Prob. 1	Prob. 2	Prob. 3	Prob. 4

Problem 1.

- Max potential(Proof by contradiction): Suppose that there is another binary tree with a greater potential than the linked list but with the same n number of nodes. It would mean that at some height of the tree there is one node having more nodes in its subtree than the equivalent node in the linked list. However this would be impossible since in a linked list, the only nodes of a tree that are not included in a node's subtree are its (grand)parent node(s). Therefore, assumption must be wrong and linked list has the maximum potential. Moreover, the same conclusion can be drawn by construction. Tree has n elements so $w(root)$ is n and child node can at most have $n-1$ nodes in its subtree and following its child can at most have $n-2$ nodes. This ends up in linked list structure.

- Min potential: Following the same reasoning in max potential, if a parent node divides its subtrees equally between children then children nodes can have the minimum potential because for potential we take the logarithms of the number of nodes and it depends on the height of the tree where summation of number of nodes in two children doesn't matter. Parent node has m nodes, first child has m_1 and so second child has $m-m_1$ nodes:

$$min\{\log(m_1)+\log(m-m_1)\}$$

  The minimum of this function is defined at $m_1 = \frac{m}{2}$ and that gives the structure of complete binary tree.

Problem 2.

Let's say we have a request of m elements with n distinct elements in it. Then if $m = n$, any order of the list with those n elements would give the same cost: $1+2+...+n$.
Now assume we have elements in the request that appear more than once, then the order matters. The goal to minimize the cost is that the access cost of an element appearing often in the request should be low. Thus if we order the list given the number of time it appears in the request (frequency), we get the static optimal ordering. The optimized cost is:

$$X_1 * F_1 + X_2 * F_2 + ... + X_n * F_n$$

Where X_i is the access cost of element i such that $X_i = i$ and F_i the frequency of i in the request such that $\forall i$, $F_i \geq F_{i+1}$. It's clear that if this last inequality is not satisfied we can only get a larger sum because $\forall i$, $X_i < X_{i+1}$.

Problem 3.

We can assume that we have a requested sequence of elements :

$$A_1, A_2, A_3, \ldots, A_{n-1}, A_n$$

Then no matter what the initial structure of our elements was, the optimal static algorithm would produce a sequence that is exactly the same with the requested one:

$$A_1, A_2, A_3, \ldots, A_{n-1}, A_n$$

Now let us suppose that the initial structure of our elements is:

$$A_n, A_{n-1}, \ldots, A_3, A_2, A_1$$

MTF actual cost analysis:

Given the above data structure and the requested sequence, the move-to-front algorithm would need to search until the end of the list each time, and then move that element to the front. Given that an element can be moved to the front in constant time, the actual cost for the given sequence would be n^2.

OPTIMAL-STATIC cost analysis:

It appears that since the sequence of elements in the optimal list is the same with the sequence of requested elements, the actual cost would be the sum of an arithmetic progression with a_1 = 1 and a_n = n. Thus the actual cost would be:

$$S = \frac{n(1+n)}{2} = \frac{n^2+n}{2}$$

So the ratio between MTF and OPTIMAL-STATIC is:

$$\begin{aligned} \lim_{n\to\infty} \frac{n^2}{\frac{n^2+n}{2}} &= \lim_{n\to\infty} \frac{2n^2}{n^2+n} \text{(using rule de l'hospital to calculate the limit)} \\ &= \lim_{n\to\infty} \frac{4n}{2n} = 2 \end{aligned}$$

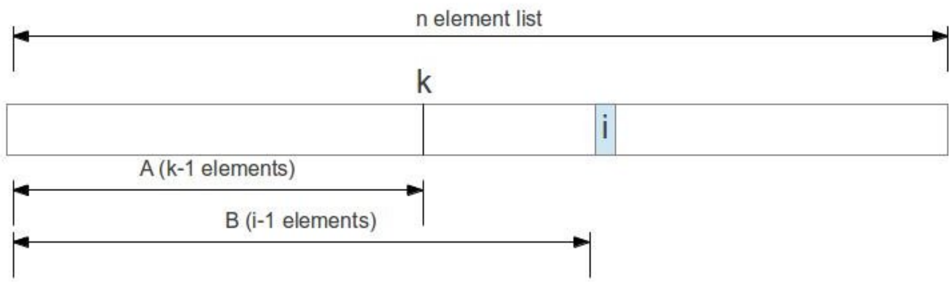
As a result, we can claim that since the cost of the move-to-front algorithm is twice as that of the static optimal algorithm in the limit, then the competitive analysis for the move-to-front algorithm against the static optimal algorithm is asymptotically tight.

Problem 4.

This question can be a good application of the *secretary problem*. Algorithm is like that:

- For n, we skip the predefined k elements while keeping the max of the seen elements in the memory.
- After k elements, we choose the next larger element than seen max. Here, larger element may not be seen, then we go to the end of the array and choose the last one.

Why this algorithm is correct is defined by a simple probability calculation:



Here, we skip k elements, and ith element is the max and we are able to choose it. If it is put into a formula:

$$P(k) = \sum_{i=1}^{n} P(i \text{ selected}|i \text{ is the best}) \cdot P(i \text{ is the best})$$

Max element can be found at the index from 1 to n because it is just a random permutation. Therefore, having max element at the index i is $\frac{1}{n}$. Moreover, since we skip the first k elements, the probability of finding max in between them is zero if max is there. Therefore, above sum until k doesn't contribute anything.

$$P(k) = \left(\sum_{i=1}^{k-1} 0 \cdot \frac{1}{n}\right) + \left(\sum_{i=k}^{n} P(i \text{ selected}|i \text{ is the best}) \cdot \frac{1}{n}\right)$$

However, if max is following the skipped elements, then it can be found:

$$P(k) = \sum_{i=r}^{n} P(\text{the best in the first } i-1 \text{ is among the first } k-1|i \text{ is the best}) \cdot \frac{1}{n}$$

For the algorithm to be able to select i, that would mean that there are no other items with value greater than i in the space between k and i(B-A). Thus, there must be at least one item in k with value greater than all the items before i. The probability of max of $i1$ elements(B) must be in first $k1$ elements(A) would be $\frac{k-1}{i-1}$. Otherwise we would choose as max a value between k and i, not the actual max.

$$\begin{aligned} P(k) &= \sum_{i=k}^{n} \frac{k-1}{i-1} \cdot \frac{1}{n} \\ &= \frac{k-1}{n} \sum_{i=k}^{n} \frac{1}{i-1} \end{aligned}$$

Optimal k can be found iteratively, setting it to 2 going upwards by calculating above result. When we arrive a probability larger than desired probability(0.25 in our case), algorithm stops. Then, by using found k, chooses probable max so skips first k elements but notes down the max of them and after k elements, picks the first element larger than known max.

However, we can calculate a k value that can work for all n. For example, by setting $k = \frac{n}{2}$ because $P(k)$ will always larger than 0.25 (to be specific, it is around 0.35).