CS 6150 - Fall 2025 - HW1
Data structures, Divide and conquer, Dynamic programming

Submission date: Friday, Sep 26, 2025 (11:59 PM)

This assignment has 6 questions, for a total of 110 points. You will still be graded out of 100, and any points you earn above 100 will count as bonus and can compensate for a low score on other homeworks. Unless otherwise specified, complete and well-reasoned formal arguments are expected in all answers.

Question	Points	Score
Prefix trees / tries	14	
Inverted Index	16	
Divide and conquer	20	
Divide and conquer: Median of medians	20	
Recurrences, recurrences	20	
Dynamic programming	20	
Total:	110	

Question 1: Prefix trees / tries .... **[14]**

Prefix trees can be used as a "poor man's binary search tree". Let us see the sense in which this is true. Suppose we have a set of N integers all in the range $[0, 2^k - 1]$ for some constant k. If we treat them as strings in base 2^d (assume d divides k) and use a trie data structure to store them:

(a) **[4]** How many characters (at most) are needed to represent any integer in the given range in base 2 (i.e. binary) and in base 2^d?

We know that with k bits we can represent 2^k different numbers in binary. Because of this fact we know that to represent any integer in the given range in base 2 we would need at most k bits.

Figuring out how many characters are needed in base 2^d is a little more tricky. The technique to do this is to set your k to something fixed like 4. Then check and see how many characters are needed to represent the largest number in the divisible values of d. In this example we would check how many characters are needed to represent 15 with $d = 1, 2, 4$. From this the patern emerges that you need $\frac{k}{d}$ characters to represent all possible numbers in the given range. See 'Problem 1a.png' for detailed work.

(b) **[4]** What is the depth of the trie constructed as described above? **The maxium depth of the trie constructed as above would simply be the number of characters needed to represent all numbers in the range. This would also be $\frac{k}{d}$**

(c) **[2]** How long is the query time and insertion time in this trie? (Assume the integer is already converted to base 2^d.)

Both the insertion time and the query time will have the same runtime. In all cases this runtime is directly related to the number of characters in the largest string in your "alphabet". This would also be $o(\frac{k}{d})$

(d) **[4]** If we decide to store N integers (from the same range) in a sorted array instead of the prefix tree, how is the query time affected by this choice of data structures?

In this case because it is a sorted array we can use binary serach to get a runtime of $O(log_2(N))$.

Question 2: Inverted Index .... **[16]**

(a) **[4]** Suppose that we have a list of documents containing a series of words as follows:

doc1 → "Data structures help algorithm design"
doc2 → "Search data"
doc3 → "Design search algorithm"
doc4 → "New data structures help search algorithm design"
doc5 → "Design new data structures"

Create an inverted index data structure for this data. Now, you are given a query to search: [algorithm, data, structures, design]. Your goal is to find documents that contain all four words. Use your inverted index to find which document contains all four words.

To set this up you can use a dictionary that is keyed by a unique word. Each key would correspond to a list of all documents that word appears in. To accomplish this simply loop through all documents and add the document name to the corespdoning "wordlist" as you see them.

```
{data: [doc1, doc2, doc4, doc5] }
{structures: [doc1, doc4, doc5] }
{help: [doc1, doc4] }
{algorithm: [doc1, doc3, doc4] }
{design: [doc1, doc3, doc4, doc5] }
{search: [doc2, doc3, doc4] }
{new: [doc4, doc5] }
```

Doc 1 and Doc 4 contain all four words

(b) **[8]** When answering search queries using an inverted index, we saw that the key step is to take the intersection of the inverted index sets corresponding to the words in the query. Suppose that these sets are of size $s_1, s_2, \ldots, s_t$ respectively (say we have t words in the query). Naïvely computing the intersection is not great if one of the s_i is large (e.g., if one of the words in the query is a frequently-occurring one). Assume that the documents in each inverted index set are sorted (say by docID). Now, give an algorithm whose running time is

$$O\big(s(\log s_1 + \log s_2 + \cdots + \log s_t)\big), \quad \text{where } s = \min_{1 \le i \le t} s_i.$$

To get the desired runtime for a search query we use the following algorihtm:

1. **Assume all inverted index "wordsets" are sorted.**
2. **For each search query (string of words) look at each inverted index associated with that word. Call this set of "wordsets" S Call the "wordset" with the shortest length "s_i"**
3. **Iteratively search through each element in the the "wordset" for s_i. For each element use binary search to search for that element in all other wordsets. Keep track of which element in the wordset s_i has the highest number of occurences across all wordsets in S and return that document.**

(c) [**4**] Why is an inverted index more efficient than a normal index for querying documents that contain a given set of words. (e.g., Searching all web pages that includes words from a particluar google search query)?

Because the number of unique words is much smaller than the number of webpages that need to be searched. With inverted index there is defintely a preprocessing cost which comes from looping through every individual document and populating the ineverted index sets. But as we saw above the amount of time it takes to search is much smaller so long as the number of documents is smaller than s_i. We no longer have to loop through every single document for every search query.

Question 3: Divide and conquer .... [**20**]

(a) [**10**] Let $A[0], A[1], \ldots, A[n]$ be an integer array of size n, where $n > 2$. We define array A to be *convex* if there exists an index 'i' such that $A[0], A[1] \cdots A[i]$ is a decreasing sequence and $A[i], A[i+1] \cdots A[n]$ is an increasing sequence. The goal is to find such index 'i' given a *convex* array A. Give an algorithm for this task that makes only $O(\log n)$ queries to the array A. [*Interesting fact:* This algorithm can be used to find the minimum value of a *convex* function without calculating the derivative.]

Because we know that the array is convex, we know that there exists a pivot point i. To find this pivot point we can use a binary search with slight modification. After we choose the center index we can check if the value directly to the right is greater than or less than the current value. If it is less than we know that our pivot point must be on the right so we can get rid of the left half of the array. If it is greater than, we know that our pivot point must be on the left so we can throw away the right. Just like binary search this has a runtime of $Olog(n)$

At every step this algorithm cuts the array in half. No matter what branch of the recursion you go down you will only be querying the array $Olog(n)$ times. No matter how many branches we end up going down (from the recursion) each branch will only take $Olog(n)$ time. Each extra branch will be covered in the constant.

When we cut the array in half don't forget to include the index you compared the right side with.

(b) [**10**] "Test pooling" is a method used when testing for a disease is expensive or if tests have limited availability. The idea is as follows: suppose we have samples drawn from n people (). Instead of testing each sample separately, samples from a subset S of the people are combined and a single test is performed on the pooled sample. If at least one of the people in S has the disease, the test comes out positive, and if none of the people in S have the disease, it comes out negative. (Let us ignore the test error for this problem, i.e. assume that the result of the test is always reliable.) It turns out that if only a few people have the disease, pooling gives a much faster way to find out who has the disease, compared to testing everyone. Suppose you have samples from n people, indexed $1, 2, \ldots n$, and you are told that **exactly four** of them have the disease. Give an algorithm that finds the indices of the persons with the disease using at most $O(\log n)$ (pooled) tests. Assume that you can draw as many samples from each person as you want. [*Hint:* First think about solving the same question when **exactly one** of them has the disease.]

To solve this problem we will use divide and conquer and recursion. At each step of the recursion we will start by cutting our list in half and checking both the left and right side for a positive case. If the left side has a positive case we will call our recursive method with the left side of the list as input. If the right side has a positive case we will call our recurisve method with the right side of the list as input. The base case of the recursion would be when the length of the input equals 1. At this point we would simply run our test a final time to decide if that index is infected. If it is we will add it to a global list.

This approach would take $O(log(n))$ for each branch of the recursion. Beucase multiple different branches will just add a constant, the final runtime would be $O(log(n))$

Question 4: Divide and conquer: Median of medians .... **[20]**

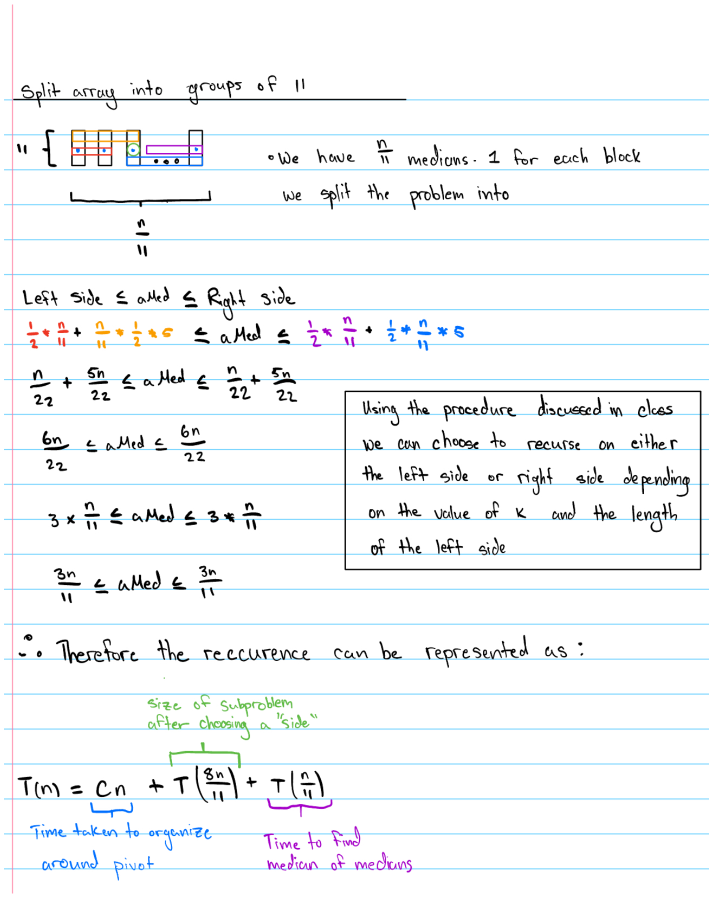
Recall the linear time selection algorithm we saw in class (median-of-medians) and answer the following questions. Please provide formal justification.

(a) [**8**] Instead of dividing the array into groups of 5, suppose we divided the array into groups of 11. Now sorting each group is slower. But what would be the recurrence we obtain? [*Hint:* What would the size of the sub-problems now be?]

Using the techniques discussed in class the first step is to prove that the median of medians is truly a median. We ensure this by making sure it is in between some fraction of the other elements. We can also use the techniques discussed in class to count how many of the elements get removed at each step. Because I had to use diagrms in my work I have attached hand written notes below

Solution:

$$T(N) \leq c * n + T(\frac{8n}{11}) + T(\frac{n}{11})$$



Split array into groups of 11

$11 \{ \quad \underbrace{\qquad\qquad}_{\frac{n}{11}}$

• We have $\frac{n}{11}$ medians. 1 for each block

we split the problem into

Left Side ≤ aMed ≤ Right side

$\frac{1}{2} * \frac{n}{11} + \frac{n}{11} * \frac{1}{2} * 5 \leq aMed \leq \frac{1}{2} * \frac{n}{11} + \frac{1}{2} * \frac{n}{11} * 5$

$\frac{n}{22} + \frac{5n}{22} \leq aMed \leq \frac{n}{22} + \frac{5n}{22}$

$\frac{6n}{22} \leq aMed \leq \frac{6n}{22}$

$3 \times \frac{n}{11} \leq aMed \leq 3 * \frac{n}{11}$

$\frac{3n}{11} \leq aMed \leq \frac{3n}{11}$

Using the procedure discussed in class we can choose to recurse on either the left side or right side depending on the value of k and the length of the left side

∴ Therefore the reccurence can be represented as :

$$T(n) = \underbrace{Cn}_{\text{Time taken to organize around pivot}} + \overbrace{T\left(\frac{8n}{11}\right)}^{\text{size of subproblem after choosing a "side"}} + \underbrace{T\left(\frac{n}{11}\right)}_{\text{Time to find median of medians}}$$

Figure 1: The New Reccurence

(b) [**6**] Analyze the runtime of your new recurrence. Is it still linear?
This problem can be sovled using the Akra Bazzi formula

$$a_1 = a_2 = 1, k = 2, b1 = \frac{8}{11}, b_2 = \frac{1}{11} \tag{1}$$
$$\frac{8}{11}^p + \frac{1}{11}^p = 1 \tag{2}$$
$$p = 0.6815 \tag{3}$$
$$T(n) = O(n^p(1 + \int_1^n \frac{u}{u^{p+1}} du)) = O(n^{0.6815}(1 + \int_1^n u^{-0.6815} du)) \tag{4}$$
$$T(n) = O(n^{0.6815}(1 + \frac{n^{0.3185-1}}{0.3185})) = O(\frac{1}{c} * n^{0.6815} * n^{0.3185}) \tag{5}$$
$$T(n) = O(n) \tag{6}$$

(c) [**6**] Say we defined "almost median" differently, and we ended up with a recurrence $T(n) = T(4n/5) + T(n/5) + n$. Does this still lead to a linear running time?
The formal way to go about it is using the akra bazi theorem. The first step is to solve for p

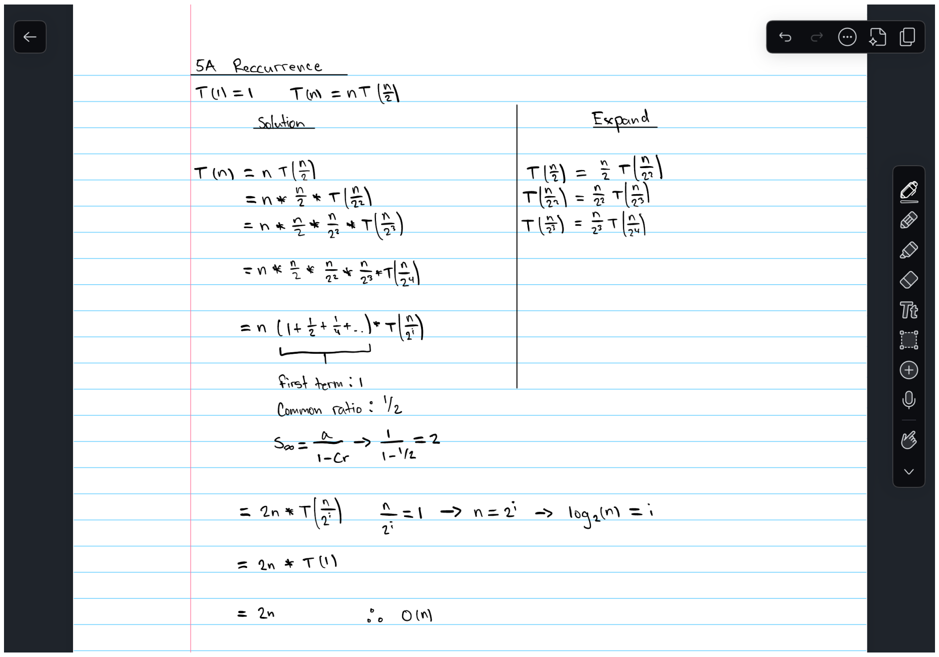
$$a_1 = a_2 = 1, k = 2, b1 = \frac{4}{5}, b_2 = \frac{1}{5} \tag{7}$$
$$p = 1 \tag{8}$$
$$\frac{4}{5}^p + \frac{1}{5}^p = 1 \tag{9}$$
$$T(n) = O(n^p(1 + \int_1^n \frac{u}{u^{p+1}} du)) = O(n(1 + \int_1^n \frac{u}{u^2} du)) \tag{10}$$
$$T(n) = O(n(1 + \ln(n))) = O(nlog(n)) \tag{11}$$

Question 5: Recurrences, recurrences .... [**20**]
Solve each of the recurrences below, and give the best $O(\cdot)$ bound you can for each of them using iterative substitution or recursion tree. Show your work.
Note: when we write $n/2$, $n/3$, etc. (any smaller quantity expressed in terms of n) we mean the floor (closest integer less than the number) of that quantity.

(a) [**8**] $T(n) = nT\left(\frac{n}{2}\right)$. Use $T(1) = 1$ as the base case.
To solve this problem we will use the plug and chug method.

$$T(n) = nT(\frac{n}{2}) \tag{12}$$
$$T(n) = n * \frac{n}{2} * T(\frac{n}{2^2}) \tag{13}$$
$$T(n) = n * \frac{n}{2} * \frac{n}{2^2} * T(\frac{n}{2^3}) \tag{14}$$
$$T(n) = n(1 + \frac{1}{2} + \frac{1}{4}) * T(\frac{n}{2^i}) \tag{15}$$
$$\frac{n}{2^i} = 1 \tag{16}$$
$$\log_2 n = i \tag{17}$$
$$T(n) = 2n * T(1) = 2n \tag{18}$$
$$O(n) \tag{19}$$

More work can be seen in the screenshot below. Once we solve the reccurence we get the following runtime $O(n)$.



5A Reccurrence

$T(1) = 1 \qquad T(n) = nT\left(\frac{n}{2}\right)$

Solution

$$T(n) = n\,T\left(\tfrac{n}{2}\right)$$
$$= n * \tfrac{n}{2} * T\left(\tfrac{n}{2^2}\right)$$
$$= n * \tfrac{n}{2} * \tfrac{n}{2^2} * T\left(\tfrac{n}{2^3}\right)$$
$$= n * \tfrac{n}{2} * \tfrac{n}{2^2} * \tfrac{n}{2^3} * T\left(\tfrac{n}{2^4}\right)$$
$$= n\,\underbrace{\left(1 + \tfrac{1}{2} + \tfrac{1}{4} + ..\right)}_{} * T\left(\tfrac{n}{2^i}\right)$$

first term : 1

Common ratio : 1/2

$$S_\infty = \frac{a}{1-Cr} \rightarrow \frac{1}{1-1/2} = 2$$

$$= 2n * T\left(\tfrac{n}{2^i}\right) \qquad \frac{n}{2^i} = 1 \rightarrow n = 2^i \rightarrow \log_2(n) = i$$
$$= 2n * T(1)$$
$$= 2n \qquad \therefore O(n)$$

Expand

$$T\left(\tfrac{n}{2}\right) = \tfrac{n}{2}\,T\left(\tfrac{n}{2^2}\right)$$
$$T\left(\tfrac{n}{2^2}\right) = \tfrac{n}{2^2}\,T\left(\tfrac{n}{2^3}\right)$$
$$T\left(\tfrac{n}{2^3}\right) = \tfrac{n}{2^3}\,T\left(\tfrac{n}{2^4}\right)$$

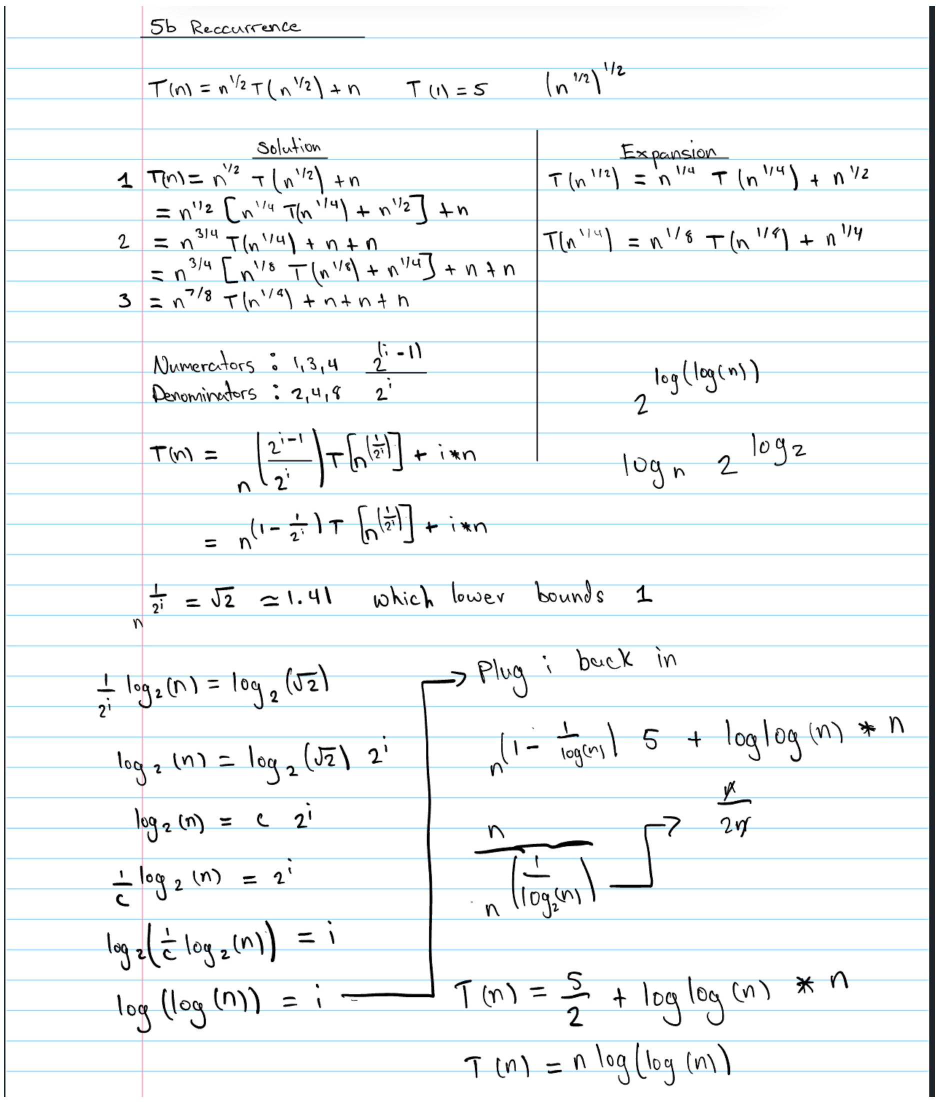
Figure 2: Solving Recurrence 5a

(b) [8] $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n$. Use $T(1) = 5$ as the base case. [*Caution: $T(1)$ may not be directly substitutable in the last step of iterative substitution.*]

We can also solve this using plug and chug. The work can be seen in the screenshot below. Once we solve the reccurence we get the following runtime $O(nlog(log(n)))$.

It is helpful to remember that $\sqrt{x} = x^{\frac{1}{2}}$ It is also important to see that the base case can't be used directly. We can use $sqrt(2) = 1.41$ becuase it bounds the base case

$$T(n) = n^{\frac{1}{2}}T(n^{\frac{1}{2}}) + n, T(1) = 5 \tag{20}$$
$$T(n) = n^{\frac{3}{4}}T(n^{\frac{1}{4}}) + n + n \tag{21}$$
$$T(n) = n^{\frac{7}{8}}T(n^{\frac{1}{8}}) + n + n + n \tag{22}$$
$$T(n) = n^{\frac{2^{i-1}}{2^i}} * T(n^{\frac{1}{2^i}}) + i * n \tag{23}$$
$$i = \log_2 \log_2 n \tag{24}$$
$$T(n) = n^{(1-\frac{1}{log(n)})} + \log_2(\log_2(n)) * n \tag{25}$$
$$T(n) = \frac{5}{2} + \log_2(\log_2(n)) * n \tag{26}$$
$$T(n) = O(\log_2(\log_2(n)) * n) \tag{27}$$



5b Reccurrence

$T(n) = n^{1/2} T(n^{1/2}) + n \qquad T(1) = 5 \qquad (n^{1/2})^{1/2}$

Solution

1 $T(n) = n^{1/2} T(n^{1/2}) + n$

$= n^{1/2} [n^{1/4} T(n^{1/4}) + n^{1/2}] + n$

2 $= n^{3/4} T(n^{1/4}) + n + n$

$= n^{3/4} [n^{1/8} T(n^{1/8}) + n^{1/4}] + n + n$

3 $= n^{7/8} T(n^{1/8}) + n + n + n$

Expansion

$T(n^{1/2}) = n^{1/4} T(n^{1/4}) + n^{1/2}$

$T(n^{1/4}) = n^{1/8} T(n^{1/8}) + n^{1/4}$

Numerators : 1,3,4 $\quad \frac{2^{i}-1}{2^i}$

Denominators : 2,4,8 $\quad 2^i$

$2^{\log(\log(n))}$

$T(n) = n^{\left(\frac{2^i-1}{2^i}\right)} T\left[n^{\left(\frac{1}{2^i}\right)}\right] + i*n$

$\log_n 2^{\log_2}$

$= n^{\left(1-\frac{1}{2^i}\right)} T\left[n^{\left(\frac{1}{2^i}\right)}\right] + i*n$

$n^{\frac{1}{2^i}} = \sqrt{2} \simeq 1.41$ which lower bounds 1

$\frac{1}{2^i} \log_2(n) = \log_2(\sqrt{2})$

$\log_2(n) = \log_2(\sqrt{2})\, 2^i$

$\log_2(n) = c\, 2^i$

$\frac{1}{c} \log_2(n) = 2^i$

$\log_2\left(\frac{1}{c}\log_2(n)\right) = i$

$\log(\log(n)) = i$

→ Plug i back in

$n^{\left(1-\frac{1}{\log(n)}\right)} 5 + \log\log(n) * n$

$\frac{n}{n^{\left(\frac{1}{\log_2(n)}\right)}} \rightarrow \frac{\cancel{n}}{2\cancel{n}}$

$T(n) = \frac{5}{2} + \log\log(n) * n$

$T(n) = n\log(\log(n))$

Figure 3: Solving Recurrence 5b

(c) [4] Imagine an algorithm B that works by halving the input size repeatedly (something like binary search where the input size for the next step of the problem is half of the previous input size, and **the other half is discarded**). However, **unlike binary search** which splits the input down the middle in constant time, B needs to determine a custom split by reading each element and deciding whether to put it in the first half or the second. This takes linear time, but still divides the input in two equal halves. [E.g. input: $\{a, b, c, d, e, f\}$, after split: $\{a, c, f\}, \{b, d, e\}$] Then, B picks one of the halves, discards the other, and continues to do so until it reaches an input size of 1, which it solves in constant time.

i. Formulate a recurrence relation for this algorithm. [*Use constants like $c_1, c_2, \ldots$ if/when needed.*]

ii. Solve the recurrence relation to determine a big-O bound on B.

(i) To come up with the recurrence we will use the standard formula:

Number of Subproblems $*$ Size of each Subproblem $*$ Cost to Divide / Combine

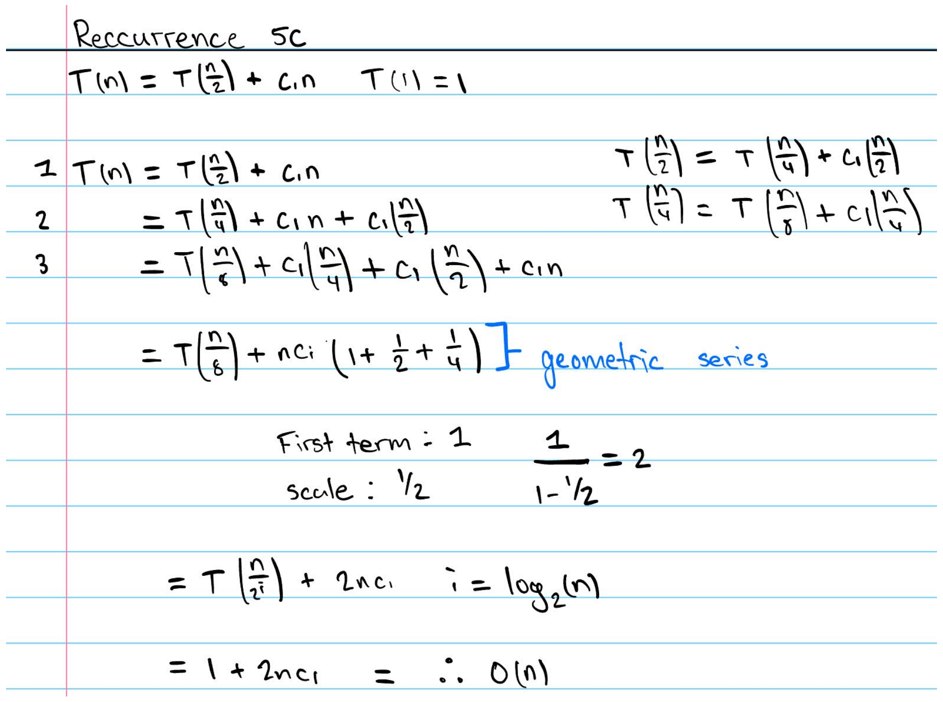
We only have 1 subproblem at each step. Each subproblem is of size ($T\frac{1}{2}$). The cost to divide and combine is simply the cost to divide which was given to be linear. Therefore our solution is

$$T(n) = T(\frac{n}{2}) + c_1 n, T(1) = 1$$

(ii) Using this information and following the work in the image below we can see that the

big-O bound is $O(n)$

$$T(n) = T(\frac{n}{2}) + c1n, T(1) = 1 \tag{28}$$
$$T(n) = T(\frac{n}{4}) + c1n + c1(\frac{n}{2}) \tag{29}$$
$$T(n) = T(\frac{n}{8}) + c1n + c1(\frac{n}{2}) + c1(\frac{n}{4}) \tag{30}$$
$$T(n) = T(\frac{n}{8}) + nci(1 + \frac{1}{2} + \frac{1}{4} + ...) \tag{31}$$
$$T(n) = T(\frac{n}{2i} + 2nci) \tag{32}$$
$$i = \log_2 n \tag{33}$$
$$O(n) \tag{34}$$



Reccurrence 5c

$T(n) = T(\frac{n}{2}) + c_1 n \quad T(1) = 1$

1 $T(n) = T(\frac{n}{2}) + c_1 n$ $\qquad T(\frac{n}{2}) = T(\frac{n}{4}) + c_1(\frac{n}{2})$

2 $= T(\frac{n}{4}) + c_1 n + c_1(\frac{n}{2})$ $\qquad T(\frac{n}{4}) = T(\frac{n}{8}) + c_1(\frac{n}{4})$

3 $= T(\frac{n}{8}) + c_1(\frac{n}{4}) + c_1(\frac{n}{2}) + c_1 n$

$= T(\frac{n}{8}) + nc_1 \left(1 + \frac{1}{2} + \frac{1}{4}\right)$] geometric series

First term : 1 $\qquad \frac{1}{1-1/2} = 2$

scale : $1/2$

$= T(\frac{n}{2^i}) + 2nc_1 \qquad i = \log_2(n)$

$= 1 + 2nc_1 = \therefore O(n)$

Figure 4: Solving Recurrence 5c

Question 6: Dynamic programming . . . . **[20]**

Imagine a village with a few houses that need a new water pipeline. We know the following locations as co-ordinates on a village map (consider XY plane): the reservoir $h_0(x_0, y_0)$, and n houses that need water, named $h_1, h_2, \ldots, h_n$ with co-ordinates $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$. The reservoir and the houses are sequenced such that their X co-ordinates are in an increasing order, i.e., $x_0 < x_1 < x_2 < \cdots < x_n$.

We have an oracle (magic procedure) called *'line_fit'* that takes a set of points (co-ordinates), performs linear regression (line fitting) on it, and returns the sum of distances of all points from the line of best fit. We do not care how this oracle works, but use it to determine how to lay the pipeline. However, we realize that if we lay a single long, straight segment of pipe using *line_fit*, the sum of distances of all the houses from the pipeline is large, and necessitates long 'offshoots' for every house (see $d_0, d_1, \ldots, d_7$ in figure 5). The lengths of offshoots can be reduced by using more pipe segments joined together instead of a single long segment. E.g. in figure 6, there are 4 segments and the sum of distances d_0 through d_7 is much smaller. But, to lay the pipeline like this, we need to weld the pipe segments to form a continuous pipeline.

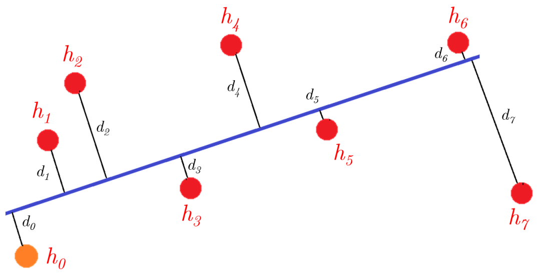


Figure 5: One segment, long offshoots

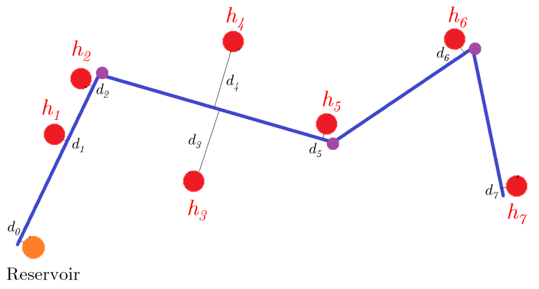


Figure 6: Four segments, short offshoots

We are now faced with the following problem. The government is only funding the pipe segments (irrespective of their lengths). We need to bear the cost of welding the segments and of the 'offshoots' for each house. The cost of welding is W dollars **per segment**, and the cost of offshoots is C dollars **per unit length** (remember that *line_fit* returns the total length of all offshoots). We need to minimize the cost of the project (welding + offshoots). If we have too few segments, we need longer offshoots for the houses, thus driving up the cost. If we have too many segments (e.g. one segment serving each house), you need shorter branches but more welded joints, which can again be costlier. In theory, the number of such arrangements we need to consider is huge (non-polynomial in n). We can imagine that there is a certain number of segments that gives us the best trade-off between welding and offshoots, and minimizes the total cost. E.g. figure 7 uses only two segments and the offshoots are not as long as in figure 5.

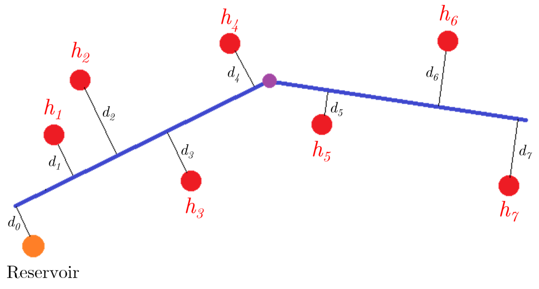


Figure 7: Two segments, not too long offshoots

There are certain constraints to be considered. The pipeline has to be one continuous entity without any forks. Each segment must serve only consecutive houses from the given sequence (e.g. a segment cannot serve h_1, h_2, h_4 without serving h_3). In other words, *line_fit* only accepts consecutive points $h_i, h_{i+1}, \ldots, h_j$ as arguments, and returns the optimal value of $\sum_{r=i}^{j} d_r$ which is the sum of distances for the line of best fit. Each house must be connected, and every segment of the pipeline must serve at least one house.

Considering these constraints and using *line_fit* as a subroutine, answer the following questions.

(a) [**4**] Write the cost optimization equation for this problem in terms of n, W, C, and *line_fit*. [*Hint: The cost of offshoots in each segment is $C \times$ the sum of distances returned by* line_fit *for that segment. The length of the segment itself does not matter.*]

Uppercase S is the number of segments

Lowercase s_i is the set of the houses in the current segment

$OPT(0, n) = \min_{0 \le i \le n} W + C * line\_fit(0, i) + OPT(i+1, n)$

(b) [**10**] Design a dynamic programming algorithm that takes the co-ordinates of the reservoir and n houses as input, and returns the minimum **cost** of the pipeline.

a: seg_start

We are using an array costs[] for our lookup table that is size n. All values in the lookup table are initialized to inf **at the start of the algorihtm.**

Algorithm 1 $OPT(a)$

```
$costs = []$                                                    ▷ All values initialized to $+\inf$
if $a == n$ then
    return $0$
end if
if $costs[a] \neq \inf$ then
    return $costs[a]$
end if
for $a \leq i \leq n$ do
    $costs[a] = \min(costs[a], (W + C * line\_fit(a, i) + OPT(i + 1, n)))$
end for
return $costs[a]$
```

(c) **[4]** Analyze the running time of the algorithm (treat the oracle as constant-time).

To analyze the runtime we realize that every single subproblem will only need to be sovled once. So if we can find out how many subproblems there are total and how long each one takes we will have our runtime.

The number of subproblems is exactly equal to the number of nodes in the problem (the number of houses + the reservoir)

When we are at the bottom of recursion where the base case happens no lookups must be made. When we are at the problem one level above that exactly one lookup must be made. When we are two levels above we have two lookups. We also have to compute the minimum of all values looked up. If you follow this pattern clear up to the start of your lookup table you get the following relationship. (Note: c is the time it takes to find the minimum over all lookup values being compared)

$$nc + (n-1)c + (n-2)c + ... = \frac{c * n(n+1)}{2}$$

Removing constants and doing big o analysis we get the following

$$O(n^2)$$

(d) **[2]** Comment on the benefit of using dynamic programming to solve this problem.

Dyanmic programming is often useful for problems where we need to try every possibility and have multiple recursive branches. We notice above that a huge part of us being able to get the runtime specified is because we know that we only have to solve every subproblem once. If we hadn't done that every single branch would have to resolve the subproblem. This would take factorial in the length of the input for every single problem. This means it is at least as bad as $n!$ but is actually higher than that.

From this it is obvious to see that dynamic programming can offer huge benefits if we are sovling the same problem mutliple times.