**Recurrence Relations Running Time** ʘ = Theta

linear/sequential search: T(n)=T(n-1)+c O(n) O = Big oh

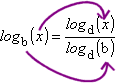
Insertion sort: T(n)=T(n-1)+n-1 O() Ω = Omega

binary search: T(n)=1+log(n) O(log(n)) € = element in

tree traversal: T(n)=2T(n/2)+1 O(n) **Change Base Formula**

insert to max heap: T(n)= O(log(n)) log b(x)=

extract from max heap O(log(n)) log d(x)/log d(b)

heapify: T(n) = ʘ(log(n))

heapsort: T(n) = T(nlog(n)) O (n \* lg (n))

**Recurrence Algorithm Big-Oh Solution**

T(n) = T(n/2) + O(1) Binary Search O(log n)

T(n) = T(n-1) + O(1) Sequential Search O(n)

T(n) = 2 T(n/2) + O(1) tree traversal O(n)

T(n) = T(n-1) + O(n) Selection Sort (other n2 sorts) O(n2)

T(n) = 2 T(n/2) + O(n) Mergesort (average case Quicksort) O(n log n)

**Heap Information**

Heap is excellent for a priority queue.

Min-heap is opposite of max heap

Height of heap is where d is the depth and first node is depth 0 (height)

Heap maximum is ʘ(1) and O(log(n)

Heap property: for every node i, other than the root, A[PARENT(i)] >= A[i]

Parent of I in array A[i] = floor(i/2)

Left child node of i Left[A[i]] = i\*2

Right child node of I right[A[i]] = (i\*2) + 1

Height of heap is: floor(log2n)

**Binary Tree Information**

A full binary tree one in which each node is either a leaf or has degree exactly 2.

**Recursion Tree Information**

height is: floor(base a logn) -1

tree has logn+1 levels

**Asymptotic Bounds**

**Showing Θ:** Method #1 - Find constants n0, c1, and c2, Method #2 - Take limit:

**You have shown f(n) ∈ Θ(g(n)):**If Method #1 leads to finding the 3 constants

If taking the limit in Method #2 leads to a constant c

**You have shown f(n) ∉ Θ(g(n)):** If Method #1 fails to lead to finding the 3 constants

If taking the limit in Method #2 leads to zero or infinity. What makes this a tight asymptotic bound? Because f(n) is sandwiched in between c1g(n) and c2g(n)

**O Notation - Asymptotic Upper Bound - Not Necessarily Tight**

Θ Notation - Asymptotic Tight Bounds

Showing O: Method #1 - Find constants n0, c, Method #2 - Take limit:

**You have shown f(n) ∈ O(g(n)):**If Method #1 leads to finding the 2 constants

If taking the limit in Method #2 leads to a constant c, then you have also shown that g(n) is a tight upper bound, i.e., **f(n) ∈ Θ(g(n))**If taking the limit in Method #2 leads to 0, then you have shown f(n) ∈ o(g(n)), i.e., that g(n) is an upper bound but it is not a tight upper bound **You have shown f(n) ∉ O(g(n)):** If Method #1 fails to lead to finding the 2 constants If taking the limit in Method #2 leads to infinity

**Ω Notation - Asymptotic Lower Bound - Not Necessarily Tight**

Showing Ω:Method #1 - Find constants n0, c, Method #2 - Take limit:

**You have shown f(n) ∈ Ω(g(n)):**If Method #1 leads to finding the 2 constants

If taking the limit in Method #2 leads to a constant c, then you have also shown that g(n) is a tight lower bound, i.e., **f(n) ∈ Θ(g(n))** If taking the limit in Method #2 leads to infinity, then you have shown f(n) ∈ ω(g(n)), i.e., that g(n) is a lower bound but it is not a tight lower bound. **You have shown f(n) ∉ Ω(g(n)):** If Method #1 fails to lead to finding the 2 constants If taking the limit in Method #2 leads to zero

**Stuff from last year**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| http://pages.iu.edu/~jholly/C455/Notes/Chapter3/limitN.gif f(n) / g(n) = | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **Result of Limit** | **What f(n) is in terms of O, Θ, Ω, o, and**ω | | | | | |  | o | O | Θ | Ω | ω | | 0 | f(n) ∈ o(g(n)) |  |  |  |  | | *c* |  | f(n) ∈ O(g(n)) | f(n) ∈ Θ(g(n)) | f(n) ∈ Ω(g(n)) |  | | ∞ |  |  |  |  | f(n) ∈ ω(g(n)) | |

Let f(n) = (4x^3 -4x)/4 and g(n) = x^2 take limit of the ratio of f(n)/g(n)

a: f(n) € ɯ(g(n)) True, b: f(n) Ω (g(n)) false, c: f(n) € ʘ (g(n)) False,

d: f(n) € O(g(n)) False, e: f(n) € o(g(n)) False

Give recurrence: T(n) = 3T(n/2) + sqrt(n)

A:number of subproblems generated: 3

B:size of subproblem: n/2

C:Cost of D(n) + C(n): sqrt(n)

D:Total number of levels in each tree: base a log(n) – 1 (minus because subtree)

E:Total number of activations per level: b^k (no idea what k is)

F:Total number of activations per level: n^(base 2 log(3)) | n^(base b log(a))

G:Cost per level:

H:Cost of the last level: theta(n^(base 2 log(3))) | n^(base b log(a))

**Recursion Tree Example**

Draw the recursion tree for the recurrence: T(n) = 3T(n/4) + c\*n2

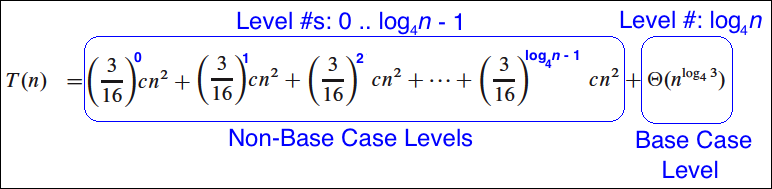
The general recurrence equation has the form:

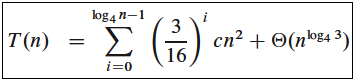
T(n) = Θ(1) if n ≤ c

= a \* T(n/b) + D(n) + C(n) otherwise

In this example, a = 3, b = 1/4, D(n) + C(n) = c\*n2

Big O for Recursive functions: Recurrence relations

Add up all costs over all log4 n levels in the tree:



Rewrite the equation above using summation notation:

**Master Method for Divide and Conquer**

Let: a ≥ 1, b ≥ 1, f(n) be a function

T(n) be the recurrence: T(n) = a \* T(n/b) + f(n)

Then T(n) can be asymptotically bounded by using one of the following cases:

If f(n) = O() for some constant ε > 0, then T(n) ∈ Θ ()

If f(n) = Θ() then T(n) ∈ Θ ( \* lg(n))

If f(n) = Ω() for some constant ε > 0, and if a \* f(n/b) ≤ cf(n) for some constant c < 1 and all sufficiently large n, then T(n) ∈ Θ(f(n))

**Method for Chip & Conquer**

Let: b = 1 (the branching factor), c > 0 (the chipping factor), f(n) be a function

T(n) be the recurrence: T(n) = b \* T(n - c) + f(n)

Then T(n) can be asymptotically bounded as follows:

If f(n) is a polynomial n α, then T(n) ∈ Θ(n α+1)

If f(n) is lg(n), then T(n) ∈ Θ(n \* lg(n))

**Method for Chip & Be Conquered**

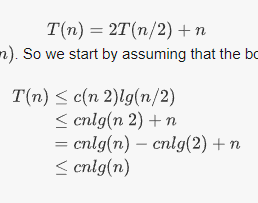
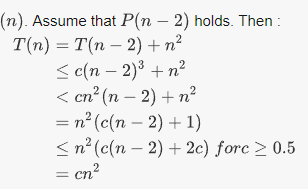
Let: b > 1 (the branching factor), c > 0 (the chipping factor), f(n) be a function

T(n) be the recurrence: T(n) = b \* T(n - c) + f(n)

Then T(n) in most cases can be bounded as T(n) ∈ Θ(b n/c)

**Substitution Method Example**

Use the substitution method to show that

 Show that T(n) = O(n^3)

**Dr H’s stupid Master method**

T(n) = aT(n/b)+f(n)^d, a>1, b>=2, c>0

Then T(n) = {

ʘ (n^d) if a< b^d,

ʘ (n^d logn) if a=b^d

ʘ (n \* base b log(a)) if a>b^d }

**Actual Master Method**

Let a >= 1 and b>1 be constants, let f .n/ be a function, and let T(n) be defined

on the nonnegative integers by the recurrence

T(n)=aT(n/b) + f(n)

where we interpret n=b to mean either floor(n/b) or roof(n/b). Then T(n) has the following asymptotic bounds:

1. if f(n) = O(n^(base b log(a-€))) for some constant €>0, then T(n)= ʘ(n^(base b log a))

2. If f(n) = ʘ(n^(base b log(a))), then T(n)= ʘ(n^(base b log a)ln(n))

3. If f(n) = Ω(n^(base b log(a+€))) for some constant €>0, and if f(n/b) <= cf(n) for

some constant c<1 and all sufficiently large n, then T(n)= ʘ(f(n)).

**Stuff from last year**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| http://pages.iu.edu/~jholly/C455/Notes/Chapter3/limitN.gif f(n) / g(n) = | |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | **Result of Limit** | **What f(n) is in terms of O, Θ, Ω, o, and**ω | | | | | |  | o | O | Θ | Ω | ω | | 0 | f(n) ∈ o(g(n)) |  |  |  |  | | *c* |  | f(n) ∈ O(g(n)) | f(n) ∈ Θ(g(n)) | f(n) ∈ Ω(g(n)) |  | | ∞ |  |  |  |  | f(n) ∈ ω(g(n)) | |

Let f(n) = (4x^3 -4x)/4 and g(n) = x^2 take limit of the ratio of f(n)/g(n)

a: f(n) € ɯ(g(n)) True, b: f(n) Ω (g(n)) false, c: f(n) € ʘ (g(n)) False,

d: f(n) € O(g(n)) False, e: f(n) € o(g(n)) False

Give recurrence: T(n) = 3T(n/2) + sqrt(n)

A:number of subproblems generated: 3

B:size of subproblem: n/2

C:Cost of D(n) + C(n): sqrt(n)

**B: T(n) = 2T(n/2)+n^4**

g(n)=n^2 show f(n) € O(g(n))

0 = f(n) <= g(n) \* c

= 3n^3+4n^2-2 <= c\*n^2

=3n+4-(2/n^2) <= c

N = 1 -> 3+4-2=5 N=2

C=5 Inequality fails C=9.5 Passes

**Given recurrences of the form T(n)=b\*T(n-c)+f(n)**

**For chip and conquer recurrence:**

if f(n) is a polynomial n^alpha, then T(n) € in ʘ(n^(alpha+1))

if f(n) is lg(n), then T(n) € ʘ(nlogn(n))

**For chip and be conquer recurrences:**

Then T(n) in most cases can be bounded as T(n) € ʘ(n^(n/c))