CS 401 :: Homework 1

Ryan Magdaleno

January 29, 2024

Problem 1. Ordering functions by their asymptotic dominance.
Order $f_1, f_2, ...,$ such that $i \geq 1$, we have $f_i(n) = O(f_i + 1(n))$.

Solution ::

$$\log(\log(n)) < \log(n) < 1000 \cdot n < n \cdot \log(n) < n^2 < n^{\log n} < 2^n < 2^{2^n}$$

Problem 2.

(a) Prove or disprove that $2^{n+1} = O(2^n)$.

Solution ::
I will use the following proof:
By definition of the limit, there exists n_0 such that for all $n \geq n_0$:

$$\frac{1}{2}c \leq \frac{f(n)}{g(n)} \leq 2c$$

If we let $c = 1$, then:

$$\frac{1}{2} \leq \frac{2^{n+1}}{2^n} \leq 2 \tag{1}$$

$$\frac{1}{2} \leq 2 \leq 2 \tag{2}$$

This inequality holds for all $n \geq n_0$.
$\therefore 2^{n+1} = O(2^n)$, given $n_0 = 1$

(b) Prove or disprove that $2^{2n} = O(2^n)$.

Solution ::
I will use the following proof:
$T(n)$ is $O(f(n))$ if:

$$\limsup_{n\to\infty} \frac{T(n)}{f(n)} < \infty$$

Let's simply the expression (3) and take the limit (4):

$$\limsup_{n\to\infty} \frac{2^{2n}}{2^n} = \limsup_{n\to\infty} 2^n \tag{3}$$

$$= \infty \tag{4}$$

$\therefore$ Because the limit is ∞, this means $2^{2n} \neq O(2n)$

Problem 3. Give the asymptotic running time of the following algorithm in Θ notation. Justify both the upper and the lower bound.

```
FUNC1(n);
s ← 0;
i ← 5;
while (i < n^2 + 7) do
    for j ← i to i^3 log i do
        s ← s + 1;
    end
    i ← i × 4;
end
```

Solution ::
Upper Bound ::

1. The outer loop runs until $i \geq n^2 + 7$, i will be multiplied by 4 each iteration. This indicates in the worst case that the outer loop contributes $O\left(\frac{\log_4(n^2+7)}{5}\right)$ to the overall complexity.

2. The inner loop runs $i^3 \log(i)$ iterations for each outer loop iteration. We can relate the inner loop to the outer loop by looking at the highest possible values of i based on the outer iteration. The max number of iterations of the inner loop can be approximated via the geometric sum of $i^3 \cdot \log(i)$ from $i = 0$ to $i = \frac{\log_4(n^2+7)}{5}$ for the function $i^3 \cdot \log(i)$. The sum approximated for a maximized i is n^2 therefore the i^3 term contributes $(n^2)^3 = n^6$. The $\log(i)$ therefore is $\log(n^2)$. In the worst case this indicates that the inner loop contributes $O(n^6 \log(n^2))$ to the overall complexity.

3. All together the upper bound of this algorithm is:

$$O\left(\frac{\log_4\left(n^2+7\right)}{5} \cdot \left(n^6 \log\left(n^2\right)\right)\right)$$

   or more simplfied:

$$O\left(\log_4\left(n^2\right) \cdot \left(n^6 \log\left(n^2\right)\right)\right)$$

Lower Bound ::

1. For the lower bound, the best case for this algorithm would be when $n = 0$ making the outer loop iterate a single time.
2. The inner loop would then iterate the constant i to $i^3 \log(i)$
3. This indicates that in the best-case scenario, the lower bound is:

$$\Omega\left(1 \cdot \left(5^3 \log(5)\right)\right)$$

and since $i = 5$ is a constant this can be simplfied to:

$$\Omega(1)$$

Big Theta Time ::

1. The dominant terms from both functions combines to give us the following theta complexity:

$$\Theta\left(n^6 \cdot \log_4\left(n^2 + 7\right)\right)$$

or simply:

$$\Theta\left(n^6 \cdot \log_4\left(n^2\right)\right)$$

Problem 4.

(a) You are given a set of n items of sizes $a_1, ..., a_n \in \mathbb{N}$, and a bin size $B \in \mathbb{N}$. Your goal is to find a maximum cardinality subset of items that all fit inside the bin. That is, you want to find a set of distinct indices $I = \{i_1, ..., i_k\} \subseteq \{1, ..., n\}$, such that

$$a_{i_1} + ... + a_{i_k} \leq B,$$

maximizing k. Design a greedy algorithm. Running time of your algorithm should be polynomial in n.

Solution ::

Given some way to keep track of an item's original index even after moving it, my algorithm would follow like so:

1. Have some sum variable s initialized to 0.
2. Sort the set that contains the items $a_1, ..., a_n$ in ascending order.
3. Add each a_i value to s and respective index to I if $s + a_i \leq B$.
4. Break summation early if (3) condition was not met.
5. Return the resultant I set.

Runtime Analysis ::

1. If we anaylyze the code in the subsequent pages, there's a single sorting step which indicates an $O(n \log n)$ time, where n is the number of items. **Dominant step.**
2. The for loop in the worst case will iterate fully when ITEMSIZESUM never becomes greater than B, indicating an $O(n)$ run time.
3. The algorithm's time complexity is $O(n \log n) + O(n)$ which simplfies to $O(n \log n)$. Therefore this algorithm indeed runs in time polynomial in n.

I've written C++23 code that follows this algorithm below. The code below can be compiled if you invoke with my provided compilation command. Please have a look at the function PROBLEMFOUR for my algorithm implementation.

```cpp
// g++ -std=c++23 -O2 -Wall £(NAME).cpp -o £(NAME).exe
#include <algorithm>
#include <iostream>
#include <vector>

struct Item { int size, index; };
bool compareItems(Item i1, Item i2) { return (i1.size < i2.size); }

std::vector<int> ProblemFour(std::vector<Item> items,
                             const int& B)
{
    std::vector<int> I;
    int itemSizeSum = 0;  // 1) s variable = 0.

    // 2) Sort items in ascending order:
    std::sort(items.begin(), items.end(), compareItems);

    // 3) Collect front items until s + a_i > B:
    for (const Item& i : items) {
        itemSizeSum += i.size;
        if (itemSizeSum > B) { break; } // 4) Break summation early.
        I.push_back(i.index);
    }
    return I; // 5) Return resultant I set.
}
```

```cpp
int main()
{
    std::vector<int> numsToAdd = {4, 10, 32, 2, 44, 132, 60, 23, 1};
    int n = (int)numsToAdd.size();
    int B = 100;

    std::vector<Item> items;
    items.reserve(n);

    for (int i = 0; i < n; ++i) {
        items.push_back({numsToAdd[i], i});
    }
    std::vector<int> I = ProblemFour(items, B);
    std::cout << "Set of n items: ";
    for (const int& size : numsToAdd) {
        std::cout << size << ' ';
    }

    std::cout << "\nReturned I set: ";
    for (const int& index : I) {
        std::cout << index << ' ';
    }
    return 0;
}
```

(b) Suppose that instead of maximizing k, we want to maximize the total size of the items in the bin; that is, we want to maximize the quantity

$$\text{size}(I) = a_{i_1} + ... + a_{i_k}.$$

Show that your greedy algorithm does not work in this case.

Solution ::

To show that my algorithm doesn't work with part b, we can show a counter example.

$A = \{1, 2, 99\}$, $B = 100$, $n = 3$

My algorithm would sort the A set and sum the values up until:

$\text{size}(I) = 3$

In this case my algorithm would terminate on $\text{size}(I) = 3$ where,

$\text{size}(I) = 1 + 2$ and $I = \{1, 2\}$ (1-based indices).

However the most optimal $\text{size}(I)$ would be $\text{size}(I) = 100$ where,

$\text{size}(I) = 1 + 99$ and $I = \{1, 3\}$.

$\therefore$ My algorithm does not work in certain cases if we are trying to maximize the quantity:

$$\text{size}(I) = a_{i_1} + ... + a_{i_k}.$$

Problem 5. A group of n people are lying on the beach. The beach is represented by the real line $\mathbb{R}$ and the location of the i-th person is some integer $x_i \in \mathbb{Z}$. Your task is to prevent people from getting sunburned by covering them with umbrellas. Each umbrella corresponds to a closed interval $I = [a, a + L]$ of length $L \in \mathbb{N}$, and the i-th person is covered by that umbrella if $x_i \in I$. Design a greedy algorithm for covering all people with the minimum number of umbrellas. The input consists of the integers $x_1, ..., x_n$, and L. The output of your algorithm should be the positions of umbrellas.

Prove that your algorithm is correct and that it runs in time polynomial in n.

Solution ::
Assuming all person indices are unique and L and n can't be negative my algorithm would be like so:

1. Create some container U for the umbrella positions.
2. Sort people indices in ascending order.
3. Place umbrella such that the first uncovered person is in the leftmost pos.
4. Count how many people got covered and add the current umbrella index to U.
5. Repeat steps $3 - 4$ until total covered people $== n$.
6. Return resultant U container.

Correctness ::

1. This algorithm is greedy because it always picks an umbrella that covers the leftmost uncovered person.
2. This algorithm will terminate because at each iteration, the algorithm selects an umbrella position that covers the leftmost uncovered person. By doing so, this algorithm ensures that all people will eventually be covered by an umbrella and will do so until the number of covered people is equal to n.
3. This algorithm is optimal because of the leftmost position selection, this will minimize the number of umbrellas required to cover subsequent people.

Runtime Analysis ::

1. If we anaylyze the code in the subsequent pages, there's a single sorting step which indicates an $O(n \log n)$ time, where n is the number of people. **Dominant step.**

2. Placing people on the beach vector always takes $O(n)$ time.

3. The main while loop iterates until every person is covered, in the worst case, each person would need their own loop iteration/umbrella. There's an inner loop that always runs $O(L+1)$ times, where L is the given constant L. Therefore, the while loop runs $O(n \cdot (L+1)) = O(n)$ times.

4. The algorithm's time complexity is $O(n \log n) + O(n) + O(n)$ which simplfies to $O(n \log n)$. Therefore this algorithm indeed runs in time polynomial in n.

   I've written C++23 code that follows this algorithm below. The code below can be compiled if you invoke with my provided compilation command. Please have a look at the function ProblemFive for my algorithm implementation.

```cpp
// g++ -std=c++23 -O2 -Wall £(NAME).cpp -o £(NAME).exe
#include <algorithm>
#include <iostream>
#include <vector>
struct BeachPoint { bool umbrella = false, covered = false, person = false; };
void printBeach(const std::vector<BeachPoint>& beach)
{
    std::cout << "\nU C P\n";
    for (const auto& bp : beach) {
        std::cout << bp.umbrella << ' '
                  << bp.covered  << ' '
                  << bp.person   << '\n';
    }
}
```

More below:

```cpp
std::vector<int> ProblemFive(std::vector<int>& personIndices,
                             const int& L,
                             const int& N)
{
    // Bad value check:
    if (L < 0 || N < 0) { return {}; }

    // Sort personIndices in ascending order:
    std::sort(personIndices.begin(), personIndices.end());

    // Set up and place people on beach:
    std::vector<BeachPoint> beach(personIndices[N - 1] + L + 1);  // Ensure space.
    std::vector<int> umbrellaIndices;                             // Returned vector.
    for (const int& personIndex : personIndices) {
        beach[personIndex].person = true;
    }

    int covered = 0, personPos, umbrellaPos;
    while (covered < N) {
        personPos = personIndices[covered];  // Get first uncovered person's index.
        umbrellaPos = personPos + (L/2);     // Calc Umbrella position like step 2.
        umbrellaIndices.push_back(umbrellaPos);

        // Place umbrella on beach and count how many got covered:
        beach[umbrellaPos].umbrella = true;
        for (int i = personPos; i <= personPos + L; i++) {
            if (beach[i].person) { covered++; }
            beach[i].covered = true;
        }
    }
    printBeach(beach);
    return umbrellaIndices;
}
```

```cpp
int main()
{
    std::vector<int> inputIndices = {1, 3, 5};
    const int L = 2, N = (int)inputIndices.size();  // 3 people to cover.

    std::vector<int> outputIndices = ProblemFive(inputIndices, L, N);

    // Print umbrella positions:
    std::cout << "\nUmbrellas: " << outputIndices.size() << '\n';
    std::cout << "Umbrella indices (0-based):\n";
    for (const int& umbrellaIndex : outputIndices) {
        std::cout << umbrellaIndex << ' ';
    }
    return 0;
}
```