CPSC 320 2021W2: Assignment 1

This assignment is due **Monday October 17, 2022 at 22:00 Pacific Time**. Assignments submitted **within 24 hours after the deadline** will be accepted, but a penalty of up to 15% will be applied. Please follow these guidelines:

- Prepare your solution using LaTeX, and submit a pdf file. Easiest will be to use the .tex file provided. For questions where you need to select a circle, you can simply change `\fillinMCmath` to `\fillinMCmathsoln` .
- Enclose each paragraph of your solution with `\begin{soln}Your solution here...\end{soln}`. Your solution will then appear in dark blue, making it a lot easier for TAs to find what you wrote.
- Start each problem on a new page, using the same numbering and ordering as this assignment handout.
- Submit the assignment via GradeScope at `https://gradescope.ca`. Your group must make a **single** submission via one group member's account, marking all other group members in that submission **using GradeScope's interface**.
- After uploading to Gradescope, link each question with the page of your pdf containing your solution. There are instructions for doing this on the CPSC 121 website, see `https://www.students.cs.ubc.ca/~cs-320/2022W1/index.php?page=assignments&menu=1&submenu=0`.

Before we begin, a few notes on pseudocode throughout CPSC 320: Your pseudocode should communicate your algorithm clearly, concisely, correctly, and without irrelevant detail. Reasonable use of plain English is fine in such pseudocode. You should envision your audience as a capable CPSC 320 student unfamiliar with the problem you are solving. If you choose to use actual code, note that you may **neither** include what we consider to be irrelevant detail **nor** assume that we understand the particular language you chose. (So, for example, do not write `#include <iostream>` at the start of your pseudocode, and avoid language-specific notation like C/C++/Java's ternary (question-mark-colon) operator.)

Remember also to **justify/explain your answers**. We understand that gauging how much justification to provide can be tricky. Inevitably, judgment is applied by both student and grader as to how much is enough, or too much, and it's frustrating for all concerned when judgments don't align. Justifications/explanations need not be long or formal, but should be clear and specific (referring back to lines of pseudocode, for example). Proofs should be a bit more formal.

On the plus side, if you choose an incorrect answer when selecting an option but your reasoning shows partial understanding, you might get more marks than if no justification is provided. And the effort you expend in writing down the justification will hopefully help you gain deeper understanding and may well help you converge to the right selection :).

Ok, time to get started...

1 Statement on Collaboration and Use of Resources

To develop good practices in doing homeworks, citing resources and acknowledging input from others, please complete the following. This question is worth 2 marks.

1. All group members have read and followed the guidelines for groupwork on assignments in CPSC 320 (see `https://www.students.cs.ubc.ca/~cs-320/2022W1/index.php?page=assignments&menu=1&submenu=3`).
   ◯ Yes ◯ No
2. We used the following resources (list books, online sources, etc. that you consulted):
3. One or more of us consulted with course staff during office hours.
   ◯ Yes ◯ No
4. One or more of us collaborated with other CPSC 320 students; none of us took written notes during our consultations and we took at least a half-hour break afterwards.
   ◯ Yes ◯ No
   If yes, please list their name(s) here:
5. One or more of us collaborated with or consulted others outside of CPSC 320; none of us took written notes during our consultations and we took at least a half-hour break afterwards.
   ◯ Yes ◯ No
   If yes, please list their name(s) here:

2 Recurrence relations for uneven splits

As stated in class, most divide and conquer algorithms divide the inputs into a number of subproblems that are all (almost) the same size. In this case, we can use the Master theorem to derive the solutions of the recurrence relations that describe their running times. Unfortunately, the Master theorem can not be applied to recurrences that correspond to algorithms that divide their inputs unevenly. This question generalizes the solution to some of the recurrence relations that appeared on one of the tutorial quizzes. We consider the following recurrence relation:

$$T(n) = \begin{cases} n^k + \sum_{j=1}^{t} a_j T(n/b_j) & \text{if } n \geq \max_{1 \leq j \leq t}\{b_j\} \\ \Theta(1) & \text{if } n < \max_{1 \leq j \leq t}\{b_j\} \end{cases}$$

You will prove a theorem that is similar to the Master theorem for this recurrence. We have provided most of the theorem, but left some blanks for you to fill in. You need to fill in the blanks and then prove the statement. Hints:

- Start by computing the work done on the first two levels of the recursion tree for that recurrence.
- Two of the three cases are fairly straightforward.
- The third case has similarities to the two questions from one of this week's tutorial quizzes.

1. [5 points] Complete the statement of the Theorem:

Let $\Delta =$ $\boxed{\sum_{j=1}^{t} a_j/b_j^k}$. Then

$$T(n) \in \begin{cases} \Theta\left(\boxed{n^k}\right) & \text{if } \Delta < 1 \\ \Theta\left(\boxed{n^k \log n}\right) & \text{if } \Delta = 1 \\ O\left(\boxed{n^{k+\log_{(\min_{j=1}^{t} b_j)} \Delta}}\right) \text{ and } \Omega\left(\boxed{n^{k+\log_{(\max_{j=1}^{t} b_j)} \Delta}}\right) & \text{if } \Delta > 1 \end{cases}$$

Your bounds for the third case will depend on n, k, Δ and $b_1, \ldots b_t$. You do not need bounds that are always tight for the third case, but the bounds you provide should be tight in the case there $t = 1$.

2. [8 points] Prove this theorem.

Solution: Let us consider the recursion tree that corresponds to the recurrence relation.

- The root of the tree performs n^k work.
- The next level down has
  - a_1 nodes that each perform $T(n/b_1) = (n/b_1)^k = n^k(1/b_1)^k$ work. Thus these a_1 nodes perform $(a_1/b_1^k)n^k$ work.
  - a_2 nodes that each perform $T(n/b_2) = (n/b_2)^k = n^k(1/b_2)^k$ work. Thus these a_2 nodes perform $(a_2/b_2^k)n^k$ work.
  - ...
  - a_t nodes that each perform $T(n/b_t) = (n/b_t)^k = n^k(1/b_t)^k$ work. Thus these a_t nodes perform $(a_t/b_t^k)n^k$ work.

  Hence this level performs $n^k \sum_{j=1}^{t} a_j/b_j^k$ work.
- In general, level i will do $n^k \left(\sum_{j=1}^{t} a_j/b_j^k\right)^i$ work.

Let

$$\Delta = \sum_{j=1}^{t} a_j/b_j^k$$

Then level i of the recursion performs $\Delta^i n^k$ work, up to the level where the first leaf can be found.

Case 1 ($\Delta < 1$): If $\Delta < 1$ then the work being done at each level is a decreasing geometric series. And so

$$T(n) \leq \sum_{i=0}^{\infty} \Delta^i n^k = \frac{n^k}{1-\Delta} \in O(n^k).$$

Also, $T(n) \geq n^k$ (the work done at the root of the tree), and so $T(n) \in \Theta(n^k)$.

Case 2 ($\Delta = 1$): When $\Delta = 1$, every level of the tree up to the level where the first leaf can be found does exactly n^k work. The first leaf can be found at level $\alpha = \log_{(\max_{j=1}^{t} b_j)} n$ and thus $T(n) \in \Omega(\alpha n^k)$. Similarly, the leaf furthest from the root is at level $\beta = \log_{(\min_{j=1}^{t} b_j)}$ and thus $T(n) \in O(\beta n^k)$. Because $\log_x n = \log_y n / \log_y x$ no matter what x and y might be, the two bounds are the same up to a constant factor, and so $T(n) \in \Theta(n^k \log n)$.

Case 3 ($\Delta > 1$): Here we have an increasing geometric series, so our lower and upper bounds will not be within a constant factor of each other. Let's start with the lower bound: the first leaf is once again at level $\alpha = \log_{(\max_{j=1}^{t} b_j)} n$. Level i of the tree does $\Delta^i n^k$ work for each i in the range $0 \le i \le \alpha$, so we have a lower bound of $n^k \sum_{i=0}^{\alpha} \Delta^i$ on $T(n)$.

The expression we now have for the lower bound is not easy to interpret, because the upper bound of the sum depends on n. It is possible to use the formula for the sum of a geometric series directly to get an equivalent, more readable, expression. We will use a slighly more devious but less algebraically painful approach by finding an alternate recurrence relation whose recursion tree has exactly α levels, and whose solution can be found using the Master theorem. We want a recurrence of the form $T'(n) = aT'(n/b)+n^k$ where the work being done on consecutive rows is n^k, Δn^k, $\Delta^2 n^k$, etc. Then $T'(n)$ will be the lower bound we want for $T(n)$.

To have exactly α levels, we need $\log_b n = \alpha$, which means $b = \max_{j=1}^{t} b_j$. This implies that $a/b^k = \Delta$, i.e. $a = b^k \Delta$. The solution to the recurrence relation $T'(n) = b^k \Delta T'(n/b)$ is in $\Theta(n^{\log_b b^k \Delta}) = \Theta(n^{\log_b b^k + \log_b \Delta}) = \Theta(n^{k+\log_b \Delta})$.

Therefore $T(n) \in \Omega(n^{k+\log_{(\max_{j=1}^{t} b_j)} \Delta})$. A similar argument using $\min_{j=1}^{t} b_j$ proves that $T(n) \in O(n^{k+\log_{(\min_{j=1}^{t} b_j)} \Delta})$.

Note: the Akra–Bazzi method is a more general that includes the theorem from this assignment as a special case.

3 Forgetful oceanographers

The British Columbia department of fisheries is asking one of its oceanographers to take water samples at regular intervals along a portion of the coast line. The samples will then be analyzed to determine the location and amounts of pollutant present in the water. Unfortunately not only is the definition of "regular intervals" left to the oceanographer, but this oceanographer is known to be forgetful, and some of the locations might get skipped. You have been hired to determine efficiently which locations are missing.

The input to your algorithm will be an array A of locations. For simplicity, we will assume each location is a non-negative integer. If the oceanographer didn't forget any location, then the array A will be of the form $[a_1, a_1 + c, a_1 + 2c, \ldots, a_1 + (n-1)c]$, where A has length n (for $n \ge 2$) and c is the (constant) distance between two consecutive locations. However let us suppose the oceanographer skipped k locations, but that neither the first nor the last location were skipped. For example, the missing locations in $[3, 6, 12, 18, 24, 27]$ might be 9, 15 and 21.

1. [2 points] Give an example that shows that if you are not told what k is, then you can not determine c, no matter how many locations the array you are given contains.

   Solution: The array given as example ($[3, 6, 12, 18, 24, 27]$) might be missing 9, 15 and 21 ($c = 3$), but it might also be missing all of 4, 5, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 19, 20, 21, 22, 23, 25 and 26 ($c = 1$).

2. [2 points] Suppose now that you are not told k, but you are given an upper bound K on its value. You will be able to determine c and k as long as the array you receive contains at least $f(K)$ locations. What is $f(K)$, and why?

Solution: If the array contains $f(K)$ values, then it has $f(K)-1$ pairs of consecutive locations. As long as one of these pairs differs by exactly c, we will be able to find the value of c by taking its difference. How many pairs do we need for that? If the array is missing at most K locations, then having $K+1$ pairs will ensure that one of the pairs does **not** have a missing location in-between them. To have $K+1$ such pairs, we need an array with at least $K+2$ elements. So

$f(K) =$ K+2

3. [3 points] Describe an algorithm to compute the values of c and k, assuming you know K and are given an array with at least $f(K)$ locations.

   Solution: Let us assume that array indexes are numbered from 1 to n. Since the array contains at least $K+2$ different values, there are only $K+1$ gaps between these values, and so at least one of the gap will not contain any one of the missing locations. Hence

   $$c = \min_{1 \le i \le n-1} \{A[i+1] - A[i]\}$$

   and the number of missing locations is

   $$k = 1 + \frac{A[n] - A[1]}{c} - n$$

   The term $1 + (A[n] - A[1])/c$ is the number of locations we would expect an array not missing any location to have, and we subtract off the actual number of locations to determine how many are missing.

4. [6 points] Finally suppose that you are told what k is. Describe an efficient algorithm that takes as input the array and the value of k, and returns an array containing all missing locations.

   Solution: Let us once again assume that array indexes are numbered from 1 to n. We use a divide and conquer algorithm. The helper is given three additional parameters: the value of c, the number of missing elements, and the output array to fill with these missing elements. We stop the recursion as soon as the number of missing elements reaches zero.

```
Algorithm find-missing-elements(A, n, k):
    B = new array[n + k]
    c = (A[n] - A[1]) / (k + n- 1)
    find-missing-helper(A, 1, n, c, k, B)
    return B

Algorithm find-missing-elements-helper(A, low, high, c, k, B):
    if k > 0:
        if high = low + 1 # Base case: missing elements must be here
            for i = A[low] + c to A[high] - c:
                add i to B
            return

        # We still have more than two elements
        mid = (low + high) // 2

        # Compute the number of missing elements on each side.
```

```
kleft = (A[mid] - A[low])/c - (mid - low)
kright = (A[high] - A[mid])/c - (high - mid)

find-missing-elements-helper(A, low, mid, c, kleft, B)
find-missing-elements-helper(A, mid, high, c, kright, B)
```

5. [2 points] What is the worst-case running time of your algorithm, as a function of k and the length n of the array?

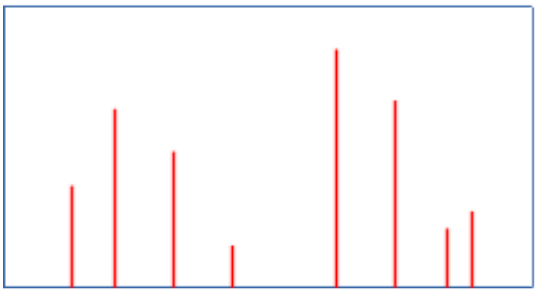
   Solution: This algorithm is rather tricky to analyze precisely. A possibly not completely tight upper-bound is that it runs in $O(k \log_2 n)$ time if $k < n/log_2 n$, in $O(n)$ time if $n/\log_2 n \leq k \leq n$, and in $O(k)$ time if $k > n$.

   A more precise analysis can show that if the k missing values are located in t of the gaps between consecutive element of A, then the number of recursive calls performed by the algorithm is at most (approximately – there are floors and ceilings involved in the exact bound) $k + t \log_2(n/t)$, where $t \leq \max\{k, n-1\}$. Unfortunately it's not entirely clear how we can turn this into a simple, closed-form function of k and n.

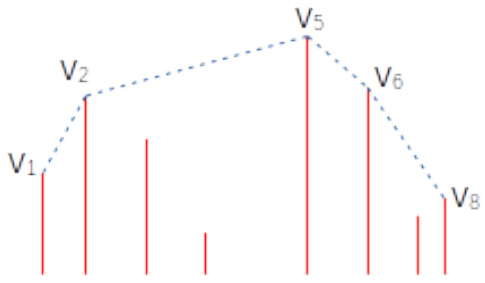
4 Rebuilding the Mars dome efficiently

Fifty years after establishing a colony on Mars, its settlers need to replace the dome that protects their colony from the martian atmosphere and cosmic rays. You have been asked to help them plan the replacement. For simplicity, we will assume that the martian surface is the x-axis, and that each of the n buildings of the colony is a vertical line segment that grows upward from the x-axis, as illustrated in the following Figure. So building i is represented by the pair (x_i, y_i) where x_i is its position along the x-axis, and y_i is its height. We will also assume that the buildings are sorted by increasing x-coordinate. That is, $x_1 < x_2 < \cdots < x_n$.

To minimize costs, the colonists are looking for a dome that uses the least material possible. Such a dome is shown in the Figure on the right: observe that it consists of a sequence of edges whose endpoints are the tops of some of the buildings, and that any horizontal line drawn through the figure either does not intersect the dome, or does so in a single line segment. We will denote an optimal dome that covers buildings x_i, x_{i+1}, ..., x_j by $D_{i,j}$. In the example in the figure, $D_{1,8}$ contains the 4 edges $\{(v_1, v_2), (v_2, v_5), (v_5, v_6), (v_6, v_8)\}$.



Current dome



New dome

1. [8 points] Design an efficient divide and conquer algorithm that takes as input a sorted array of n buildings, each represented by the x and y coordinates of its roof, and returns $D(1, n)$.

   Solution: The high-level description of the algorithm is similar to that of `Mergesort`, and can be written as follows, assuming that `B` is the array containing the coordinates of the tops of the buildings, sorted by increasing x-coordinate.

```
Algorithm BuildDome(B)
  return BuildDomeHelper(B, 0, length(B) - 1)

Algorithm BuildDomeHelper(B, first, last)
  //
  // Base case: only one building. We return an array with a single point.
  //
  if length(B) = 1 then
    return B

  //
  // Recursive case
  //
  mid ← ⌊ (first + last)/2 ⌋
  Dl ← BuildDomeHelper(B, first, mid)
  Dr ← BuildDomeHelper(B, mid+1, last)
  return DomeMerge(Dl, Dr)
```

The difficult part of the algorithm is how to merge the solutions to the two subproblems efficiently. We know from question 2 of one of the tutorial quizzes that there is exactly one edge of the solution that does not come from `Dl` or `Dr`. We also know (helped by question 3 of the same tutorial quiz) that this edge has the property that every point of `Dl` and `Dr` is either on it, or below it. There is in fact only one such edge; the issue is how to find it. Note that it does not necessarily connect the top of the highest building to another point.

We could find the edge using a (rather tricky to reason about) double binary search. However there is a simpler algorithm that will suffice for our purposes. We initially consider the rightmost point v_l of `Dl` and the leftmost point v_r of `Dr`. If the line segment that connects them does not have the property we need (that is, the neighbors of v_l on `Dl` and v_r on `Dr` are not both below the line through that line segment) then we delete either v_l (if the neighbor of v_l on `Dl` isn't below the line) or v_r (in the other case). We then repeat the process until the condition is satisfied. In pseudo-code, we get:

```
Algorithm DomeMerge(Dl, Dr)
  while true do
    //
    // We'll exit by jumping out of the loop
    //
    vl ← Dl[length(Dl) - 1]
    vr ← Dr[0]

    l ← line through vl, vr
    if length(Dl) > 1 and not isBelow(Dl[length(Dl) - 2], l) then
      delete vl from Dl
      continue

    if length(Dr) > 1 and not isBelow(Dr[1], l) then
      delete vr from Dr
      continue

    //
    // The line segment connecting vl to vr is the one we want
    //
```

```
    break
  endwhile

  return concatenate(Dl, Dr)
```

A pessimistic analysis of our algorithm would state that function `DomeMerge` runs in $O(n)$ time in the worst case, and that therefore the running time of algorithm `BuildDomeHelper` is described by the recurrence relation

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & \text{if } n \geq 2 \\ \Theta(1) & \text{if } n = 1 \end{cases}$$

and is therefore in $\Theta(n \log n)$. However this is overly pessimistic. Here's a more accurate analysis: the recursion tree has exactly $2n - 1$ nodes. If we ignore the time taken by function `DomeMerge` and the recursive calls, each call to `BuildDomeHelper` takes constant time. So the running time is in $\Theta(n)$ plus the time taken by the calls to function `DomeMerge`.

But what about the calls to `DomeMerge`? We will call `DomeMerge` $2n - 1$ times. Each iteration of the `while` loop except the last one deletes a point from the initial contents of B. So there can be at most $n - 2$ such iterations (the final dome must have at least two points). There are exactly $2n - 1$ "last" iteration of the `while` loop, one per call to `DomeMerge`. So the total amount of time spent in the calls to `DomeMerge` is in $\Theta(k(n-2) + k(2n-1))$ where k is an upper bound on the running time of a single iteration of the `while` loop (all of the steps can be performed in constant time, including checking if a point is on, above or below the line l). That is, the total time spent in the calls to `DomeMerge` is in $\Theta(n)$.

Putting all of these facts together, we can conclude that algorithm `BuildDome` runs in $\Theta(n)$ time.