CSC 6013 Algorithms and Discrete Structures   Michael Bradley

Class Notes for Week 7:
Decrease-and-Conquer Algorithms

Read these sections from the textbook: 2.1 - 2.2, 12.1 - 12.3, 22.1 - 22.4

Decrease-and-Conquer Algorithms
Decrease by a constant
A. Insertion sort
B. Topological sort
Decrease by a constant factor
C. Binary search
D. Fake coin detection
E. Russian peasants' multiplication
Decrease by a variable amount
F. Euclidean Greatest Common Divisor
G. Lomuto partition
H. K'th order statistic

Preview of weekly assignments:

- Discussion board post
- Quiz
- Coding assignment
- Worksheet

Decrease-and-Conquer Algorithms

In this unit we will investigate the decrease-and-conquer algorithm design strategy. This strategy takes a problem and recursively decreases its size in some way. Eventually, the problem will be decreased in such a way that it is easy to solve. We then solve the problem and use that solution to arrive at a solution to the original problem.

The basic idea of Decrease-and-Conquer is to exploit the relationship between an instance of a problem and the solution to a smaller instance of the problem. Essentially, we do some work and solve a smaller version of the same problem.

We will classify Decrease-and-Conquer algorithms into three different categories.

1. **Decrease by a constant**: The size of the instance is reduced by subtracting the same constant on each iteration of the algorithm (normally one).
Example: Computing a^n by realizing $a^n = a \cdot a^{n-1}$. This yields the recurrence:
$$f(n) = \begin{cases} a \cdot f(n-1) & if\ n > 0 \\ 1 & if\ n = 0 \end{cases}$$
In this example, the problem size is reduced from n to n-1.

2. **Decrease by a constant factor**: The size of the instance is reduced by dividing by the same constant factor on each iteration of the of the algorithm (normally the factor is two).
Example: Computing a^n by realizing $a^n = \left(a^{\frac{n}{2}}\right)^2$. This yields the recurrence:
$$f(n) = \begin{cases} \left(a^{\frac{n}{2}}\right)^2 & if\ n > 0\ and\ even \\ a \cdot \left(a^{\frac{n-1}{2}}\right)^2 & if\ n > 0\ and\ odd \\ 1 & if\ n = 0 \end{cases}$$
In this example, the problem size is reduced from n to n/2.

3. **Decrease by a variable size**: The size-reduction for an instance varies from one iteration to the next.
Example: Euclid's algorithm for the GCD. We will see this soon.
We will see that for this example, the problem size is reduced from max(n, m) to a smaller integer that varies with the particular values of n and m.

Decrease by a Constant Algorithms

A. Insertion Sort

We consider the sorting problem again. Note: This is a brute force algorithm; it is not recursive.

Basic idea:
After three steps, an array
    A = [35, 7, 81, 23, 17, 56, 40, …..]
might look like:
    A = [7, 23, 35, 81, 17, 56, 40, …..]
with elements A[0] through A[3] sorted and elements A[4] through A[n-1] not yet examined.

Observation: To sort an array A[0..n-1] it is sufficient to place element A[n-1] into the sorted sequence A[0..n-2].

Q: How do we place the element in the correct location?
A: The answer to this question provides a recursive description of sorting, but it is inefficient. It is more efficient to work bottom up (i.e., from the base case).

Intuition: We will sort the sequence by decreasing the instance to sort by one on each iteration of the algorithm. After the first iteration of the sort, we will guarantee that a sequence of length one is sorted.

In general, after iteration k completes, we will have a k-element sorted sequence and an n-k unsorted sequence.

The insertion sort algorithm is as follows: After i passes, i = last large and s = last small element.

```
1   # Input:  An array A of integers.
2   # Output: A sorted in increasing order.
3   def InsertionSort(A):
4       for i in range(1, len(A)):
5           v = A[i]
6           j = i - 1
7
8           # Insert v into the sorted sequence A[0 .. j - 1].
9           # This is achieved by shifting elements over until we
10          # have a free cell and then filling it.
11          while j >= 0 and A[j] > v:
12              A[j + 1] = A[j]
13              j = j - 1
14          A[j + 1] = v
```

- Let's look at an example run of the algorithm on the input $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Notationally gray cells are in the sorted subarray, dotted arrows indicate a movement of an element, black cells represent the current number to be placed, and solid arrow.

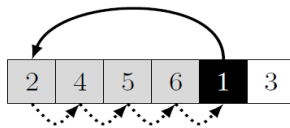 1. Place the number 2 in the sorted sequence.

5	2	4	6	1	3
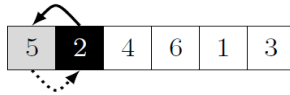
  2. Place the number 4 in the sorted sequence.

2	5	4	6	1	3

  3. Place the number 6 in the sorted sequence. Note, there is no change in the position of the number.

2	4	5	6	1	3

  4. Place the number 1 in the sorted sequence.

2	4	5	6	1	3

  5. Place the number 3 in the sorted sequence.

1	2	4	5	6	3
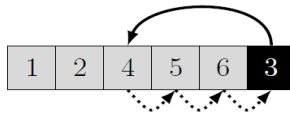
  6. The finalized sequence is:

1	2	3	4	5	6
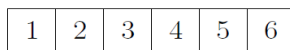
Let us formally analyze the running time of insertion sort by constructing a table.

Line	Cost	Count
4	c_1	n-1
5	c_2	n-1
6	c_3	n-1
9	c_4	1+2+3+…+n-1 = (n-1)n/2
10	c_5	1+2+3+…+n-1 = (n-1)n/2
11	c_6	1+2+3+…+n-1 = (n-1)n/2
12	c_7	n-1

$T(n) = c_1(n-1) + c_2(n-1) + c_3(n-1) + c_4(n-1)n/2 + c_5(n-1)n/2 + c_6(n-1)n/2 + c_7(n-1)$
$= (.5c_4 + .5c_5 + .5c_6)n^2 + (c_1 + c_2 + c_3 - .5c_4 - .5c_5 - .5c_6 + c_7)n + (-c_1 - c_2 - c_3 - c_7)1$
$= O(n^2)$

The worst case of insertion sort occurs when the array is initially arranged from largest to smallest and we want to sort if from smallest to largest. For this instance, at step k, A[k] is smaller than all entries before it in the array. The while loop runs through k-1 iterations so that all array elements in slots A[0] through A[k-1] are moved so A[k] can be inserted into slot A[0]. $1 + 2 + 3 + … + n\text{-}1 = $ (n-1)n/2, so the running time of the insertion sort algorithm is $O(n^2)$.

The best case is when the array is already sorted into ascending order. With each array element, we exit the while loop after a single comparison, so there are n-1 comparisons. Each element is copied into v and then copied right back where it came from.

Q: Why is insertion sort a "decrease by a constant amount" algorithm? Given an array of n numbers, after one array element is inserted into its proper place in the sorted portion of the array, there is one less item to be sorted.

B. Topological Sort

The basic idea of topological sort is that we are given a graph and we are asked to create a list of all the vertices so that the list satisfies this property:

        Whenever there is an edge from u to v, then u must be earlier in the list than v.

We illustrate this with the following simple example.
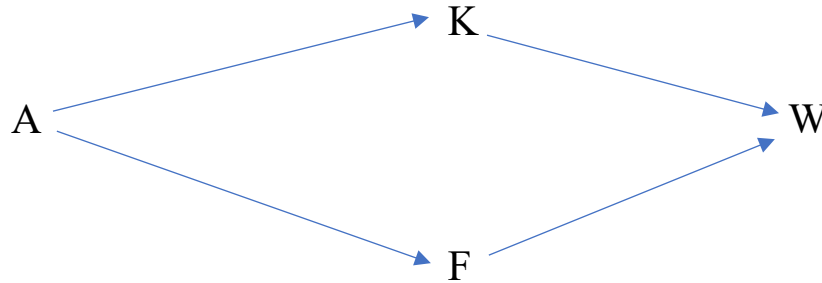
Represent this directed graph with an adjacency matrix

	A	F	K	W
A				
F				
K				
W				

Represent this directed graph with adjacency lists

A
F
K
W

2 possible solutions:

In non-technical terms: A list of vertices is topologically sorted if you never have a later vertex pointing to an earlier vertex.

In order to understand topological sorting, we must first recall a few things about directed graphs.

Directed Graph (Digraph): A graph G = (V, E) is a graph where V is the set of vertices, E is a set of edges, and (u, w) in E does <u>not</u> imply (w, u) is in E.

A directed graph is called **acyclic** if the graph has no cycles in it. This is often referred to as a DAG.

Both BFS and DFS make sense for directed graphs. The forests have four types of edges.
1. Tree Edges: An edge (v, u) if u was first visited by exploring edge (v, u)
2. Back Edges: An edge between a vertex and its ancestor.
3. Forward Edges: An edge between a vertex and its descendant.
4. Cross Edges: Edges that only join siblings or the parent's siblings (uncles or aunts) in the tree. In other words, edges that are not tree, back, or forward edges.

An interesting computation on DAGs is a topological sort.
Definition (Topological Sort Problem).
Input: Given a directed acyclic graph G = (V, E).
Output: A sequence of all v in V such that every edge in (u, w) in E, u is listed in the sequence before w.

Applications
- instruction scheduling in program compilation;
- resolving symbol dependencies in linkers;
- how to progress through the CS major;
- any dependency graph you can dream up.

The common solution (assuming the graph is a DAG)
1. Run the DFS = depth first search.
2. As each vertex becomes a dead end (has no explorable edges), push the node label onto a stack.
3. Once the DFS has finished, pop each element off (being sure to print each element as it is popped off the stack.)

Q: Why does this work?
A: Since there are no back edges, when a vertex v is pushed onto the stack, there will not be vertices u with edge (u, v) below it. Otherwise, we would be in violation of the "no back edges" rule.

The decrease-and-conquer solution.
Repeat the following until the graph has one vertex.
- Locate a vertex with in-degree zero (a vertex that has only one vertex pointing to it).
- Remove the vertex (and all associated edges) and add it to the end of the list of topologically sorted vertices.
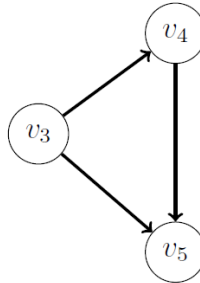
In the above, we can settle ties arbitrarily.

Second example: Given the graph below, we solve the topological sorting problem.
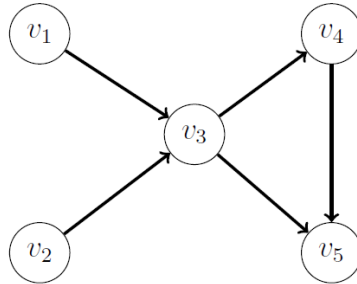
1. WLOG[3] we add vertex v_1 to the topologically sorted list. Thus our list is $L = v_1$.
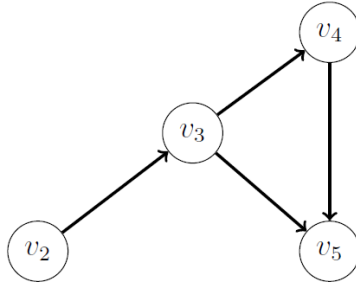
2. The only vertex with in-degree zero is vertex v_2 so we remove that from the graph and add it to the topologically sorted list. Thus, $L = v_1, v_2$.

3. The only vertex with in-degree zero is vertex v_3 so we remove that from the graph and add it ot the topologically sorted list. Thus, $L = v_1, v_2, v_3$.

4. We can remove vertex v_4 as it has in-degree zero, so our topologically sorted listed becomes $L = v_1, v_2, v_3, v_4$.

(v_5)

5. Lastly, we add v_5 to the topologically sorted list thus making our list $L = v_1, v_2, v_3, v_4, v_5$.

Q: Do you see why this works?

Q: Will this reveal a cycle in the graph? How?

Q: Why is topological sort a "decrease by a constant amount" algorithm? Given an array of n numbers, after one array element is removed to the queue, there is one less item in the graph to be sorted.

Q: What is the recurrence relation for this algorithm?

Depending on what work is being counted, we have
$$T(n) = x + T(n-1), \quad T(1) = y$$
for some constants x and y

Decrease-by-a-Constant-Factor Algorithms

Recall: What is a decrease by constant-factor-algorithm?
The first decrease-by-a-constant factor algorithm we will look at is the Binary search.

D. Binary Search

We have seen this binary search previously as an example of a recursive algorithm, but we are revisiting it since it is in this subclass of decrease-and-conquer algorithms.

Binary search assumes that the input array is sorted in increasing order. The algorithm works by comparing the middle element in the array A[0 .. n -1] to the search key k. There are three cases that could occur

1. If $A\left[\left\lfloor \frac{n-1}{2} \right\rfloor\right] = k$ then we have found the element.
2. If $A\left[\left\lfloor \frac{n-1}{2} \right\rfloor\right] > k$ then we recursively search in the subarray $A\left[0 .. \left\lfloor \frac{n-1}{2} \right\rfloor - 1\right]$.
3. If $A\left[\left\lfloor \frac{n-1}{2} \right\rfloor\right] < k$ then we recursively search in the subarray $A\left[\left\lfloor \frac{n-1}{2} \right\rfloor + 1 .. n - 1\right]$.

The book likes the iterative solution to binary search, we will look at the recursive solution.

```
1   # Input:  An array A in sorted order, end > start > 0,
2   #    and key k.
3   # Output:  Index i such that A[i] = k or None if no
4   #    match is found.
5   def BinarySearch(A, start, end, k):
6     m = math.floor((end + start) / 2)
7     if start > end:
8       return None
9     else:
10       if A[m] ==  k:
11         return m
12       elif A[m] > k:
13         return BinarySearch(A, start, m - 1, k)
14       else:
15         return BinarySearch(A, m + 1, end, k)
```

Note: You may need to import math.

The starting call is BinarySearch (A, 0, n–1, k).

Q: What is the running time of BinarySearch?

A: Clearly it is limited by the number of calls/comparisons which we can set up a recurrence relation for. In the worst case must split the array in half until we end up with one element.

The recurrence relation that describes the work done by the binary search algorithm is:
$$T(n) = T(n/2) + 2, \; T(0) = 0$$

Ideas on finding the closed form of the recurrence?
Solve the recurrence relation.

$$
\begin{aligned}
T(n) &= T(n/2) + 2 \\
&= (T(n/4) + 2) + 2 = T(n/4) + 4 \\
&= (T(n/8) + 2) + 4 = T(n/8) + 6 \\
&= (T(n/16) + 2) + 6 = T(n/16) + 8
\end{aligned}
$$
in general, after k steps,
$$= T(n/2^k) + 2k$$

For simplicity, we can assume that n is an integer power of 2, say $n = 2^{\lg(n)}$
Then, when $k = \lg(n)$, we get $n/2^k = n/n = 1$, so we get

$$
\begin{aligned}
&= T(n/2^{\lg(n)}) + 2\lg(n) \\
&= T(1) + 2\lg(n) \\
&= 2 + 2\lg(n)
\end{aligned}
$$

So, the algorithm is O(lg n).

Or, by the master method, $T(n) = T(n/2) + 2$ can be rewritten as $T(n) = 1*T(n/2) + 2*n^0$. With $a = 1$, $b = 2$, and $d = 0$, when we compare a with b^d, we get $1=2^0$, so we are in the case 2 which is $O(n^d \lg(n)) = O(\lg(n))$.

Q: Why is binary search a "decrease by a constant factor" algorithm? Given an array of n numbers and a searchkey, after one "look" into the middle of the array, if we do not find the key we are searching for, we repeat the process with only half of the array elements. So, the size of the problem decreases from n to n/2.

Example C-1: Search for the searchkey 12 in the sorted array of 11 elements:

A = [2, 5, 9, 12, 17, 23, 27, 31, 36, 39, 44] where the subscripts range from 0 to 10.

The initial function call is BinarySearch (A, 0, 10, 12).

m = 5, A[5]=23, 23 ≠ 12, and 23 > 12 so continue the search in the left half A[0, 4]

The next function call is BinarySearch (A, 0, 4, 12).

m = 2, A[2]=9, 9 ≠ 12, and 9 < 12 so continue the search in the right half A[3..4].

The next function call is BinarySearch (A, 3, 4, 12).

m = 3, A[3]=12, 12 == 12, so the searchkey was found in slot m = 3.

Example C-2: Search for the searchkey 15 in the sorted array of 11 elements

A = [2, 5, 9, 12, 17, 23, 27, 31, 36, 39, 44] where the subscripts range from 0 to 10.

The initial function call is BinarySearch (A, 0, 10, 12).

m = 5, A[5]=23, 23 ≠ 15, and 23 > 15 so continue the search in the left half A[0..4].

The next function call is BinarySearch (A, 0, 4, 12).

m = 2, A[2]=9, 9 ≠ 15, and 9 < 15 so continue the search in the right half A[3..4].

The next function call is BinarySearch (A, 3, 4, 12).

m = 3, A[3]=12, 12 ≠ 15, and 12 < 15 so continue the search in the right half A[4..4].

The next function call is BinarySearch (A, 4, 4, 12).

m = 4, A[4]=17, 17 ≠ 15 so continue the search in the right half A[5..4].

The next function call is BinarySearch (A, 5, 4, 12).

5>4 so there are no more entries in the array to search. Searchkey 15 was not found.

D. Fake-Coin Detection

Imagine for the moment that you have access to a balance scale and n identical looking coins. One of these coins is fake; it is lighter than a real coin. (We could make the corresponding change to our algorithm if the fake coin was known to be heavier.) The scale is "coarse grained"; all you get is weight relationships, not actual values.

Your goal is to design an algorithm that determines which coin is fake using as few weighings as possible.

Do we have any ideas? Think decrease by a factor and conquer style algorithms.

Here is one way to solve the problem.

Divide the coins into two piles of size

$$\lfloor \frac{n}{2} \rfloor$$

If n is odd, then set one coin aside.
If the two piles are the same weight, then the coin that was set aside is the fake coin.
If the two piles don't weigh the same and there is more than one coin in each pile, repeat the process with the lighter pile, because it contains the fake coin. If there is only one coin in each pile, the light "pile" is the light/fake coin.

Example: Trace through the algorithm with n = 100 coins, one of which is light.

Q: This algorithm is O(lg n) where the fundamental unit of work is weighing two piles of coins. Why?

$$n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow \dots \rightarrow 1 \quad \lg(n) \text{ steps}$$

Note: $1{,}000 \approx 2^{10}$ so it would take at most 10 weighings to find the one fake coin. Similarly, a million $\approx 2^{20}$ and a billion $\approx 2^{30}$.

Q: Why is fake coin detection a "decrease by a constant factor" algorithm? Given a set of n coins, after one weighing, if we do not find the fake/light coin, we repeat the process with only half of the coins. So, the size of the problem decreases from n to n/2.

Example D-1: Let O be the lighter/fake coin and X be standard coins.

X X X X X X O X X X X X (The fake coin is the 7th of 12 coins.)

Weigh two groups of 6:

X X X X X X   >   O X X X X X

Split the light group into two halves and weigh these:

O X X   <   X X X

Split the light group into two "halves" setting aside the extra final coin, and weigh the "halves":

O < X

The left coin is fake/light.

Example D-2: Let O be the lighter/fake coin and X be standard coins.

X X X X X X X X O X X X (The fake coin is the 9th of 12 coins.)

Weigh two groups of 6:

X X X X X X   >   X X O X X X

Split the light group into two halves and weigh these:

X X O   <   X X X

Split the light group into two "halves" setting aside the extra final coin, and weigh the "halves":

X = X

So the fake coin must be the one (O) set aside.

Example D-3: Let O be the lighter/fake coin and X be standard coins.

X X X O X X X X X X X (The fake coin is the 4th of 11 coins.)

Weigh two groups of 5 with the last coin set aside, and weigh the "halves":

X X X O X   <   X X X X X   X

Split the light group into two halves with the last coin set aside, and weigh the "halves":

X X   >   X O   X

Split the light group into two halves setting aside the extra final coin, and weigh the "halves":

X > O

So the fake coin must be the one on the right (O).

E. Russian Peasants Multiplication

Another decrease by a constant factor algorithm is the Russian Peasant's Multiplication
algorithm.
Admittedly, it is a rather esoteric algorithm.
Does find uses in efficient multiplication of binary numbers.
Given two integers n and m, we want to compute the product nm.

We can compute the product using the following recurrence relation

$$p(n, m) = \begin{cases} p\left(\frac{n}{2}, 2m\right), & \text{If } n \text{ is even} \\ p\left(\frac{n-1}{2}, 2m\right) + m, & \text{If } n \text{ is odd} \\ m, & \text{If } n = 1 \end{cases}$$

n is the size of the problem

Example E-1: To calculate 8 * 7, p(8, 7) = p(4, 14) = p(2, 28) = p(1, 56) = 56

Example E-2: To calculate 10 * 3, p(10, 3) = p(5, 6) = p(2, 12) + 6 = p(1, 24) + 6 = 24 + 6 = 30

Example E-3: To calculate 7* 5, p(7, 5) = p(3, 10) + 5 = [p(1, 20) + 10] + 5 = [20 + 10] + 5 = 35

Q: What would be the running time of an algorithm that implements this recurrence relation? Keep track of the recursive calls as the fundamental unit of work.

Since the algorithm terminates when the first parameter == 1, the number of recursive calls in the worst case would be given by the recurrence relation
$$T(n) = T(n/2) + 1, \ T(1) = 0.$$
Use the master method to solve the recurrence relation to find the Big-Oh class for the worst-case workload.

$T(n) = 1 * T(n/2) + n^0$. With $a = 1$, $b = 2$, $d = 0$, we get $1 = 2^0$ so we are in case 2 with
$\quad O(n^0 lg(n)) = O(lg(n))$.

If we wanted to count as work each piece of arithmetic, we would note that each recursive call involves at most one division, one multiplication, one subtraction, and one addition. So, in the worst case we have
$$T(n) = T(n/2) + 4, \ T(1) = 0.$$
$T(n) = 1 * T(n/2) + 4n^0$. With $a = 1$, $b = 2$, $d = 0$, we again get $1 = 2^0$ so we are in case 2 with

$\quad O(n^0 lg(n)) = O(lg(n))$.

Q: Why is this algorithm a "decrease by a constant factor" algorithm? The size of the problem (as measured by the first parameter) decreases by a factor of 2 with each recursive call.

Note: Not every reduce-by-a-constant-factor algorithm will reduce by a factor of 2 and will thus be O(lg n), although this was true for the three examples we played with.

Decrease-by-a-Variable Size Algorithms

The size-reduction for an instance varies from one iteration to the next.

F. Euclidean Greatest Common Divisor Algorithm

Recall that the greatest common divisor of two non-negative integers m and n is the largest integer d such that there exists two positive integers a and b where m = ad and n = bd.

The algorithm presented here is attributed to Euclid, a Greek mathematician who wrote the text *Elements* and lived around 325 BC.

```
1        Input: Two non-negative integers m > 0 and n > 0
2        Output: The greatest common divisor, d, of m and n
3        def GCD(m, n):
4        if n==0:
5                return m
6        elif:
7                return GCD(n, m%n)        #  m%n produces the remainder after dividing by n
```

Example F-1. Trace through this algorithm with the instance GCD(30, 70):
GCD(30, 70) = GCD(70, 30) = GCD(30, 10) = GCD(10, 0) = 10

Example F-2. Trace through this algorithm with the instance GCD(60, 24):
GCD(60, 24) = GCD(24, 12) = GCD(12, 0) = 12

Example F-3. Trace through this algorithm with the instance GCD(45, 17)::
GCD(45, 17) = GCD(17, 11) = GCD(11, 6) = GCD(6, 5) = GCD(5, 1) = GCD(1,0) = 1
The two numbers 45 and 17 are "relatively prime"; they have no common factors other than 1.

We will see that for this algorithm, the problem size is reduced from max(n, m) to a smaller integer that varies with the particular values of n and m, so we are in the reduce by a variable amount subclass.

G. Lomuto Partition

To partition an array means to select an element p called the pivot and arrange the elements A such that
- Every element less than p is left of p in the array.
- Every element greater than p is right of p in the array.

array elements < p	p = pivot	array element > p

One such partitioning algorithm is called Lomuto's Partitioning algorithm.
The basic idea is as follows:
Think of an array A as a subarray A[l .. r] ($0 \leq l \leq r \leq n - 1$) composed of three contiguous segments.
The segments are as follows:
1. A segment with elements less than pivot p.
2. A segment of elements greater than pivot p.
3. A segment of elements yet to be compared to pivot p.

At any point during the run of this algorithm, our array looks like:

p = pivot	array elements < p	array element > p	unexamined array elements

* Our basic setup looks like this

l s i r

p	< p	> p	?

- The algorithm proceeds as follows:
  * Starting with $i = l + 1$, scan the subarray $A[l .. r]$
  * At each iteration compare the first element in the unknown segment with p.
  * If $A[i] \geq p$ simply increment i.
  * If $A[i] < p$, increment s and swap $A[i]$ and $A[s]$. Lastly, we must increment i.
  * After the entire unknown segment has been processed, swap the pivot p with element $A[s]$.

example G-1: A = [10 , 15, 7 , 13 , 6 , 2]

Using Lomuto partitioning, we select 10 as the pivot, with s=0 and i=1.

10	15, 7, 13, 6, 2

Since A[i] = A[1] = 15 > pivot = 10, simply increment i to i=2.

10	15	7, 13, 6, 2

Since A[i] = A[2] = 7 < pivot = 10, increment s to 1, swap A[i] = A[2] = 7 and A[s] = A[1] = 15, and increment i to i=3. We now have

10	7	15	13, 6, 2

Since A[i] = A[3] = 13 > pivot = 10, simply increment i to i=4.

10	7	15, 13	6, 2

Since A[i] = A[4] = 6 < pivot = 10, increment s to 2, swap A[i] = A[4] = 6 and A[s] = A[2] = 15, and increment i to i=5. We now have

10	7, 6	13, 15	2

Since A[i] = A[5] = 3 < pivot = 10, increment s to 3, swap A[i] = A[5] = 2 and A[s] = A[3] = 13, and increment i to i=6. We now have

10	7, 6, 2	15, 13

Since i=6 is out of range, we perform one final swap of the pivot A[l]=A[0]=10 with A[s] = A[3]=2 to get the final partitioned array
    A = [2, 7, 6 , 10 , 15 , 13] with s=3.

2, 7, 6	10	15, 13

Note that the array is partitioned about the pivot 10 because the entries 2, 7, 6, are all < 10 and the entries 15, 13 are >10.

example G-2: A = [25 , 38, 16 , 42 , 89 , 12, 20]

The formal algorithm is:

```
1   # Input: A subarray A[left .. right]
2   # Output: A partition of A[left .. right] and the new pivot.
3   def LomutoPartition(A, left, right):
4     p = A[left]
5     s = left
6     for i in range(left + 1, right + 1):
7       if A[i] < p:
8         s = s + 1
9         Swap(A, s, i)
10    Swap(A, left, s)
11    return s
```

Note: Basically, s = index of the last small element and i = index of the last big element. And the pivot ends up in slot s after the final swap.

How many swaps, in the worst case, does LomutoPartition make?

Why is partitioning helpful in computing the k-th order statistic?
If p is actually in location s = k - 1 then the k-th order statistic has been found.
Otherwise,
- If s > k - 1 then, the k-th order statistic is in the left half of the array.
- If s < k - 1 then, the k-th order statistic is in the right half of the array.
Notice that the partitioning algorithm just reduces the size of the subarray that we must consider. Moreover, it is decreasing the partition by a variable amount!

We apply LomutoPartition recursively to solve the problem.

H. Selection / k-th order statistic / Quickselect

Selecting the largest, smallest, second largest, fifth smallest, etc entry in an array is actually a frequently occurring problem.

Definition (Selection Problem).
Input: *An n element sequence* $A = a_1, a_2, \ldots, a_n$ *and a number* $k \in \{1, 2, \ldots, n\}$
Output: *The k-th smallest element in A.*
The k-th smallest element in a sequence A is sometimes called the k-th order statistic.

Examples
k = 1: The smallest element in the sequence.
k = n: The largest element in the sequence.
$k = \left\lfloor \frac{n}{2} \right\rfloor$: The median element in the sequence.

One way to solve this problem is to sort the array A of numbers and then look at location A[k]. While this works, it is costly.
The best-known solution is O (n lg n).
This will work but, we can do better.

One way we can do better is through the use of a partitioning (such as Lomuto partitioning). Recall that in partitioning we select an element p called the pivot and arrange the elements A such that
- Every element less than p is left of p in the array.
- Every element greater than p is right of p in the array.

The algorithm QuickSelect is one method, that uses partitioning, to solve the k-th order statistic selection problem.

```python
# Input: Subarray A[left .. right] and integer k.
# Ouptut: The value of the k-th smallest element in
#    A[left .. right].
def QuickSelect(A, left, right, k):
    s = LomutoPartition(A, left, right)
    if s == k - 1:
        return A[s]
    elif s > left + k - 1:
        return QuickSelect(A, left, s, k)
    else:
        return QuickSelect(A, s + 1, right, k)
```

example H-1: Let's look at an example of Quickselect for sequence 4; 1; 10; 8; 7; 12; 9; 2; 15 with k = 5 and the pivots shown in bold.

1. Run `LomutoPartition`.

   (a) $\overset{s}{4}, \overset{i}{1}, 10, 8, 7, 12, 9, 2, 15$

   (b) $\mathbf{4}, \overset{s}{1}, \overset{i}{10}, 8, 7, 12, 9, 2, 15$

   (c) $\mathbf{4}, \overset{s}{1}, 10, 8, 7, 12, 9, \overset{i}{2}, 15$

   (d) $\mathbf{4}, 1, \overset{s}{2}, 8, 7, 12, 9, \overset{i}{10}, 15$

   (e) $\mathbf{4}, 1, \overset{s}{2}, 8, 7, 12, 9, 10 \; \overset{i}{15}$

   (f) $2, 1, \mathbf{4}, 8, 7, 12, 9, 10, 15$

2. Since, $s = 2$ is smaller than $k - 1 = 4$ we proceed with the right half of the partitioned array. Run `LomutoPartition`

   (a) $\overset{s}{8}, \overset{i}{7}, 12, 9, 10, 15$

   (b) $\mathbf{8}, \overset{s}{7}, \overset{i}{12}, 9, 10, 15$

   (c) $\mathbf{8}, \overset{s}{7}, 12, 9, 10, \overset{i}{15}$

   (d) $7, \mathbf{8}, 12, 9, 10, 15$

3. Since, $s = k - 1 = 4$ we stop as we have found the 5th order statistic which is 8.

What is the worst case for Quickselect?
It depends on LomutoPartition.
Every element ends up in same segment after partitioning!

What is the running time of Quickselect in the worst case?
If we have the worst case for partitioning every time, then we have

$$\sum_{i=1}^{n} i \in \Theta\left(n^2\right)$$

running time.

As a note, average case efficiency of Quickselect is linear.
It is also worth noting that Quickselect actually computes the n-k largest elements in the array.