Design & Analysis of Algorithms

Lecture 19

Recursion

Recursion

- Given an array L, find max among numbers between position `start` and `end` (inclusive)

```
findmax (L, start, end) {
    if (start == end)
        return  L[start]
    else {
        mid = ⌊(start+end)/2⌋
        x = findmax(L,start,mid)
        y = findmax(L,mid+1,end)
        if (x>y) return x
        else return y
    }
}
```

Recursion

- Given an array L, find max among numbers between position `start` and `end` (inclusive)

```
findmax (L, start, end) {
    if (start == end)
        return  L[start]
    else {
        mid = ⌊(start+end)/2⌋
        x = findmax(L,start,mid)
        y = findmax(L,mid+1,end)
        if (x>y) return x
        else return y
    }
}
```

Time T(n) taken by `findmax(L,a,a+n-1)`?

Recursion

- Given an array L, find max among numbers between position `start` and `end` (inclusive)

```
findmax (L, start, end) {
    if (start == end)
        return  L[start]
    else {
        mid = ⌊(start+end)/2⌋
        x = findmax(L,start,mid)
        y = findmax(L,mid+1,end)
        if (x>y) return x
        else return y
    }
}
```

Time T(n) taken by `findmax(L,a,a+n-1)`?

$T(1) = c_1$

Recursion

- Given an array L, find max among numbers between position `start` and `end` (inclusive)

```
findmax (L, start, end) {
    if (start == end)
        return  L[start]
    else {
        mid = ⌊(start+end)/2⌋
        x = findmax(L,start,mid)
        y = findmax(L,mid+1,end)
        if (x>y) return x
        else return y
    }
}
```

Time T(n) taken by `findmax(L,a,a+n-1)`?

$T(1) = c_1$

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_2$

Recursion

- Given an array L, find max among numbers between position `start` and `end` (inclusive)

```
findmax (L, start, end) {
    if (start == end)
        return  L[start]
    else {
        mid = ⌊(start+end)/2⌋
        x = findmax(L,start,mid)
        y = findmax(L,mid+1,end)
        if (x>y) return x
        else return y
    }
}
```

Time T(n) taken by `findmax(L,a,a+n-1)`?

$T(1) = c_1$

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_2$

Binary recursion tree with c_2 on each internal node. c_1 at leaves.

Recursion

- Given an array L, find max among numbers between position `start` and `end` (inclusive)

```
findmax (L, start, end) {
    if (start == end)
        return  L[start]
    else {
        mid = ⌊(start+end)/2⌋
        x = findmax(L,start,mid)
        y = findmax(L,mid+1,end)
        if (x>y) return x
        else return y
    }
}
```

Time T(n) taken by `findmax(L,a,a+n-1)`?

$T(1) = c_1$

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_2$

Binary recursion tree with c_2 on each internal node. c_1 at leaves.

$T(n) = O(\text{number of nodes})$

Recursion

- Given an array L, find max among numbers between position `start` and `end` (inclusive)

```
findmax (L, start, end) {
    if (start == end)
        return  L[start]
    else {
        mid =  ⌊(start+end)/2⌋
        x = findmax(L,start,mid)
        y = findmax(L,mid+1,end)
        if (x>y) return x
        else return y
    }
}
```

Time T(n) taken by `findmax(L,a,a+n-1)`?

$T(1) = c_1$

$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c_2$

Binary recursion tree with c_2 on each internal node. c_1 at leaves.

$T(n) = O(\text{number of nodes})$

$T(n) = O(n)$

Question

- Time taken by find3max(L,a,a+n) is

  A. $\Theta(n)$
  B. $\Theta(n \log n)$
  C. $\Theta(n^2)$
  D. $\Theta(n^3)$
  E. None of the above

```
find3max (L, st, en) {
    if (st == en)
        return  L[st]
    else {
        mid1 = st + ⌊(en-st+1)/3⌋
        mid2 = st + 2* ⌊(en-st+1)/3⌋
        x = find3max(L,st,mid1)
        y = find3max(L,mid1+1,mid2)
        z = find3max(L,mid2+1,en)
        if (x≥y ∧ x≥z) return x
        if (y≥x ∧ y≥z) return y
        if (z≥x ∧ z≥y) return z
    }
}
```

Merging Two Sorted Lists

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Merging Two Sorted Lists

- Maintain the invariant that a list K has a prefix of the final merged list. X_1, X_2 have the rest of L_1, L_2.

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Merging Two Sorted Lists

- Maintain the invariant that a list K has a prefix of the final merged list. X_1, X_2 have the rest of L_1, L_2.
  - Base case: K=empty, X_1=L_1, X_2=L_2

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Merging Two Sorted Lists

- Maintain the invariant that a list K has a prefix of the final merged list. X_1, X_2 have the rest of L_1, L_2.
  - Base case: K=empty, X_1=L_1, X_2=L_2
  - Inductively, move the smaller of first(X_1) and first(X_2) to the end of K

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Merging Two Sorted Lists

- Maintain the invariant that a list K has a prefix of the final merged list. X_1, X_2 have the rest of L_1, L_2.
  - Base case: K=empty, X_1=L_1, X_2=L_2
  - Inductively, move the smaller of first(X_1) and first(X_2) to the end of K
  - Terminating condition: Both X_1 and X_2 are empty

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Merging Two Sorted Lists

- Maintain the invariant that a list K has a prefix of the final merged list. X_1, X_2 have the rest of L_1, L_2.
  - Base case: K=empty, $X_1=L_1$, $X_2=L_2$
  - Inductively, move the smaller of first(X_1) and first(X_2) to the end of K
  - Terminating condition: Both X_1 and X_2 are empty
- Time taken (as a function of $n = |L_1|+|L_2|$) ?

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Merging Two Sorted Lists

- Maintain the invariant that a list K has a prefix of the final merged list. X_1, X_2 have the rest of L_1, L_2.
  - Base case: K=empty, $X_1=L_1$, $X_2=L_2$
  - Inductively, move the smaller of first(X_1) and first(X_2) to the end of K
  - Terminating condition: Both X_1 and X_2 are empty
- Time taken (as a function of $n = |L_1|+|L_2|$) ?
  - When finished K has n elements

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Merging Two Sorted Lists

- Maintain the invariant that a list K has a prefix of the final merged list. X_1, X_2 have the rest of L_1, L_2.
  - Base case: K=empty, X_1=L_1, X_2=L_2
  - Inductively, move the smaller of first(X_1) and first(X_2) to the end of K
  - Terminating condition: Both X_1 and X_2 are empty
- Time taken (as a function of $n = |L_1|+|L_2|$) ?
  - When finished K has n elements
  - Each element gets added to K exactly once

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Merging Two Sorted Lists

- Maintain the invariant that a list K has a prefix of the final merged list. X_1, X_2 have the rest of L_1, L_2.
  - Base case: K=empty, $X_1=L_1$, $X_2=L_2$
  - Inductively, move the smaller of first(X_1) and first(X_2) to the end of K
  - Terminating condition: Both X_1 and X_2 are empty
- Time taken (as a function of $n = |L_1|+|L_2|$) ?
  - When finished K has n elements
  - Each element gets added to K exactly once
  - Each iteration adds exactly one element to K (in O(1) time)

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Merging Two Sorted Lists

- Maintain the invariant that a list K has a prefix of the final merged list. X_1, X_2 have the rest of L_1, L_2.
  - Base case: K=empty, $X_1=L_1$, $X_2=L_2$
  - Inductively, move the smaller of first(X_1) and first(X_2) to the end of K
  - Terminating condition: Both X_1 and X_2 are empty
- Time taken (as a function of $n = |L_1|+|L_2|$) ?
  - When finished K has n elements
  - Each element gets added to K exactly once
  - Each iteration adds exactly one element to K (in O(1) time)
  - T(n) = O(n)

```
merge (L1, L2 : ascending lists) {
    K = empty-list; X1 = L1; X2 = L2;
    while (X1 not empty or X2 not empty) {
        if (X2 empty)
            x = pop(X1)
        else if (X1 empty)
            x = pop(X2)
        else if ( first(X1) ≤ first(X2) )
            x = pop(X1)
        else
            x = pop(X2)
        append(K,x)
    }
    return K
}
```

Graph Reachability

```
reachable (G: graph; s,t: nodes in G) {
    unmark all nodes in G
    M = empty-list
    mark s; insert(M, s)
    while (M not empty) {
        x := pop(M)
        if (x=t) return true
        for each neighbor y of x
            if (y unmarked) {
                mark y; insert(M, y)
            }
    }
    return false
}
```

Graph Reachability

- Check if s, t connected

```
reachable (G: graph; s,t: nodes in G) {
    unmark all nodes in G
    M = empty-list
    mark s; insert(M, s)
    while (M not empty) {
        x := pop(M)
        if (x=t) return true
        for each neighbor y of x
            if (y unmarked) {
                mark y; insert(M, y)
            }
    }
    return false
}
```

Graph Reachability

- Check if s, t connected
- Explore graph starting from s, without getting into an infinite loop

```
reachable (G: graph; s,t: nodes in G) {
    unmark all nodes in G
    M = empty-list
    mark s; insert(M, s)
    while (M not empty) {
        x := pop(M)
        if (x=t) return true
        for each neighbor y of x
            if (y unmarked) {
                mark y; insert(M, y)
            }
    }
    return false
}
```

Graph Reachability

- Check if s, t connected
- Explore graph starting from s, without getting into an infinite loop
  - Mark each element already reached

```
reachable (G: graph; s,t: nodes in G) {
    unmark all nodes in G
    M = empty-list
    mark s; insert(M, s)
    while (M not empty) {
        x := pop(M)
        if (x=t) return true
        for each neighbor y of x
            if (y unmarked) {
                mark y; insert(M, y)
            }
    }
    return false
}
```

Graph Reachability

- Check if s, t connected
- Explore graph starting from s, without getting into an infinite loop
  - Mark each element already reached
  - If t marked, then reachable from s

```
reachable (G: graph; s,t: nodes in G) {
    unmark all nodes in G
    M = empty-list
    mark s; insert(M, s)
    while (M not empty) {
        x := pop(M)
        if (x=t) return true
        for each neighbor y of x
            if (y unmarked) {
                mark y; insert(M, y)
            }
    }
    return false
}
```

Graph Reachability

- Check if s, t connected
- Explore graph starting from s, without getting into an infinite loop
  - Mark each element already reached
  - If t marked, then reachable from s
  - Time taken?

```
reachable (G: graph; s,t: nodes in G) {
    unmark all nodes in G
    M = empty-list
    mark s; insert(M, s)
    while (M not empty) {
        x := pop(M)
        if (x=t) return true
        for each neighbor y of x
            if (y unmarked) {
                mark y; insert(M, y)
            }
    }
    return false
}
```

Graph Reachability

- Check if s, t connected
- Explore graph starting from s, without getting into an infinite loop
  - Mark each element already reached
  - If t marked, then reachable from s
  - Time taken?
- Each edge {x,y} inspected at most twice (As each node is inserted/popped at most once)

```
reachable (G: graph; s,t: nodes in G) {
    unmark all nodes in G
    M = empty-list
    mark s; insert(M, s)
    while (M not empty) {
        x := pop(M)
        if (x=t) return true
        for each neighbor y of x
            if (y unmarked) {
                mark y; insert(M, y)
            }
    }
    return false
}
```

Graph Reachability

- Check if s, t connected
- Explore graph starting from s, without getting into an infinite loop
  - Mark each element already reached
  - If t marked, then reachable from s
  - Time taken?
- Each edge {x,y} inspected at most twice (As each node is inserted/popped at most once)
- T(|E|) = O(|E|)

```
reachable (G: graph; s,t: nodes in G) {
    unmark all nodes in G
    M = empty-list
    mark s; insert(M, s)
    while (M not empty) {
        x := pop(M)
        if (x=t) return true
        for each neighbor y of x
            if (y unmarked) {
                mark y; insert(M, y)
            }
    }
    return false
}
```

Binary Search

Binary Search

- Find where a desired object occurs (if at all) in a sorted list of objects

Binary Search

- Find where a desired object occurs (if at all) in a sorted list of objects
  - Objects can be compared with each other (using a total ordering)

Binary Search

- Find where a desired object occurs (if at all) in a sorted list of objects
  - Objects can be compared with each other (using a total ordering)
- Simple idea:

Binary Search

- Find where a desired object occurs (if at all) in a sorted list of objects
  - Objects can be compared with each other (using a total ordering)
- Simple idea:
  - Check if desired object = middle one in the list

Binary Search

- Find where a desired object occurs (if at all) in a sorted list of objects
  - Objects can be compared with each other (using a total ordering)
- Simple idea:
  - Check if desired object = middle one in the list
  - If not, comparing with the middle one lets you see if it could be in the left half or the right half of the list (since the list is sorted)

Binary Search

- Find where a desired object occurs (if at all) in a sorted list of objects
  - Objects can be compared with each other (using a total ordering)
- Simple idea:
  - Check if desired object = middle one in the list
  - If not, comparing with the middle one lets you see if it could be in the left half or the right half of the list (since the list is sorted)
  - Recursively search in that half

Binary Search

- Find where a desired object occurs (if at all) in a sorted list of objects
  - Objects can be compared with each other (using a total ordering)
- Simple idea:
  - Check if desired object = middle one in the list
  - If not, comparing with the middle one lets you see if it could be in the left half or the right half of the list (since the list is sorted)
  - Recursively search in that half
  - Depth of recursion, for an n element list $\leq \lceil \log_2 n \rceil$

Binary Search

Binary Search

- Zeroing in on the answer by shrinking the range by half each time

Binary Search

- Zeroing in on the answer by shrinking the range by half each time

Binary Search

- Zeroing in on the answer by shrinking the range by half each time

Binary Search

- Zeroing in on the answer by shrinking the range by half each time

Binary Search

- Zeroing in on the answer by shrinking the range by half each time

Binary Search

- Zeroing in on the answer by shrinking the range by half each time

Binary Search

- Zeroing in on the answer by shrinking the range by half each time
- Traversing an implicit binary tree

Binary Search

- Zeroing in on the answer by shrinking the range by half each time
- Traversing an implicit binary tree
- Nodes contain the mid-elements of the range under them

Binary Search

- Zeroing in on the answer by shrinking the range by half each time
- Traversing an implicit binary tree
- Nodes contain the mid-elements of the range under them

Binary Search

- Zeroing in on the answer by shrinking the range by half each time

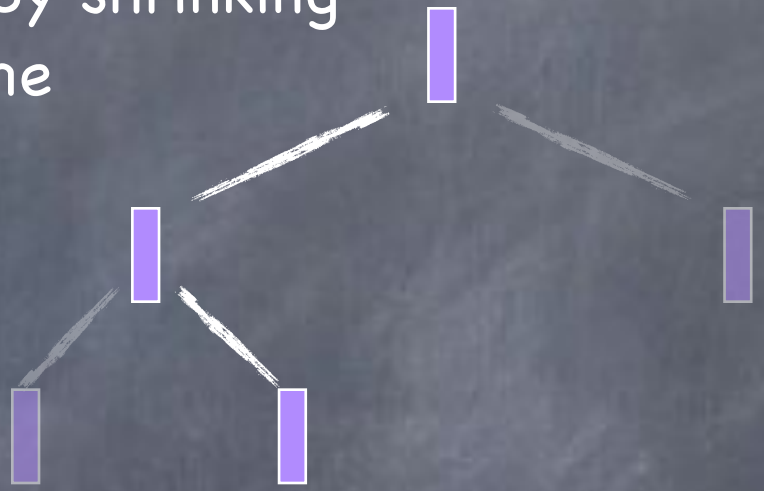
- Traversing an implicit binary tree
- Nodes contain the mid-elements of the range under them
- At each node compare the desired object with the object at the node

Binary Search

- Zeroing in on the answer by shrinking the range by half each time
- Traversing an implicit binary tree
- Nodes contain the mid-elements of the range under them
- At each node compare the desired object with the object at the node

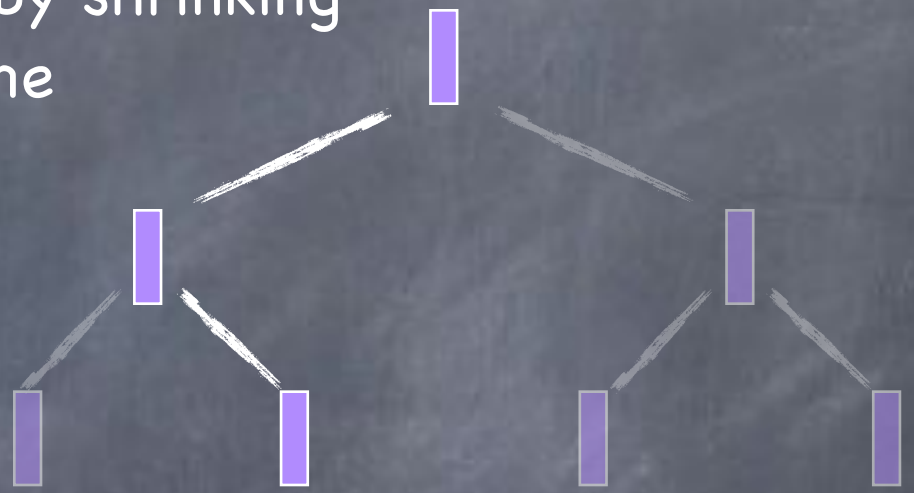
Binary Search

- Zeroing in on the answer by shrinking the range by half each time
- Traversing an implicit binary tree
- Nodes contain the mid-elements of the range under them
- At each node compare the desired object with the object at the node



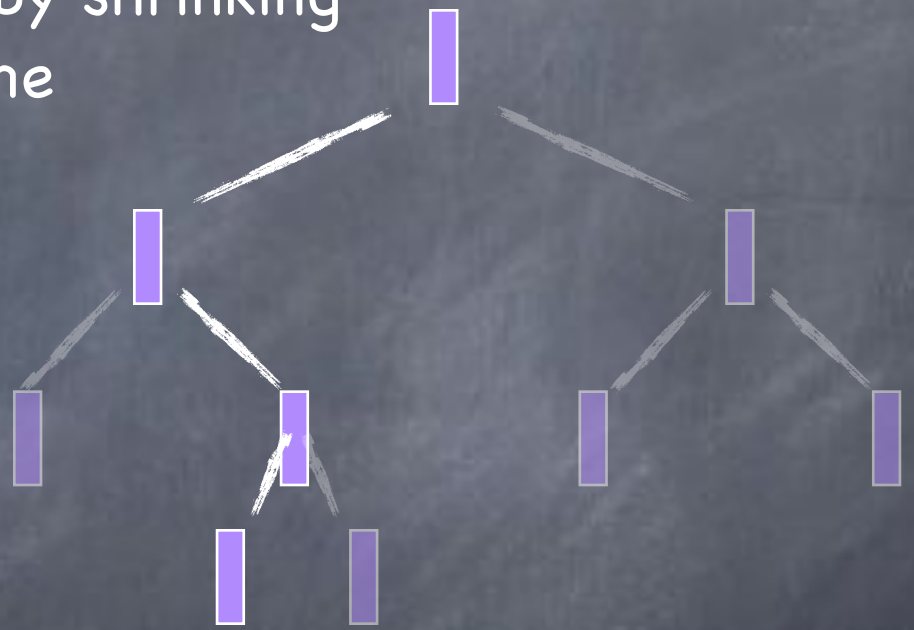
Binary Search

- Zeroing in on the answer by shrinking the range by half each time
- Traversing an implicit binary tree
- Nodes contain the mid-elements of the range under them
- At each node compare the desired object with the object at the node



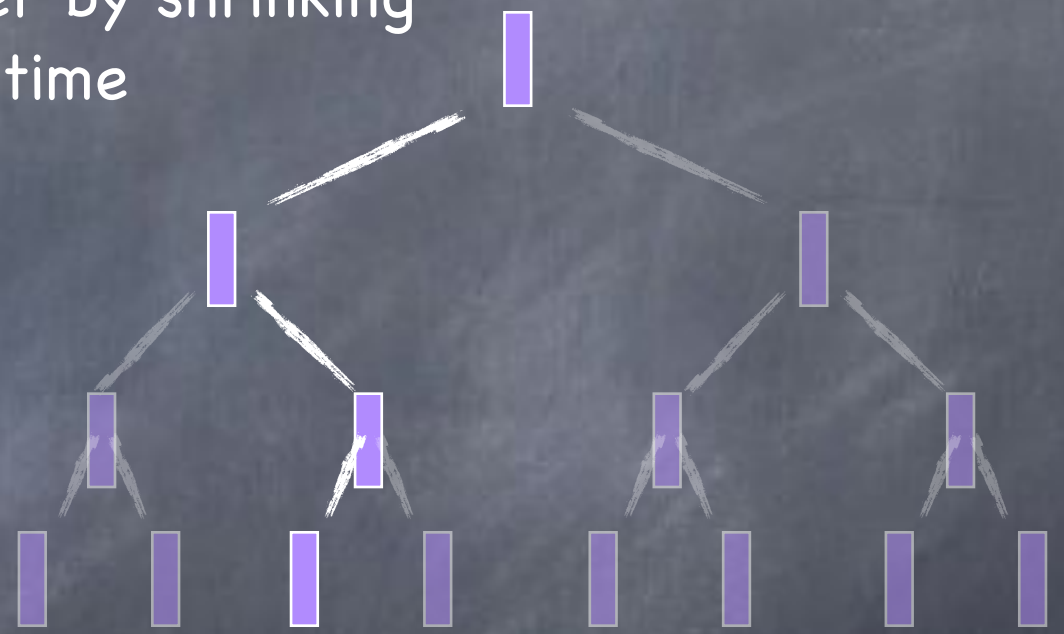
Binary Search

- Zeroing in on the answer by shrinking the range by half each time
- Traversing an implicit binary tree
- Nodes contain the mid-elements of the range under them
- At each node compare the desired object with the object at the node



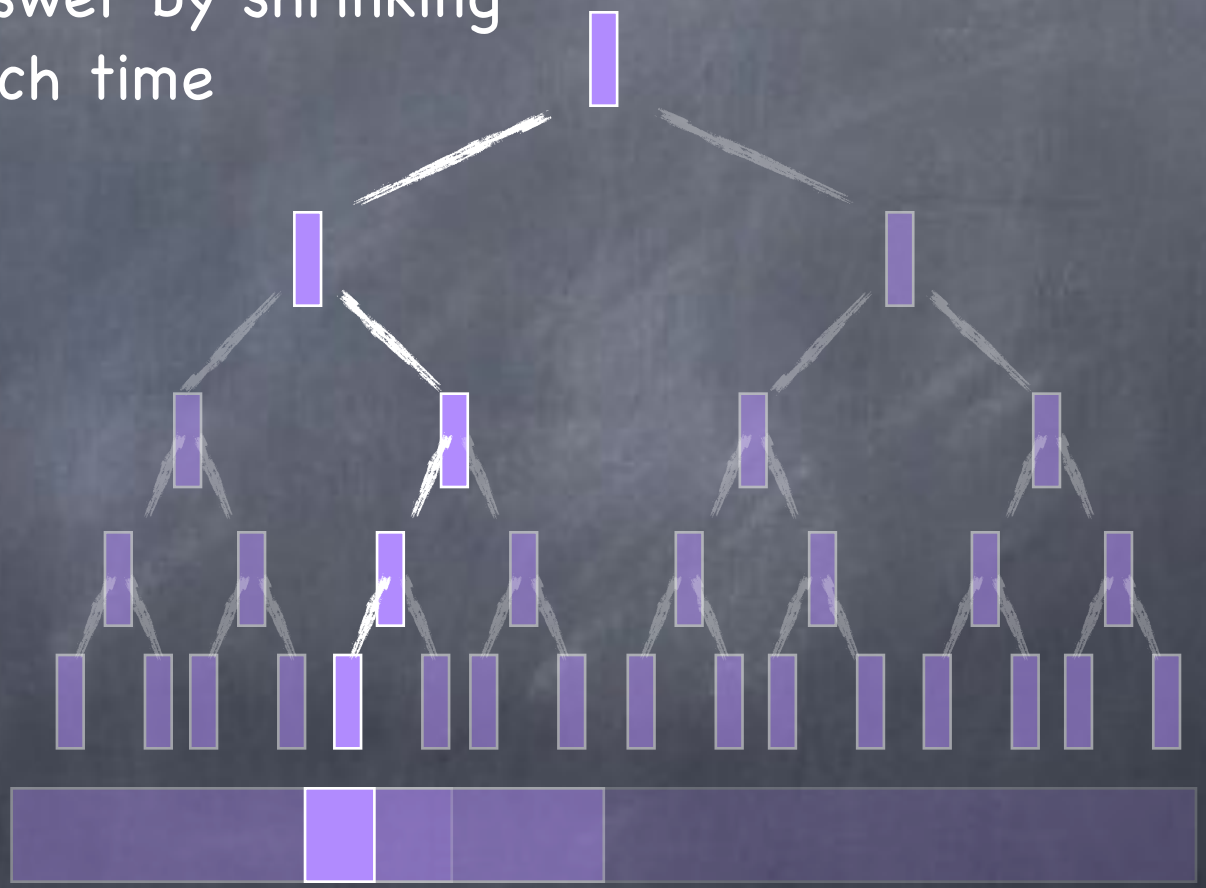
Binary Search

- Zeroing in on the answer by shrinking the range by half each time
- Traversing an implicit binary tree
- Nodes contain the mid-elements of the range under them
- At each node compare the desired object with the object at the node



Binary Search

- Zeroing in on the answer by shrinking the range by half each time
- Traversing an implicit binary tree
- Nodes contain the mid-elements of the range under them
- At each node compare the desired object with the object at the node



Binary Search

Binary Search

- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)

Binary Search

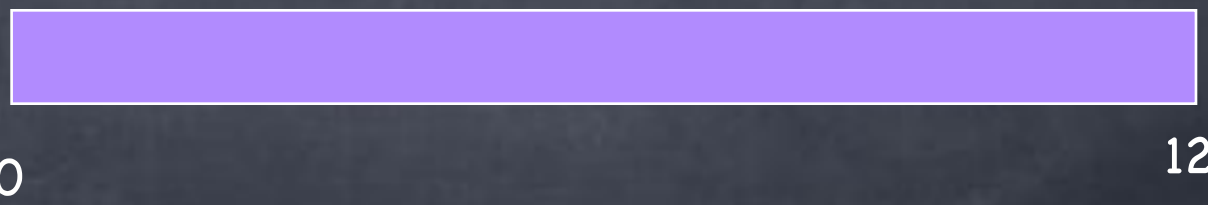
- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)

Binary Search

- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)

Binary Search

- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)

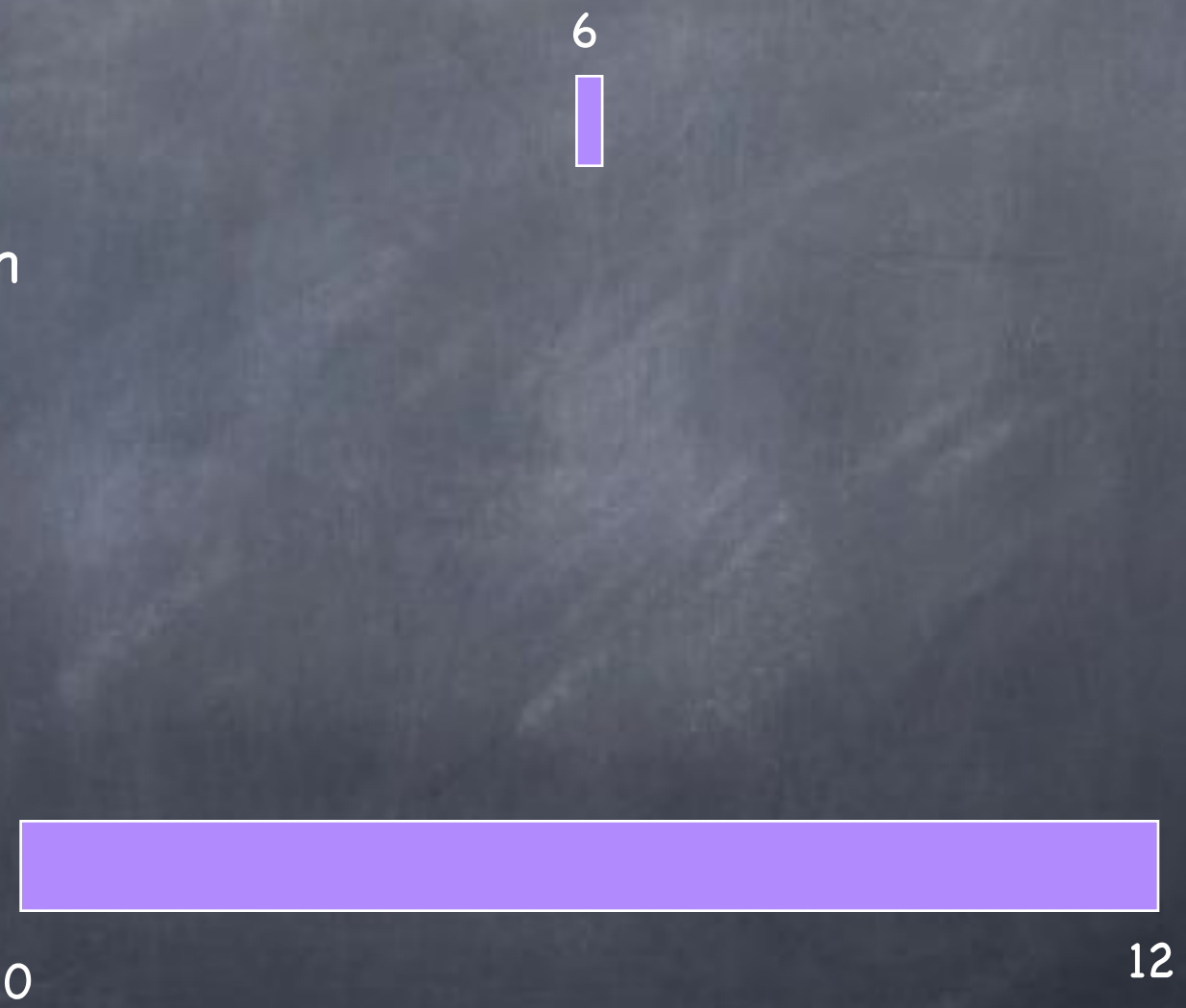


Binary Search

- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)
- How to compare desired object (here √n) with middle element m?

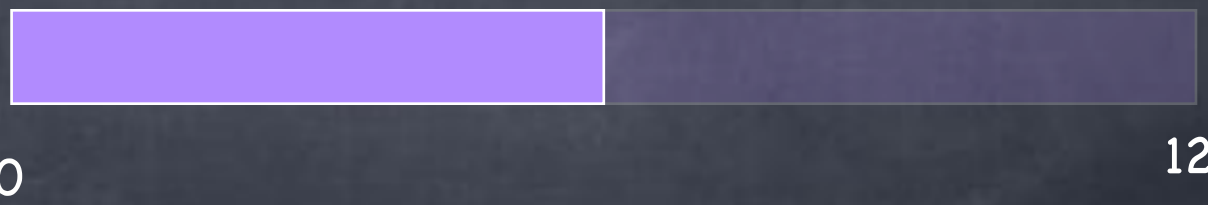
Binary Search

- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)
- How to compare desired object (here $\sqrt{n}$) with middle element m?
  - compare n and m^2



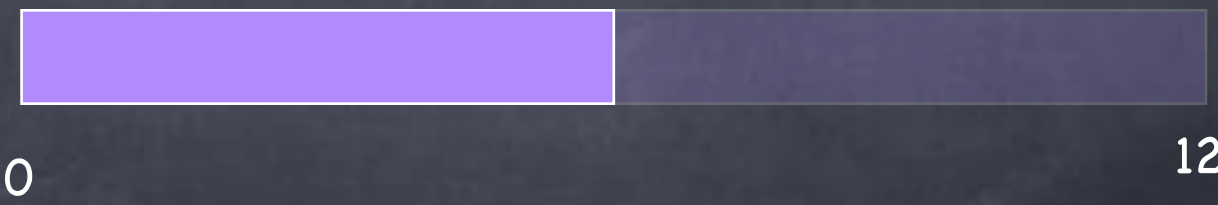
Binary Search

- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)
- How to compare desired object (here $\sqrt{n}$) with middle element m?
  - compare n and m^2



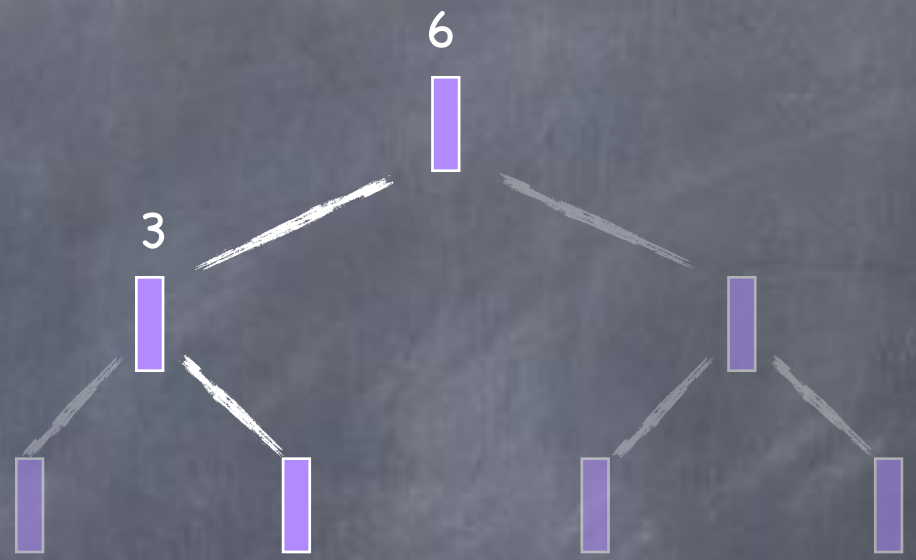
Binary Search

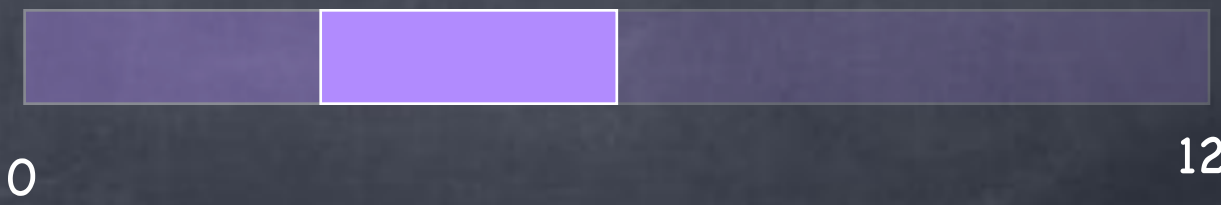
- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)
- How to compare desired object (here $\sqrt{n}$) with middle element m?
  - compare n and m^2



Binary Search

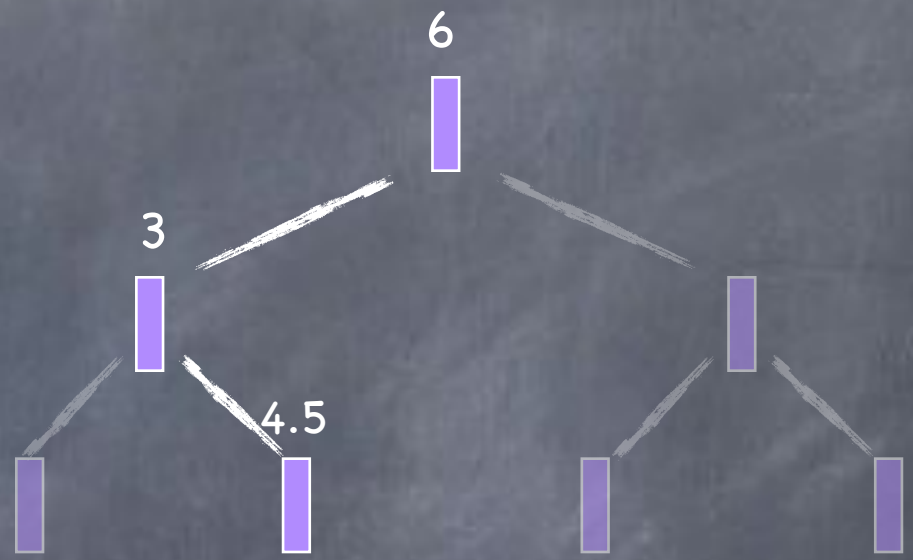
- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)
- How to compare desired object (here $\sqrt{n}$) with middle element m?
  - compare n and m^2

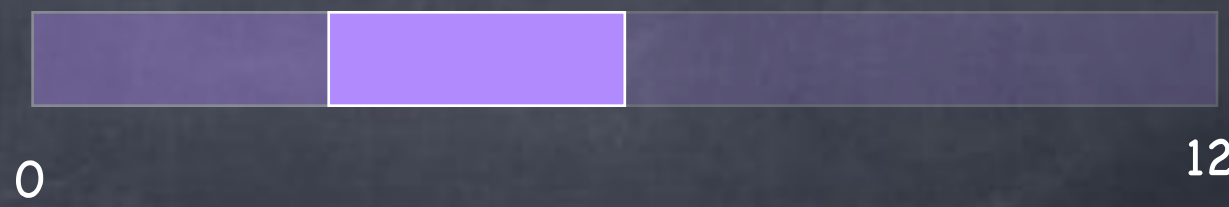




Binary Search

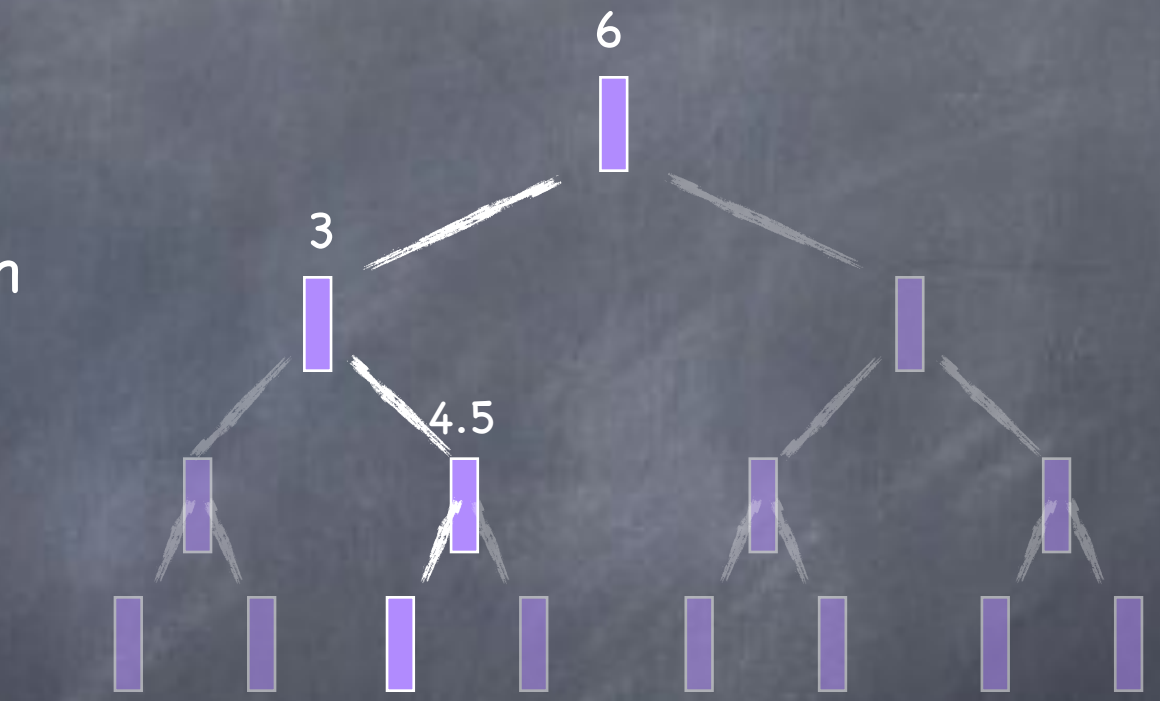
- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)
- How to compare desired object (here $\sqrt{n}$) with middle element m?
  - compare n and m^2

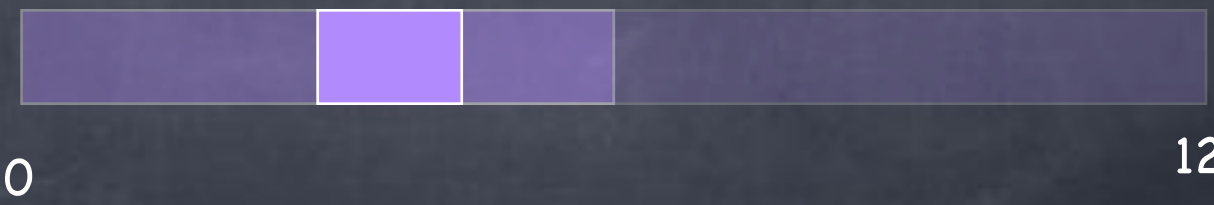




Binary Search

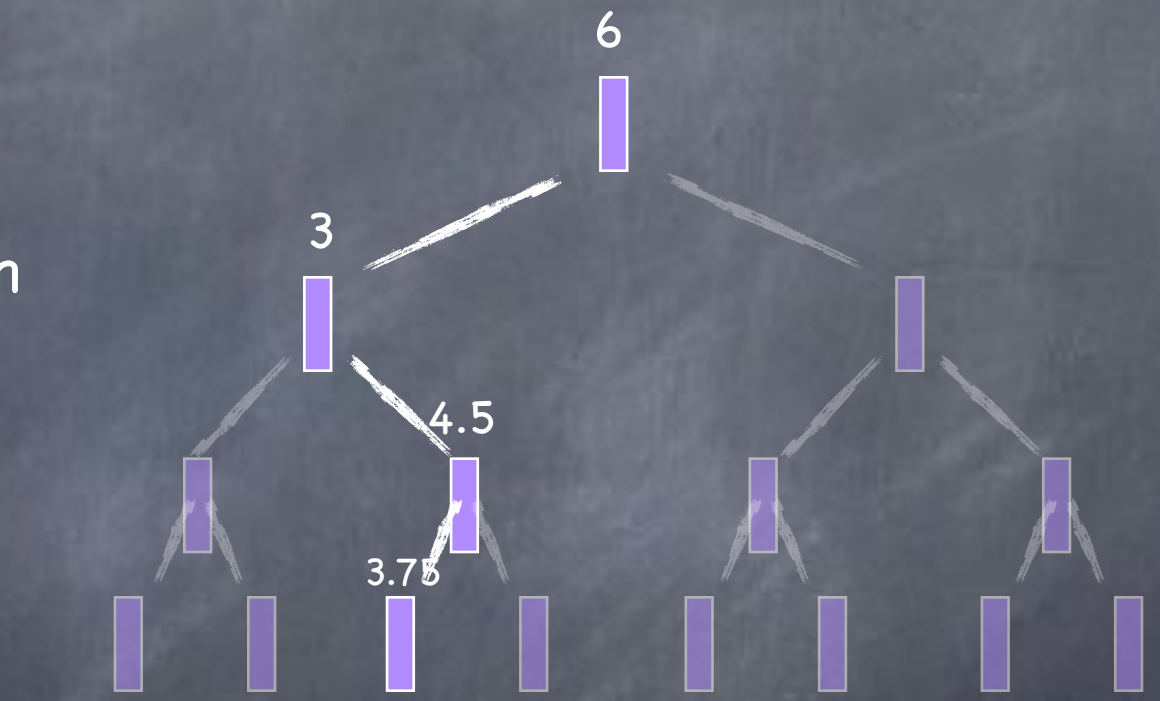
- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)
- How to compare desired object (here $\sqrt{n}$) with middle element m?
  - compare n and m^2





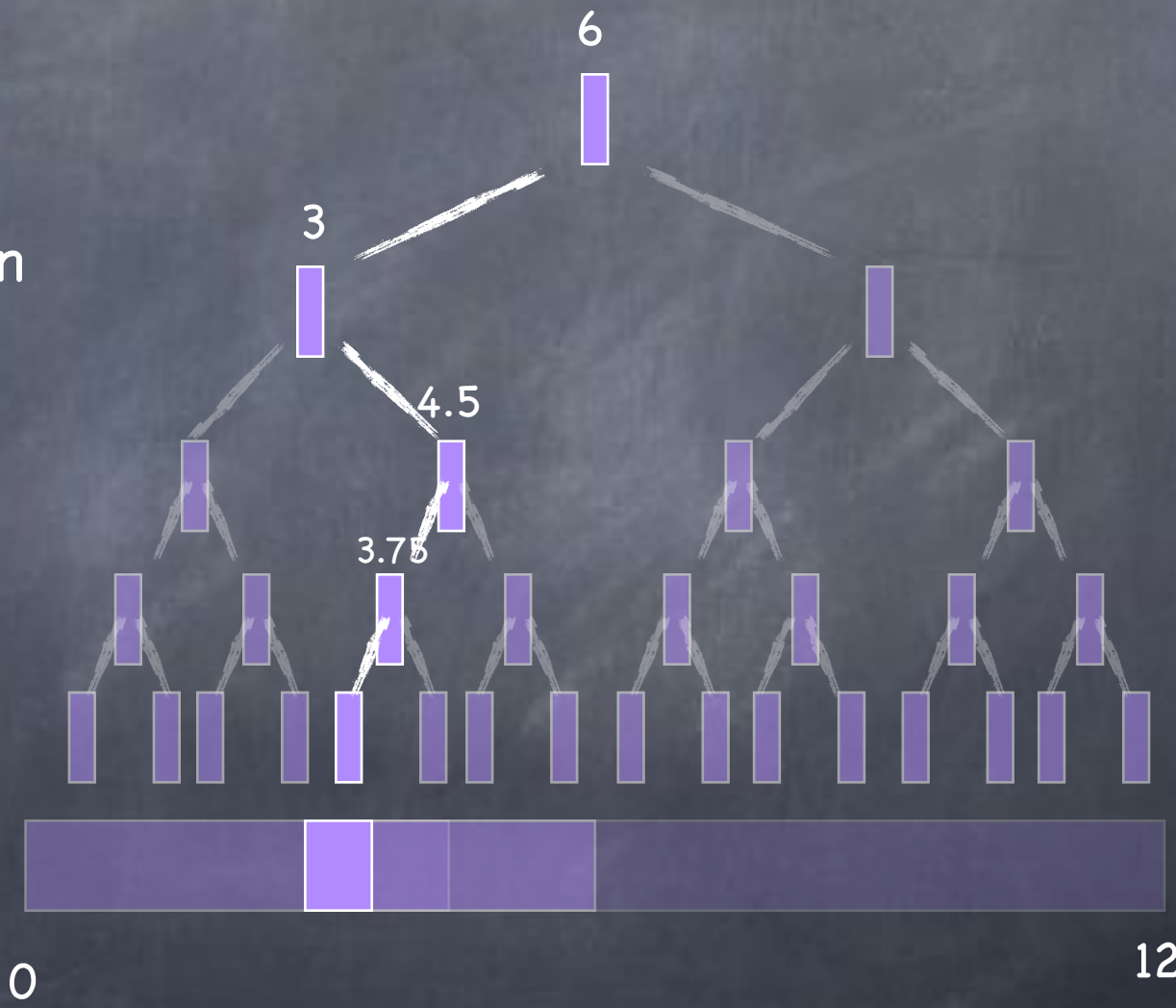
Binary Search

- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)
- How to compare desired object (here $\sqrt{n}$) with middle element m?
  - compare n and m^2



Binary Search

- Example: finding (up to required precision) the square root of a number n>1 (using only comparison and multiplication)
- Initial range: [0,n] (say)
- How to compare desired object (here √n) with middle element m?
  - compare n and m^2



Merge Sort

Merge Sort

- Sorting by divide-and-conquer

Merge Sort

- Sorting by divide-and-conquer
  - Split the list into two (unless a single element)

Merge Sort

- Sorting by divide-and-conquer
  - Split the list into two (unless a single element)
  - Sort each list recursively

Merge Sort

- Sorting by divide-and-conquer
  - Split the list into two (unless a single element)
  - Sort each list recursively
  - Merge the sorted lists into a single sorted list

Merge Sort

- Sorting by divide-and-conquer
  - Split the list into two (unless a single element)
  - Sort each list recursively
  - Merge the sorted lists into a single sorted list
- T(n) = 2T(n/2) + time to merge

Merge Sort

- Sorting by divide-and-conquer
  - Split the list into two (unless a single element)
  - Sort each list recursively
  - Merge the sorted lists into a single sorted list
- $T(n) = 2T(n/2) + \text{time to merge}$
- $T(n) = 2T(n/2) + d\,n$ [in fact, $T(n) \leq 2T(n/2)+dn+c$]

Merge Sort

- Sorting by divide-and-conquer
  - Split the list into two (unless a single element)
  - Sort each list recursively
  - Merge the sorted lists into a single sorted list
- $T(n) = 2T(n/2) + \text{time to merge}$
- $T(n) = 2T(n/2) + d\,n$ [in fact, $T(n) \leq 2T(n/2)+dn+c$]
  - Depth of recursion = $O(\log n)$

Merge Sort

- Sorting by divide-and-conquer
  - Split the list into two (unless a single element)
  - Sort each list recursively
  - Merge the sorted lists into a single sorted list
- $T(n) = 2T(n/2) + \text{time to merge}$
- $T(n) = 2T(n/2) + d\,n$ [in fact, $T(n) \leq 2T(n/2)+dn+c$]
  - Depth of recursion = $O(\log n)$
  - $T(n) = O(n \log n)$

Big Number Arithmetic

Big Number Arithmetic

- Usually multiplication/addition are a single operation in a CPU

Big Number Arithmetic

- Usually multiplication/addition are a single operation in a CPU
- But not possible when an integer has too many digits to fit into a processor's registers

Big Number Arithmetic

- Usually multiplication/addition are a single operation in a CPU
- But not possible when an integer has too many digits to fit into a processor's registers
- Can break up the integer into smaller pieces, and compute on them

Big Number Arithmetic

- Usually multiplication/addition are a single operation in a CPU
- But not possible when an integer has too many digits to fit into a processor's registers
- Can break up the integer into smaller pieces, and compute on them
  - e.g. Addition with carry: each operation works on single digit numbers (takes 3 numbers and gives two numbers)

Big Number Arithmetic

- Usually multiplication/addition are a single operation in a CPU
- But not possible when an integer has too many digits to fit into a processor's registers
- Can break up the integer into smaller pieces, and compute on them
  - e.g. Addition with carry: each operation works on single digit numbers (takes 3 numbers and gives two numbers)
  - To add two n-digit numbers: O(n) operations

Big Number Arithmetic

- Usually multiplication/addition are a single operation in a CPU
- But not possible when an integer has too many digits to fit into a processor's registers
- Can break up the integer into smaller pieces, and compute on them
  - e.g. Addition with carry: each operation works on single digit numbers (takes 3 numbers and gives two numbers)
  - To add two n-digit numbers: O(n) operations
    - As fast as possible: need to at least read all the digits

Big Number Arithmetic

- Usually multiplication/addition are a single operation in a CPU
- But not possible when an integer has too many digits to fit into a processor's registers
- Can break up the integer into smaller pieces, and compute on them
  - e.g. Addition with carry: each operation works on single digit numbers (takes 3 numbers and gives two numbers)
  - To add two n-digit numbers: O(n) operations
    - As fast as possible: need to at least read all the digits
    - (Remember: the number N has O(log N) digits)

Big Number Arithmetic

Big Number Arithmetic

- Multiplication of two large numbers

Big Number Arithmetic

- Multiplication of two large numbers
  - First attempt: $x = x_0 + 2\,x_1$, where x_1 has one digit less
    Similarly, $y = y_0 + 2\,y_1$. So $x \cdot y = x_0y_0 + 2\,(x_0y_1 + x_1y_0) + x_1y_1$.

Big Number Arithmetic

- Multiplication of two large numbers
  - First attempt: $x = x_0 + 2\,x_1$, where x_1 has one digit less
    Similarly, $y = y_0 + 2\,y_1$. So $x \cdot y = x_0y_0 + 2\,(x_0y_1 + x_1y_0) + x_1y_1$.
  - $T(n) = T(n-1) + O(n)$ (and $T(1)=O(1)$). So $T(n) = O(n^2)$

Big Number Arithmetic

- Multiplication of two large numbers
  - First attempt: $x = x_0 + 2\,x_1$, where x_1 has one digit less
    Similarly, $y = y_0 + 2\,y_1$. So $x \cdot y = x_0y_0 + 2\,(x_0y_1 + x_1y_0) + x_1y_1$.
  - $T(n) = T(n-1) + O(n)$ (and $T(1)=O(1)$). So $T(n) = O(n^2)$
  - Can we do better by dividing the problem differently?

Big Number Arithmetic

- Multiplication of two large numbers
  - First attempt: $x = x_0 + 2\,x_1$, where x_1 has one digit less
    Similarly, $y = y_0 + 2\,y_1$. So $x \cdot y = x_0y_0 + 2\,(x_0y_1 + x_1y_0) + x_1y_1$.
  - $T(n) = T(n-1) + O(n)$ (and $T(1)=O(1)$). So $T(n) = O(n^2)$
  - Can we do better by dividing the problem differently?
    - $x = x_0 + 2^{n/2}\,x_1$ where x_0, x_1 have n/2 digits each
      (assuming n is a power of 2)

Big Number Arithmetic

- Multiplication of two large numbers
  - First attempt: $x = x_0 + 2\,x_1$, where x_1 has one digit less Similarly, $y = y_0 + 2\,y_1$. So $x \cdot y = x_0y_0 + 2\,(x_0y_1 + x_1y_0) + x_1y_1$.
  - $T(n) = T(n-1) + O(n)$ (and $T(1)=O(1)$). So $T(n) = O(n^2)$
  - Can we do better by dividing the problem differently?
    - $x = x_0 + 2^{n/2}\,x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)
    - $x \cdot y = x_0y_0 + 2^{n/2}(x_0y_1 + x_1y_0) + 2^n x_1y_1$, where all 4 products are of n/2 digit numbers (mult. by power of 2 and addition take O(n) time)

Big Number Arithmetic

- Multiplication of two large numbers
  - First attempt: $x = x_0 + 2\,x_1$, where x_1 has one digit less Similarly, $y = y_0 + 2\,y_1$. So $x \cdot y = x_0y_0 + 2\,(x_0y_1 + x_1y_0) + x_1y_1$.
  - $T(n) = T(n-1) + O(n)$ (and $T(1)=O(1)$). So $T(n) = O(n^2)$
  - Can we do better by dividing the problem differently?
    - $x = x_0 + 2^{n/2}\,x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)
    - $x \cdot y = x_0y_0 + 2^{n/2}(x_0y_1 + x_1y_0) + 2^n x_1y_1$, where all 4 products are of n/2 digit numbers (mult. by power of 2 and addition take O(n) time)
    - $T(n) = 4T(n/2) + \Theta(n)$. Still $T(n)=\Theta(n^2)$.

Big Number Arithmetic

- Multiplication of two large numbers
  - First attempt: $x = x_0 + 2\,x_1$, where x_1 has one digit less Similarly, $y = y_0 + 2\,y_1$. So $x \cdot y = x_0y_0 + 2\,(x_0y_1 + x_1y_0) + x_1y_1$.
  - $T(n) = T(n-1) + O(n)$ (and $T(1)=O(1)$). So $T(n) = O(n^2)$
  - Can we do better by dividing the problem differently?
    - $x = x_0 + 2^{n/2}\,x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)
    - $x \cdot y = x_0y_0 + 2^{n/2}(x_0y_1 + x_1y_0) + 2^n x_1y_1$, where all 4 products are of n/2 digit numbers (mult. by power of 2 and addition take O(n) time)
    - $T(n) = 4T(n/2) + \Theta(n)$. Still $T(n)=\Theta(n^2)$.
    - Can we do better?

Big Number Arithmetic

Big Number Arithmetic

- Multiplication of two large numbers

Big Number Arithmetic

- Multiplication of two large numbers
  - $x = x_0 + 2^{n/2} x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)

Big Number Arithmetic

- Multiplication of two large numbers
  - $x = x_0 + 2^{n/2} x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)
  - $x \cdot y = x_0y_0 + 2^{n/2} (x_0y_1 + x_1y_0) + 2^n x_1y_1$

    $= x_0y_0 + 2^{n/2}[(x_0+x_1)(y_0+y_1) - x_0y_0 - x_1y_1] + 2^n x_1y_1$

Big Number Arithmetic

- Multiplication of two large numbers
  - $x = x_0 + 2^{n/2} x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)
  - $x \cdot y = x_0y_0 + 2^{n/2} (x_0y_1 + x_1y_0) + 2^n x_1y_1$

    $= x_0y_0 + 2^{n/2}[(x_0+x_1)(y_0+y_1) - x_0y_0 - x_1y_1] + 2^n x_1y_1$
    - Only 3 multiplications (and reusing products). All of them on numbers about n/2 digits each

Big Number Arithmetic

- Multiplication of two large numbers
  - $x = x_0 + 2^{n/2} x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)
  - $x \cdot y = x_0y_0 + 2^{n/2} (x_0y_1 + x_1y_0) + 2^n x_1y_1$

    $= x_0y_0 + 2^{n/2}[(x_0+x_1)(y_0+y_1) - x_0y_0 - x_1y_1] + 2^n x_1y_1$
    - Only 3 multiplications (and reusing products). All of them on numbers about n/2 digits each
    - $T(n) = 3T(n/2) + O(n)$. $T(1) = O(1)$.

Big Number Arithmetic

- Multiplication of two large numbers
  - $x = x_0 + 2^{n/2} x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)
  - $x \cdot y = x_0y_0 + 2^{n/2} (x_0y_1 + x_1y_0) + 2^n x_1y_1$
    $= x_0y_0 + 2^{n/2}[(x_0+x_1)(y_0+y_1) - x_0y_0 - x_1y_1] + 2^n x_1y_1$
    - Only 3 multiplications (and reusing products). All of them on numbers about n/2 digits each
    - $T(n) = 3T(n/2) + O(n)$. $T(1) = O(1)$.
      - Recursion tree: each level (internal or leaves) has 3^k nodes, with $n/2^k$ on each node. Level sum = $O(n \cdot (3/2)^k)$. k = 0 to $\log_2 n$. (Level sum from last level dominates.)

Big Number Arithmetic

- Multiplication of two large numbers
  - $x = x_0 + 2^{n/2} x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)
  - $x \cdot y = x_0y_0 + 2^{n/2} (x_0y_1 + x_1y_0) + 2^n x_1y_1$
    $= x_0y_0 + 2^{n/2}[(x_0+x_1)(y_0+y_1) - x_0y_0 - x_1y_1] + 2^n x_1y_1$
    - Only 3 multiplications (and reusing products). All of them on numbers about n/2 digits each
    - $T(n) = 3T(n/2) + O(n)$. $T(1) = O(1)$.
      - Recursion tree: each level (internal or leaves) has 3^k nodes, with $n/2^k$ on each node. Level sum = $O(n \cdot (3/2)^k)$. k = 0 to $\log_2 n$. (Level sum from last level dominates.)
      - $T(n) = O(n(3/2)^{\log\_2 n}) = O(2^{\log\_2 n} \cdot (3/2)^{\log\_2 n}) = O(3^{\log\_2 n})$
        $= O(3^{\log\_3 n \times \log\_2 3}) = O(n^{\log\_2 3}) = O(n^{1.585..})$

Big Number Arithmetic

- Multiplication of two large numbers
  - $x = x_0 + 2^{n/2} x_1$ where x_0, x_1 have n/2 digits each (assuming n is a power of 2)
  - $x \cdot y = x_0y_0 + 2^{n/2} (x_0y_1 + x_1y_0) + 2^n x_1y_1$
    $= x_0y_0 + 2^{n/2}[(x_0+x_1)(y_0+y_1) - x_0y_0 - x_1y_1] + 2^n x_1y_1$
    - Only 3 multiplications (and reusing products). All of them on numbers about n/2 digits each
    - $T(n) = 3T(n/2) + O(n)$. $T(1) = O(1)$.
      - Recursion tree: each level (internal or leaves) has 3^k nodes, with $n/2^k$ on each node. Level sum = $O(n \cdot (3/2)^k)$. k = 0 to $\log_2 n$. (Level sum from last level dominates.)
      - $T(n) = O(n(3/2)^{\log\_2 n}) = O(2^{\log\_2 n} . (3/2)^{\log\_2 n}) = O(3^{\log\_2 n})$
        $= O(3^{\log\_3 n \times \log\_2 3}) = O(n^{\log\_2 3}) = O(n^{1.585..})$

Can do even better, but more involved