University of Washington
Department of Computer Science and Engineering
CSE 421, Winter 2024

January 17, 2024

Homework 3, Due Wednesday, January 24, 11:59 pm, 2024

Turnin instructions: Electronic submission on gradescope using the CSE 421 gradescope site. Submit the assignment as a PDF, with separate pages for different numbered problems. Problems consisting of multiple parts (e.g., 2a, 2b) can be submitted on the same page.

Problem 1 (10 points):

The paragraphing problem is: Given a set of words $w_1, \ldots, w_n$ with word lengths $l_1, \ldots, l_n$, break the words into consecutive groups, such that the sum of the lengths of the words in each group is less than a fixed value K. (We will ignore the issue of putting spaces between words or hyphenation; these are minor details.) The words remain in the original order, so the task is just to insert line breaks to ensure that each line is less than length K.

The greedy algorithm for line breaking is to pack in as many words as possible into each line, e.g., to put words into a line one at a time until the length bound K is reached, and break the line before the word w_r that caused the the bound to be exceeded.

Prove that the Greedy Algorithm is optimal in the sense that it produces a paragraph with the smallest number of lines. For a formal proof, induction is recommended. The key to this problem is coming up with the right induction hypothesis.

Problem 2 (10 points):

Let $w_1, \ldots, w_n$, and $v_1, \ldots, v_n$ and K be positive real numbers. Give an algorithm to find values $a_1, \ldots, a_n$, with $0 \leq a_i \leq 1$ to maximize

$$\sum_{i=1}^{n} a_i v_i$$

such that

$$\sum_{i=1}^{n} a_i w_i \leq K.$$

Argue that your algorithm is correct. Hint: You will want to use a greedy algorithm. The run time should be $O(n \log n)$.

Problem 3 (10 points):

A standard way for evaluating heuristic algorithms is to compare the value of the solution with an optimal solution. In the case of a problem to *maximize* an objective function (such as the number of non-intersecting intervals in the Interval Scheduling problem), we look at the maximum ratio of the optimal solution to the solution found by the algorithm as a function of the input size n. More

specifically, let $Opt(I)$ be the value of the optimal solution on instance I, and $A(I)$ be the value of the solution found by Algorithm A. We define the *Approximation Ratio*, $R_A(n)$ to be:

$$R_A(n) = \max_{I \text{ of size } n} \frac{Opt(I)}{A(I)}.$$

The approximation ratio R_A is $\lim_{n\to\infty} R_A(n)$. The larger the approximation ratio, the worse the approximation is. In the lecture, we showed that for interval scheduling, earliest deadline first (EDF) is the optimal algorithm. We introduced three other heuristics, the earliest startime first (ESF), the shortest interval first (SIF), and the minimal intersection first (MIF). This problem is to determine the approximation rations for these three algorithms.

Problem 3a (3 points):

Show that the approximation ratio for earliest starttime first is ∞.

Problem 3b (4 points):

Show that the approximation ratio for shortest interval first is 2.

Problem 3c (3 points):

Show that the approximation ratio for minimal intersection first at least $\frac{4}{3}$ and at most 2.

Problem 4:

In this problem, and in the next, you will implement algorithms for interval scheduling. For the test data, you will have to construct a generator to create random instances of interval scheduling. Implement the following mechanism for generating random intervals from $[0, 1.0)$. Use double precision floats for your interval endpoints.

Random Interval Generator

Generate n random intervals. You can do it in a few ways, pick your favorite one:

1. For each interval, generate 2 numbers in $[0.0, 1.0)$ at random. Let the smaller one be the left endpoint and the larger one be the right endpoint.
2. For each interval, generate the left endpoint by choosing a point x in $[0.0, 1.0)$ at random. Generate the right endpoint y by choosing a point in $[x, 1.0)$ at random.
3. For each interval, generate the right endpoint by choosing a point y in $[0.0, 1.0)$ at random. Generate the left endpoint x by choosing a point in $[0.0, y)$ at random.
4. For each interval, generate a random interval length l from $(0.0, 1.0)$ at random. Generate the left endpoint x by choosing a point in $[0.0, 1 - l)$ and your right endpoint would be $y = x + l$.
5. Anything else as long as it's random and the results aren't trivial! (Your TAs think) it's fun to think about how different interval distributions affects the number of non-overlapping intervals.

You are free to write in any programming language you like. The quality of your algorithm may be graded, but the actual quality of the code will not be graded. The expectation is that you write the algorithmic code yourself - but you can use other code or libraries for supporting operations. Submit your code as a PDF.

Programming Problem 4a (3 points):

Write a random interval generator.
For this problem, turn in a PDF of your code, and the output of two runs of size 10.

Programming Problem 4b (4 points):

Implement the Earliest Deadline First algorithm for the Interval Scheduling Problem to find a maximum set of disjoint intervals. Submit your code as a PDF. Your implementation should be $O(n \log n)$.

Programming Problem 4c (3 points):

How does the size of the the solution grow as a function of n. Run your program across a range of input sizes to get an estimate of the growth of the solution. How does the solution grow? (You should be able to run this on instances of up to size $n = 1,000,000$ without much difficulty, which is big enough to see a good trend.)

Problem 5:

Implement the greedy algorithms for interval scheduling using the smallest interval first heuristic and the fewest intersections first heuristic, and compare the heuristics with the earliest deadline first heuristic. You do *not* need to focus on efficient implementations for these heuristics. While it is probably possible to simulate these heuristics in $O(n \log n)$ time you may implement simpler approaches which will have runtimes of $O(n^2)$ or $O(n^3)$.

Programming Problem 5a (7 points):

Run a series of experiments to test how the three heuristics perform. You should test the same input on three different heuristics to how better the optimal algorithm is. You can also run experiments to see how much of a difference there is in the average performance of the heuristics are for a number of different runs. Run across a range of sizes of n to see what differences you can detect as n grows. (The size of n you can work with will depend on the efficiency of your implementation. Even with an $O(n^3)$ algorithm, $n = 10,000$ should be feasible.)

Programming Problem 5b (3 points):

Discuss the results from your experiments. Do they align with the with the worst case analysis from problem 3?