Analysis and Complexity of Algorithms

Marián Kurčina

ZS 2024

xkurcinam@stuba.sk
marian.kurcina2003@gmail.com

Table of Contents

Task 1

Design and implement an algorithm to solve the problem of scheduling with specified deadlines. Determine a schedule with the maximum total profit, where each task has a profit that will only be obtained if the task is scheduled before its deadline.
Inputs: The number of tasks and an array of integers deadline, indexed from 1 to n, where deadline[i] is the deadline for the i-th task. The array is sorted in ascending order based on profits.
Outputs: The optimal sequence of tasks.

1.1 Consider the following tasks, deadlines, and profits, and use the scheduling algorithm with specified deadlines to maximize the total profit. Display the optimal sequence of tasks with the maximum profit on the screen.

Job	Deadline	Profit
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55

The solution

To solve the problem, I used Algorithm 4.4 from *Foundations of Algorithms* by Richard E. Neapolitan, as mentioned in the task. The program works as follows:

Arrays with indices, deadlines, and profits are provided and can be modified. After loading the data, a maximum sort is performed to sort the data (in Task 1, nothing changes, but the code is compatible with Task 1.1). After sorting the data by profit in ascending order, the schedule function is invoked, which operates as described in the pseudocode from the task.

The schedule function takes the following parameters:

- n: number of tasks
- deadline[]: array of deadlines
- jobs_inx[]: indices of the jobs
- job_profit[]: array of profits for the tasks.

At the beginning of the function, the maximum deadline is determined, which indicates the number of tasks that can be completed. This value will be used to create an array for the final order of tasks. This array is initialized to 0.

Then, the algorithm proceeds to assign tasks to the final task order array in the following manner: The algorithm attempts to add a task to the position corresponding to its deadline, as if it were being done at the last moment. If that position is occupied, it tries previous positions one by one. If it reaches the beginning of the array, it rejects the task and continues with the next one. After going through all the tasks in the list, the program displays the best task arrangement and the profit.

Output for the input from the task

Input:

Job	Deadline	Profit
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55

Output:

```
Best job sequence:
7
1
3
2
Total Profit: 170
```

Analysis of the algorithm's complexity

d = maximum deadline (at most n)

max sort – **O(n^2)**
find_max – **O(n)**

schedule:
- Initialization to 0 – **O(d)**
- Task assignment – **O(n^2)**
- Overall complexity – O(d+ n^2) → **O(n^2)**

Overall complexity without sorting - O(n+d+n^2) → **O(n^2)**

Task 2

Consider the scheduling procedure in the "Scheduling with Deadlines" algorithm (Algorithm 4.4). Let d be the maximum deadline for n tasks. Modify the procedure to add tasks as late as possible to the schedule, but not later than their deadline. This can be done by initializing d+1 disjoint sets containing the integers 0, 1, ..., d. Let small(S) be the smallest element in the set S. When a task is scheduled, find the set S containing the minimum of its deadline and n. If small(S) = 0, the task is rejected. Otherwise, it is scheduled at time small(S) and the set S is merged with the set containing small(S) − 1. Assume we are using the Disjoint Set Data Structure III from Appendix C. Show that this version has a time complexity of θ(n log m), where m is the minimum of d and n.

The solution

In my implementation, I used the pseudocode from page 732 of *Foundations of Algorithms* by Richard E. Neapolitan. In my code, I use modified functions from the pseudocode since I was unable to implement it fully. As in the pseudocode, I also defined a structure for working with the "smallest" parameter, which also contains a pointer to the parent. I don't need the "depth" parameter, as I work with root elements when merging sets. The parent of an element always points to the element above it in the hierarchy, so I can navigate to the root of the set, which contains the correct "smallest" value. The `merge` function works by combining two trees, where the root of one tree points to the root of the other. The entire set will have the same root, and therefore the same "smallest" value. The `small` function simply navigates to the root of the tree and finds the "smallest" value. At the beginning, the tasks are sorted as in task 1, in descending order by profit. Then, I find the maximum deadline value and create an array of sets and an array where I will store the best solution. After that, the scheduling process begins, where it goes through all the tasks and tries to schedule them by finding the "smallest" value. The "smallest" value indicates the latest position where the task can be executed. If the function returns a value smaller than 1, it means there is no available spot for that task. However, if it returns a value greater than 0, I can schedule the task and merge the time slot where it is executed with the previous set. After going through all the tasks, I display the best schedule.

Output for the input from the task

Input:

Job	Deadline	Profit
1	2	40
2	4	15
3	3	60
4	2	20
5	3	10
6	1	45
7	1	55

Output:

```
Best job sequence:
7
1
3
2
Total Profit: 170
```

Analysis of the algorithm's complexity

d = maximum deadline (at most n)

max sort – **O(n^2)**

find_max – **O(n)**

Initialization of Sets – **O(d)**
Initialization of the Plan Array – **O(d)**

Planning:

small – In an ideal tree, the complexity of finding the root of the tree is log c, where c is the number of elements in the tree. → **O(log d)**
merge – **O(1)**
Scheduling for n elements = **O(n*(log d))**

Total complexity without sorting: O(n+2d+n*(log d)) → **O(n*log d)**, which can be written as **O(n*log m),** where m is the minimum of d and n, since d is always less than or equal to n.

Task 3

Use a greedy approach to write an algorithm that minimizes the number of moves of records in the problem of merging n sets. Use double merging, where two sets are merged at each step.

a: Provide two different inputs for your implementation and analyze each step of your algorithm.

b: Analyze your algorithm and show the results using Big-O notation.

The solution

The files are represented using a list of their lengths, as this is the only parameter of interest, and the process of merging doesn't need to be implemented. To solve the problem, I use a queue to hold the lengths of the original files and later the lengths of the merged files. The algorithm always picks the two smallest records from the queue, merges their lengths, and adds the new record back into the queue. In the end, the queue will contain a single record that represents the final file created from all the previously merged files. My code for the queue comes from my solution last year for implementing a queue in the DSA course. I have two functions related to the queue: get_min, which returns the smallest element (the one at index 0) and rebalances the queue by adjusting the parent-child relationships so that the smallest element is truly at the front, and add, which inserts an element into the queue in the correct position to maintain the order. This algorithm is greedy because, at each step, it makes a decision based on the current locally optimal solution.

Algorithm Analysis

Let the input be a list of file lengths: [1,2,3,4,5,6,7] a [1,2,15,1,10,8,2]
First, these values will be inserted into the queue. The queue will look like this:

[1,2,3,4,5,6,7] a [1,1,2,2,8,10,15]

At each step, it will extract the two smallest elements from the queue, so first it will be 1,2 and 1,1. These will be added together and inserted back into the queue. The queue will look like this:

[3,3,4,5,6,7] a [2,2,2,8,10,15].

During the merging process, it would look like this:

[3,3,4,5,6,7] →3 + 3→6→[4,5,6,6,7]
[4,5,6,6,7] →4 + 5→9→[6,6,7,9]
[6,6,7,9] →6 + 6→12→[7,9,12]
[7,9,12] →7 + 9→16→[12,16]
[12,16] →12 + 16→28→[**28**]

[2,2,2,8,10,15] →2 + 2→4→[2,4,8,10,15]
[2,4,8,10,15] →2 + 4→6→[6,8,10,15]
[6,8,10,15] →6 + 8→14→[10,14,15]
[10,14,15] →10 + 14→24→[15,24]
[15,24] →15 + 24→39→[**39**]

The result of the program is the length of the final file, which is 28 and 39.

Algorithm Complexity Analysis

add – **O(log n)**
get_min **– O(log n)**

Adding n elements to the **queue – O(n*log n)**

Merging Files (the number of merges is n-1)– O((n-1)*(3*log n)) → **O(n*log n)**

Task 4

Generate Huffman code for a given set of characters without constructing the Huffman tree.

Character	Frequency
A	45
B	13
C	12
D	16
E	9
F	5

a: Use canonical Huffman coding.
b: Use approximate coding based on frequency.
c: Provide examples of encoding and decoding and analyze both approaches using Big-O notation.

The solution

At the beginning, the data is sorted by frequency from highest to lowest. Then, the user selects the method for determining the code length for each character.

Type A:

For each character, structures info and groups are created.

- info contains details about frequency, code, code length, and the character itself.
- groups contains an array of characters and the sum of their frequencies, which helps in merging characters.

Characters are progressively merged into groups, always combining the two smallest groups. The number of merges for each character determines its code length.

The merging process works similarly to Task 3, using a queue. The two smallest groups are taken from the queue, merged, their information is updated, and the new group is inserted back into the queue. When updating information, the code length for each character in the two merged groups is increased by 1.

After merging all groups, character information is sorted again from highest to lowest frequency. If two characters have the same frequency but different code lengths, they are sorted from shortest to longest code (this solved an issue where three or more characters had the same highest frequency).
If the first element has a code length of 1, it is incremented to prevent it from being a prefix for another code.
This info list is used to find the codes.

Type B:

For each element, the formula $\lceil -\log_2 p \rceil$ is used, where $p = \frac{frekvencia\ pre\ prvok}{súčet\ všetkých\ frekvencií}$. This formula determines the code length for each character, which is then stored in info.

Encoding process:

- Variables c (current code length) and temp (decimal value to be converted into binary) are used.
- When the code length changes compared to the previous character, temp is multiplied by 2 (left shift), and c is incremented by 1.
- temp is then converted to binary, forming the character's code.
  This ensures that no code is a prefix of another.

Encoding and Decoding:

- **Decoding:** Reads the string character by character, storing the read symbols in a variable. After each digit is added, it checks if the sequence matches a known code. If a match is found, it prints the corresponding character and clears the variable. If not, it reads the next symbol.
- **Encoding:** Reads a character, finds it in the info list, prints its code, and moves to the next character.

Examples of Encoding and Decoding

Having encoded letters from the assignment using Canonical Huffman coding:
A=00, B=010, C=011, D=100, E=1010, F=1011

Decoding examples:
000101001011010 → adcfd
1001010000000010 → ceaaad

Encoding examples:
adcfd → 000101001011010
ceaaad → 1001010000000010

Having encoded letters from the assignment using Frequency-Based Approximation:
A=00, B=010, C=011, D=1000, E=1001, F=10100

Decoding examples:
000101001011010 → adebd
1001010000000010 → edaaad

Encoding examples:
adcfd → 000101001011010
ceaaad → 1001010000000010

Complexity Analysis of Encoding and Decoding

m = Length of the string
Encoding – m times checking among at most n characters → **O(m * n)**
Decoding – m times checking among at most n characters → **O(m * n)**

Complexity Analysis of the Algorithm

Type A - Canonical Huffman Coding:
- Creating and filling the info and groups arrays: O(n)
- Adding to the queue: O(logn), adding all elements: O(nlog n)
- **Merging elements:**
  - Retrieving an element from the queue: O(log n)
  - Copying character arrays into one: maximum number of characters is n → O(n)
  - Incrementing the code length for merged characters: maximum count is n → $O(n^2)$
  - Adding back to the queue: O(log n)
  - **Total complexity:** $O(n^2)$
  - Total complexity for n repetitions: $O(n^3)$
- Min sort: $O(n^2)$

Total complexity - Canonical Huffman Coding: $O(n^3)$

Type B – Frequency-based Encoding:
- Summing frequencies: O(n)
- Assigning information to the info array: O(n)
- **Encoding characters:**
  - m = maximum code length
  - Conversion to binary: m can be at most n → O(n)
  - Encoding all n characters: $O(n^2)$

Total complexity - Frequency-based Encoding: $O(n^2)$