- Dijkstra -> stacks and recursion NOTE: Introduction to Algorithms has a more generalized and detailed overview of techniques

Recursion and Trees

- a function that calls itself
- requires a termination condition
- recursion and trees are tightly connected
- mathematical recurrences and recursion in programs are connected as well
- additional items:
  - divide and conquer
  - dynamic programming
  - tree traversal
  - depth-first (and breadth-first) search

Recursive Algorithms

- solves a problem by solving one or more smaller instances of the problem

```
int factorial(int N) {
    if (N == 0) return 1;
    return N*factorial(N-1);
}
```

- (the mathematical concept/definition of) recurrence relations are recursively defined functions
- any recursively defined function can be expressed using loops and any computation using loops can be defined recursively
- Euclid's algorithm

```
int gcd(int m, int n) {
    if (n == 0) return m;
    return gcd(n, m % n);
}
```

- evaluate prefix expressions (example of a recursive descent parser)

```
char *a; int i;
int eval() {
    int x = 0;
    while (a[i] == ' ') {
        i++;
```

```
    }
    if (a[i] == '+') {
        i++;
        return eval() + eval();
    }
    if (a[i] == '*') {
        i++;
        return eval() * eval();
    }
    while ((a[i] >= '0') && (a[i] <= '9')) {
        x = 10 * x + (a[i++] - '0');
    }
    return x;
}
```

- the definition of linked data structures is inherently recursive
- linked list recursive functions:

```
int count(link x) {
    if (x == 0) {
        return 0;
    }
    return 1 + count(x->next);
}
void traverse(link h, void visit(link)) {
    if (h == 0) {
        return;
    }
    visit(h);
    traverse(h->next, visit);
}
void traverseR(link h, void visit(link)) {
    if (h == 0) {
        return;
    }
    traverseR(h->next, visit);
    visit(h);
}
void remove(link& x, Item v) {
    while (x != 0 && x->item == v) {
        link t = x;
        x = x->next;
        delete t;
    }
    if (x != 0) {
        remove(x->next, v);
    }
}
```

Divide and Conquer

- uses two recursive calls, each split to roughly half of the input

- divide and conquer to find the max

```
Item max(Item a[], int l, int r) {
    if (l == r) {
        return a[l];
    }
    int m = (l + r) / 2;
    Item u = max(a, l, m);
    Item v = max(a, m+1, r);
    if (u > v) {
        return u;
    } else {
        return v;
    }
}
```

- Towers of Hanoi:

```
void hanoi(int N, int d) {
    if (N == 0) {
        return;
    }
    hanoi(N-1, -d);
    shift(N, d);
    hanoi(N-1, -d);
}
```

- draw a ruler (recursive divide and conquer):

```
void rule(int l, int r, int h) {
    int m = (l+r)/2;
    if (h > 0) {
        rule(l, m, h-1);
        mark(m, h);
        rule(m, r, h-1);
    }
}
```

- draw a ruler (non-recursive):

```
void rule(int l, int r, int h) {
    for (int t = 1, j = 1; t <= h; j += j, t++) {
        for (int i = 0; l+j+i <= r; i += j+j) {
            mark(l+j+i, t);
        }
```

```
    }
}
```

- the ruler drawing problem can be used to demonstrate a number of relationships
    - n bit numbers
    - effects of preorder vs postorder and bottom-up order
    - fractals
    - combine and conquer
- properties:
    - a recursive function that divides a problem of size n into two independent non-empty parts that it solves recursively calls itself less than n times
    - recursive Towers of Hanoi produces a solution with 2 ^ n - 1 moves
- must be aware of the stack during recursion -> both the size and the elements stored in each frame
- additional related topics:
    - divide into parts of varying size
    - divide into n parts
    - divide into overlapping parts
    - do varying amounts of work in the non-recursive part of the algorithm

Dynamic Programming

- divide and conquer's sub-problems are independent
- when the sub-problems are not independent the solutions become more complicated because direct recursive approaches take excessive time
- bottom-up dynamic programming:
    - mathematical recurrences work with integers recursively
    - these functions can be solved by computing all of the function values in order starting with the smallest and computing the value at each step using values from previous steps
    - (possibly uses an array to store previous values and access them)
- top-down dynamic programming:
    - simpler corollary to bottom-up dynamic programming (at same cost or less cost)
    - the recursive program saves each computed value as its last action and checks for the computed value to avoid re-computation as its first action
    - the running time is reduced to linear running time
    - sometimes called memoization
- dynamic Fibonacci:

```
int F(int i) {
    static int knownF[maxN];
    if (knownF[i] != 0) {
        return knownF[i];
     }
    int t = i;
    if (i < 0) {
        return 0;
    }
    if (i > 1) {
        t = F(i-1) + F(i-2);
    }
    return knownF[i] = t;
}
```

- recursive knapsack:

```
int knap(int cap) {
    int i, space, max, t;
    for (i = 0, max = 0; i < N; i++) {
        if ((space = cap-items[i].size) >= 0) {
            if ((t = knap(space) + items[i].val) > max) {
                max = t;
            }
        }
    }
    return max;
}
```

- non-recursive knapsack:

```
int knap(int M) {
    int i, space, max, maxi = 0, t;
    if (maxKnown[M] != unknown) return maxKnown[M];
    for (i = 0, max = 0; i < N; i++)
        if ((space = M-items[i].size) >= 0)
            if ((t = knap(space) + items[i].val) > max) {
                max = t;
                maxi = i;
            }
        }
    }
    maxKnown[M] = max; itemKnown[M] = items[maxi];
    return max;
}
```

- properties:
    - dynamic programming reduces the running time of a recursive function to be at most the time required to evaluate the function for all arguments less than or equal to the given argument, treating a recursive call as constant

- top-down vs bottom-up
  - top-down -> save known values
  - bottom-up -> precompute known values
  - prefer top-down because:
    - it is a mechanical transformation of a natural problem solution
    - the ordering of sub-problem computation takes care of itself
    - may not need to compute all of the answers to all of the sub-problems

Additional Notes

- See Introduction to Algorithms for a more detailed discussion
- NOTE: dynamic programming is characterized by:
  - the overlap between sub-problems
  - optimal sub-structure (an optimal solution can be constructed efficiently from optimal solutions of its sub-problems)
  - (non-overlapping sub-problems leads to divide and conquer solutions)
- top-down:
  - solve the top of the input tree first in terms of the sub-problems recursively and store solutions to sub-problems in a table (memoization)

```
return_type memoization(p, n) {
    let r[0...n] be a new array
    for (i = 0 to n) {
        r[i] = some invalid value;
    }
    return memoize_aux(p, n, r);
}

return_type memoize_aux(p, n, r) {
    if (r[n] is defined (i.e. greater than 0 or a valid value)) {
        return r[n];
    } else {
        // this is just generic to signal the action of computing smaller values
        // and storing them/returning them
        q = -infinity; // (i.e. invalid value)
        for (i = 1 to n) {
            q = recursively calculate q
        }
        r[n] = q;
        return q;
    }
```

- bottom-up:
  - start by solving sub-problems first and then solve larger and larger problems (using a tabular form (tabulation))

```
return_type bottom_up(p, n) {
    let r[0...n] be a new array
```

```
    r[0] = 0 // base case
    for (j = 1 to n) {
        q = -infinity // (i.e. invalid value)
        for (i = 1 to j) { // working from sub-problems upward
            calculate q
        }
        r[j] = q;
    }
    return r[n]
}
```

Additional Notes

factorial

```
unsigned int factorial(unsigned int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

- possible non-recursive implementation

```
unsigned int factorial(unsigned int n) {
    int t;
    int i;
    for (t = 1, i = 1; i <= n; i++) {
        t *= i;
    }
    return t;
}
```

- better

```
unsigned int factorial(unsigned int n) {
    int a = 1;
    for (int i = n; i > 1; i--) {
        a *= i;
    }
```

```
    return a;
}
```

power

```
double power(double x, unsigned int n) {
    if (n == 0) {
        return 1.0;
    } else {
        return x * power(x, n - 1);
    }
}
```

- non-recursive

```
double power(double x, unsigned int n) {
    double a = 1.0;
    for (int i = n; i > 1; i--) {
        a *= x;
    }
    return a;
}
```

tail recursion

- just a glorified loop

```
void tail(int i) {
    if (i > 0) {
        std::cout << i << " ";
        tail(i - 1);
    }
}
```

- iterative equivalent

```
void iterativeTail(int i) {
    for (; i > 0; i--) {
        std::cout << i << " ";
```

```
    }
}
```

nontail recursion

- the call stack can store information without having to explicitly store it

```
void reverse() {
    char ch;
    std::cin.get(ch);
    if (ch != '\n') {
        reverse();
        std::cout.put(ch);
    }
}
```

- non-recursive

```
void iterativeReverse() {
    char stack[80];

    int ch;
    int i = 0;
    while (ch = (std::cin.get() && ch != '\n')) {
        stack[top] = ch;
        i++;
    }
    for (; i >=0; i--) {
        std::cout.put(stack[i]);
    }
}
```

- with separate stack

```
void nonRecursiveReverse() {
    int ch;
    while (ch = (std::cin.get() && ch != '\n')) {
        stack.push(ch);
    }
    while (!stack.empty()) {
        std::cout.put(stack.pop());
```

```
    }
}
```

indirect recursion

- when one function calls another function that in turn calls the first (at some point in the ensuing call stack)

```
receive() -> decode() -> store() -> receive() -> decode() -> store()
```

nested recursion

- function defined in terms of itself with the additional definition of parameters in terms of original function

backtracking

- recursively travel all possible paths to find the correct solution
- relates to trying all possible permutations
- generically:
    - from starting point
        - test for accept or reject and return if either
        - while solution has not been found
            - recursively try each possibility in current path
            - if no solution, return to starting point
        - if no solution return false
- backtracking is really just depth-first search on an implicit graph (or tree)

```
void backtrack(candidate) {
    if (reject(instance, candidate)) {
        return;
    }
    if (accept(instance, candidate)) {
        return output(instance, candidate);
    }
    element s = first(instance, candidate);
    while (s != null_case) {
        backtrack(s);
        s = next(instance, s)l
    }
```

```
}
```

- recursive

```
boolean solve(Node n) {
    if n is a leaf node {
        if the leaf is a goal node, return true
        else return false
    } else {
        for each child c of n {
            if solve(c) succeeds, return true
        }
        return false
    }
}
```

- non-recursive

```
boolean solve(Node n) {
    put node n on the stack;
    while the stack is not empty {
        if the node at the top of the stack is a leaf {
            if it is a goal node, return true
            else pop it off the stack
        }
        else {
            if the node at the top of the stack has untried children
                push the next untried child onto the stack
            else pop the node off the stack

    }
    return false
}
```

NOTE: review recursive descent parser here

linear recursion

- general outline:
  - if n == 0
    - return constant

    - else return function(n - 1)

binary recursion

- general outline:
    - if n,m == base case:
        - return constant
    - else return function(start, n, n + m / 2) + function(start, n + m / 2, m)

multiple recursion

- generalized from binary recursion
- much like dynamic program, multiple possible configurations,

general design

- test for base case
- recur

eliminating tail recursion

- iterate or explicitly use a stack