Assignment 11, Data-Structures

Oleg Sivokon

<2016-03-06 Sun>

Contents

1 Problems

1.1 Problem 1

Count the number of compare and copy operations required to sort the two sequences given below using insertion sort:

1. $\frac{n}{2}, \frac{n}{2} - 1, \ldots, 2, 1, n, n-1, \ldots, \frac{n}{2}, \frac{n}{2} - 1.$
2. $n, 1, n-1, 2, \ldots, \frac{n}{2} + 2, \frac{n}{2} - 1, \frac{n}{2} + 1, \frac{n}{2}.$

1.1.1 Answer 1

Assuming we sort in ascending order, observe that the our task is to repeat the same operation twice (viz. to reverse two sorted arrays of the size exactly half of n.) Reversing individual arrays will require in case of one-length array 0 `swap` operations, in case of two-length array, 1 `swap` operation, and when we go further, we would need to do the same amount of work we did in the case of $n-1$, and then need to do $n-1$ more swaps to bring the first element of the original input to the back.

This gives the recurrence (for reversing the array):

$$
\begin{aligned}
&R(1) = 0 \\
&R(2) = 1 \\
&R(3) = R(2) + (3-1) = 3 \\
&R(4) = R(3) + (4-1) = 6 \\
&\quad \ldots \\
&R(n) = R(n-1) + n - 1 = \sum_{i=0}^{n-1} i = \frac{n(n-1)+2}{2} \ .
\end{aligned}
$$

Hence the total amount of `swap` operations needed to sort the given array is $T(n) = 2R(\frac{n}{2}) = n(n+1)$.

It is easy to see that the asymptotic complexity of $T(n)$ is $O(n^2)$ since since $\lim_{n\to\infty} \frac{n^2}{n(n-1)} = 1$.

1.1.2 Answer 2

Assuming we sort in ascending order, in the first step we make one comparison and swap. In the next step the first element will stand in its place, while the second element will need to move two positions to the end. The one before last element will need to move one position back.

The same will happen once we increment further. I.e. now two elements will stand in their place, but now the largest element will need to move two positions to the end, and so will the second largest element. The third smalles element will need to move one position to the front.

Let now $T(n)$ denote the number of `swap` operations we perform for any given n, then:

$$
\begin{aligned}
T(2) &= 1 \\
T(4) &= T(2) + 2 + 1 \\
T(6) &= T(4) + 2 + 2 + 1 \\
T(8) &= T(6) + 2 + 2 + 2 + 1 \\
&\dots \\
T(n) &= T(n-2) + n - 1 \ .
\end{aligned}
$$

This recurrence is easily recognizable as being just $(n-1)^2$. Hence, as expected, insertion sort requires roughly quadratic number of swaps, i.e. $O(n^2)$.

1.2 Problem 2

Given a sorted array $A[1 \dots n]$ where all elements are unique integers:

1. Write a predicate that asserts whether the given array is dense (has no gaps) in time $\Theta(1)$.
2. Write a predicate that given that A is sparse finds the element v which doesn't apear in A but is smaller than its largest element and is greater than its smallest element.

1.2.1 Answer 3

Algorithm 1 Assert S is a sparse array

```
procedure is-sparse(S)
    x ← first(S)
    y ← last(S)
    len ← length(S)
    return y − x > len
end procedure
```

Algorithm 2 Finds the first gap in sparse array S

```
procedure binsearch-missing(S)
    len ← length(S)
    start ← 0 end ← len/2
    cut ← slice(S, start, end)
    while end − start > 1 do
        if is-sparse(cut) then
            end ← end − (end − start)/2
        else
            start ← end − 1
            end ← end + (len − end)/2
        end if
    end while
    return end + 1
end procedure
```

Real code (compiled in C99):

Note that you will need the support code located in this derictory

```
bool is_sparse(const array* sorted) {
    int* first = (int*)sorted->elements[0]->val;
    int* last = (int*)sorted->elements[sorted->length - 1]->val;

    return (int)*last - (int)*first >= sorted->length;
}

size_t binsearch_missing(const array* sparse) {
    size_t start = 0, end = sparse->length / 2;
    array* cut = slice(sparse, start, end);

    while (end - start > 1) {
        if (is_sparse(cut)) {
            end -= (end - start) / 2;
        } else {
            start = end - 1;
            end += (sparse->length - end) / 2;
        }
        free_array(cut);
        cut = slice(sparse, start, end);
    }
    return end + 1;
}

void report(array* tested, char* message) {
    printf(message, to_string((printable*)tested));
    if (!is_sparse(tested)) {
        printf("Array is dense.\n");
    } else {
        printf("Array is sparse.\n");
        size_t missing = binsearch_missing(tested);
        printf("The first gap is at: %d\n", (int)missing);
    }
}

int main() {
    report(make_sparse_sorted_array(
        10, 13, 7, int_element_generator),
           "Created sparse array: %s.\n");
```

```
    return 0;
}
```

```
Created sparse array: [13, 17, 18, 24, 30, 31, 35, 41, 45, 49].
Array is sparse.
The first gap is at: 2
```

1.3 Problem 3

Given a list of m real numbers S, a similar list of n real numbers T and a real number z, write an algorithm that finds a pair of elements in $x \in S$ and $t \in T$ s.t. $s + t = z$.

1.3.1 Answer 4

Algorithm 3 Find $s \in S$ and $t \in T$ s.t. $s + t = z$

```
procedure summands-of(S, T, z)
    if length(S) < length(T) then
        shortest ← sorted(S)
        longest ← T
    else
        shortest ← sorted(T)
        longest ← S
    end if
    for val ∈ longest do
        diff ← z − val
        (pos, found) ← binsearch(shortest, diff)
        if found then
            other ← elt(shortest, pos)
            return (val, other)
        end if
    end for
    return failure
end procedure
```

Real code compiled in C99:

```
pair* summands_of(const array* a,
                  const array* b,
                  const float z,
                  comparison_fn_t cmp) {
    pair* result = make_pair();
    array* shortest;
    array* longest;
    size_t i;

    if (a->length < b->length) {
        shortest = sorted((array*)a, cmp);
        longest = (array*)b;
    } else {
        shortest = sorted((array*)b, cmp);
        longest = (array*)a;
    }
    for (i = 0; i < longest->length; i++) {
        float* val = longest->elements[i]->val;
        printable_float* diff = make_printable_float(z - *val);
        size_t pos = binsearch(shortest, (printable*)diff, cmp);
        if (pos >= shortest->length) continue;
        float* other = shortest->elements[pos]->val;
        result->first =
            (printable*)make_printable_float((float)*val);
        result->last =
            (printable*)make_printable_float((float)*other);
            break;
    }
    return result;
}

int main() {
    int ints[7] = {1, 2, 3, 4, 5, 6, 7};
    float sum = 13.0;
    array* test = make_array_from_pointer(
        ints, 7, float_element_generator);

    printf("Floats: %s\n", to_string((printable*)test));
    pair* summands = summands_of(test, test, sum, compare_floats);
    printf("%f = %s + %s\n",
           sum,
           to_string(summands->first),
           to_string(summands->last));
    return 0;
}
```

```
Floats: [1.000000, 2.000000, 3.000000, 4.000000, 5.000000, 6.000000, 7.000000]
13.000000 = 7.000000 + 6.000000
```

1.4 Problem 4

Show example of a function f satisfying $f(n) \neq \Omega(n)$ and $f(n) \neq O(n)$.

1.4.1 Anwser 5

Recall the definition of $O(n)$: $f(n) = O(f(n))$ as $n \to \infty$ precisely when $\forall (x \geq x_0) : \big|f(n)\big| \leq M\big|f(n)\big|$, where M and x_0 are some real numbers. The definition of Ω is similar, but asking to find a real constant M s.t. starting with x_0 all values of $\big|f(n)\big|$ are less than $M\big|f(n)\big|$.

One way to come up with the function which isn't its own upper or lower bound is to take an oscilating function, for example:

$$f(n) = \begin{cases} 1, & \textbf{if}\, n \equiv 0 \mod 2 \\ 0, & \textbf{if}\, n \equiv 1 \mod 2 \end{cases}$$

Clearly there is no such n_0 for which all values of $f(n)$ are greater than $f(n_0)$, similarly, there are no such n_0 that all values of $f(n)$ for $n > n_0$ are smaller than any multiple of $\big|f(n)\big|$.

1.5 Problem 5

Given following functions:

$$f_1(n) = max\left(\sqrt{n^3} \times \lg n, \sqrt[3]{n^4} \times \lg^5 n\right)$$
$$f_2(n) = \begin{cases} n \times \lg^3 n, & \textbf{if } n = 2k \\ n^3 \times \lg^3 n, & \textbf{if } n = 2k+1 \end{cases}$$
$$f_3(n) = n^{\lg \lg n} + n^{1000000} \times \lg^{100000} n$$
$$f_4(n) = \begin{cases} n^n \times 2^{n!}, & \textbf{if } n \leq 2^{1000000} \\ \sqrt{n^{\lg n}}, & \textbf{if } n > 2^{1000000} \end{cases}$$

for each pair of them assert O, o, Ω, ω and Θ.

1.5.1 Answer 6

	f_1, f_2	f_1, f_3	f_1, f_4	f_2, f_3	f_2, f_4	f_3, f_4
O	$f_1 = O(f_2)$	$f_1 = O(f_3)$	$f_1 = O(f_4)$	$f_2 = O(f_3)$	$f_2 = O(f_4)$	$f_3 = O(f_4)$
o	$f_1 = o(f_2)$	$f_1 = o(f_3)$	$f_1 = o(f_4)$	$f_2 = o(f_3)$	$f_2 = o(f_4)$	$f_3 = o(f_4)$
Ω	$f_2 = \Omega(f_1)$	$f_3 = \Omega(f_1)$	$f_4 = \Omega(f_1)$	$f_3 = \Omega(f_2)$	$f_4 = \Omega(f_2)$	$f_4 = \Omega(f_3)$
ω	$f_2 = \omega(f_1)$	$f_3 = \omega(f_1)$	$f_4 = \omega(f_1)$	$f_3 = \omega(f_2)$	$f_4 = \omega(f_2)$	$f_4 = \omega(f_3)$
Θ	$f_1 \neq \Theta(f_2)$	$f_1 \neq \Theta(f_3)$	$f_1 \neq \Theta(f_4)$	$f_2 \neq \Theta(f_3)$	$f_2 \neq \Theta(f_4)$	$f_3 \neq \Theta(f_4)$

Discussion

We can simplify the calculations by noticing that f_1 is sub-linear, i.e. it is dominated by $O(n)$, while all other functions are at least linear.

We are only interested in the second case of f_4, which is easier to rewrite as $n^{\frac{1}{2}\lg n}$.

Similarly, we can simplify f_3 by taking its fastest growing term: $n^{\lg \lg n}$. In other words, both f_3 and f_4 exhibit exponential growth.

Finally, f_2 is qubic in its worst case (again, the logarithmic factor is dominated by n^3 asymptotically.)

We can see that f_4 grows faster than f_3 by comparing the exponents: $\frac{1}{2}\lg n > \lg\lg n$ for some n since $\frac{1}{2}\lg n = \lg\sqrt{n} > \lg\lg n$ since $\lg n = O(\sqrt{n})$.

Neither function is of the same order as the other, thus it is never the case that $f_i = \Theta(f_j)$.