**Asymptotic Bounds**

O(g(n)) = f(n): there exist positive constants c and no such that 0 ≤ f1(n) ≤ cg(n) for all n ≥ n0 🡪 g(n) is an upperbound

Ω(g(n)) = f(n): there exist positive constants c and no such that 0 ≤ cg(n) ≤ f1(n) for all n ≥ n0 🡪 g(n) is an lowerbound

Θ(g(n)) = f(n): there exist positive constants c and no such that 0 ≤ c1g(n) ≤ f1(n) ≤ c2g(n) for all n ≥ n0 🡪 g(n) bounds on both sides

*Theta implies O and Ω because it will be an upper and lower bound*

**Limit Method Symmetry Property Muster Method (Decrease and Conquer)**

The limit as n->∞ f(n)/g(n) = If f(n) = Θ(g(n)) then g(n) = Θ(f(n)) Determine a recurrence relation when the number of inputs is decreasing by a

Zero then f(n) = O(g(n)) constant instead of being divided

+constant = Θ(g(n)) If we can put the equation into the form T(n) = aT(n-b) + f(n)

∞ = Ω(g(n)) where a, b > 0 and d > 0 where f(n) = O(nd) or Θ(nd) then:

If a < 1: T(n) = O(nd)

**Master Method (Divide and Conquer)** If a = 1: T(n) = O(nd+1)

Determine the recurrence relation from the provided code If a > 1: T(n) = O(ndan/b)

Look for loops and recurrences as summations/divisions

If we can get the equation in the form T(n) = aT(n/b) = f(n) **Designing a Recurrence**

where a ≥ 1, b > 1 and f(n) > 0; then calculate n ^ logb a Determine what parts need to be fed back to the original alogrithm

Case 1: f(n) = O(n ^ logb a – ε) **MINUS** then T(n) = Θ(n ^ logb a) For binary search for instance, sends half of the array back each time

Case 2: f(n) = Θ(n ^ logb a ) **EQUAL** then T(n) = Θ(n ^ logb a \* lg n) Fastest sort is mergeSort at n lg n

Case 3: f(n) = Ω(n ^ logb a + ε) **PLUS**  Insertion sort runs at n2

If Case 3 then must also prove the regularity condition that a\*f(n/b) ≤ c\*f(n)

for some c, where f(n) is from the original recurrence equation. If it still

passes regularity condition then then T(n) = Θ(f(n))

Be careful that constants are greater than or equal to 1, that f(n) is positive

**void merge(int passedarr[], int low, int mid, int high) Order of Growth Ternary Sort**

int size1 = mid-low+1; 22^n int mid1 = low + (high-low)/3;

int size2 = high-mid; nn int mid2 = (low + 2\*high)/3;

int LeftArr[size1], RightArr[size2]; n! if(mid1 <= mid2) {

3n if(A[mid1] == x)

for (int i = 0; i < size1; i++) n\*2n return mid1;

LeftArr[i] = passedarr[low + i]; 2n  if(A[mid2] == x)

for (int i = 0; i < size2; i++) n3  return mid2;

RightArr[i] = passedarr[mid + 1 + i]; n2 log(n) if(A[mid1] > x)

n2  return ternarySearch(A, low, mid1-1, x);

int idxleft = idxright = 0 n log (n) else if (A[mid2] > x)

int idxmerge = low; n return ternarySearch(A, mid1+1, mid2-1,x)

√n else

while (idxleft < size1 && idxright < size2) log3 n return ternarySearch(A,mid2+1,high,x)

if (LeftArr[idxleft] <= RightArr[idxright]) log2 n

passedarr[idxmerge] = LeftArr[idxleft]; log n

idxleft++; √log n

else log(log(n))

passedarr[idxmerge] = RightArr[idxright]; log(log(log(n))

idxright++; 1

idxmerge++; 1/n

while (idxleft < size1) **insertSort(A[], n)**

passedarr[idxmerge] = LeftArr[idxleft]; for(int i=1; i < count; i++) {

idxleft++; int key = numvec[i];

idxmerge++; int j = i-1;

while((j >= 0) && (numvec[j] > key))

while (idxright < size2) numvec[j+1] = numvec[j];

passedarr[idxmerge] = RightArr[idxright]; j--;

numvec[j+1] = key;

idxright++;

idxmerge++;

**void mergeSort(int numarr[] , int low, int high)**

if (low < high)

int mid = (low+high)/2;

mergeSort(numarr, low, mid);

mergeSort(numarr, mid+1, high);

merge(numarr, low, mid, high);

**Dynamic Programming Fib as DP Recursive [Builds up a memo table] (Run time: Theta(n))**

Works to solve exponential growth rate problems in polynomial time memo = {}

Generally works to solve min/max problems fib(n) {

**Recurrences with overlapping subproblems** if(n in memo) {return memo[n]}

Optimal overall solution requires optimal subproblems if(n <= 1) { f = n }

Use previous subproblems to find solution to future subproblem else {f = fib(n-1) + fib(n-2)

Memoize the results (like in Fibonacci sequence, storing already computed values) memo[n] = f

Steps to DP: return f

* Define subproblems
* Guess part of the solution **Fib as Bottoms Up [Build from Bottom Up] (Run time: Theta(n))**
* Relate subproblem solutions fib = {}
* Recurse + memoize or build a DP bottom-up table fib[0] = 0; fib[1] = 1
* Solve original problem for k = 2 to n

Run time of Recurse+Memo = #subproblems \* time per subproblem fib[k] = fib[k-1] + fib[k-2]

return fib[k]

**Binomial (n,k) Runtime and Space Complexity (Theta(n\*k)) LCS (Longest Common Subsequence) LCS Code**

for i <- 0 to n 0 i = 0 or j = 0 for i = 1 to m && j = 1 to n C[i,0] && C[j,0] = 0

for j <- 0 to min(i,k) c[i,j] = c[i-1, j-1] + 1 x[i] = y[i] for i = 1 to m

if j = 0 or j = 1 max(c[i,j-1], c[i-1, j] otherwise for j = 1 to n

C[i, j] = 1 if (Xi == Yi)

else C[i,j] = C[i-1, j-1] + C[i-1,j] c[i,j] = c[i-1, j-1] + 1

return C[n,k] else c[i,j] = max(c[i,j-1], c[i-1, j]

return c

**Knapsack(Theta(nW))**

B[k-1, w] if wk > w

B[k,w] = max(B[k-1,w], B[k-1, w-wk]+bk) wk ≤ w

**Hotel (traveling a distance looking for optimal stopping points)**

OPT[j] = min(OPT[j], OPT[i] + (aj-ai)2) – Theta(n2)

**Canoe (starting at a point and deciding to change path) RodCutting Coin Change**

C[i] = min(C[k] + R[k][i] for 1 ≤ k ≤ i ≤ n – Theta(n2) let r[0…n] be a new array for (int amt = 1; amt <= amount; amt++)

n = #row[R] r[0] = 0 coinReq[amt] = INF

C[1] = 0 for j = 1…n for(int j = 0; j < coins.length; j++)

for i = 2…n q = INF if(coins[j] <= amt)

min = R[1,i] for i = 1…j coinReq[amt] = Math.min(coinReq[amt-coins[j]+1,

for k = 2…i-1 q = max(q, p[i] + r[j-i]) coinReq[amt]);

if C[k] + R[k,i] < min r[j] = 1

min = C[k] + R[k,i] return r[n] This assumes a second array of coins with denominations

C[i] = min OPT = max(p[i], p[i] +r[j-i] Can also do like Knapsack and create 2-D array

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

**Greedy Algorithms**

Greedy Algorithms are used when you do **not** have overlapping subproblems

Greedy works well on unit time problems with consistent increments (1 unit length, 1 time interval)

Fractional Greedy Knapsack takes the item with the highest value/weight relationship and takes everything it can

For the scheduling problem of unit length time with penalty, order by penalty and then see which problems can be resolved by their deadline…early programs do in time slot i or a previous slot, late programs can slot at the end.

**Palindrome (diagonal 2-D matrix like Canoe) Greedy Properties**

if (input[i] == input[j] Show that you can make a localized optimized choice

T[i][j] = T[i+1][j+1]+2 Show that it creates/leaves an optimal subproblem

else Show that it creates the optimal solution

T[i][j] = max(T[i+1][j], T[i][j-1])

    for (int k = 3; k <= n; ++k)

        for (int i = 0; i < n-k+1 ; ++i)

            int j = i + k - 1;

              if (table[i+1][j-1] && str[i] == str[j])