Recurrence Relations

Class 13

Definitions

- you are very familiar with function definitions in math
- a function is defined with an algebraic rule

$$f(x) = x^2 - 3x + 2$$

- it can be translated directly into a C++ function

```
double f(double x)
{
  return x * x - 3 * x + 2;
}
```

Sequences

- there is another type of definition
- commonly used to define sequences of values
- the Fibonacci sequence can be listed as $\{1, 1, 2, 3, 5, \ldots\}$
- and it can also be defined by a rule

$$f(n) = f(n-1) + f(n-2) \text{ given } f(0) = f(1) = 1$$

- this type of rule is called a recurrence relation
- with initial conditions

Recurrence Relations

- a recurrence relation is an equation or inequality
- defines an arbitrary element in a sequence in terms of one or more of its predecessors

Recurrence Relations

- a recurrence relation is an equation or inequality
- defines an arbitrary element in a sequence in terms of one or more of its predecessors

- a recursive algorithm implements a recurrence relation
- a recurrence relation describes a recursive algorithm

Recurrence to Recursion

- recurrence relations translate into code
- the initial conditions turn into base cases
- the code has recursive calls

```
unsigned fib(unsigned n)
{
  if (n == 0 || n == 1)
  {
    return 1;
  }
  return fib(n - 1) + fib(n - 2);
}
```

Solving Recurrence Relations

- to solve a recurrence relation means to give a formulation for an arbitrary element in a sequence in terms that does not use any other elements in the sequence
- a solution is also called a closed form
- there are many techniques for solving recurrence relations
- we will only look at one, as our interest is in using their results

Solving by Substitution

Let

$$T(n) = T(n{-}1) + n \text{ given } T(0) = 0$$

- off to the side, replace every occurrence of n with $n - 1$
- we can do this because n is arbitrary
- this substitution gives us

$$\begin{aligned} T(n-1) &= T(n-1-1) + (n-1) \\ &= T(n-2)+(n-1) \end{aligned}$$

Solving by Substitution

Let

$$T(n) = T(n-1) + n \text{ given } T(0) = 0$$

- off to the side, replace every occurrence of n with $n-1$
- we can do this because n is arbitrary
- this substitution gives us

$$\begin{aligned} T(n-1) &= T(n-1-1) + (n-1) \\ &= \boxed{T(n-2)+(n-1)} \end{aligned}$$

- now substitute this expression for $T(n-1)$ back into the original formulation, to give

Solving by Substitution

Let

$$T(n) = T(n-1) + n \text{ given } T(0) = 0$$

- off to the side, replace every occurrence of n with $n-1$
- we can do this because n is arbitrary
- this substitution gives us

$$\begin{aligned} T(n-1) &= T(n-1-1) + (n-1) \\ &= \boxed{T(n-2) + (n-1)} \end{aligned}$$

- now substitute this expression for $T(n-1)$ back into the original formulation, to give

$$T(n) = T(n-2) + (n-1) + n$$

Solving by Substitution

- using the original formulation, off to the side substitute every occurrence of n by $n-2$ to get

$$T(n-2) = T(n-3) + (n-2)$$

- and use this expression for $T(n-2)$ in the last expression of the previous slide

$$\begin{aligned} T(n) &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \end{aligned}$$

Solving by Substitution

- continuing the series, we have

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + (n-1) + n \\ &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \end{aligned}$$

- how long can this process go on?

Solving by Substitution

- the series ends at the initial condition (base case) $T(0) = 0$

$$
\begin{aligned}
T(n) &= T(n-1) + n \\
&= T(n-2) + (n-1) + n \\
&= T(n-3) + (n-2) + (n-1) + n \\
&\vdots \\
&= T(n-(n-1)) + (n-(n-2)) + (n-(n-3)) + \cdots \\
&\qquad + (n-1) + n \\
&= T(n-n) + (n-(n-1)) + (n-(n-2)) + (n-(n-3)) \\
&\qquad + \cdots + (n-1) + n \\
&= 0 + 1 + 2 + \cdots + n \\
&= \frac{n(n+1)}{2}
\end{aligned}
$$

Analysis

- thus we have the closed form

$$\begin{aligned} T(n) &= T(n-1) + n \text{ given } T(0) = 0 \\ &= \frac{n(n+1)}{2} \end{aligned}$$

- the solution of a recurrence relation is identical to the analysis of its matching recursive algorithm
- we analyze recursive algorithms by
  - writing the recurrence relation for the algorithm
  - solving that recurrence relation
- thus we have

$$T(n) \in \Theta(n^2)$$

Analyzing Recursive Functions

- unfortunately, many recurrence relations are hard to solve
- substitution only works when the terms differ in position by exactly one $n \to n - 1$
- however, most of the recurrence relations we deal with in this course do not have this form

The Master Theorem

- most of our recurrence relations instead have this form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- where $f(n)$ is a polynomial of degree d e.g., $f(n) = kn^d$
- this cannot be solved by substitution
- we will use the Master Theorem for this form of recurrence relation

Example

use the Master Theorem to analyze the recursive algorithm whose behavior is expressed by

$$T(n) = 4T\left(\frac{n}{2}\right) + kn$$

Example

use the Master Theorem to analyze the recursive algorithm whose behavior is expressed by

$$T(n) = 4T\left(\frac{n}{2}\right) + kn$$

- $a = 4$
- $b = 2$
- $d = 1$

Example

use the Master Theorem to analyze the recursive algorithm whose behavior is expressed by

$$T(n) = 4T\left(\frac{n}{2}\right) + kn$$

- $a = 4$
- $b = 2$
- $d = 1$

We must now ask the question:

$$a \overset{?}{<} b^d$$

$$4 \overset{?}{<} 2^1$$

$$4 \not< 2^1 \text{ No}$$

Example

- $a = 4$
- $b = 2$
- $d = 1$

Next ask:

$$a \stackrel{?}{=} b^d$$

$$4 \stackrel{?}{=} 2^1$$

$$4 \neq 2^1 \text{ No}$$

Example

- $a = 4$
- $b = 2$
- $d = 1$

Finally ask:

$$a \overset{?}{>} b^d$$

$$4 \overset{?}{>} 2^1$$

$$4 > 2^1 \text{ Yes!}$$

Therefore

$$\begin{aligned} T(n) &\in \Theta(n^{\log_2 4}) \\ &\in \Theta(n^2) \end{aligned}$$

Analysis

- note the Master Theorem does not actually solve the recurrence relation
- we did not arrive at a closed form
(because we did not specify a base case)
- with the Master Theorem, we go directly from recurrence relation to analysis, without solving for a closed form
- therefore, we do not need to specify a base case

Binary Search

- binary search is a classic recursive algorithm
- a recursive algorithm consists of
    1. one or more checks for base case(s)
    2. some amount of local work
    3. one or more recursive calls
- basic operations
    - recursive calls themselves are not counted
    - return statements themselves are not counted

    - generating arguments for recursive calls is counted
    - generating a value to return is counted
    - local work is counted

Binary Search

2	3	5	11	17	23	29

1. if the range of elements is **empty**, return not-found sentinel

Binary Search

2	3	5	11	17	23	29

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into 3:
   2.1 the very middle element

Binary Search

2	3	5	11	17	23	29

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into 3:
   2.1 the very middle element
   2.2 the elements to the **left** of middle

Binary Search

2	3	5	11	17	23	29

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into 3:
   2.1 the very middle element
   2.2 the elements to the **left** of middle
   2.3 the elements to the right of middle

Binary Search

2	3	5	11	17	23	29

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into 3:
   2.1 the very middle element
   2.2 the elements to the **left** of middle
   2.3 the elements to the right of middle
3. if the searched-for value is the middle element, you've found it and you're done

Binary Search

2	3	5	11	17	23	29

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into 3:
   2.1 the very middle element
   2.2 the elements to the **left** of middle
   2.3 the elements to the right of middle
3. if the searched-for value is the middle element, you've found it and you're done
4. else if the searched-for value is **smaller** than the middle element, repeat step 1 on the **left half**

Binary Search

2	3	5	11	17	23	29

1. if the range of elements is **empty**, return not-found sentinel
2. divide the range of elements to search into 3:
   2.1 the very middle element
   2.2 the elements to the **left** of middle
   2.3 the elements to the right of middle
3. if the searched-for value is the middle element, you've found it and you're done
4. else if the searched-for value is **smaller** than the middle element, repeat step 1 on the **left half**
5. else repeat step 1 on the right half

Binary Search

look at code

Binary Search Analysis

base case determination

line 5: 1 operation

line 6: 3 operations

local work

line 8: 3 operations

lines 9 and 13: 2 operations

line 11 or 15: 1 operation (lines mutually exclusive)

total: 10 operations

Binary Search Analysis

- how many recursive calls?

- how big is the input for the recursive call?
- can the algorithm end early?

Binary Search Analysis

- how many recursive calls?
  - either line 9 or line 13, but never both
  - therefore one recursive call
- how big is the input for the recursive call?
- can the algorithm end early?

Binary Search Analysis

- how many recursive calls?
  - either line 9 or line 13, but never both
  - therefore one recursive call
- how big is the input for the recursive call?
  - the size of the range is half the size of the original
- can the algorithm end early?

Binary Search Analysis

- how many recursive calls?
  - either line 9 or line 13, but never both
  - therefore one recursive call
- how big is the input for the recursive call?
  - the size of the range is half the size of the original
- can the algorithm end early?
  - yes, because of line 19 `return mid;`
  - not because of line 22
    (special case, equivalent to zero input size)

Binary Search Analysis

- running this algorithm thus involves
  - $\leq$ because the whole process might end early
  - 10 operations
  - 1 recursive call on a range of size $\frac{n}{2}$

$$T(n) \leq aT\left(\frac{n}{b}\right) + kn^d$$

Binary Search Analysis

- running this algorithm thus involves
  - $\leq$ because the whole process might end early
  - 10 operations
  - 1 recursive call on a range of size $\frac{n}{2}$

$$T(n) \leq aT\left(\frac{n}{b}\right) + kn^d$$

Binary Search Analysis

- running this algorithm thus involves
  - $\leq$ because the whole process might end early
  - 10 operations
  - 1 recursive call on a range of size $\frac{n}{2}$

$$T(n) \leq aT\left(\frac{n}{b}\right) + kn^d$$

$$\leq 1T\left(\frac{n}{2}\right) + 10n^0$$

Binary Search Analysis

- running this algorithm thus involves
  - $\leq$ because the whole process might end early
  - 10 operations
  - 1 recursive call on a range of size $\frac{n}{2}$

$$
\begin{aligned}
T(n) &\leq aT\left(\frac{n}{b}\right) + kn^d \\
&\leq 1T\left(\frac{n}{2}\right) + 10n^0
\end{aligned}
$$

$$T(n) \in O(\lg n)$$

Lower Bound

- the previous slide resulted in a big-Oh result
- this is an upper bound
- now we need to consider the lower bound
- direct observation of the code shows that a single run of the algorithm could find the requested element and end the algorithm immediately
- thus the minimum number of basic operations is 9 (nothing to do in line 19)

$$\begin{aligned} T(n) &\geq 9 \\ &\in \Omega(1) \end{aligned}$$

- this is typical for search algorithms

Complete Analysis

- putting it all together, for recursive binary search, we have:
  - the input size is the size of the range of array elements
  - the algorithm can terminate early, so there are distinct best and worst cases
  - there are a constant 10 basic operations each time
  - the analysis is:

$$\begin{aligned} T(n) &\in O(\lg n) \\ &\in \Omega(1) \end{aligned}$$

Binary Search Considerations

- the analysis assumes the data are already in the vector, in order
- the analysis only treats the search function itself, not the entire program
- just to put n items into a vector is $\in \Theta(n)$
- sorting the n-item vector is $\in \Theta(n \lg n)$
- so the overall program:
  1. populate a vector with n elements
  2. sort the vector
  3. perform one binary search on the array
  4. report results
- is $\in O(n \lg n)$
- but once the vector is built, successive binary searches on that array would each be $\in O(\lg n)$

A Key Takeaway

- from a slide in Class 1:
- for our purposes, the simplest definition of a logarithm is

  How many times can you divide a number n by 2
  (using integer division)
  before the result is 1 or 0?

- binary search works by dividing the size of the vector by two for each recursion
- this division is what causes the algorithm to have logarithmic analysis