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

Problem 1.

We can use contradiction or construction:

max: sth like suppose there is another binary tree with greater potential than the binary list but with the same n number of nodes that would mean that at some height of the tree there is one node having more nodes in its sublist 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 parent nodes

well we can just say that the w(root) = n-1, and the biggest weight for its child is n-2 constructing such a tree will end up with a linked list

min: well summing n/2 + n/2 or (n/2)-1 + (n/2)+1, doesnt matter, we get n BUT by doing n/2 + n/2 we will have the tree with the smallest height -¿ smallest number of element in the sum

and besides this way we maximize the number of nodes with no children

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 :

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

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

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

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

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

MTF actual cost analysis:

Given the above data structure and the requested sequence, the move-to-front alg orithm would need to search until the end of the list each time, and then move t hat element to the front. Given that an element can be moved to the front in con stant time, the actual cost for the given sequence would be n2.

OPTIMAL-STATIC cost analysis:

It appears that since the sequence of elements in the optimal list is the same w ith the sequence of requested elements, the actual cost of would be the sum of an arithmetic progression with a1 = 1 and an = n. Thus the actual cost would be

$$S = \frac{n(1+n)}{2} = \frac{n2+n}{2}$$

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

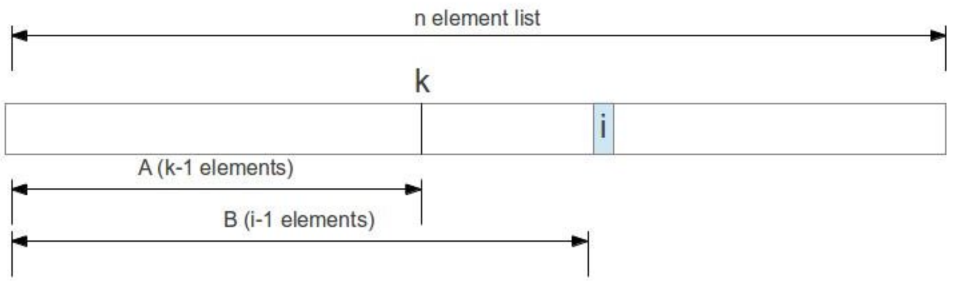
As a result, we can claim that since the cost of the move-to-front algorithm is twice as that of the static optimal algorithm in the limit, then the competitive analysis for the move-to-front algorithm against the static optimal algorithm is asymptotically tight.

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:



$$
\begin{aligned}
P(k) &= \sum_{i=1}^{n} P(i \text{ selected}|i \text{ is the best}) \cdot P(\text{applicant } i \text{ is the best}) \\
&= (\sum_{i=1}^{k-1} 0 \cdot \frac{1}{n}) + (\sum_{i=r}^{n} P(\text{the best in the first } i-1 \text{is among the first } k-1|i \text{ is the best}) \cdot \frac{1}{n}) \\
&= \sum_{i=k}^{n} \frac{k-1}{i-1} \cdot \frac{1}{n} \\
&= \frac{k-1}{n} \sum_{i=k}^{n} \frac{1}{i-1}
\end{aligned}
$$

Since we skip first k elements, the probability of finding max in between them is zero if max is there. However, if max is following the skipped elements, then we can find it with probability $\frac{k-1}{i-1}$ because i is the max and we choose it so max of $i-1$ elements must be in first $k-1$ elements. Otherwise, max would be between k and i and we would choose it, not the actual max.

k can be found iteratively, setting it to 2 going upwards by calculating above result. When we arrive a probability larger than desired probability, 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.

Here, we provide reference *python* implementation for k finding and testing of found k on all permutations:

```
from itertools import permutations
from sys import argv

def find_k(n):
  total_prob = 0
  for k in xrange(2, n+1):
    for i in xrange(k, n+1):
      total_prob += 1.0/(i-1)
    total_prob *= ((k-1.0)/n)
    if total_prob > 0.25:
      return k

def find_max(l, skip):
  i, m = 0, -1
  for e in l:
    if i <= skip:
      m = max(m, e)
      i += 1
    else:
      if e > m:
        return e
  return l[-1]

def fact(n):
  if n <= 1:
    return n
  else:
    return n * fact(n-1)

n = int(argv[1])
cnt, k = 0.0, find_k(n)
for l in permutations(range(1, n+1)):
  if find_max(l, k) == n:
    cnt += 1
print "n: %d skip: %d prob: %.3f" % (n, k, cnt / fact(n))
```