University of Moratuwa, Sri Lanka

Faculty of Engineering

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

BSc Engineering, Semester 7 (2014 Intake)
Feb – Jul 2018

CS4522 Advanced Algorithms

Assignment 2 (worth 12%; due 23rd July 2018)

Note the following important points before answering this assignment:

Use standard notations. All other notations and any assumptions must be clearly explained. Some questions may not be considered during grading; but students should answer all questions. Submission instructions are given at the end.
You must include in your submission:

- the names and registration numbers of any students with whom you worked together on this assignment (As already informed, while discussions with other students is a positive aspect, plagiarism, copying, giving or receiving aid are strictly not permitted. Working together cannot extend to writing the answers).

- the list of other resources (books, websites,…) you used in preparing your answers.

1. [20 marks] Fig. Q1 shows a flow network. Edge labels indicate capacity of each edge. Answer (a) and (b) below.

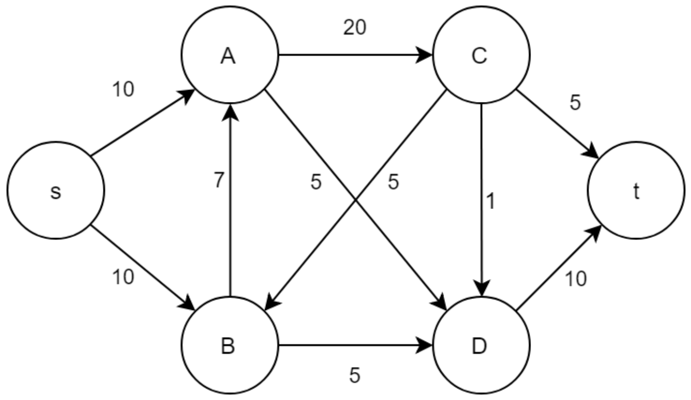

Fig Q1

(a). [15 marks] Find a maximum flow for the flow network. Show your work.
(b). [5 marks] Show all minimum s-t cuts in the flow network in Fig Q1.

2. [25 marks] Suppose you are given a large $n \times n$ matrix $A = (a_{ij})$. You are expected to compute an n-element vector B = (b_i) from the matrix A as follows: each element b_i in B is the sum of n elements in row i of A. That is, $b_i = \sum_{j=1}^{n} a_{ij}$.

(a). [10 marks] Give a parallel algorithm to compute B using parallel loops (i.e., `parallel for`), trying to parallelize as much as possible. State any assumptions. Briefly explain it.
(b). [10 marks] Give a parallel algorithm to compute B using nested parallelism (i.e., `spawn`) with a divide-and-conquer strategy. State any assumptions and briefly explain how it works.
(c). [2 marks] Write recurrences for the *work* and *span* using your algorithm in (b) above.
(d). [3 marks] Compute the *work*, *span* and *parallelism* for the computation in terms of n, using the recurrences you wrote in (c) above.

3. [30 marks] Let's consider implementing a stack as an array. Let the bottom of stack be the left-end of the array and the stack grow left-to-right, which means the top of stack is the rightmost element in the array. With push operations, the stack grows to the right and with pop operations the stack shrinks to the left.

When we discussed implementing a table as an array, we observed that doubling the array size whenever it gets full is a good strategy because the *amortized cost* of insertion was O(1) per insertion. (We did not discuss deletion at that point).

Now, let us try to apply the same amortized analysis method for a *stack implemented as an array*, considering both push and pop operations. When we do a push, we will double the array size if the array is full. When we do a pop, the stack gets small. We don't want to hold onto memory space unnecessarily, so we want to consider resizing the array to make it smaller, when it is suitable. That is to make the array smaller occasionally. We will consider two strategies for this.

Say k = number of elements and L = size of array.

(a). [15 marks] Strategy #1 (S1): When we pop at k = L/2, we cut the array size in half. Is this a good strategy? Why or why not?
(b). [15 marks] Strategy #2 (S2): When we pop at k = L/4, we cut the array size in half. Is this a good strategy? Why or why not?

4. [25 marks] Suppose you are given a *sorted array* A of size N (it contains N integers) and you have to search for an element *x* in it. You will first implement two search algorithms for this, as stated in parts (a) and (b) below, in either Java or Python. Next you will execute them to perform search operations and report timing results for the three cases where N=10^5, 10^6 and 10^7. For each case, you will perform 1000 searches, where each search is for a different *x*. You must have a mix of integers to search for such that 300 of them are not in the array (which will result in failure of search) and the remaining 700 integers are spread roughly evenly in A. Then the average of the time taken for the 1000 searches will give the search time for the case.

(a). [9 marks] Implement an *algorithm* that uses *binary search*. It should return a valid index if *x* is in A and -1 otherwise. If this algorithm is recursive, then your other algorithm in (b) below must also be recursive. Execute and measure the elapsed time to search for 1000 numbers and then obtain the average search time for each case of N, as described above. Report your results using the tabular format given below.

(b). [12 marks] Implement a *randomized binary search* algorithm to search for *x* in A by modifying your algorithm in (a) or otherwise. If your algorithm in (a) above is recursive, then your randomized algorithm must also be recursive. Execute and measure the elapsed time to search for 1000 numbers and then obtain the average search time for each case of N, as described above. Report your results using the tabular format given below.

(c). [4 marks] Compare and discuss about the performance of implementations in (a) and (b) and the results you obtained.

Array Size N	Average Time for Search (units*)	
	Binary Search	Randomized Search
10^5		
10^6		
10^7		

*units examples: milli seconds (ms), micro seconds (µs)

Submission Instructions: At the end, you are required to submit a zip file that contains the PDF document that has your answers to the questions 1-4 and your source code files for Q4; name the submitted zip file as “A2_<UoMRegNo>.zip” where <UoMRegNo> is your UoM Registration number (e.g., 140011X). Make sure the size of the uploaded file is less than 1MB.