COS 397 – week 03

Divide and Conquer. Greedy Algorithms.

Divide and conquer – algorithmic technique

Definition: Solve a problem, either directly because solving that instance is easy (typically, because the instance is small) or by *dividing* it into two or more smaller instances. Each of these smaller instances is (usually) *recursively* solved, and the solutions are combined to produce a solution for the original instance.
Note: The technique is named "divide and conquer" because a problem is conquered by dividing it into several smaller problems. This technique yields elegant, simple and quite often very efficient algorithms. Well-known examples include:

heapify,
Heap (heap property) is a *complete tree* where every *node* has a *key* more extreme (greater or less) than or equal to the key of its *parent*. Usually understood to be a *binary heap*.
Heapify is rearrange a *heap* to maintain the *heap property*. If the root node's key is not more extreme, swap it with the most extreme child key, then *recursively* heapify that child's subtree. The child subtrees must be heaps to start.

merge sort, ***quicksort***,

binary search.
(Why is binary search included? The dividing part picks which segment to search, and "the solutions are combined" trivially: take the answer from the segment searched. Segments not searched are "recursively solved" by the null operation: they are ignored.) A similar principle is at the heart of several important data structures such as binary search tree, multiway search trees, tries, skip lists, multidimensional search trees (k-d trees, quadtrees), etc.

Divide-and-conquer is a top-down technique for designing algorithms that consists of dividing the problem into smaller subproblems hoping that the solutions of the subproblems are easier to find and then composing the partial solutions into the solution of the original problem.

Little more formally, divide-and-conquer paradigm consists of following major phases:

- Breaking the problem into several sub-problems that are similar to the original problem but smaller in size,
- Solve the sub-problem recursively (successively and independently), and then
- Combine these solutions to subproblems to create a solution to the original problem.

Binary Search (simplest application of divide-and-conquer)
Binary Search is an extremely well-known instance of divide-and-conquer paradigm. Given an ordered array of n elements, the basic idea of binary search is that for a given element we "probe" the middle element of the array. We continue in either the lower or upper segment of the array, depending on the outcome of the probe until we reached the required (given) element.
Problem Let $A[1 \ldots n]$ be an array of non-decreasing sorted order; that is $A[i] \leq A[j]$ whenever $1 \leq i \leq j \leq n$. Let q be the query point. The problem consists of finding q in the array A. If q is not in A, then find the position where q might be inserted. Formally, find the index i such that $1 \leq i \leq n+1$ and $A[i-1] < x \leq A[i]$.
Sequential Search
Look sequentially at each element of A until either we reach at the end of an array A or find an item no smaller than 'q'. Sequential search for '*q*' in array A:

```
for i = 1 to n do
    if A [i] ≥ q then
        return index i
    return n + 1
```

Analysis: This algorithm clearly takes a θ(r), where r is the index returned. This is Ω(n) in the worst case and O(1) in the best case. If the elements of an array A are distinct and query point *q* is indeed in the array then loop executed (*n* + 1) / 2 average number of times. On average (as well as the worst case), sequential search takes θ(n) time.

Binary Search

Look for *q* either in the first half or in the second half of the array *A*. Compare *q* to an element in the middle, *n/2,* of the array. Let $k = n/2$. If $q \leq A[k]$, then search in the $A[1 \ldots k]$; otherwise search $T[k+1 \ldots n]$ for *q*. Binary search for *q* in subarray $A[i \ldots j]$ with the promise that $A[i-1] < x \leq A[j]$

```
If i = j then return i
k = (i + j)/2
if q ≤ A [k]
    then return Binary Search [A [i..k], q]
    else return Binary Search [A[k+1..j], q]
```

Analysis. Binary Search can be accomplished in logarithmic time in the worst case, i.e., $T(n) = \theta(\log n)$. This version of the binary search takes logarithmic time in the best case.

Iterative Version of Binary Search. Interactive binary search for *q*, in array $A[1 \ldots n]$

```
if(q > A [n]) return n + 1
i = 1; j = n;
while (i < j)
{ k = (i + j)/2;
  if (q ≤ A [k]) j = k;
  else i = k + 1;
}
return i
```

Fast Exponentiation

Suppose that we need to compute the value of a^n for some reasonably large *n*. Such problems occur in primality testing for cryptography.

The simplest algorithm performs *n* – 1 multiplications, by computing $a \times a \times \ldots \times a$. However, we can do better by observing that:

If *n* is even, then $a^n=(a^{n/2})^2$. If *n* is odd, then $a^n=a(a^{n/2})^2$. In either case, we have halved the size of our exponent at the cost of at most two multiplications, so O(log *n*) multiplications suffice to compute the final value.

```
int power(int a, int n)
{
  if (n == 0) return(1);
  x = power(a,n/2);
  if (n%2==0)return x*x;
  else return a*x*x;
}
```

This simple algorithm illustrates an important principle of “divide and conquer”. It always pays to divide a job as evenly as possible. This principle applies to real life as well. When *n* is not a power of two, the problem cannot always be divided perfectly evenly, but a difference of one element between the two sides cannot cause any serious imbalance.

Merge Sort

```
void merge(int a[],
           int beg, int mid, int end)
{ int n = end - beg + 1;
  int b[n];
  int i1 = beg;
  int i2 = mid + 1;
  int j = 0;
  while ((i1 <= mid) && (i2 <= end))
  { if (a[i1] < a[i2]) {b[j]=a[i1];
i1++;}
    else {b[j]=a[i2]; i2++;}
    j++;
  }
  while(i1 <= mid) {b[j]=a[i1]; i1++;
j++;}
  while(i2 <= end) {b[j]=a[i2]; i2++;
j++;}
  for(j=0; j<n; j++) a[beg+j] = b[j];
}
```

Example run:

```
int a[]=
{5,9,10,12,17,8,11,20,32};
merge(a,0,4,8);
```

(5,9,10,12,17)(8,11,20,32) ⇒ ()
(9,10,12,17)(8,11,20,32) ⇒ (5)
(9,10,12,17)(11,20,32) ⇒ (5,8)
(10,12,17)(11,20,32) ⇒ (5,8,9)
(12,17)(11,20,32) ⇒ (5,8,9,10)
(12,17)(20,32) ⇒ (5,8,9,10,11)
(17)(20,32) ⇒ (5,8,9,10,11,12)
()(20,32) ⇒ (5,8,9,10,11,12,17)
()(32) ⇒ (5,8,9,10,11,12,17,20)
()() ⇒ (5,8,9,10,11,12,17,20,32)

Sorting program:

```
void merge_sort(int a[],int beg, int end)
{ if (beg == end) return;
  int mid = (beg + end) / 2;
  merge_sort(a, beg, mid);
  merge_sort(a, mid + 1, end);
  merge(a, beg, mid, end);
}
```

Quick-Sort

Divide-and-Conquer sort algorithm.
The idea in Quick-Sort is to partition the array into 3 sections.
L - the elements smaller than the pivot
E - the elements equal to the pivot
G - the elements larger than the pivot
and than call Quick-Sort recursively with L and G

```
QuickSort(L)
  pick random pivot x in L
  partition L into:
    L < x   , elements less than x
    E = x  , elements equal to x
    G > x  , elements greater than x
  QuickSort(L)
  QuickSort(G)
  L = Concatenate(L, E, G)
  L = Concatenate(L, E, G)
```

Quick Sort Algorithm:

```
void qsort(int a[], int L, int R)
{ int i=L, j=R;
  int x=a[(i + j)/2];
  do
  { while (a[i] < x) i++;
    while (a[j] > x) j--;
    if (i <= j)
    { swap(a[i], a[j]); i++; j--; }
  }
  while (i <= j);
  if (L < j) qsort(a, L, j);
  if (i < R) qsort(a, i, R);
}
```

<u>**worst case:**</u>
The pivot is unique minimum or maximum element $\Rightarrow$ L or G has size n-1
$T(n) = T(n-1) + O(n) \Rightarrow T(n) = O(n^2)$

<u>**Best case:**</u>

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$$

TASK:
Given a set S of n elements and an index k ($1 \le k \le n$), **the k-smallest element** is the k-th element when the elements are sorted from the smallest to the largest.

O(n) <u>expected time</u> algorithm for finding the k-smallest element in the mean case.

<u>**Example:**</u>
For the given set of numbers: {6, 3, 2, 4, 1, 1, 2, 6,}
The 4-smallest element is 2 since in the 2 is the 4'th element in the sorted set {1, 1, 2, 2, 3, 4, 6, 6}.

Solution:
The algorithm is based on the Quick-Sort algorithm.

```
Select(k, S)
   pick x in S
   partition S into:
      L < x
      E = x
      G > x
   if k ≤ length(L)
      return Select(k, L)
   else if k ≤ length(L) + length(E)
           return x
        else
           return Select(k - length(L) - length(E), G)
```

In the worst case, all the elements are larger than x (L is empty) and length(E) = 1 $\Rightarrow$
it takes: n + (n-1) + (n-2) + ... + 1 = O(n^2)

In the mean case, similar to quick-sort, half of the elements in L are good pivots, for which the size of L and G are each less than $\frac{3n}{4}$.

Therefore: $T(n) \le T\left(\frac{3n}{4}\right) + O(n) = O(n)$

TASK:

Given an array of n numbers, suggest an O(n) expected time algorithm to determine whether there is a number in A that appears more than $\frac{n}{2}$ times.

Solution:

If *x* is a number that appears more than $\frac{n}{2}$ times in A, than *x* is the $\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right)$-smallest in A.

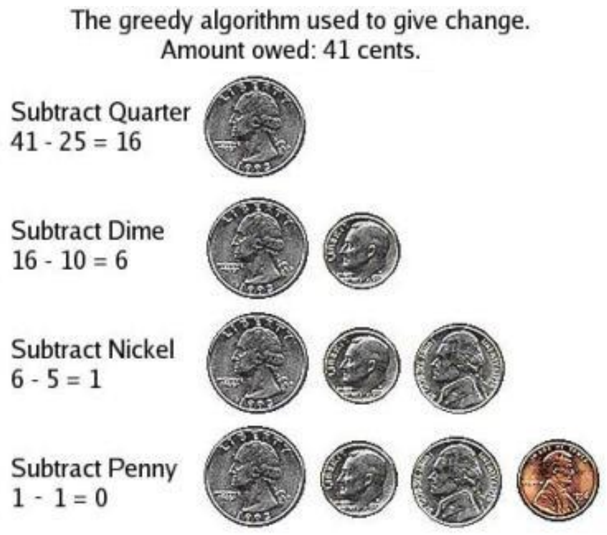
Therefore, the algorithm is:

```
$x \leftarrow$ Select ($\left(\left\lfloor\frac{n}{2}\right\rfloor+1\right)$, A)
$apps \leftarrow 0$
for  i ← 1 to n do:
   if (A[i] = $x$)
        $apps$ ++
if  $apps > \frac{n}{2}$
      return TRUE
else return FALSE
```

Time Complexity :
In the mean case, *Select* algorithm runs in O(n), computing *apps* takes O(n) as well.
Total run time in the mean case: O(n)

GREEDY APPROACH

The greedy Algorithm works by making the decision that seems most promising at any moment; it never reconsiders this decision, whatever situation may arise later. As an example consider the problem of "Making Change":



The greedy algorithm determines the **minimum amount** of US coins to give while making change. These are the steps a human would take to emulate a greedy algorithm. Greed manifests itself in this process because the algorithm picks the coins of highest value first.

Problem Make a change of a given amount using the smallest possible **number of coins**.

Informal Algorithm
- Start with nothing.
- at every stage without passing the given amount.
  - add the largest to the coins already chosen.

Formal Algorithm
Make change for *n* units using the least possible number of coins.

```
MAKE-CHANGE (n)
    C ← {100, 25, 10, 5, 1}     // constant.
    S ← {};                      // set that will hold the solution set.
    sum ← 0 sum of item in solution set
    WHILE sum not = n
        x = largest item in set C such that sum + x ≤ n
        IF no such item THEN
            RETURN    "No Solution"
        S ← S and {value of x}
        sum ← sum + x
    RETURN S
```

Characteristics and Features of Problems solved by Greedy Algorithms

Construct the solution in an optimal way. Algorithm maintains two sets. One contains chosen items and the other contains rejected items.
The greedy algorithm consists of four functions.
1. A function that checks whether chosen set of items provide a solution.
2. A function that checks the feasibility of a set.
3. The selection function tells which of the candidates is the most promising.
4. An objective function, which does not appear explicitly, gives the value of a solution.

Structure Greedy Algorithm

- Initially the set of chosen items is empty i.e., solution set.
- At each step
  - item will be added in a solution set by using selection function.
  - IF the set would no longer be feasible
    - reject items under consideration (and is never consider again).
  - ELSE IF set is still feasible THEN
    - add the current item.

Definitions of feasibility

A feasible set (of candidates) is promising if it can be extended to produce not merely a solution, but an optimal solution to the problem. In particular, the empty set is always promising why? (because an optimal solution always exists)

A greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, reducing each problem to a smaller one.

The "greedy-choice property" and "optimal substructure" are two ingredients in the problem that lend to a greedy strategy.

It says that a globally optimal solution can be arrived at by making a locally optimal choice.

Egyptian fractions: The ancient Egyptians only used fractions of the form $^1/_n$ so any other fraction had to be represented as a *sum of such unit fractions* and, furthermore, all the unit fractions were different! An example:
$^6/_7 = {^1/_2} + {^1/_3} + {^1/_{42}}$
This problem is solved using the greedy method.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimization methods like dynamic programming. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees, Dijkstra's algorithms for finding Single-Source Shortest paths, and the algorithm for finding optimum Huffman trees.
An algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally, *optimal solution* for some *optimization problems*, but may find less-than-optimal solutions for some instances of other problems.

Examples

An activity selection problem:

Suppose we have a set S = {1,2…n} of n proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity i has a start time s_i and a finish time f_i, where $s_i \leq f_i$. If selected, activity i takes place during the half-open time interval $[s_i, f_i)$. Activities i and j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. The Activity selection problem is to select a maximum-size set of mutually compatible activities.

In order to solve the activity selection problem, we make a greedy choice of activity 1. Upon making such a choice the reduced sub problem is a proper subset of the original problem. So by induction greedy algorithm yields optimal solution.

Knapsack problem:

A thief robbing a store finds n item; the ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integer W. What items should he take? (This is called the 0-1 knapsack problem because each item must either be taken or left behind. In the fractional knapsack problem the thief can take fraction of item rather than to make binary choices.)

In fractional knapsack problem, the thief can take as much as possible an item with greatest value per pound. But in the 0-1 knapsack case since the thief has to choose between items fully he may not be able to optimally fill the knapsack with items of greater value.

A greedy algorithm for the activity-selection problem:
Assumptions**:** Input activities are in order by non-decreasing finishing time, and input s and f (starting and finishing times) are represented as arrays.
Pseudocode:

```
Greedy-Activity-Selector(s,f)
n  ← length[s]; A ← {1}; J ← 1
For i ← 2 to n Do
 if $s_i$ >= $f_i$ then A ← A U {i}
   j ← i
return A
```

Proof of correctness:
To prove, *s*how that the activity problem satisfied;

- Greedy choice property.
- Optimal substructure property.

Let *S = {1, 2, . . . , n}* be the set of activities. Since activities are in order by finish time. It implies that activity 1 has the earliest finish time. Suppose, $A \subseteq S$ is an optimal solution and let activities in *A* are ordered by finish time. Suppose, the first activity in *A* is *k*. If k = 1, then A begins with greedy choice then there is nothing to proof here. If k $\neq$ 1, we want to show that there is another solution *B* that begins with greedy choice, activity 1. Let $B = A - \{k\} \cup \{1\}$. Because f_1 <= f_k, the activities in *B* are disjoint and since B has same number of activities as *A*, i.e., $|A| = |B|$, *B* is also optimal.

Once the greedy choice is made, the problem reduces to finding an optimal solution for the problem. If *A* is an optimal solution to the original problem *S*, then *A' = A - {1}* is an optimal solution to the activity-selection problem $S' = \{i \in S: s_i >= f_i\}$. If we could find a solution *B'* to *S'* with more activities then *A',* adding 1 to *B'* would yield a solution *B* to *S* with more activities than *A*, there by contradicting the optimality.

Linear Partition Problem
Given a list of positive integers, $s_1, s_2, ..., s_N$, and a bound B, find smallest number of contiguous sublists s.t. each sum of each sublist ≤ B. I.e.: find partition points $0 = p_0, p_1, p_2, ..., p_k = N$ such that for j = 0, 1, ..., k–1,

$$\sum_{i=p_j+1}^{p_{j+1}} s_i \le B$$

Greedy algorithm:
Choose p_1 as large as possible. Then choose p_2 as large as possible. Etc.

Greedy is optimal for this Linear Partition Problem
Theorem: Given any valid partition $0 = q_0, q_1,..., q_k = N$, then for all j, $q_j \le p_j$. (The p_i's are greedy solution.)

Proof: (by induction on k).
Base Case: $p_0 = q_0 = 0$ (by definition).
Inductive Step: Assume $q_j \le p_j$.

We know $\sum_{i=q_j+1}^{q_{j+1}} s_i \leq B$ (since q's are valid).

So $\sum_{i=p_j+1}^{q_{j+1}} s_i \leq B$ (since $q_j \leq p_j$).

So $q_{j+1} \leq p_{j+1}$ (since Greedy chooses p_{j+1} to be as large as possible subject to constraint on sum).

Greedy approximation

- In many cases where exact algorithms are difficult to find, or all known ones are inefficient, the greedy approach can provide a simple and effective heuristic, or approximate solution method.
- Usually such greedy heuristics for hard problems are too simple to have good performance, but there are some exceptions.
- As an example, we shall consider a greedy approximation algorithm for the important Set Cover problem.

Greedy approximation: the Set Cover problem

Set Cover Optimization Problem (SC-O): Given a base set *B* and a collection of finite subsets *S* = {*S*1, . . . ,*S*n} of *B*, determine the smallest number of sets in *S*, whose union covers all of *B*.

A candidate solution {*S0*1, . . . ,*S0*k } $\subseteq$ *S* to a given instance of the SC-O problem is feasible if $\bigcap_{i=1}^{k} S_{oi} = B$, and its cost is the number of sets picked, i.e. *k*. An optimal solution is a feasible solution with minimum cost.

The Set Cover problem is NP-complete, so there are not likely to exist any efficient exact solution algorithms for it.

A greedy approximation heuristic

A straightforward greedy heuristic for the Set Cover problem is the following:

```
Repeat until all elements in $B$ are covered:
    Pick set $S_i$ with most uncovered elements
```

This does not always yield optimal solutions. (Example!). However, one can establish the following bound:

Claim. Suppose *B* contains *n* elements and the optimal cover consists of *k* sets. Then the greedy algorithm will pick at most *k* ln *n* sets.

I.e. the greedy algorithm is guaranteed to be always at most a factor of ln *n* worse than the optimum. We say that it has an approximation bound of ln *n*.