**Recurrence Relations Running Time**

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

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

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

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

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

extract from max heap O(log(n))

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.

Minimum height is: floor(log2n)

**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

**Substitution Method Information**

**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:

E:Total number of activations per level:

**Mathematical induction**

P(n) is a propositional function over positive integers and we want to prove ∀n: n ≥ 1 P(n)

To prove this using induction we must show:

P(1) is true

P(k) → P(k+1)

In other words [P(1) ^ (∀k: P(k) → P(k+1))] → ∀n: P(n)

We need to show that P(k+1) cannot be false when P(k) is true

This is accomplished by assuming P(k) to be true and then using this hypothesis (i.e., that P(k) is true) to show that P(k+1) must also be true.

If it turns out that we cannot show that P(k+1) is true when P(k) is assumed to be true, then there are two possibilities:

We just are not clever enough to prove what needs to be proved; or

The entire proof fails because it is not possible to show tat P(k+1) holds.

Example: Use in induction to prove that the sum of the 1st n odd positive integers = n2

P(n):

Base Step: P(1)

n = 1 then:

(2 \* 1) - 1 = 12

2 - 1 = 1

1 = 1

Inductive Step: P(k) -> P(k+1)

Inductive Hypothesis (IH): assume P(k) is true, then under this assumption show that P(k+1) must also be true.

What does it mean to assume P(k) is true?

By plugging k into the summation (above), it expands to: 1 + 3 + 5 + ... + 2k - 1 = k2

This is what we get to assume is true, this is our hypothesis.

Now show P(k+1) is true, i.e., show that the following equation holds:

1 + 3 + ... + 2k - 1 + 2k + 1 = (k+1)2

Start with the left hand side of the previous equation (highlighted in yellow below):

1 + 3 + ... + 2k - 1 + 2k + 1 = (1 + 3 + ... + 2k - 1) + 2k + 1 (apply Associative law to introduce parenthesis)

= k2 + 2k + 1 (apply IH - replace part separated by parenthesis)

= (k+1)2 (Use Factoring)

**Big O for Recursive functions: Recurrence relations**

In part A students are asked to write the function ValsLess:

struct Tree

{

int info;

Tree \* left;

Tree \* right;

Tree(int value, Tree \* lchild, Tree \* rchild)

: info(value), left(lchild), right(rchild)

{ }

};

bool ValsLess(Tree \* t, int val)

bool IsBST(Tree \* t)

// postcondition: returns true if t represents a binary search

// tree containing no duplicate values;

// otherwise, returns false.

{

if (t == NULL) return true; // empty tree is a search tree

return ValsLess(t->left,t->info) &&

ValsGreater(t->right,t->info) &&

IsBST(t->left) &&

IsBST(t->right);

}

void DoStuff(apvector<int> & a, int left,int right)

// postcondition: a[left] <= ... <= a[right]

{

int mid = (left+right)/2;

if (left < right)

{

DoStuff(a,left,mid);

DoStuff(a,mid+1,right);

Combine(a,left,mid,right);

}

}

T(n) = 2 T(n/2) + O(n) [the O(n) is for Combine]

T(1) = O(1)

T(n) = 2 T(n/2) + n = 2 [2 T(n/4) + n/2] + n = 4 T(n/4) + 2n = 4 [2 T(n/8) + n/4] + 2n = 8 T(n/8) + 3n = 16 T(n/16) + 4n = 2k T(n/2k) + k n

n/2k = 1 OR n = 2k OR log2 n = k

Continuing with the previous derivation we get the following since k = log2 n:

= 2k T(n/2k) + k n = 2log2 n T(1) + (log2n) n = n + n log2 n [remember that T(1) = 1] = O(n log n)

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

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

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

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

T(n) = 2 T(n/2) + O(n)

For an n-element vector a the call FindKth(a,0,n-1,k) returns the kth element in a:

int FindKth(const apvector<int> & a, int left, int right)

// post: return the k-th element in a

{

int pivot = Partition(a,left,right)

if (pivot == k) return a[k];

else if (k < pivot) return FindKth(a, left, pivot-1, k);

else return FindKth(a,pivot+1, right, k);

}

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

Where the O(n) term comes from Partition. Note that there is only one recursive call made in FindKth.

For an n-element vector a the call FindKth(a,0,n-1,k) returns the kth element in a:

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:

int FindKth(const apvector<int> & a, int left, int right)

// post: return the k-th element in a

{

int pivot = Partition(a,left,right)

if (pivot == k) return a[k];

else if (k < pivot) return FindKth(a, left, pivot-1, k);

else return FindKth(a,pivot+1, right, k);

}

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

Where the O(n) term comes from Partition. Note that there is only one recursive call made in FindKth.

**Substitution method**

Use the substitution method to show that T(n) ∈ O(n lg n)

where T(n) = 1 when n = 1

2T(⌊ n/2 ⌋) + n when n > 1

By definition of Big-O, what we have to prove is: T(n) ≤ c \* n lg n

Part 1) Write the induction hypothesis based on what is to be shown:

For this example proof, the IH is: TIH(x) ≤ c\* x \* lg(x)

In this inequality, we use x for the variable so as not to confuse it with n in the recurrence equation

Because this is Big-O, which means f(n) is bounded from the top by g(n), we must use ≤ as the inequality in the IH

Part 2) Begin the proof by doing the first 3 Steps:

Note: this is a mechanical process that works the same for all Big-O proofs

Step 1) T(n) = 2T(⌊n⁄2⌋) + n - Rewrite the given recurrence equation for T(n) when not at the base case, in this case when n >1

Step 2) ≤ 2TIH(⌊n⁄2⌋) + n - Change the equality to an inequality (i.e., change = to ≤) and substitute TIH into the inequality (i.e., TIH(x) for T(n)) and do substitution of ⌊n⁄2⌋ for x

Step 3) = 2(c\*⌊n⁄2⌋lg⁡(⌊n⁄2⌋)) + n - Expand TIH(x) to c\*x\*lg(⁡x), while at same time letting x = ⌊n⁄2⌋

At this point the proof is setup and ready for the (usually) more difficult Part 3 (below)

Part 3) Use algebra and properties of inequalities and logs to manipulate the right hand side from Part 2, Step 3

= 2(c \* ⌊n/2 ⌋ lg(⌊ n/2 ⌋)) + n (1) This line is from Part 2, Step 3 (above), start Part 3 here

≤ 2(c \* n/2 lg(n/2)) + n (2) Simplify (1) above by using the property of floor function: ⌊x⌋ ≤ x

this allows us to eliminate ⌊ ⌋ (i.e., floor functions)

Notice that we're saying equation (line 1) ≤ (line 2), we're not saying (line 1) = (line 2).

This is because of the ≤ in the floor function property: ⌊x⌋ ≤ x

= c \* n lg(n/2) + n (3) Simplify (2) to get (3)

= c \* n lg(n) + c \* n lg(1/2) + n (4) Simplify (3) by using log property: lg(a\*b) = lg(a) + lg(b)

From line (3) a = n, and b = 1/2

= c \* n lg(n) - c \* n lg(2) + n (5) Simplify (4) by using log property: lg(1/a) = -lg(a)

= c \* n lg(n) - c \* n + n (6) Simplify from (5): lg(2) = 1

Now we're close to showing T(n) ≤ c \* n lg n, which is what we're out to prove

≤ c \* n lg n (7) Now we can claim line (6) ≤ (7), but we have to prove it

We prove it by finding the constant 'c' where the inequality holds: (6) ≤ (7) (see below)

The proof fails if that 'c' cannot be found

For what constant 'c' is line (6) <= (7)?

Write those two lines (lines 6 and 7) down as follows, then do algebra to solve for the constant 'c':

c \* n lg(n) - cn + n ≤ c \* n lg(n)

-cn + n ≤ 0

-cn ≤ -n

c ≥ 1

By definition of Big-O, 'c' is required > 0, i.e., because we're working in the Quadrant I

Part 4) Find n0

To do this part evaluate the recurrence and the closed form solution of T(n), call them TR(n) for the recurrence and TC(n) for the closed form solution. Make sure that TR(n) ≤ TC(n)

Try n0 = 1 Try n0 = 2

TR(n) ≤ TC(n)

TR(1) ≤ TC(1)

1 ≤ c(1)lg(1)

1 ≤ c \* 0

fails for n0 = 1 TR(n) ≤ TC(n)

TR(2) ≤ TC(2)

2\*T(⌊ 2/2 ⌋) + 2 ≤ c(2)lg(2)

2\*T(1) + 2 ≤ 2c

4 ≤ 2c

succeeds when c ≥ 2 and n0 = 2

Finally

What we've shown is that TR(n) ≤ TC(n) for all n ≥ n0, and c ≥ 2

So TR(n) ∈ O(n lg n)

**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)