CSCI 262
Data Structures

4 – Analysis of Recursive Algorithms,
    Binary Search,
    Merge Sort

Recursive Function Analysis
Here’s a simple recursive function which raises one number to a (non-negative) power:

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

What is the 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:

```
main()
power()
power()
power()
power()
power()
power()
```

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?

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. This one is doubly-recursive:

double power(double n, unsigned k)
if k == 0 return 1
else if k == 1 return n
else return power(n, ceiling(k/2)) * power(n, floor(k/2))

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.

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

- How many levels are in our tree?
- How much work is done at each level?
Analyzing Power 2

We do constant work in power.
So our work is less than or equal to:

\[ \text{some constant} \times (1 + 2 + 4 + \ldots + k/2 + k) \]

\[ = \text{some constant} \times k \times \left( \frac{1}{k} + \ldots + \frac{1}{4} + \frac{1}{2} + 1 \right) \]

The sum \( 1 + 1/2 + 1/4 + \ldots + 1/k < 2 \), so our total is

\[ < 2 \times \text{some constant} \times k = O(k), \text{ same as before!} \]

Correctness

Does this work?

Try it: let \( k = 11 \)

\[ m = \text{power} (n, 5) \]

\[ k \text{ is odd so} \]

\[ \text{return } (m \times m \times n) = (n^5 \times n^5 \times n) = n^{11} \checkmark \]

A Smarter Way

Here’s a better way:

\[
\text{double power(double n, unsigned k)
if k == 0 return 1
double m = power(n, [k/2])
if k is even
return m \times m
else
return m \times m \times n}
\]

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 \( O(k) \).
Divide and Conquer
- Split problem into multiple smaller subproblems
- 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:
```java
// 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
```

Complexity: O(N)

**Binary Search**
Search for a value in a sorted list.
```java
// 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
```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

**Analysis of Binary Search**
Comparing with pivot
- Compare with pivot
- Return or choose new pivot

O(1)
```
 N elements
```

O(1)
```
 N/2 elements
```

O(1)
```
 N/4 elements
```

```
1
```

Worst case: element not found

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

**Merge Sort**
Another divide & conquer algorithm:
Merge Sort

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

```python
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 Illustrated

Analysis of Mergesort

- Split = \( O(1) \)
- Merge = \( O(n) \)
- \( 2 \times \text{Split} = O(1) \)
- \( 3 \times \text{Merge} = O(n^2) \)

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

```cpp
#include <algorithm>

void sort(begin_iterator, end_iterator)
```

#### sort example

**Sorting a vector:**

```cpp
#include <algorithm>

int main()

    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:**

```cpp
#include <algorithm>

int main()

    vector<int> foo = {16, 4, 23, 1, 2, 17, 6};
    sort(foo.begin(), foo.end()); // {2, 4, 6, 10, 17, 23}
    sort(foo.begin(), foo.end(), rev); // {10, 17, 16, 6, 4, 2, 1}
    return 0;
```

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

```cpp
return 0;
```
Up Next

- Reading: Chapter 12.4 – 12.6, 12.7 optional
- Friday, September 7
  - Lab 3
- Monday, September 10
  - Lab 3 due
  - Project 1 – Image Editor due
  - TBA