CS 112 - Data Structures

Sesh Venugopal

Sequential Search
Average Case

Sequential Search on a list: Array or Linked List

```
// on an array of integers
for (int i=0; i < arr.length; i++) {
    if (target == arr[i]) {
        return true;
    }
}
return false;
```

```
// on a linked list, Node has int data
for (Node ptr=front; ptr != null;
        ptr=ptr.next) {
    if (target == ptr.data) {
        return true;
    }
}
return false;
```


Big O running time is the same for either version:

Success:

Worst case: O(n), Best case: O(1)

Failure: O(n)

Sequential Search on a list: Average Count for Success

Basic Operation: Comparing target against list (array or linked list) item

What is the *average* number of comparisons for success?

A quick calculation would be to average out the best case (1 comparison) and the worst case (n comparisons), so (n+1)/2

Sequential Search on a list: Average Count for Success

A more general approach to compute average, like you would do to find average of a bunch of n numbers, is to add up the numbers and and divide by n (e.g. your average score on 3 exams).

In the case of sequential search, the numbers to be averaged are the comparisons for matching at various positions.

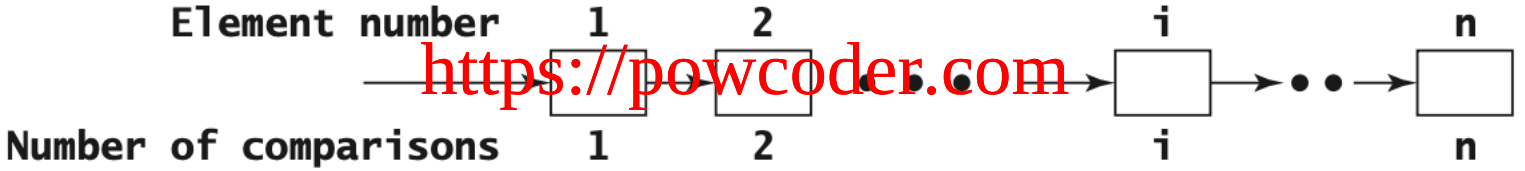
If the elements of the list have positions numbered 1 thru n, and it takes C_i comparisons to match at the i-th position

then the average number of comparisons over all items is:

$$\frac{C_1 + C_2 + \cdots + C_n}{n}$$

Sequential Search on a list: Average Count for Success

$$\frac{C_1 + C_2 + \cdots + C_n}{n}$$



It takes 1 comparison to match against the 1st item (C_1), 2 comparisons to match against the second item (C_2), etc.

So the average number of comparisons for success is:

```
(1+2+3+…+n)/n = n*(n+1)/2/n = (n+1)/2
```

Average Count for Success: The Assumption of Equal Likelihood

$$\frac{C_1 + C_2 + \cdots + C_n}{n}$$

This formula we are using for the average *assumes that all matches are equally likely*.

For instance, if you have a list of 10 items, and 100 matches are made on the list, then under this assumption, each item would be matched 10 times (i.e. same number of matches for all items)

So we can rethink the formula as a weighted average:

C_1*1/n + C_2*1/n + … + C_n*1/n = 1/n*(C_1 + C_2 + … + C_n)

Each item's comparison count for match is weighted with its likelihood or probability of being matched – which is 1/n since all items are equally likely to be matched

Average Count for Success: General Formula, Random Likelihoods

In practice, it is absurd to believe that all items will be equally likely to be matched on. More generally, there will be biases toward a few items—they will be a lot more likely to be matched on than others. (Think of Google search trends, with its day-to-day fluctuations.)

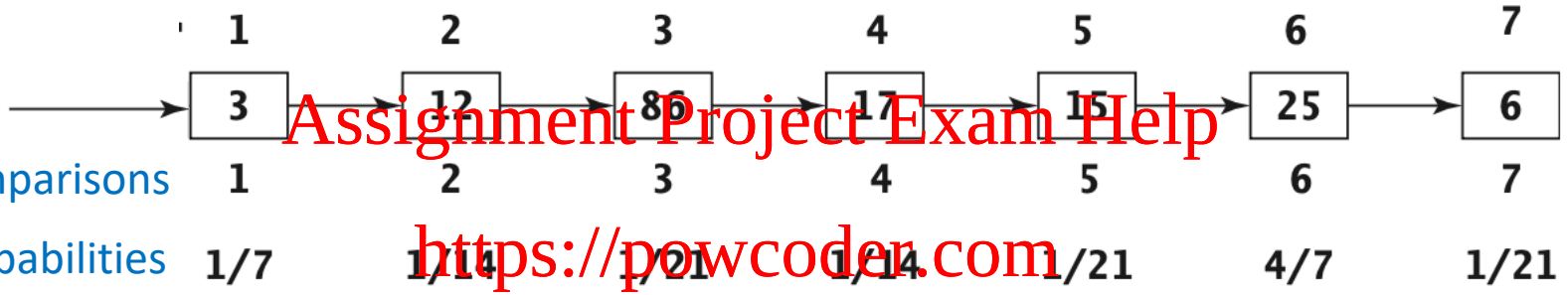
So, in practice, we will need to assign a different probability of match for each item, say P_i for the *i*-th item . Which generalizes the weighted comparisons formula for average to:

$$C_1*P_1 \; + \; C_2*P_2 \; + \; \ldots \; + \; C_n*P_n$$

Each item's comparison count for match is weighted with its likelihood or probability of being matched – but the probabilities are all different, in general (of course the sum of all probabilities equals 1)

Average Count for Success: General Formula, Random Likelihoods

Here's an example:

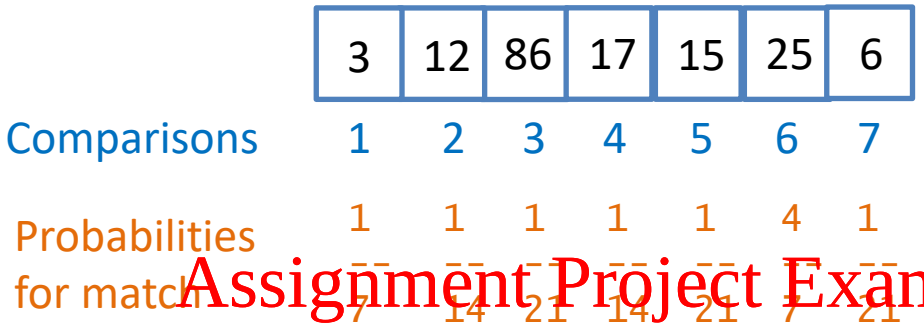


Note that the value 3 (4/7) is twice as likely to be matched as 12 (1/14), and 25 (4/7) is 4 times as likely to be matched as 3.

Using the weighted average formula gives us the following for average number of comparisons:

$$1 \times \frac{1}{7} + 2 \times \frac{1}{14} + 3 \times \frac{1}{21} + 4 \times \frac{1}{14} + 5 \times \frac{1}{21} + 6 \times \frac{4}{7} + 7 \times \frac{1}{21} = \frac{99}{21} = 4.7$$

Rearranging items based on search pattern

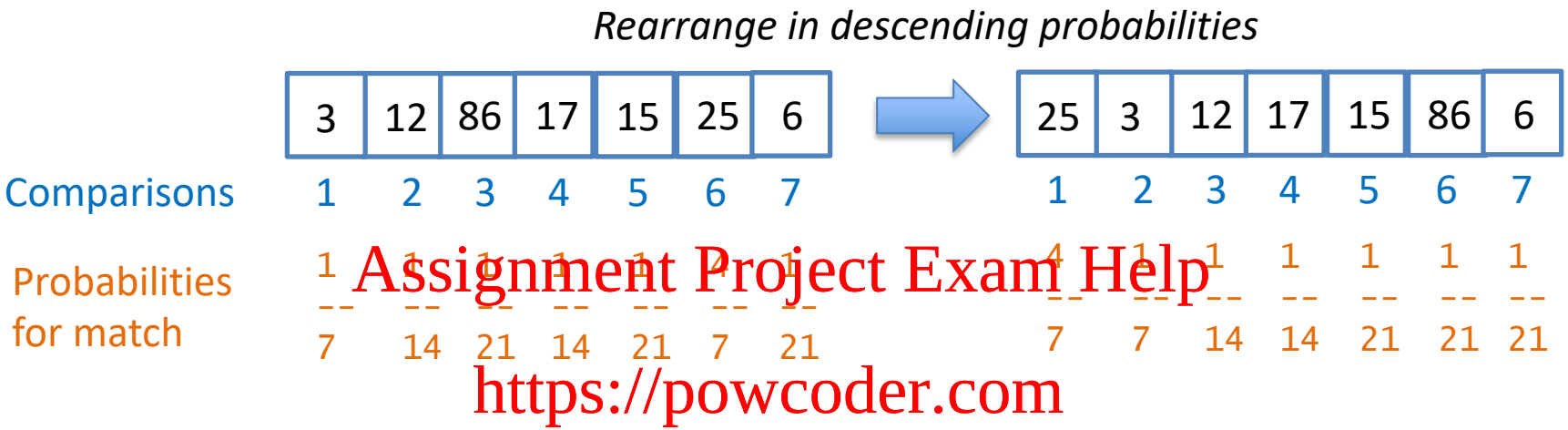


Since the value 25 is 4 times as likely to be matched on as value 3, shouldn't 25 be all the way up front so that the search time (number of comparisons) is reduced on average?

And similarly for all other items: the more likely they are to be matched on, the more toward the front they should be, to reduce the number of comparisons to get to match on them

Rearranging the items in *decreasing order of their match probabilities* results in the minimum average number of comparisons.

Rearranging items based on search pattern



Average number of comparisons for success AFTER rearrangement:


```
1*4/7 + 2*1/7 + 3*1/14 + 4*1/14 + 5*1/21 + 6*1/21 + 7*1/21 = 2.2
```

The average after rearrangement is less than half the original average of 4.7

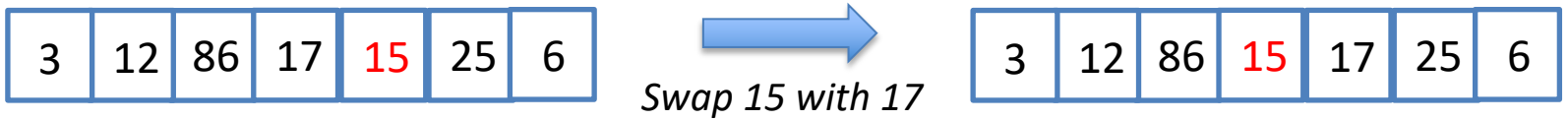
Adaptive Shuffling

Rearranging ALL items in one shot, knowing the spread of probabilities of matches, is only possible after the spread has attained long-term stability (will change very little over time). But this kind of long-term stability is not very likely to occur in practice – there is usually lots of fluctuation in short periods of time.

There is an alternative: shuffle "adaptively" by moving an item, as soon as it is matched, toward the front (the expectation is that it is quite likely to be matched again very soon) – moving toward front will reduce comparisons for match.

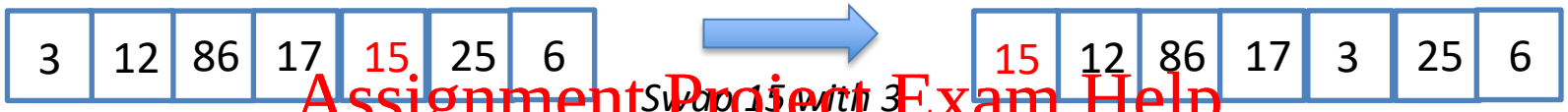
There are two possible kinds of adaptive moves.

One option is to move the just-matched item one spot toward the front.



Adaptive Shuffling

Another, more aggressive option, is to move the just-matched item all the way to the first spot



But it would now push the previous 1st item (e.g. 3) backward, thereby undoing (perhaps drastically) the previous move that brought that item (3) to the 1st spot.

Adaptive Shuffling

A better way to bring the just-matched item to the 1st spot is shuffle all preceding items one step back (like in insertion sort!)

But this would be much more expensive, O(n) time in the worst case

However, it would only take O(1) time in a linked list

General Formula, Random Likelihoods: Widely Applicable

The general formula for weighted average given probabilities for match can be applied to ANY search algorithm on ANY data structure

$$C_1 * p_1 + C_2 * p_2 + \ldots + C_n * p_n$$

The actual number of comparisons (C_i's) for match would depend on the algorithm and the data structure

For example, this formula can be applied for binary search on a sorted array – you will need to compute the number of comparisons to find a match at each array position (the middle would be 1 comparison), then multiply by the associated probabilities for match.