CS5800 ALGORITHMS S'22 — ABHI SHELAT
DUE WED FEB 9, 2022 AT 11.59P VIA GRADESCOPE

H2.SOLUTIONS

PROBLEM 1 *BlueY*

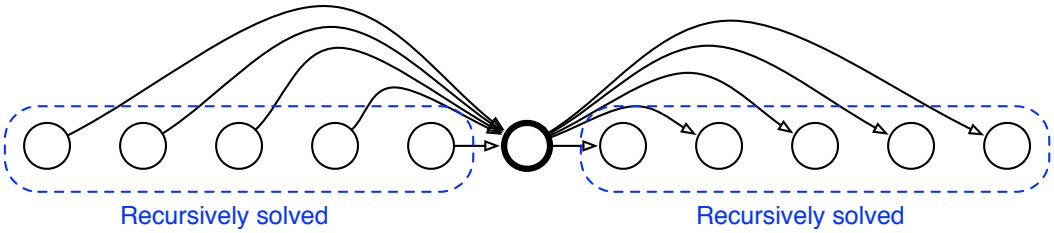
The BlueY company plans to colonize planetary systems for several nearby star systems; each system has n orbital planets of interest that start nearest from star to farthest. A key to habitable star system is good transport logistics which connects each of the n planets or objects of interest to each other via shuttles. BlueY claims to be able to develop such a system.

1. One approach to a system is to have a spaceshuttle run from the nearest planet to the farthest point as a traditional route, making a stop at every planet along the route. The system would be cheap because it only requires $O(n)$ route segments for a system of n orbital objects. However, a person traveling from planet $i = 0$ to planet $j = n$ must travel through all n segments. This system will be slow for that person.

2. You can have a special express ship run from every planet to every other destination. No person will every wait through any unnecessary segments no matter where they start and end. However, this system requires $\Theta(n^2)$ segments and will be expensive.

Suggest a better compromise: Use a divide-and-conquer technique to design a shuttle system that uses $\Theta(n \log n)$ route segments and which requires a person to wait through at most 1 extra segment when going from any planet i to any planet j (as long as $i \leq j$, i.e., we only consider far-bound routes for simplicity, and thus all ships run from the star to the farthest object only). In other words, a transport can move stuff from any i to any j by using at most 2 route segments.

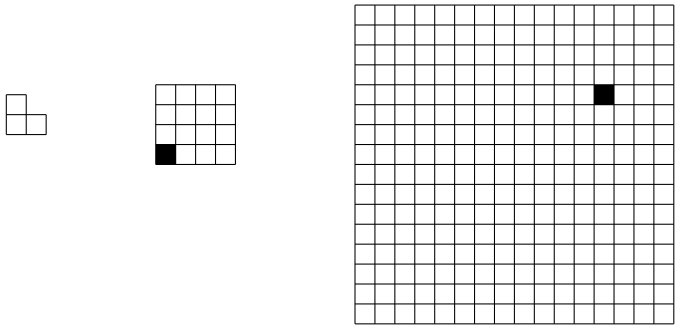
Solution: For any $n \leq 5$, 6 segments suffice. To solve larger n, find the middle planet. Recursively build a shuttle system for the left half and the right half. These systems will handle any traveller who starts and ends *entirely* in the left half or *entirely* in the right half. Now consider some traveler who starts at i in the left half and travels to a planet j in the right half. We can handle this case by adding a route segment from every point on the left half to the middle planet, and then adding a segment from the middle planet to every planet in the right half. Thus, the traveller who starts in i on the left half can take one segment to go to the middle stop, and then another segment to get to their final destination j in the right half. The total number of segments is therefore $T(n) = 2T(n/2) + n$ which solves to $\Theta(n \log n)$ by the Master's theorem.



The key insight here is that the problem is just like the Arbitrage problem in which there are three cases: (a) a morning trade, (b) an afternoon trade, or (c) a buy in the morning and a sell in the afternoon. Trades and route segments are roughly the same in this pattern; the way to handle the third case, is also slightly different: namely, we need to observe that in $\Theta(n)$ segments, we can connect any planet in the left half to any planet in the right half in at most 2 hops.

PROBLEM 2 *Missing box*

An *ℓ-tile* is an special shaped tile formed by 1-by-1 adjacent squares. The problem is to cover any 2^b-by-2^b chessboard that has one missing square (anywhere on the board) with such tiles. Such tiles must cover all squares except the missing one and no two tiles can overlap.



(a) (b) (c)

Figure 1: (a) An ℓ-tile (b) A 4×4 instance of the problem (c) A 16×16 instance

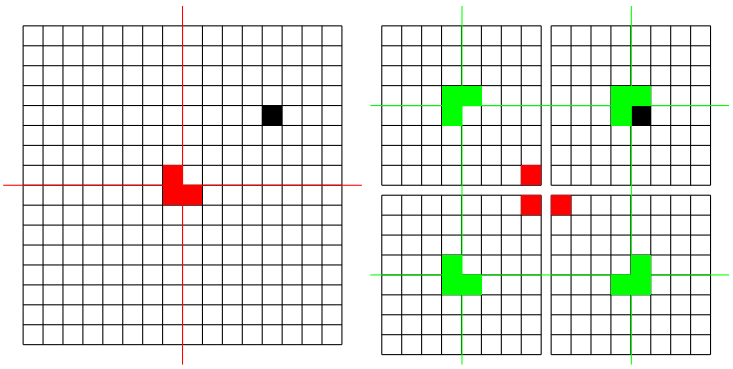
An instance is specified by b (which determines the size of the board), and the coordinates $(x, y) \in [1, \ldots, 2^b] \times [1, \ldots, 2^b]$ of the missing square in the board. A solution to an instance consists of a list of triples, where each triple describes the position of one of the tiles.

1. Describe a divide and conquer strategy for solving this problem. (Hint: Start with the base case.) Brute force search will not receive credit.

   Solution: Base case, if $b = 1$, then a single triomino, appropriately rotated, solves the problem. For larger b, divide the board into 4 sub-boards of size $b - 1$. The missing square must occur in one of the four sub-board, the other three sub-boards will be full boards. We can transform these three other sub-boards into a smaller instance of the tiling problem by using one extra triomino in order to cover one square from these three sub-boards. We can then recursively solve the four problems. See below for a diagram that shows two levels of the recursion (first in red, then in green).

2. Provide pseudo-code for your solution. You may use macros such as UPPERRIGHT(A) or LOWERLEFT(A) to refer to the upper-right (lower-left) quadrant of a two-dimensional array A.

   Solution:



TrioSolver(b, x, y)

```
1  if b = 1
2     then rotate a triomino in order to the cover the three squares.
3          return said triomino.
4  (Imagine Dividing board into four sub-boards of size b − 1 each.)
5  if (x, y) ∈ UR(b) Place triomino in center facing right.
6  elseif (x, y) ∈ UL(b) Place triomino in center facing left.
7  elseif (x, y) ∈ LR(b) Place triomino in center facing down-right.
8  else (x, y) ∈ LL(b) Place triomino in center facing down-left.
9  Solve all 4 sub-problems recursively.
10 return concatenation of solutions.
```

Notice that after line 8, all four sub-boards have one missing square, and are therefore instances of the same problem of size $b-1$.

3. Show a tight asymptotic bound for the running time of your solution.

Solution: $T(b) = 4T(b-1) + \Theta(1)$. This recurrence solves to $T(b) = \Theta(4^b)$.

Use the guess and check method. $T(b) < 4^b - d$. This is true for the base case. Suppose it is true for all b up to $b^* - 1$. Consider

$$T(b^*) = 4T(b^* - 1) + c$$

where c is the constant in the Θ-notation. By the inductive hypothesis,

$$T(b^*) < 4(4^{b^*-1} - d) + c$$

This implies that

$$T(b^*) < 4^{b^*} - 4d + c < 4^{b^*} - d$$

if $3d > c$. The lower bound follows a similar argument.

Alternatively, let $n = 2^b$, and $S(n) = T(b)$, then we can get $S(n) = 4S(n/2) + \Theta(1)$. By case 1 of Master Theorem, we get $S(n) = \Theta(n^2)$, so $T(b) = \Theta((2^b)^2) = \Theta(4^b)$.

PROBLEM 3 *Why 5 in Median?*

Recall the deterministic selection algorithm:

SELECT$(A[1,\ldots,n], i)$

1 Base case if $|A| < 5$.
2 $p \leftarrow$ MEDIANOFMEDIANS(A)
3 $A_\ell, A_r, i_p \leftarrow$ PARTITION(A, p)
4 **if** $i_p = i$ **return** $A[i_p]$
5 **elseif** $i_p < i$ **return** SELECT$(A_r, i - i_p)$
6 **else return** SELECT$(A_\ell, i_p - i)$

MEDIANOFMEDIANS$(A[1\,..\,n])$

1 Divide A into lists of 5 elements. If only one element, return it.
2 Compute the median of each small list, store these medians in a new list B
3 $p \leftarrow$ SELECT$(B, \lceil n/10 \rceil)$
4 **return** p

1. Suppose Line 1 of MEDIANOFMEDIANS divides A into lists of 3 elements each instead of 5 elements and line 3 is modified to pick the $\lceil n/6 \rceil^{\text{th}}$ element. State an upper and lower bound on the size of A_ℓ. Be as precise as you can.

   Solution:

   There are $\lceil n/3 \rceil$ lists, each of which has a median. Let x be the median of these medians; there will be $\lceil n/6 \rceil^{\text{th}}$ medians that are smaller than x. Each of these smaller medians has 1 smaller (or equal) element. As a result, we are guaranteed that

   $$|A_\ell| > 2 \cdot \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{3} \right\rceil \right\rceil - 2 \right) \geq \frac{n}{3} - 4$$

   By a similar analysis, we can argue that A_r is also just as big. Therefore,

   $$|A_\ell| \leq \frac{2n}{3} + 4$$

2. Analyze the running time of SELECT under the 3-element version of MEDIANOFMEDIANS.

   Solution: $S(n) = \Theta(n \log n)$. The first two steps of MEDIANOFMEDIANS3 take $\Theta(n)$ time. Therefore, $P(n) = S(\lceil n/3 \rceil) + \Theta(n)$. All together, we have that

   $$\begin{aligned} S(n) &= P(n) + S(2n/3 + 4) + \Theta(n) \\ &= S(\lceil n/3 \rceil) + S(2n/3 + 4) + \Theta(n). \end{aligned}$$

   Let c be the constant factor in the $\Theta(n)$ term, i.e. $S(n) = S(\lceil n/3 \rceil) + S(2n/3 + 4) + cn$. For the lower bound first, we guess that this recurrences solves to $S(n) = \Omega(n \log n)$.

Assume that $S(n) \geq d_1 n \log n$ for all $n < n_0$, for a constant d_1 to be determined. Now consider

$$\begin{aligned}
S(n_0) &= S(n_0/3) + S(2n_0/3+4) + cn_0 \\
&\geq d_1(n_0/3)\log(n_0/3) + d_1(2n_0/3+4)\log(2n_0/3+4) + cn_0 \\
&\geq d_1(n_0/3)(\log n_0 - \log 3) + d_1(2n_0/3)(\log n_0 + \log(2/3)) + cn_0 \\
&\geq d_1 n_0 \log n_0 + cn_0 - d_1(n_0/3)\log 3 + d_1 n_0(2/3)\log(2/3) \\
&= d_1 n_0 \log n_0 + cn_0 - d_1(n_0/3)\left(\log 3 - 2\log(2/3)\right) \\
&\geq d_1 n_0 \log n_0 + cn_0 - d_1 n_0 \\
&\geq d_1 n_0 \log n_0.
\end{aligned}$$

The second last line follows from the fact that $(\log 3 - 2\log(2/3)) = \log_2 \frac{27}{4} < \log_2 8 = 3$. We set the constant $d_1 \leq c$ so the last line holds. In the second step, we can ignore the $+4$ terms because we are determining a lower bound and the log function is an increasing function.

To show the upper bound, assume that $S(n) \leq d_2 n \log n$ for all $n < n_0$, for a constant d_2 to be determined. Now consider

$$\begin{aligned}
S(n_0) &= S(n_0/3) + S(2n_0/3+4) + cn_0 \\
&\leq d_2(n_0/3)\log(n_0/3) + d_2(2n_0/3+4)\log(2n_0/3+4) + cn_0 \\
&\leq d_2(n_0/3)(\log n_0 - \log 3) + d_2(2n_0/3+4)\log n_0 + cn_0
\end{aligned}$$

This last line follows because for sufficiently large n_0, n_0 will be larger than $\frac{2}{3}n_0 + 4$. Back to the proof, we can conclude

$$\begin{aligned}
&= d_2(n_0/3)(\log n_0 - \log 3) + d_2(2n_0/3+4)\log n_0 + cn_0 \\
&= d_2 n_0 \log n_0 - \frac{d_2 \log 3}{3} n_0 + 4d_2 \log n_0 + cn_0 \\
&\leq d_2 n_0 \log n_0.
\end{aligned}$$

We set $d_2 > \frac{3}{\log 3}c$ so that for sufficiently large n_0, the $-\frac{d_2 \log 3}{3}n_0$ term will overwhelm the $+4d_2 \log n_0 + cn_0$ terms, thus $-\frac{d_2 \log 3}{3}n_0 + 4d_2 \log n_0 + cn_0 < 0$, therefore the last line holds.

PROBLEM 4 *Peaks*

On a clear day, the shores of lake Zurich offer a sublime view of the alpine mountainscape. Define the *left alpine function*, denoted $a_\ell(s)$, of a mountain scape s as the total number of times that a peak is taller than one of its left neighboring peaks. The *right alpine function*, $a_r(\cdot)$, is defined analogously. The alpine mountainscape s below has 8 peaks, $a_\ell = 19$, the contribution of each peak is listed below the building.

Design and analyze a divide and conquer algorithm that computes both the left and right alpine functions of a mountainscape s with n peaks. The input $A[1, \ldots, n]$ consists of the heights of each peak in left to right order; assume

all peaks have unique heights. Your solution should have a running time of $\Theta(n \log n)$ and should include an analysis of the running time.

Solution: We use a divide and conquer approach: split the input in half, recursively determine the left and right alpine scores of both halves, and then recombine them in time $O(n)$ in order to get an algorithm with running time $T(n) = 2T(n/2) + O(n) = \Theta(n \log n)$.

The issue is to devise a way to combine the left and right scores from each half into a global alpine score. The simple approach of just checking each element in the left half with each element in the right half would lead to an $O(n^2)$ algorithm. Instead, note that if the two smaller halves are sorted, then it is easy to compute—say—the global right alpine score: just add the number of times each element in the left half is larger than an element in the right half. Because the lists are sorted, this can be done just using one pass through the left and right sub-arrays. More specifically, suppose we find that the first peak in the left sorted list is taller than the first j buildings in the right sorted list. Then we immediately know that all remaining peaks in the left list will also be taller than these first j and we can add appropriately to the n_ℓ score. Then, we can start comparing the second peak of the left list starting from position j of the right list.

Unfortunately, if we sort each right and left list separately, that would increase the work to $2T(n/2) + (n/2)\log(n/2)$ which is no longer a $\Theta(n \log n)$ solution. Instead, we slightly modify the job of the recursive function as we did in class. We require that *in addition* to computing the left/right alpine scores, the function must also sort the list. In other words, we add the merge-sort algorithm into this procedure as follows:

Algorithm 1 $(n_\ell, n_r, C[1, \ldots, n]) \leftarrow$ alpine(A[1,. . . ,n])

1: If list has size 1 return $(0, 0, A)$.
2: $(n_\ell, n_r, L) \leftarrow$ alpine$(A[1, \ldots, n/2])$
3: $(m_\ell, m_r, R) \leftarrow$ alpine$(A[n/2 + 1, \ldots, n])$
4: $o_\ell \leftarrow$ Combine-Left$(L[1, \ldots, n/2],\ R[1, \ldots, n/2])$
5: $o_r \leftarrow$ Combine-Right$(L[1, \ldots, n/2],\ R[1, \ldots, n/2])$
6: Merge lists $L[1, \ldots, n/2], R[1, \ldots, n/2]$ into list C
7: **return** $(n_\ell + m_\ell + o_\ell, n_r + m_r + o_r, C)$

Steps 1,4,5,6 require $\Theta(n)$ time, and steps 2, 3 require $T(n/2)$ time for a total running time of $2T(n/2) + \Theta(n) = \Theta(n \log n)$ as desired. All that remains is to show how the Combine-Left and Combine-Right functions work. Here, we only illustrate the left case in Algorithm 2; the right case is similar. First, correctness. The procedure maintains two pointers, r, ℓ into the lists R, L respectively. If the value $R[r]$ is greater than $L[\ell]$, then peak r and all peaks that are taller than r contribute to the left alpine score. In this case, we adjust the left score and increment the left pointer ℓ. (See lines 4,5). Otherwise, if peak r is smaller, it does not contribute, so we move on to the next peak in the right list (see line 7). For running time, notice that each execution of the loop increments either ℓ or r. Therefore, the loop can run at most time $2n$. The remaining operations take constant time.

An interesting observation is that each pair of peaks (i, j) must contribute to either the left skyline or the right skyline since either peak i is taller than peak j or vice versa. This implies that the sum of the left and right alpine score must equal the number of unique pairs of peaks, i.e. $n(n+1)/2$. Thus another way to compute both is to compute the left score and subtract that value from the number of unique pairs.

Algorithm 2 Combine-Left($L[1, \ldots, n]$, $R[1, \ldots, n]$) (Assume L, R are sorted.)

```
Initialize return value to N ← 0 and pointers ℓ, r ← 1
while r ≤ n and ℓ ≤ n do
    if R[r] > L[ℓ] then
        N ← N + (n − r + 1)
        Increment ℓ
    else
        Increment r
    end if
end while
return N
```