CSC2710 Analysis of Algorithms — Fall 2022
Unit 5 - Decrease & Conquer Algorithms

Book: §2.1 – §2.2; §12.1 – §12.3; §22.4

5.1 Decrease-and-Conquer Strategy

- 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.
- We will classify Decrease-and-Conquer algorithms into three different categories.
  1. **Decrease by a constant**: The size of the instance is reduced by the same constant on each iteration of the algorithm (normally one).
     - Example: Computing a^n by realizing $a^n = a \cdot a^{n-1}$. Notice that this yields the recurrence:

$$f(n) = \begin{cases} af(n-1), & n > 0 \\ 1, & n = 0 \end{cases}$$

  2. **Decrease by a constant factor**: The size of the instance is reduced by the same constant factor on each iteration of the of the algorithm (normally the factor is two).
     - Example: Computing a^n by realizing $a^n = \left(a^{\frac{n}{2}}\right)^2$ if n is even. In the case n is odd we have $a^n = \left(a^{\frac{n-1}{2}}\right)^2 \cdot a$. Notice we have the recurrence[1]:

$$a^n = \begin{cases} \left(a^{\frac{n}{2}^{2}}\right), & n > 0 \text{ and } \exists d \in \mathbb{Z}^* \text{ such that } n = 2d \\ a\left(a^{\frac{n-1}{2}}\right)^2, & n > 0 \text{ and } \nexists d \in \mathbb{Z}^* \text{ such that } n = 2d \\ 1, & n = 0 \end{cases}$$

     - **Practice**: What do you think the running time of this algorithm is?
  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.

[1]This method is used *a lot* in cryptographic algorithms with the slight modification. The algorithm is called the square-and-multiply method

5.2 Decrease-by-a-Constant Algorithms

We'll talk about 2 algorithms where the input is decreased by a constant size every iteration.

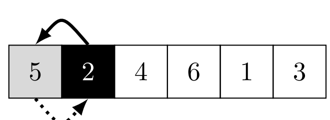
5.2.1 Insertion Sort

- We consider the sorting problem again.
  - **Practice**: What is the sorting problem?
- **Observation**: to sort an array $A[0..n-1]$ it is sufficient to place element $A[n-1]$ into the sorted sequence $A[0..n-2]$.
  - **Practice**: How do we place the element in the correct location?
  - This is observation provides a recursive description of sorting. This 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 each iteration of the algorithm.
  - After the first iteration of the sort we will guarantee that an sequence of length one is sorted.
  - In general, after iteration k completes, we will guarantee that we have an k element sorted sequence and an $n-k$ unsorted sequence.
    - Note: this does *not* mean every element is necessarily in its correct final location for any iteration $k < n$.
- The insertion sort algorithm is as follows:

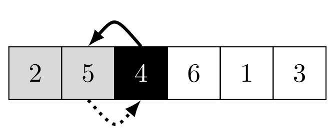
```python
# Input: An array A of integers
# Output: A is in sorted order
def InsertionSort( A ):
    for i in range( 1, len(A) ):                        # Line 1
        v = A[i]                                        # Line 2
        j = i-1                                         # Line 3
        # Insert A[i] into the sorted sequence (first part of A)
        while( j >= 0 and A[j] > v ):                   # Line 4
            A[j + 1] = A[j]                             # Line 5
            j = j - 1                                   # Line 6
        A[j + 1] = v                                    # Line 7
```

- 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.

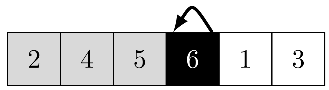
  1. Place the number 2 in the sorted sequence.



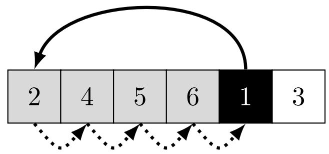
  2. Place the number 4 in the sorted sequence.



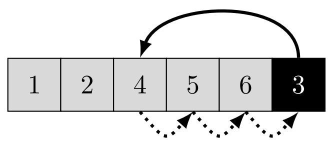
3. Place the number 6 in the sorted sequence. Note, there is no change in the position of the number.



4. Place the number 1 in the sorted sequence.



5. Place the number 3 in the sorted sequence.



6. The finalized sequence is:

1	2	3	4	5	6

- Let's analyze the running time of insertion sort by constructing a table.

Line	**Cost**	**Count**	
1	c_1	$n-1$	
2	c_2	$n-1$	
3	c_3	$n-1$	
4	c_4	$\sum_{i=1}^{n-1} t_i$	t_i is the number of iterations performed for iteration j.
5	c_5	$\sum_{i=1}^{n-1} t_i$	
6	c_6	$\sum_{i=1}^{n-1} t_i$	
7	c_7	$n-1$	

  - What is the value of t_i in the worst case?
    * In the worst case t_i is i.
  - What is $\sum_{i=1}^{n-1} t_i$, the dominating term, in the worst case?

$$\begin{aligned}\sum_{i=1}^{n-1} t_i &= \sum_{i=1}^{n-1} i \\ &= \frac{(n-1)\,n}{2} && \text{(by the Gaussian sum)} \\ &\le n^2\end{aligned}$$

  - **Practice**: What is the running time of this algorithm?

5.2.2 Topological Sorting

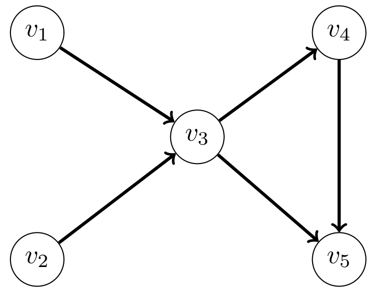
- In order to understand Topological sorting we must first recall a few things about directed graphs.
  - **Directed Graph**(Digraph): A graph $G = (V, E)$ is a graph where V is the set of vertices, E is a set of edges, and $(u, w) \in E$ does not imply $(w, u) \in E$.
  - A directed graph is called *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 visted 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 parents 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 topologoical sort.

  Definition 1 (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.*

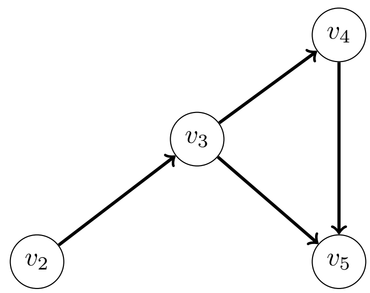
- Applications
  - instruction scheduling in program compilation;
  - resolving symbol dependencies in linkers;
  - how to progress through the CS major;
  - any dependency graph you can dream up.
- The common solution (assuming the graph is a DAG)
  1. Run the DFS.
  2. As each vertex becomes a dead 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 its popped off the stack.)
- Why does this work?
  - Since there are no backedges, when a vertex v is pushed on to the stack there will *not* be vertices u with edge (u, v) below it. Otherwise, we would be in violation of the no backedges rule.
- The decrease and conquer solution.
  - Repeat the following until the graph has one vertex.
    - Locate a vertex with in-degree zero[2].
    - 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.

[2]Recall, in-degree of a vertex v is the number of edges $(u, v) \in E$.

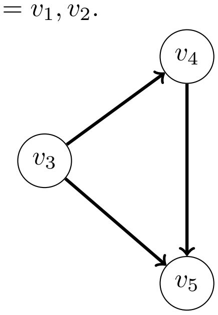
- **Practice**: What kind of decrease and conquer algorithm is this?
- Example: Given the graph below, we solve the topological sorting problem.



1. WLOG[3] we add vertex v_1 to the topologically sorted list. Thus our list is $L = v_1$.



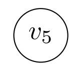
2. The only vertex with in-degree zero is vertex v_2 so we remove that from the graph and add it to the topologically sorted list. Thus, $L = v_1, v_2$.



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, $L = v_1, v_2, v_3$.

[3] "Wah-log" without loss of generality. It means making a certain choice doesn't affect the correctness of our argument.

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



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

- **Practice**: Do you see why this works?
- **Practice**: Will this process reveal a cycle in the graph? If so how?

5.3 Decrease-by-a-Constant-Factor Algorithms

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

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

  - The starting call is BinarySearch $(A, 0, n-1, k)$.
- What is the running time of BinarySearch?
  - 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.

$$C(n) = C\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + 1$$

  - The initial condition is $C(1) = 1$.
- Ideas on finding the closed form of the recurrence?
  - Idea: Let $n = 2^k$ and solve the recurrence for k.

$$\begin{aligned}
C\left(2^k\right) &= C\left(2^{k-1}\right) + 1 \\
&= \left(C\left(2^{k-2}\right) + 1\right) + 1 \\
&\vdots \\
&= (C(1)) + k && \text{By the fact } k - k = 0 \text{ which gives us } 2^0 = 1 \\
&= k + 1
\end{aligned}$$

  - What's k?
    - $k = \lg n$.
  - So we have that the closed form of the recurrence is

$$C(n) = \lfloor \lg n \rfloor + 1 = \lceil \lg(n+1) \rceil .$$

  - This all implies that the algorithm is $\Theta(\lg n)$.

5.3.1 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. Your goal is to design an algorithm that determines which coin is fake.
  - It is prudent to use a few weighings as possible.
  - The scale is coarse grained all you get is weight relationships not actual values.
  - We will assume that the fake coin does *not* weigh the same as a genuine coin. In fact, it is lighter
- 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 the coin put aside is the fake coin.
  - If the two piles don't weigh the same, repeat the process with the lighter pile.
- **Practice**: I claim my solution is $\Theta(\lg n)$. Why am I right?

5.3.2 Russian Peasants Multiplication

- Another decrease by a constant factor algorithms 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

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

- **Practice**: What would be the running time of an algorithm that implements this recurrence relation?

5.4 Variable-Size-Decrease Algorithm

- **Practice**: What does it mean when we say the algorithm is a variable size decrease and conquer?
- We start by looking at an *extremely* frequently occurring problem.

  Definition 2 (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.
- **Practice:** What is a brute-force solution for this problem?
- 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*
  - 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.
  - One such partitioning algorithm is called *Lomuto's Partitioning* algorithm.
  - The basic idea is as follows:
    * Think of an array A as a a subarray $A[l..r]$ ($0 \leq l \leq r \leq n-1$) composed for 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.
    * 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 $A[s]$.

- The formal algorithm is as follows:

```
# Input: A subarray A[l..r]
# Output: Partition of A[l..r] and the new position of the pivot
def LumotoPartition(A, left, right):
    p = A[left]
    s = left
    for i in range(left+1, right+1):
        if( A[i] < p ):
            s = s + 1
            Swap(A, s, i)
    Swap(A, left, s)
    return s
```

- **Practice**: 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 paritioning algorithm just reduces the size of the subarray that we must consider.
    * Moreover, it is decreasing the partition by a varaible amount!
  - We apply LomutoPartition recursively to solve the problem.
- The algorithm QuickSelect is one method, that uses partitioning, to solve the selection problem.

```
# Input: A subarray A[left .. right] of A and an integer k
#    such that 1 <= k <= right - left + 1
# Output: The value of the k-th smallest element in A[left .. right]
def QuickSelect( A, left, right, k ):
    s = LumotoPartition(A, left, right)            # Line 1
    if( s == k-1 ):                                # Line 2
        return A[s]                                # Line 3
    elif( s > left + k - 1):                       # Line 4
        return QuickSelect( A, left, s-1, k )      # Line 5
    else:                                          # Line 6
        return QuickSelect( A, s+1, right, k )     # Line 7
```

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

(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 5^{th} 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.

5.4.1 Interpolation Search

- Similar to binary search *except* it considers the value of the search key in order to find the array's element to be compared with the search key.
- In particular, the elements of the array are assumed to be sorted in increasing order.
  - Intuition: treat array as a sequence of points. Our goal is to use two of the points, the start and end of the subarray, to estimate the the location of a key k in the array.
    - The x-coordinates of the points are indices
    - The y-coordinates of the points are the values stored at in the array.
- General Idea: assume values in the array increase linearly and and perdict[4] the location of the key using the point-slope form of a line over the endpoints of the subarray. Then we peredict the x- coordinate for the key k by solving for x.
  - Recall the point-slope form of a line is given by

    $$y - y_1 = m\left(x - x_1\right),$$

    where m is the slope of the line.
  - On the iteration dealing with left most element $A[l]$ and right-most element $A[r]$ we have two points $(l, A[l])$ for (x_1, y_1) and $(r, A[r])$ for (x, y). The point slope form is then:

    $$y - A[l] = \frac{A[r] - A[l]}{r - l}(x - l)$$

  - In order to determine the likely x-coordinate we simply solve for x

    $$\begin{aligned}
    y - A[r] &= \frac{A[r] - A[l]}{r - l}(x - l) \\
    \implies (y - A[l])(r - l) &= (A[r] - A[l])(x - l) \\
    \implies \frac{(y - A[l])(r - l)}{(A[r] - A[l])} &= x - l \\
    \implies \frac{(y - A[l])(r - l)}{(A[r] - A[l])} + l &= x
    \end{aligned}$$

  - The x-coordinate must be an integer in order to be an index so we take the floor of the result.

    $$x = \left\lfloor \frac{(y - A[l])(r - l)}{(A[r] - A[l])} \right\rfloor + l$$

  - If we substitute in k, then we have

    $$x = \left\lfloor \frac{(k - A[l])(r - l)}{(A[r] - A[l])} \right\rfloor + l$$

  - If $A[x] = k$ we have found the element.
  - If $A[x] > k$ we repeat the process for subarray $A[x + 1..r]$.
  - If $A[x] < k$ we repeat the process for subarray $A[l..x - 1]$.
- On average, the interpolation search algorithm performs $\lg\lg n + 1$ comparisons.

[4] This is technically called interpolation.

5.4.2 Game of Nim

- Another place where we see the decrease by a variable amount and conquer is in the game of Nim.
- Nim is a two player game.
  - **Setup**: Construct n piles $p_1, p_2, \dots p_n$ and place t_i tokens in pile p_i.
  - **Rules**: Players alternate removing tokens from piles. On there turn, a player may remove as many tokens as they want from exactly one pile.
  - **Winning**: A player wins if they are the last player that is able to remove tokens.
- It turns out, that if we encode the pile p_i sizes in binary b_i, the goal for a player is to remove enough tokens from some pile such that $b_1 \oplus b_2 \oplus \cdots \oplus b_n = 0$
- This is a really fun problem I encourage you to look at the book and read up more on this fun little game.

Challenge Problems

- Recall the structure and behavior of a Binary Search Tree (BST).
  - How does a search for a key in a binary search tree work?
    * Why is this considered a variable decrease and conquer problem?
    * What is the search time in the worst case?
    * What is the search time in the average case?
  - How do you insert a new element into a binary search tree?
    * Why is this considered a variable decrease and conquer problem?
    * What is the insertion time in the worst case?
    * What is the insertion time in the average case?
- Implement both binary search and interpolation search algorithms
  - Populate an array of 10,000 integers with random numbers between 0 and 1,000,000.
  - Sort them each (just use built-in sort functionality)
  - Implement both search algorithms.
  - Search for 1,000,000 random integers in the arrays, using a timing function to time the running time of each function. Which has a better average search time?
- Determine a decrease-and-conquer algorithm to solve the *highest peak* problem. Your algorithm should find the element that is the furthest distance above (greater than) its two neighbors, thus being the highest peak.