Brian Loughran
Johns Hopkins
Algorithms
3/10/2020

1. [15 points] The following problem is known as the Dutch Flag Problem. In this problem, the task is to rearrange an array of characters R, W, and B (for red, white, and blue, which are the colors of the Dutch national flag) so that all the R's come first, all the W's come next, and all the B's come last. Design a linear, in-place algorithm that solves this problem.

One important component of this algorithm is a method swap(a, b), which swaps the colors at index a and b.

The general idea is to iterate over the list and place each value in the list where it belongs. The list starts as all unknowns, and as you move through the list the position of each element in the list becomes known. We track as we go through the list the index at the right end of the R's, called low, the index at the right of the W's, called mid, and the index at the left of the B's, called high. The region to the left of low, then will be all R, the region between low and mid will be all W, and the region to the right of high will be all B. The remaining region between mid and high can be anything, and as we iterate over the list we will reduce the size of the region between mid and high until the unknown region shrinks to nothing.

Each iteration of the code will read the value at index mid. If the value is R, we swap the values at mid and low, and increment index low and mid. If the value is W, we do not swap and we increment the index of mid. If the value is B, we swap the values at mid and hi and decrement the value of high.

```
dutch_flag(colors):
 low = 0
 mid = 0
 high = len(colors)                    // set the indicies for low, mid and high
 while mid < high:                     // iterate until we reduce unknown area to nothing
  if colors[mid] == "R":               // check if the color is red
   swap(low, mid)                      // swap the values of low and mid
   low++
   mid++                               // increment low and mid
  else if colors[mid] == "W":          // check if the color is white
   mid++                               // increment the value of mid
  else if colors[mid] == "B":          // check if the color is blue
   swap(mid, high)                     // swap the value of mid and high
   high--                              // decrement high
```

2. Give an O(n lg k)-time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hints: Use a heap for k-way merging.)

For this algorithm, to understand the runtime of the algorithm, we must first understand heaps, particularly min heaps, and some operations and their runtimes. Below are some functions that we will use with our implementation of a heap, where n is the size of the heap:

Brian Loughran
Johns Hopkins
Algorithms
3/10/2020

getMin() – Get the root value of the heap O(1)
popMin() – Remove the root value of the heap and maintain the heap property O(lgn)
insert() – Insert a new value to the heap O(lgn)

Now knowing a little bit about some operations we can do with heaps we can design an algorithm which sorts k sorted lists into one sorted list. We start by popping the first index of each of the k lists and inserting those values into a min heap. While the heap is not empty, we popMin() the root value of the heap and insert that value into the result array. Then, if the list that the value we just extracted is not empty, we insert that value into the heap. Once the heap is empty, we will have a sorted result array in O(nlgk) time. The pseudocode is shown below:

```
sortLists(lists):                             // lists is a list of lists, e.g. [[1, 2], [3, 6, 7], [4, 10]]
  result = []
  for list, index in enumerate(lists):        // iterate each list (O(k)
    heap.insert(list.pop(0), index)
    // insert the value in the heap O(lgk) -> note we are storing a tuple to the heap. This is to track which
list the value came from

  while !heap.empty()                         // iterate until the heap is empty O(n)
    value, index = heap.popMin()              // pop the minimum value in the heap O(1)
    result.append(value)                      // append the minimum remaining value to the result
    try:
      heap.insert(lists[index].pop(0), index)        // pop next lowest value in the list to the heap O(lgk)
    catch:
      // do nothing – if pop() throws an exception that list is empty
```

Clearly just analyzing the loops we can see that the runtime of the above algorithm is
O(klgk + nlgk) = O(nlgk)
As an added bonus, using pop() will allow us to solve this problem in O(n) space.

This algorithm works because the value at the root of the heap will always be the minimum remaining value. Since the values in the heap are the minimum in their respective lists, and the value at the root of the min heap is the minimum of the minimums, it must be the lowest value in all the lists.

Brian Loughran
Johns Hopkins
Algorithms
3/10/2020

3. [35 points] ***Collaborative Problem***– CLRS 8-7: The 0-1 Sorting Lemma and Column Sort: A *compare-exchange* operation on two array elements $A[i]$ and $A[j]$, where $i < j$, has the form

Algorithm 1 Compare and Exchange Values

```
function CompareExchange($A$,$i$,$j$)
    if $A[i] > A[j]$ then
        exchange $A[i]$ with $A[j]$
    end if
end function
```

After the compare-exchange operation, we know that $A[i] \leq A[j]$. An *oblivious compare-exchange algorithm* operates solely by the sequence of pre-specified compare-exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any compare-exchange algorithm:

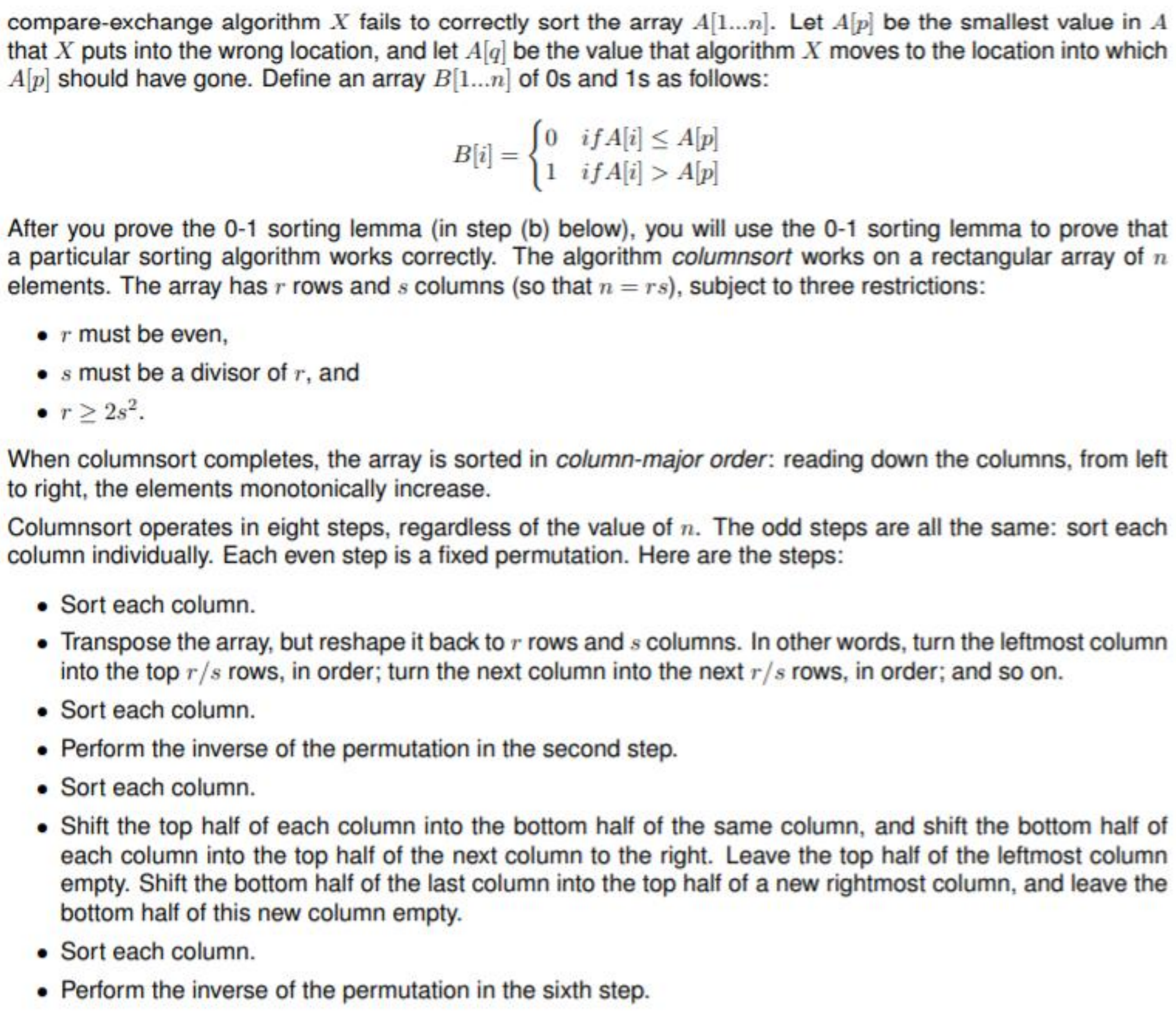
Algorithm 2 Insertion Sort with CompareExchange Operations

```
function InsertionSort($A$)
    for $j \leftarrow 2$ to $A.length$ do
        for $i \leftarrow j - 1$ downto 1 do
            CompareExchange($A$,$i$,$i + 1$)
        end for
    end for
end function
```

The **0-1 sorting lemma** provides a powerful way to prove that an oblivious compare-exchange algorithm produces a sorted result. It states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivious compare-exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0-1 input. Assume that an oblivious

Brian Loughran
Johns Hopkins
Algorithms
3/10/2020



compare-exchange algorithm X fails to correctly sort the array $A[1...n]$. Let $A[p]$ be the smallest value in A that X puts into the wrong location, and let $A[q]$ be the value that algorithm X moves to the location into which $A[p]$ should have gone. Define an array $B[1...n]$ of 0s and 1s as follows:

$$B[i] = \begin{cases} 0 & if A[i] \leq A[p] \\ 1 & if A[i] > A[p] \end{cases}$$

After you prove the 0-1 sorting lemma (in step (b) below), you will use the 0-1 sorting lemma to prove that a particular sorting algorithm works correctly. The algorithm *columnsort* works on a rectangular array of n elements. The array has r rows and s columns (so that $n = rs$), subject to three restrictions:

- r must be even,
- s must be a divisor of r, and
- $r \geq 2s^2$.

When columnsort completes, the array is sorted in *column-major order*: reading down the columns, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of n. The odd steps are all the same: sort each column individually. Each even step is a fixed permutation. Here are the steps:

- Sort each column.
- Transpose the array, but reshape it back to r rows and s columns. In other words, turn the leftmost column into the top r/s rows, in order; turn the next column into the next r/s rows, in order; and so on.
- Sort each column.
- Perform the inverse of the permutation in the second step.
- Sort each column.
- Shift the top half of each column into the bottom half of the same column, and shift the bottom half of each column into the top half of the next column to the right. Leave the top half of the leftmost column empty. Shift the bottom half of the last column into the top half of a new rightmost column, and leave the bottom half of this new column empty.
- Sort each column.
- Perform the inverse of the permutation in the sixth step.

Brian Loughran
Johns Hopkins
Algorithms
3/10/2020

Figure 1 shows an example of the steps of columnsort with $r = 6$ and $s = 3$. Even though this example violates the requirement that $r \geq 2s^2$, it happens to work.

Although it might seem hard to believe that columnsort actually sorts, you will use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies because we can treat columnsort as an oblivious compare-exchange algorithm. A couple of definitions will help you apply the 0-1 sorting lemma. We say an area of an array is **clean** if we know that it contains either all 0s or all 1s. Otherwise, the area might contain mixed 0s and 1s, in which case it is **dirty**. From part (d) on, assume that the input array contains only 0s and 1s and that we can treat it as an array with r rows and s columns.

(a) [4 points] Argue that $A[q] > A[p]$ so that $B[p] = 0$ and $B[q] = 1$.

(b) [4 points] To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort the array B correctly.

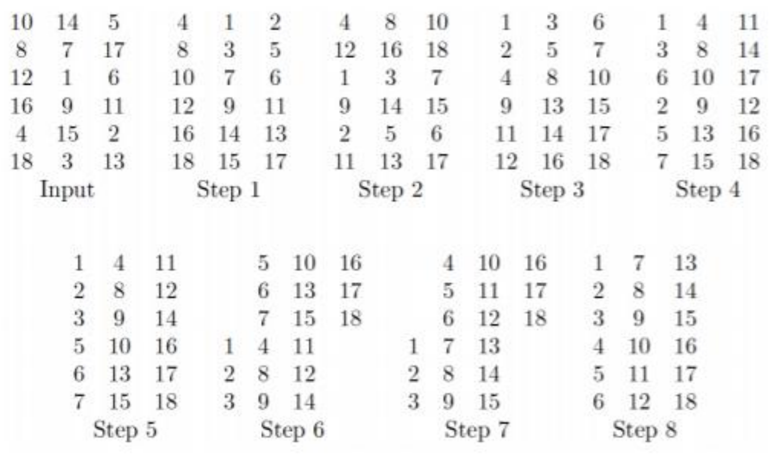
(c) [4 points] Argue that we can treat columnsort as an oblivious compare-exchange algorithm, even if we do not know what sorting method the odd steps use.

(d) [4 points] Prove that after the first three steps, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.

(e) [4 points] Prove that after the fourth step the array, read in column-major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle.

(f) [5 points] Prove that steps five through eight produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.

Figure 1: Example Execution of Columnsort



10	14	5	4	1	2	4	8	10	1	3	6	1	4	11
8	7	17	8	3	5	12	16	18	2	5	7	3	8	14
12	1	6	10	7	6	1	3	7	4	8	10	6	10	17
16	9	11	12	9	11	9	14	15	9	13	15	2	9	12
4	15	2	16	14	13	2	5	6	11	14	17	5	13	16
18	3	13	18	15	17	11	13	17	12	16	18	7	15	18
	Input			Step 1			Step 2			Step 3			Step 4	

1	4	11		5	10	16		4	10	16	1	7	13
2	8	12		6	13	17		5	11	17	2	8	14
3	9	14		7	15	18		6	12	18	3	9	15
5	10	16	1	4	11		1	7	13		4	10	16
6	13	17	2	8	12		2	8	14		5	11	17
7	15	18	3	9	14		3	9	15		6	12	18
	Step 5			Step 6				Step 7				Step 8	

(g) [5 points] Now suppose that s does not divide r. Prove that after steps one through three, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. How large must r be, compared with s, for columnsort to correctly sort when s does not divide r.

(h) [5 points] Suggest a simple change to step one that allows us to maintain the requirement that $r \geq 2s^2$ even when s does not divide r, and prove that with your change, columnsort correctly sorts.

Brian Loughran
Johns Hopkins
Algorithms
3/10/2020

a) We can prove A[q] > A[p] by definition. If A[p] is the first value that is misplaced, that means that all of the values to the left of position p are less than A[p], and all of the other values are greater than A[p]. Since all of the values to the left of position p are properly placed, A[q] cannot be any of the values to the left of position p, and thus A[q] > A[p]

b) Let a be a sequence which is not sorted by comparator network S. The sequence S(a) = B is unsorted if there is a position p such that B[k] > B[k+1]. Using the mapping:

$$B[i] = \begin{cases} 0 & if A[i] \le A[p] \\ 1 & if A[i] > A[p] \end{cases}$$

So for our unsorted sequence B there is a value B[k] = 1 and B[k+1] = 0. This means that the 0/1 sequence B is not sorted by the comparator network S(a). This proves that if there is an arbitrary sequence a this is not sorted by S, there is also a sequence containing only 0/1 that is not sorted by S. Correspondingly, if there is no 0/1 sequence not sorted by another better comparator network C, there can be no sequence of any contents that is not sorted by C. Thus, if all 0/1 sequences are sorted by C, then all sequences of any contents can also be sorted by C. This proves the 0-1 sorting lemma.

c) The even steps of the columnSort algorithm are value-independent, meaning that they do not look at the value that they are moving, they just move values blindly, thus are oblivious by definition. The odd numbered steps do not specify a sorting algorithm, thus we can use whatever algorithm we want and get the same result. Given in the problem statement is an oblivious compare-exchange algorithm, which we can use for the odd numbered steps. Since all of the steps are oblivious, then columnSort as a whole must be oblivious, meaning we can apply the 0-1 sorting lemma.

d) After step 1 of the algorithm, it is known that each column will contain some number of 0's and some number of 1's, with just 1 0 -> 1 transition per column. Each column will map to r/s rows, and since s is a divisor of r, r/s will be a whole number value. After the mapping each column maps to just one dirty row, while the rest can be considered clean. That is, there can be at most s dirty rows after the second step. The third step consists of sorting each column again, which will move the clean rows of 0's up to the top, the clean rows of 1's to the bottom, and the number of at most s dirty rows will be in the middle.

e) Expanding on the prior argument that after step 3 there are at most s dirty rows after step 3, we note that there are at most s^2 elements in these rows, a multiplication of the s rows by the s columns. Since each row is mapped in column major order in step 4, and because at the end of step 3 we have some number of rows with 0's at the beginning and some number of rows with 1's at the end, reading on column-major order will produce some number of 0's at the beginning, some number of 1's at the end, and the at most s^2 dirty bits between the 0's and 1's.

f) Since by definition r >= 2s^2, the dirty section is at most a half column long. This means that the dirty section will either be fully contained in one column, or it will wrap from one column to another, but the dirty section will never wrap more than two columns. If all of the dirty bits are in the same column, sorting the column in step 5 will result in a sorted column major sequence. The subsequent steps will not interfere with the sorted sequence, since steps 6-8 consist only of sorting again and swapping values. However, if the values wrap two columns, sorting just the columns in step 5 will not produce a sorted result. Since the dirty values are no more than a half column in length, shifting the top half of each column to the bottom half of the adjacent column will force all of the dirty values into the same column. Doing the column sort specified in step 7 will sort any remaining dirty bits, and transposing those values in step 8 will result in a fully sorted 0-1 output. According to the 0-1 sorting lemma, which states that if an oblivious compare-exchange algorithm sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values. Thus, columnSort correctly sorts inputs of any value

g) Just like as explained in d), after step 1 of the algorithm, it is known that each column will contain some number of 0's and some number of 1's, with just 1 0 -> 1 transition per column. However, after step 2 if s does no divide r there may be both a 0 -> 1 transition and a 1 -> 0 transition in a single row. There may be at most s 0 -> 1 transitions (like in the previous problem), and at most s-1 1 -> 0 transitions, resulting in a dirty region of at most 2s-1 rows. Similar to step e), we can multiply the 2s-1 dirty rows by the s columns to produce 2s^2-s dirty bits. Since the subsequent steps of the algorithm only worked when the column size was greater than or equal to twice the number of dirty bits, we will apply the same logic here and find that r >= 4s^2-2s.

h) One simple change to the algorithm would be to pad the end of the array with rows containing +inf values until s divides r cleanly, running the sorting algorithm, then trimming the extraneous values from the end of the array when you are done. This works because when mapping with the 0 -> 1 sorting lemma, these values would all map to 1's, and the sorting algorithm would sort them just the same as any other value. Since padding the end of the array with +inf satisfies the 0 -> 1 sorting lemma, the columnSort algorithm will correctly sort the values even with +inf appended to the end.

4. [15 points] CLRS 5.3-3: Suppose that instead of swapping element $A[i]$ with a random element from the subarray $A[i...n]$, we swapped it with a random element from anywhere in the array.

Algorithm 3 Permute with All

```
function PermuteWithAll($A$)
    $n \leftarrow A.length$
    for $i \leftarrow 1$ to $n$ do
        swap $A[i]$ with $A[$Random$(1,n)]$
    end for
end function
```

Does this code produce a uniform random permutation? Why or why not?

Brian Loughran
Johns Hopkins
Algorithms
3/10/2020

Given a list with n items, there are a total of n! possible orderings of the list (for example, if you have the items [1, 2, 3], you can order them as [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], or [3, 2, 1], with 3! = 6 distinct possibilities.

We also note that swapping each item in a list with n items with another item in the list will produce n^n possible swapping combinations. For example, given a list of size 2, the possible swaps can be [1 swapped with 1, 2 swapped with 1], [1 swapped with 1, 2 swapped with 2], [1 swapped with 2, 2 swapped with 1], and [1 swapped with 2, 2 swapped with 2]. Each of the n^n possibilities has an equal possibility of occurring, thus each swapping combination has a probability of 1/n^n off occurring.

To produce a uniform random permutation, each of the n^n swapping possibilities must map to each of the n! possible list combinations. However, this cannot happen if (n^n)/n! is not a whole number (this would indicate that some of the possible orderings of the list must be more likely than others). And (n^n)/n! is NOT a while number in the case that n>2 (ex. n=3 -> 27/6 which is not a whole number, and n=5 -> 3125/120 which is also not a whole number). Thus, for n>2, this cannot produce a uniform random permutation.

5. [20 points] Collaborative Problem: A number of peer-to-peer systems on the internet are based on overlay networks. Rather than using the physical internet as the network on which to perform computation, these systems run protocols by which nodes choose collections of virtual “neighbors” so as to define a higher-level graph whose structure may bear little or no relation to the underlying physical network. Such an overlay network is then used for sharing data and services, and it can be extremely flexible compared with a physical network, which is hard to modify in real time to adapt to changing conditions.

Peer-to-peer networks tend to grow through the arrival of new participants who join by linking into the existing structure. This growth process has an intrinsic effect on the characteristics of the overall network. Recently, people have investigated simple abstract models for network growth that might provide insight into the way such processes behave in real networks at a qualitative level.

Here is a simple example of such a model. The system begins with a single node v1. Nodes then join one at a time; as each node joins, it executed a protocol whereby it forms a directed link to a single other node chosen uniformly at random form those already in the system. More concretely, if the system already contains nodes v1, ..., vk−1 and node vk wishes to join, it randomly selects one of v1, ..., vk−1 and links to that node. Suppose we run this process until we have a system consisting of nodes v1, ..., vn; the random process described above will produce a directed network in which each node other than v1 has exactly one outgoing 3 edge. On the other hand, a node may have multiple incoming links, or none at all. The incoming links to a node vj reflect all the other nodes whose access into the system is via vj ; so if vj has many incoming links, this can place a large load on it. Then to keep the system load-balanced, we would like all the nodes to have a roughly comparable number of incoming links. That is

Brian Loughran
Johns Hopkins
Algorithms
3/10/2020

unlikely to happen, however, since nodes that join earlier in the process are likely to have more incoming links than nodes that join later. Let us try to quantify this imbalance as follows.

(a) [10 points] Given the random process described above, what is the expected number of incoming links to node vj in the resulting network? Give an exact formula in terms of n and j, and also try to express this quantity asymptotically (via an expression without large summations) using Θ(·) notation.

We can start by examining a few base cases. The first case we can examine is a single node system. In this system, there will be no connections at all. So the expected connections for node 1 is 0.

Now let's assume we have a two node system. In this system, there will be one connection from the second node to the first node. Thus the first node will have 1 connection, and the second node will have 0 connections.

Now let's look at a 3 node system. The connection for the third node is equally likely to connect to the first node as it is to the second. Thus, the first node will have 1.5 expected connections, the second node will have 0.5 expected connections, and the third node will have 0 expected connections.

We can begin to see a pattern here. Each time a node is added to a network with k existing nodes, the k-1 nodes before the new node each have an equal probability of getting the connection. Thus, we construct the following summation, where n is the total number of nodes in the system and j is the index of the node that we want to get the expected number of connections:

$$\sum_{k=j}^{n-1} \frac{1}{k}$$

Using our 3 node system from before (n=3, j=1), the first node will have (1/1 + 1/2) connections, the second node (n=3, j=2) will have (1/2) connections, and the third node (n=3, j=3) will have 0 connections. This indicates that our summation is probably right.

Splitting this summation gives us the following equivalent summation:

$$\sum_{k=1}^{n-1} \frac{1}{k} - \sum_{k=1}^{j-1} \frac{1}{k}$$

According to Appendix A.7 from the book we can simplify this to:

$$[\ln(n) + O(1)] - [\ln(j) + O(1)]$$

Which simplifies to the following in theta notation when reducing the O(1)'s and using the principle of subtracted logarithms:

Brian Loughran
Johns Hopkins
Algorithms
3/10/2020

$$\theta(\ln\left(\frac{n}{j}\right))$$

(b) [10 points] Part (a) makes precise a sense in which the nodes that arrive early carry an "unfair" share of connections in the network. Another way to quantify the imbalance is to observe that, in a run of this random process, we expect many nodes to end up with no incoming links. Give a formula for the expected number of nodes with no incoming links in a network grown randomly according to this model.

As stated above, every time we add a node to the system, each node has an equal probability of getting the connection. This infers that the probability each node does *not* get the connection is the complement of the probability that it does get the connection (so if there are 3 nodes in the system and we add a fourth, each node has a 1/3 probability of getting the connection, and a 2/3 probability of not getting the connection). For a given node, each time a new node is added to the system we multiply the probability of a new node being added by the probability there is already a connection. Thus, for a single node, the probability of a connection in a n-node system at index j is given by:

$$\prod_{k=j}^{n-1} 1-\frac{1}{k}=\frac{j-1}{n-1}$$

Summing over each node we can get the total number of nodes with a connection. The summation looks like:

$$\sum_{j=1}^{n}\frac{j-1}{n-1}$$

This telescoping series sums up to n/2. Thus, the expected number of nodes with no connection is n/2, or half of the nodes.