Decrease & conquer

- decrease by constant
  - sequential search, insertion sort, topological sort
- decrease by constant factor
  - binary search, fake coin problem
- decrease by variable amount
  - BST search, selection problem

Decrease & Conquer

- based on exploiting the relationship between a solution to a given instance of a problem and a solution to a smaller instance
  - once the relationship is established, it can be exploited either bottom-up or top-down

EXAMPLE: sequential search of N-item list
- checks the first item, then searches the remaining sublist of N-1 items

EXAMPLE: binary search of sorted N-item list
- checks the middle item, then searches the appropriate sublist of N/2 items

3 major variations of decrease & conquer
1. decrease by a constant (e.g., sequential search decreases list size by 1)
2. decrease by a constant factor (e.g., binary search decrease list size by factor of 2)
3. decrease by a variable amount
Decrease by a constant

decreasing the problem size by a constant (especially 1) is fairly common
- many iterative algorithms can be viewed this way
  e.g., sequential search, traversing a linked list
- many recursive algorithms also fit the pattern
  e.g., \[ N! = (N-1)! \times N \quad a^n = a^{n-1} \times a \]

EXAMPLE: insertion sort (decrease-by-constant description)

to sort a list of \( N \) comparable items:
1. sort the initial sublist of (N-1) items
2. take the Nth item and insert it into the correct position

Top-down (recursive) definition

can use recursion to implement this algorithm
- this a top-down approach, because it starts with the entire list & recursively works down to the base case

```java
public static <T extends Comparable<? super T>>
  void insertionSortRec(T[] items) {
    Decrease.insertionSortRecHelper(items, items.length-1);
  }

private static <T extends Comparable<? super T>>
  void insertionSortRecHelper(T[] items, int lastIndex) {
    if (lastIndex > 0) {
      Decrease.insertionSortRecHelper(items, lastIndex-1);
      T value = items[lastIndex];
      int j = lastIndex-1;
      while (j >= 0 && items[j].compareTo(value) > 0) {
        items[j+1] = items[j];
        j--;
      }
      items[j+1] = value;
    }
  }
```
Bottom-up (iterative) definition

for most decrease-by-constant algorithms
- top-down recursion is not necessary and in fact is less efficient
- instead, could use a loop to perform the same tasks in a bottom-up fashion

```java
public static <T extends Comparable<? super T>>
void insertionSort(T[] items) {
    for (int i = 1; i < items.length; i++) {
        T value = items[i];
        int j = i - 1;
        while (j >= 0 && items[j].compareTo(value) > 0) {
            items[j+1] = items[j];
            j--;
        }
        items[j+1] = value;
    }
}
```

Big-Oh of insertion sort

analyzing the recursive version
- cost function?
- worst case Big-Oh?

```java
public static <T extends Comparable<? super T>>
void insertionSortRec(T[] items, int lastIndex) {
    int j = lastIndex-1;
    while (j >= 0 && items[j].compareTo(value) > 0) {
        items[j+1] = items[j];
        j--;
    }
    items[j+1] = value;
}
```

analyzing the iterative version
- worst case Big-Oh?

```java
public static <T extends Comparable<? super T>>
void insertionSort(T[] items) {
    for (int i = 1; i < items.length; i++) {
        T value = items[i];
        int j = i - 1;
        while (j >= 0 && items[j].compareTo(value) > 0) {
            items[j+1] = items[j];
            j--;
        }
        items[j+1] = value;
    }
```
Analysis of insertion sort

insertion sort has advantages over other $O(N^2)$ sorts

- best case behavior of insertion sort?
  - what is the best case scenario for a sort?
  - does insertion sort take advantage of this scenario?

  does selection sort?

- what if a list is partially ordered? (a fairly common occurrence)
  - does insertion sort take advantage?

Another example

we considered the problem of topological sorting in 321

- given a directed graph, find an ordering of the vertices such that if edge $(v_i, v_j)$ is in the graph, then $v_i$ comes before $v_j$ in the ordering
- as long as there are no cycles in the graph, at least one (and possibly many) topological sorts exists

![Graph](image)

C1, C2, C3, C4, C5 or C2, C1, C3, C4, C5

topological sorting is useful in many applications

- constructing a job schedule among interrelated tasks
- cell evaluation ordering in spreadsheet formulas
- power rankings of sports teams

- may sometimes generate a topological sort to verify that an ordering is possible, then invest resources into optimizing
DFS-based topological sort

we considered an algorithm based on depth first search in 321
- traverse the graph in a depth-first ordering (using a stack for reached vertices)
- when you reach a dead-end (i.e., pop a vertex off the stack), add it to a list
- finally, reverse the list

[Diagram of a graph with vertices C1 to C5 and arrows indicating the order of vertices]

Decrease-by-1 topological sort

alternatively,
- while the graph is nonempty
  1. identify a vertex with no incoming edges
  2. add the vertex to a list
  3. remove that vertex and all outgoing edges from it

[Diagram of a graph showing the decrease-by-1 process and the resulting order of vertices]

The solution obtained is C1, C2, C3, C4, C5
Decrease by a constant factor

A more efficient (but rare) variation of decrease & conquer occurs when you can decrease by an amount proportional to the problem size.

- e.g. binary search → divides a problem of size N into a problem of size N/2

\[
\text{Cost}(N) = \text{Cost}(N/2) + C = (\text{Cost}(N/4) + C) + C = \text{Cost}(N/4) + 2C = (\text{Cost}(N/8) + C) + 2C = \text{Cost}(N/8) + 3C = \ldots = \text{Cost}(1) + (\log_2 N)C = C \log_2 N + C' \Rightarrow O(\log N)
\]

Fake coin problem

Given a scale and N coins, one of which is lighter than all the others. Identify the light coin using the minimal number of weighings.

- solution?
- cost function?
- big-Oh?
What doesn't fit here

decrease by a constant factor is efficient, but very rare

it is tempting to think of algorithms like merge sort & quick sort
  ▪ each divides the problem into subproblems whose size is proportional to the original
  ▪ key distinction: these algorithms require solving multiple subproblems, then
    somehow combining the results to solve the bigger problem
  ▪ we have a different name for these types of problems: divide & conquer
  ▪ NEXT WEEK

Decrease by a variable amount

sometimes things are not so consistent
  ▪ the amount of the decrease may vary at each step, depending on the data

EXAMPLE: searching/inserting in a binary search tree
  ▪ if the tree is full & balanced, then each check reduces the current tree into a subtree
    half the size
  ▪ however, if not full& balanced, the sizes of the subtrees can vary
  ▪ worst case: the larger subtree is selected at each check
  ▪ for a linear tree, this leads to O(N) searches/insertions
  ▪ in general, the worst case on each check is selecting the larger subtree
  ▪ recall: randomly added values produce enough balance to yield O(log N)
Selection problem

suppose we want to determine the kth order statistic of a list
- i.e., find the kth largest element in the list of N items
  (special case, finding the median, is the \(N/2\)th order statistic)
- obviously, could sort the list then access the kth item directly
  \(\Rightarrow O(N \log N)\)
- could do k passes of selection sort (find 1st, 2nd, ..., kth smallest items)
  \(\Rightarrow O(k \times N)\)
- or, we could utilize an algorithm similar to quick sort called quick select

recall, quick sort works by:
  1. partitioning the list around a particular element (i.e., moving all items \(\leq\) the pivot element to its left, all items \(>\) the pivot element to its right)
  2. recursively quick sorting each partition

Lomuto partition

we will assume that the first item is always chosen as pivot
- simple, but "better" strategies exist for choosing the pivot
- mark the first index as the pivot index
- traverse the list, for each item < the pivot
  increment the pivot index & swap the item into that index
- finally, swap the pivot into the pivot index

\[
\begin{array}{cccccccc}
5 & 8 & 7 & 2 & 1 & 6 & 4 \\
5 & 8 & 7 & 2 & 1 & 6 & 4 \\
5 & 8 & 7 & 2 & 1 & 6 & 4 \\
5 & 8 & 7 & 2 & 1 & 6 & 4 \\
5 & 2 & 7 & 8 & 1 & 6 & 4 \\
5 & 2 & 1 & 8 & 7 & 6 & 4 \\
5 & 2 & 1 & 4 & 7 & 6 & 8 \\
4 & 2 & 1 & 5 & 7 & 6 & 8 \\
\end{array}
\]
Lomuto partition implementation

for both quick sort & quick select, we need to be able to partition a section of the list

- assume parameters left & right are the lower & upper indexes of the section to be partitioned

```java
private static <T extends Comparable<? super T>>
    int partition(T[] items, int left, int right) {
        int pivotIndex = left;
        for (int i = left+1; i < right; i++) {
            if (items[i].compareTo(pivot) < 0) {
                pivotIndex++;
                T temp = items[pivotIndex];
                items[pivotIndex] = items[i];
                items[i] = temp;
            }
        }
        T temp = items[pivotIndex];
        items[pivotIndex] = items[left];
        items[left] = temp;
        return pivotIndex;
    }
```

Quick select algorithm

to find the kth value in a list:

1. partition the list (using Lomuto’s partitioning algorithm)
2. let pivotIndex denote the index that separates the two partitions
3. if k = pivotIndex+1, then
   return the value at pivotIndex
4. if k < pivotIndex+1, then
   recursively search for the kth value in the left partition
5. if k > pivotIndex+1, then
   recursively search for the (k – (pivotIndex+1))th value in the right partition

here, pivotIndex = 3
if k = 4, then answer is in index 3 = 5
if k = 2, then find 2nd value in
if k = 5, then find 1st value in
Quick select implementation

can be implemented recursively or iteratively
- recursive solution requires a private helper method that utilizes the left & right boundaries of the list section
- as with partition, the code must adjust for the changing boundaries

public static <T extends Comparable<? super T>>
T quickSelect(T[] items, int k) {
  return Decrease.quickSelect(items, 0, items.length - 1, k);
}

private static <T extends Comparable<? super T>>
T quickSelect(T[] items, int left, int right, int k) {
  int pivotIndex = Decrease.partition(items, left, right);
  if (k == pivotIndex + 1 - left) {
    return items[pivotIndex];
  } else if (k < pivotIndex + 1 - left) {
    return Decrease.quickSelect(items, left, pivotIndex - 1, k);
  } else {
    return Decrease.quickSelect(items, pivotIndex + 1, right, k - (pivotIndex - left + 1));
  }
}

Efficiency of quick select

analysis is similar to quick sort

- if the partition is perfectly balanced:
  \[ \text{Cost}(N) = \text{Cost}(N/2) + C \cdot N + C \]
  \[ = (\text{Cost}(N/4) + C \cdot N/2 + C) + C \cdot N + C \]
  \[ = \cdots \]
  \[ = \text{Cost}(1) + 2C \cdot N + (\log_2 N)C \]
  \[ \Rightarrow O(N) \]

- in the worst case, the pivot chosen is the smallest or largest value
  this reduces the algorithm to decrease-by-1 \( \Rightarrow O(N^2) \)

- as long as the partitioning is reasonably balanced, get \( O(N) \) behavior in practice
  there is a complex algorithm for choosing the pivot that guarantees linear performance