A[k...n]$ and $\forall q \in [1..k-1]$, $A[q] \geq A[k]$. It follows

  $$\forall q \in [1..k], \forall p \in [k+1...n], A[q] \geq A[p].$$

c) Identify some edges cases of the algorithm and verify if the algorithm has the correct output.

- If `A[1..n]` is empty, then the algorithm only initialize `sum` to be zero and returns it.
- If `A[1..n]` only contains one element, the outer loop will be executed only once and guarantees the `A[1..n]` contains the biggest element, which is the only element in the array. The algorithm returns the initialized sum (0) plus the only element in the array ($A[1]$).
- For a general case, the outer loop guarantees that `A[1..n]` contains the biggest k elements. In the body of the outer loop, the algorithm calculates the sum of the first k elements in the array `A[1..n]` and returns it. The returned value is the sum of the biggest k elements.

d) Conduct an exact analysis of the running time of algorithm `algo1`.

Instruction	# of times executed	Cost
$sum := 0$	1	c_1
for $i := 1$ **to** k **do**	$k+1$	c_2
$maxi := i$	k	c_3
for $j := i$ **to** n **do**	$\left(kn - \frac{k(k-1)}{2}\right)^* + k^{**}$	c_4
if $A[j] > A[maxi]$ **then**	$kn - \frac{k(k-1)}{2}$	c_5
$maxi := j$	$\alpha\left(kn - \frac{k(k-1)}{2}\right)^{***}$	c_6
$sum := sum + A[maxi]$	k	c_7
$swp := A[i]$	k	c_8
$A[i] := A[maxi]$	k	c_9
$A[maxi] := swp$	k	c_{10}
return sum	1	c_{11}

* $(n) + (n-1) + \ldots + (n-(k-1)) = kn - (0 + 1 + ... + k - 1) = kn - \frac{k(k-1)}{2}$

** k times for terminating loops

*** $0 \leq \alpha \leq 1$

$$T(n) = c_1 + c_2(k+1) + c_3 k + c_4(kn - \frac{k(k-1)}{2} + k) + c_5(kn - \frac{k(k-1)}{2}) + c_6(\alpha(kn - \frac{k(k-1)}{2})) + (c_7 + c_8 + c_9 + c_{10})k + c_{11}$$

In conclusion, $T(n) = k * n$.

e) Determine the best and the worst case of the algorithm. What is the running time and asymptotic complexity in each case?

Best case

In the best case, the array has already been sorted in descending order, hence we do not need to run `maxi := j`, i.e., $\alpha = 0$. In this case,

$$T_{\text{best}}(n) = c_1 + c_2(k+2) + c_3k + c_4(kn - \frac{k(k+1)}{2} + k) + c_5((k+1)n - \frac{k(k+1)}{2}) + 0 + (c_7 + c_8 + c_9 + c_{10})k + c_{11}$$

$$T_{\text{best}}(n) = O(k * n)$$

Worst case

Similarly, in the worst case, the array is sorted in ascending order and we have to update `maxi` every time, i.e., $\alpha = 1$. In this case, $T_{\text{worst}}(n) = c_1 + c_2(k+2) + c_3k + c_4(kn - \frac{k(k+1)}{2} + k) + c_5((k+1)n - \frac{k(k+1)}{2}) + c_6((k+1)n - \frac{k(k+1)}{2}) + (c_7 + c_8 + c_9 + c_{10})k + c_{11}$

$$T_{\text{worst}}(n) = O(k * n)$$

Asymptotic complexity of best and worst case

$$T_{\text{best}}(n) = O(k * n)$$

$$T_{\text{worst}}(n) = O(k * n)$$