CSCI 262
Data Structures

10 – Analysis of Recursive Algorithms

Binary Search

Merge Sort

Analysis of

RECURSIVE ALGORITHMS

Recursive Function Analysis

Here's a simple recursive function (in pseudo-code) which raises one number to a (non-negative) power:

```
double power(double n, unsigned k)
   if k == 0 return 1
   return n * power(n, k-1)
```

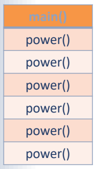
What is the "Big O" cost of `power()`?

Analyzing Power

- First, note that we want to analyze `power()` in terms of k, not n (why?)
- Now, ask the following two questions:
  - How much work do we do within `power()`, excluding the recursive call?
  - How many calls do we make to `power()`?

Analyzing Power

We can think of this another way by visualizing our call stack, and ask these questions:

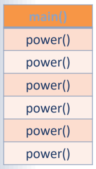


How much work at each level?

How many levels?

Analyzing Power

```
double power(double n, unsigned k)
     if k == 0 return 1
     return n * power(n, k-1)
```



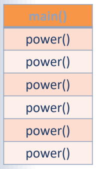
How much work at each level?

One comparison, one multiplication

How many levels?

Analyzing Power

```
double power(double n, int k)
     if k == 0 return 1
     return n * power(n, k-1)
```



How much work at each level?

One comparison, one multiplication

How many levels?

How many times can we subtract 1 before we get to k == 0?

Analyzing Power

Analysis:

2 operations per level * k levels

= 2k operations

In "Big O", we drop constants, so that's O(k).

Analyzing Power 2

Suppose we try a different approach. Note that

$n^k = n^{k/2} \times n^{k/2}$

However, we need to keep our exponents integral, so instead we can do

$n^k = n^{\lceil k/2 \rceil} \times n^{\lfloor k/2 \rfloor}$

The expression $\lceil x \rceil$ is called the *ceiling* of x, and means that we round up to the nearest integer. $\lfloor x \rfloor$ is called the *floor* of x, and means we round down.

Analyzing Power 2

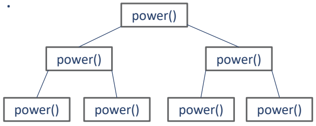
If we apply this formula in pseudo-code, we get:

```
double power(double n, unsigned k)
    if k == 0 return 1
    else if k == 1 return n
    else return power(n, ⌈k/2⌉) * power(n, ⌊k/2⌋)
```

This base case is needed now, to ensure we get a factor of n in somewhere!

Analyzing Power 2

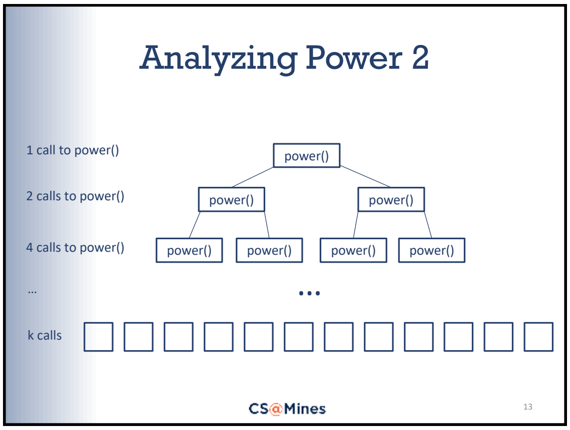
- Now things are more complicated, because each call to `power()` turns into two more calls to `power()`, etc.
- Instead of a stack, we can visualize this as a "call tree":



- How many calls to `power()` here?

Analyzing Power 2

- For these kinds of problems, easier to approximate using an ideal case:
  - Assume k is power of 2: $k = 2^p$
  - Now we divide k evenly in half at each level – no "ragged" levels
- How many levels are in our tree?
- How much work is done at each level?



Analyzing Power 2

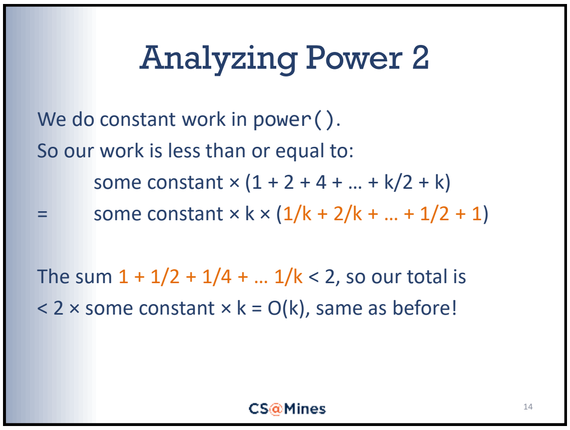
1 call to power()

2 calls to power()

4 calls to power()

…

k calls



Analyzing Power 2

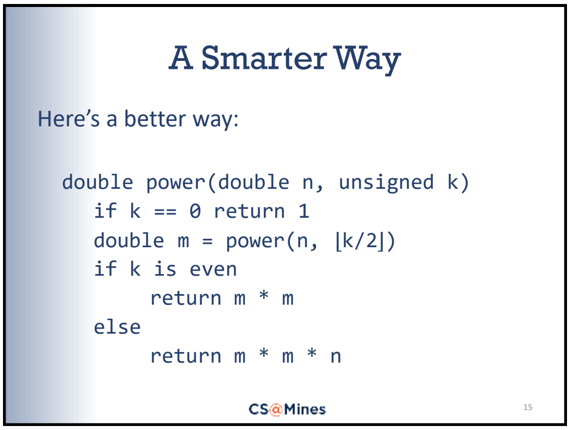
We do constant work in `power()`.

So our work is less than or equal to:

some constant × (1 + 2 + 4 + … + k/2 + k)

= some constant × k × (1/k + 2/k + … + 1/2 + 1)

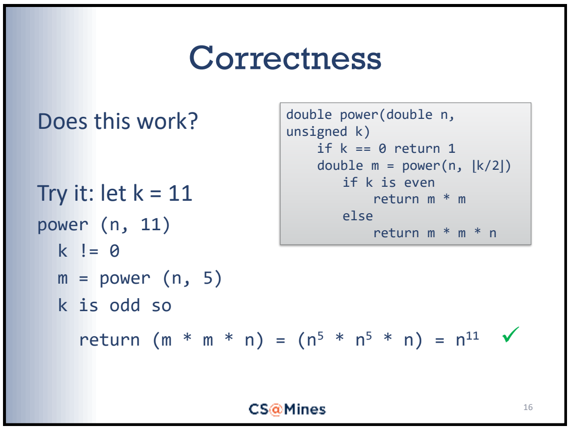
The sum 1 + 1/2 + 1/4 + … 1/k < 2, so our total is < 2 × some constant × k = O(k), same as before!



A Smarter Way

Here's a better way:

```
double power(double n, unsigned k)
   if k == 0 return 1
   double m = power(n, ⌊k/2⌋)
   if k is even
        return m * m
   else
        return m * m * n
```



Correctness

Does this work?

```
double power(double n,
unsigned k)
   if k == 0 return 1
   double m = power(n, ⌊k/2⌋)
        if k is even
           return m * m
        else
           return m * m * n
```

Try it: let k = 11

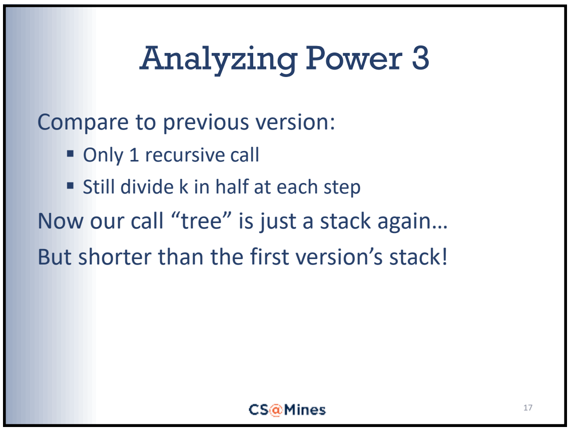
`power (n, 11)`

`k != 0`

`m = power (n, 5)`

`k is odd so`

`return (m * m * n) =` (n^5 * n^5 * n) = n^{11} ✓



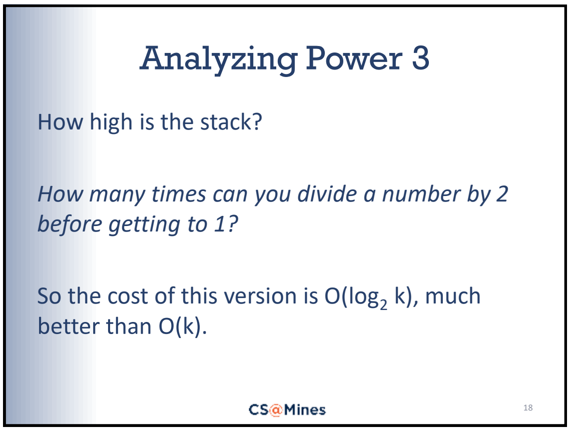
Analyzing Power 3

Compare to previous version:

- Only 1 recursive call
- Still divide k in half at each step

Now our call "tree" is just a stack again…

But shorter than the first version's stack!



Analyzing Power 3

How high is the stack?

How many times can you divide a number by 2 before getting to 1?

So the cost of this version is $O(\log_2 k)$, much better than O(k).

Searching with

DIVIDE AND CONQUER

Divide and Conquer

- Split problem into multiple smaller sub-problems
- Solve the sub-problems *recursively*
- Recombine solutions afterwards
- When splitting/recombination can be done efficiently, this approach is a winner

Linear Search

Search for a value in a sorted list.

Obvious approach:

```
// find element k in sorted list x containing n elements
search(x, k)
for i = 1 to n
     if x[i] == k return i
return  NOTFOUND
```

Pseudocode usually starts at index =1, not index = 0

Complexity: O(N)

Binary Search

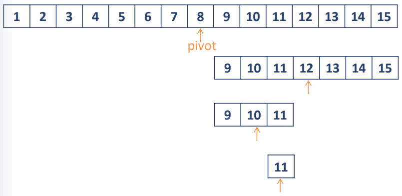
Search for a value in a sorted list.

```
// find element k in sorted list x containing n elements
binary_search(x, k)
    if x is empty
            return NOTFOUND
    pivot = n/2    // look at element halfway through list
    if  x[pivot] == k
            return pivot  // if found, return
    else if k < x[pivot]  // else search left or right sublist
            return binary_search(x[1 : pivot-1], k)
    else
            return binary_search(x[pivot+1 : n], k)
```

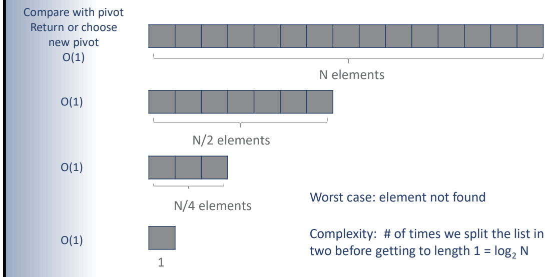
Binary Search Example

Search for a value in a sorted list.

Example: search for 11 in the list 1-15



Analysis of Binary Search



Worst case: element not found

Complexity: # of times we split the list in two before getting to length 1 = $\log_2 N$

Another divide & conquer algorithm:

MERGE SORT

Merge Sort

- Divide and Conquer algorithm for sorting
  - Split input list in half
  - Sort the halves recursively
  - Merge the sorted lists

```
// x is an unsorted list of n elements
merge_sort(x)
  n = length(x)
  if n == 1 return x
  left = merge_sort(x[1 : n/2])
  right = merge_sort(x[n/2 + 1 : n])
  return merge(left, right)
```

Merge Sort

- Divide and Conquer algorithm for sorting
  - Split input list in half
  - Sort the halves recursively
  - Merge the sorted lists

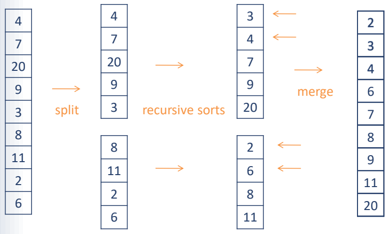
```
// x is an unsorted list of n elements
merge_sort(x)
  n = length(x)
  if n == 1 return x
  left = merge_sort(x[1 : n/2])
  right = merge_sort(x[n/2 + 1 : n])
  return merge(left, right)
```

This is where the magic happens!

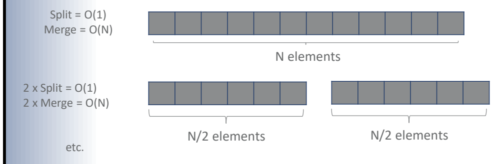
Merge Sort

```
merge(a, b)
    // treat a, b as stacks or queues
    y = empty list
    loop
        if a is empty
            append b to y, return y
        else if b is empty
            append a to y, return y
        else if top(a) < top(b)
            append pop(a) to y
        else
            append pop(b) to y
```

Merge Sort Illustrated



Analysis of Mergesort



Complexity: ?

Interlude

LOGARITHMS AND BIG O

About Logarithms

- $\log_b b^k = k$
- For any b, $\log_2 x = \log_b x / \log_b 2$

↑ This shows that the base doesn't matter in "big O" – all bases are just a constant factor from base 2.

- Because "$\log_2 x$" comes up so often, it is often abbreviated to "lg x" in computer science

---

SORTING IN THE STL

Sorting in Standard Library

- Sorting in the C++ standard library
  - Works on *random access iterators*
  - Works on vectors, strings, and arrays

```
#include <algorithm>
void sort(begin_iterator, end_iterator)
```

`sort` example

Sorting a vector:

```
#include <algorithm>
…
vector<int> vec = {17, 42, 100, -3, 50};
sort(vec.begin(), vec.end());

for (int n: vec) cout << n << " ";
```

Output:

```
-3 17 42 50 100
```

Another `sort` example

Sorting a string:

```
#include <algorithm>
…
string s = "Hello, world!";
…
sort(s.begin(), s.end());
cout << s << endl;
```

Output:

```
 !,Hdelllloorw
```

sort Notes

- Elements of container must be comparable using “<“
    - Depending on application, may be able to overload “<“ for items to be sorted
    - Otherwise, have to supply a separate bool valued function as a third parameter to sort:

```
bool rev(int a, int b) {
    return b < a;     // default comparison is a < b
}

int main() {
    vector<int> foo = {16, 4, 23, 1, 2, 17, 6};

    sort(foo.begin(), foo.end());      // {1, 2, 4, 6, 16, 17, 23}
    sort(foo.begin(), foo.end(), rev); // {23, 17, 16, 6, 4, 2, 1}
    return 0;
}
```

Up Next

- Wednesday, February 13
    - Midterm Review
- Friday, February 15
    - Lab 6 – Ancient Algorithms
- Monday, February 18
    - NO CLASS – President’s Day
- Wednesday, February 20
    - Midterm 1 – IN CLASS
- Friday, February 22
    - Optional programming contest – fun & prizes!