**COS 460 – Algorithms (week 04)**

**Design of Algorithms. Divide and Conquer. Greedy Algorithms. Backtracking.**

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.
*Note: Speaking about operating systems, "heap" refers to memory from which chunks can be allocated.*

***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
```

## Example: Closest Pair

**Input:**

```
P  =  {p(1), p(2) ,..., p(n) }
```

where *p(i) = ( x(i), y(i) ).*

A set of *n* points in the plane.

**Output**

The *distance* between the two points that are closest.

**Note:** The distance *DELTA( i, j )* between *p(i)* and *p(j)* is defined by the expression:

```
Square root of { (x(i)-x(j))^2 + (y(i)-y(j))^2 }
```

**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:

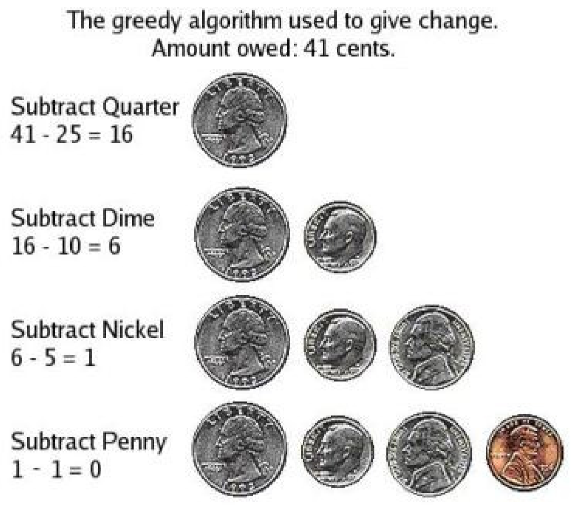
If *n* is even, then $a^n=(a^{n/2})^2$. If *n* is odd, then $a^n=a(a^{n/2})^2$. In either case, we have halved the size of our exponent at the cost of at most two multiplications, so O(log *n*) multiplications suffice to compute the final value.

```
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.

## 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**

To 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 $s_i \le f_i$. 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.

**Spanning Trees:**

A spanning tree of a graph G is a sub graph of G that is a tree and contains all the vertices of G. We construct spanning tree whenever we want to find a simple, cheap and yet efficient way to connect a set of terminals (computers, cites, factories, etc.). Spanning trees are important because of following reasons.

- Spanning trees construct a sparse sub graph that tells a lot about the original graph.
- Spanning trees a very important in designing efficient routing algorithms.
- Some hard problems (e.g., Steiner tree problem and traveling salesman problem) can be solved approximately by using spanning trees.
- Spanning trees have wide applications in many areas, such as network design, etc.

It is easy to see that this algorithm has the property that each edge is examined at most once. Algorithms, like this one, examine each entity at most once and decide its fate once and for all during that examination. The obvious advantage of this greedy approach is that we do not have to spend time reexamining entities.

**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, show 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.

**Greedy algorithm for minimum spanning tree problem:**
A minimum spanning tree (MST) of a weighted graph G is a spanning tree of G whose edges sum is minimum weight. In other words, MST is a tree formed from a subset of the edges in a given undirected graph, with two properties: it includes every vertex of the graph and the total weight of all the edges is as low as possible.

One of the most elegant minimum spanning tree algorithms:

- Examine the edges in graph in order of non-decreasing weight (If two or more weights are equal then order them arbitrarily.)
- Decide whether each edge will be included in the spanning tree.

Note that each time a step of the algorithm is performed, one edge is examined. If there are only a finite number of edges in the graph, the algorithm must halt after a finite number of steps. Thus, the time complexity of this algorithm is clearly O (n), where n is the number of edges in the graph.

**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 linear partition

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

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

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

**BACKTRACKING AND SEARCHING**

Whether a graph is an explicit or implicit structure in describing a problem, it is often the case that *searching* the graph structure may be necessary. Thus it is required to have methods which

- Can *mark* nodes in a graph which have already been *examined.*
- Determine which node should be examined next.
- Ensure that every node in the graph **can** (but not necessarily **will**) be visited.

These requirements must be realized subject to the constraint that the search process respects the structure of the graph.

**Any new node examined must be adjacent to some node that has previously been visited.**

So, *search methods* implicitly describe an **ordering** of the nodes in a given graph.

Suppose a problem may be expressed in terms of detecting a particular class of subgraph in a graph. Then the *backtracking* approach to solving such a problem would be:

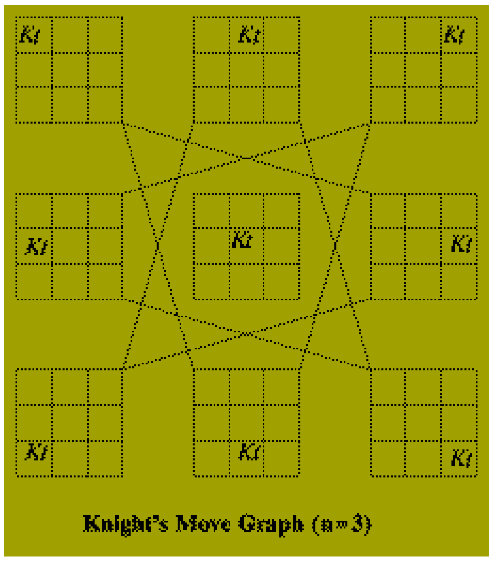
Scan each node of the graph, *following a specific order*, until

- A subgraph constituting a solution has been found **or**
- It is discovered that the subgraph built so far cannot be extended to be a solution.

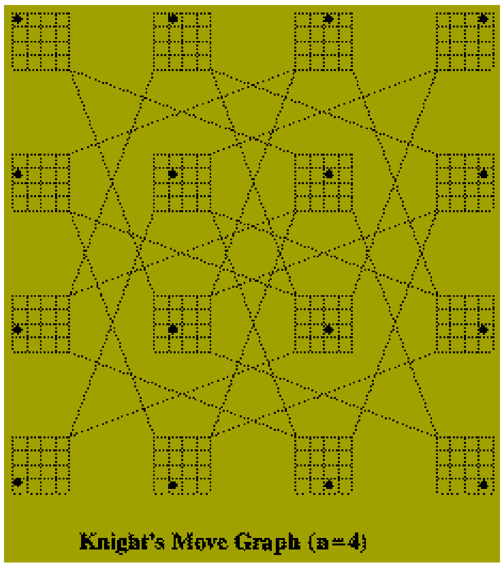
If (2) occurs then the search process is `*backed-up*' until a node is reached from which a solution might still be found.

**Simple Example: Knight's Tour**

Given a natural number, *n*, describe how a Knight should be moved on an *n times n* chessboard so that it visits every square exactly once and ends on its starting square. The *implicit graph* in this problem has $n^2$ nodes corresponding to each position the Knight must occupy. There is an edge between two of these nodes if the corresponding positions are *one move apart*. The sub-graph that defines a solution is a cycle which contains each node of the implicit graph.



Knight's Move Graph (n=3)


Knight's Move Graph (n=4)

**Depth-First Search**

The Knight's Tour algorithm organizes the search of the implicit graph using a **depth-first** approach. Depth-first search is one method of constructing a **search tree** for *explicit graphs*.
Let *G(V,E)* be a connected graph. A **search tree** of *G(V,E)* is a **spanning tree**, *T(V, F)* of *G(V,E)* in which the nodes of *T* are labelled with unique values *k* (*1* < = *k* < = |*V*|) which satisfy:

- A distinguished node called the **root** is labeled *1*.
- If *(p,q)* is an edge of *T* then the label assigned to *p* is less than the label assigned to *q*.

The labeling of a search tree prescribes the *order* in which the nodes of *G* are to be scanned.
Given an undirected graph *G(V,E)*, the depth-first search method constructs a search tree using the following recursive algorithm:

```
procedure depth_first_search (G(V,E) : graph; v : node;
                               lambda : integer; T : in outsearch_tree)
begin
   label(v) := lambda; lambda := lambda+1;
   for each w such that {v,w} in E do
     if label(w) = 0 then
       Add edge {v,w} to T;
       depth_first_search(G(V,E),w,lambda,T);
     end if;
   end loop;
end
begin
   for w in V do label(w) = 0;
   lambda := 1;
   depthfirstsearch ( G(V,E), v, lambda, T);
end;
```

The running time of the algorithm is *O*(|*E*|) since each edge of the graph is examined only once.

**The *N* by *N* Queens Problem**. In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. A chess board has 8 rows and 8 columns. The standard 8 by 8 Queen's problem asks how to place 8 queens on an ordinary chess board so that none of them can hit any other in one move.
Two solutions are not essentially distinct if you can obtain one from another by rotating your chess board, or placing in in front of a mirror, or combining these two operations.
An obvious modification of the 8 by 8 problem is to consider an *N* by *N* "chess board" and ask if one can place *N* queens on such a board. This is impossible if *N* is 2 or 3, and it is reasonably straightforward to find solutions when *N* is 4, 5, 6, or 7. The problem begins to become difficult for manual solution precisely when *N* is 8. Probably the fact that this number coincidentally equals the dimensions of an ordinary chess board has contributed to the popularity of the problem.

**All solutions of the 8 × 8 Queens Puzzle**

**Configuration 1**
```
Q – · – · – · –
– · – · Q · – ·
· – · – · – · Q
– · – · – Q – ·
· – Q – · – · –
– · – · – · Q ·
· Q · – · – · –
– · – Q – · – ·
```

**Configuration 2**
```
Q – · – · – · –
– · – · – Q – ·
· – · – · – · Q
– · Q · – · – ·
· – · – · – Q –
– · – Q – · – ·
· Q · – · – · –
– · – · Q · – ·
```

**Configuration 3**
```
· Q · – · – · –
– · – Q – · – ·
· – · – · Q · –
– · – · – · – Q
· – Q – · – · –
Q · – · – · – ·
· – · – · – Q –
– · – · Q · – ·
```

**Configuration 4**
```
· Q · – · – · –
– · – · Q · – ·
· – · – · – Q –
Q · – · – · – ·
· – Q – · – · –
– · – · – · – Q
· – · – · Q · –
– · – Q – · – ·
```

**Configuration 5**
```
· Q · – · – · –
– · – · Q · – ·
· – · – · – Q –
– · – Q – · – ·
Q – · – · – · –
– · – · – · – Q
· – · – · Q · –
– · Q · – · – ·
```

**Configuration 6**
```
· Q · – · – · –
– · – · – Q – ·
Q – · – · – · –
– · – · – · Q ·
· – · Q · – · –
– · – · – · – Q
· – Q – · – · –
– · – · Q · – ·
```

**Configuration 7**
```
· Q · – · – · –
– · – · – Q – ·
· – · – · – · Q
– · Q · – · – ·
Q – · – · – · –
– · – Q – · – ·
· – · – · – Q –
– · – · Q · – ·
```

**Configuration 8**
```
· Q · – · – · –
– · – · – · Q ·
· – Q – · – · –
– · – · – Q – ·
· – · – · – · Q
– · – · Q · – ·
Q – · – · – · –
– · – Q – · – ·
```

**Configuration 9**
```
· Q · – · – · –
– · – · – · Q ·
· – · – Q – · –
– · – · – · – Q
Q – · – · – · –
– · – Q – · – ·
· – · – · Q · –
– · Q · – · – ·
```

**Configuration 10**
```
· – Q – · – · –
– · – · Q · – ·
· Q · – · – · –
– · – · – · – Q
Q – · – · – · –
– · – · – · Q ·
· – · Q · – · –
– · – · – Q – ·
```

**Configuration 11**
```
· – Q – · – · –
– · – · Q · – ·
· – · – · – · Q
– · – Q – · – ·
Q – · – · – · –
– · – · – · Q ·
· Q · – · – · –
– · – · – Q – ·
```

**Configuration 12**
```
· – Q – · – · –
– · – · – Q – ·
· Q · – · – · –
– · – · Q · – ·
· – · – · – · Q
Q · – · – · – ·
· – · – · – Q –
– · – Q – · – ·
```