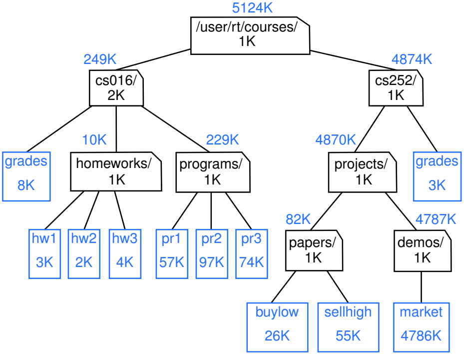
Complexity of Recursive Functions

Jianguo Lu

January 17, 2025

Overview

Motivation: Count disk usage



- The same problem **recurs**
- We need to re-apply the same method to solve the remaining data

What is a recursive program

Recursion

A method makes one or more calls to itself. A recursive program

- Has one or more base cases, and the handling of each base case should not use recursion;
- The recursive call is on a portion of the input data;
- Every possible chain of recursive calls must eventually reach a base case.

An example recursive program:

- Base case: n=0
- Recursive call on a portion of the input data: n-1
- Reach the base case: n, n-1, ...0.

```
int factorial(int n) {
   if (n == 0) return 1;
   else  return n * factorial(n-1);
 }
```

Input should be decreasing

Recursion

A method makes one or more calls to itself. A recursive program

- Has one or more base cases, and the handling of each base case should not use recursion;
- The recursive call is on a smaller portion of the input data;
- Every possible chain of recursive calls must eventually reach a base case.

What happens when running the following program:

```
int factorial(int n) {
    if (n==0) return 1;
    else  return n * factorial(n);
 }
```

Input should be decreasing

Recursion

A method makes one or more calls to itself. A recursive program

- Has one or more base cases, and the handling of each base case should not use recursion;
- The recursive call is on a smaller portion of the input data;
- Every possible chain of recursive calls must eventually reach a base case.

What happens when running the following program:

```
int factorial(int n) {
   if (n =0) return 1;
   else  return n * factorial(n);
 }
```

Decreasing alone is not enough

What happens when running the following program:

```
double factorial(double n) {
   if (n==0) return 1;
   else  return n * factorial(n/2.0);
 }
```

Why recursion

- Higher abstract level
- Models its mathematical definition
- Easier to write and verify

$$factorial(n) = \begin{cases} 1, & \text{if n=0} \\ n \times factorial(n-1), & \text{otherwise} \end{cases} \tag{1}$$

Compare this

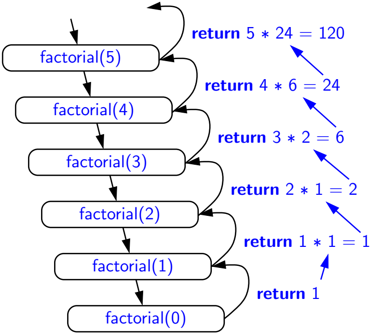
```
int factorial(int n) {
   if (n == 0) return 1;
   else  return n * factorial(n-1);
}
```

with this:

```
int factorial_iteration(int n) {
    int fac=1;
    for (int i=1; i<=n; i++){
        fac = fac*i;
     return fac;
}
```

Execution trace of recursion: Factorial example

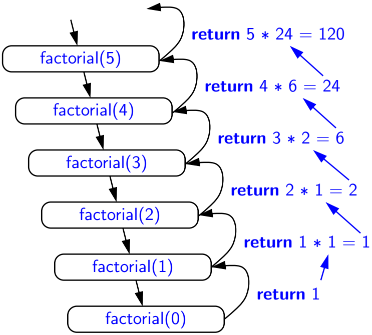
```
int factorial(int n) {
    if (n == 0) return 1;
   else  return n * factorial(n-1);
}
```



•

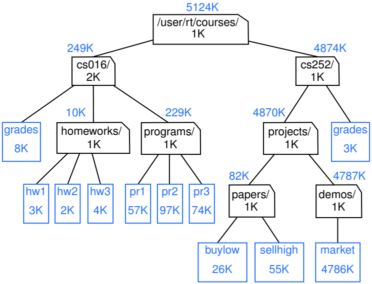
Complexity of Factorial program

```
public static int factorial(int n) {
  if (n == 0) return 1;
  else  return n * factorial(n-1);
}
```



- There are $n + 1$ calls. i.e., $n, n-1, \ldots, 0$
- Individual activation is constant time
- Hence the complexity is $O(n)$.
- There are more rigorous inference methods for recurrences.

Count disk usage




```java
public static long diskUsage(File root) {
    long total = root.length();
    if (root.isDirectory()) {
      for (String childname : root.list()) {
        File child = new File(root, childname);
        total += diskUsage(child);
      }
    }
    return total;
  }
```

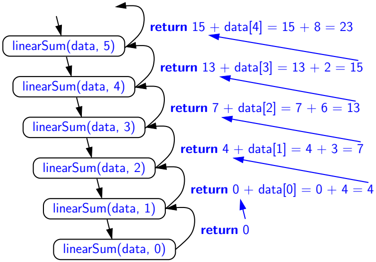
From linear recursion to binary recursion: Summation example

The problem: summation of the elements in an array

4	3	6	2	8

Linear recursion: recur once

```
public static int linearSum(int[] data, int n) {
  if (n == 0)  return 0;
  else  return linearSum(data, n-1) + data[n-1];
}
```



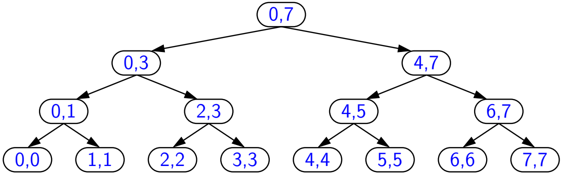
From linear recursion to binary recursion: Summation example

Binary recursion

In binary recursive method is a method that has two recursive calls

```
int binarySum(int[] data, int low, int high) {
    if (low > high)              return 0;
    else if (low == high)        return data[low];
    else {  int mid = (low + high) / 2;
      return binarySum(data,low,mid)+binarySum(data,mid+1,high);
    }
}
```

What is the complexity of binary summation?

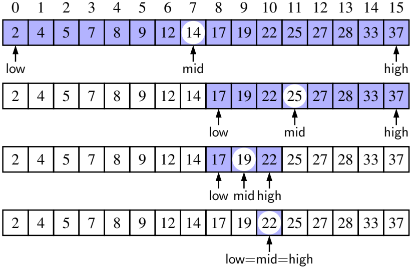


Search from an sorted array

- Sequential search is $O(n)$
- Better search method?
- Note that the array is sorted.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

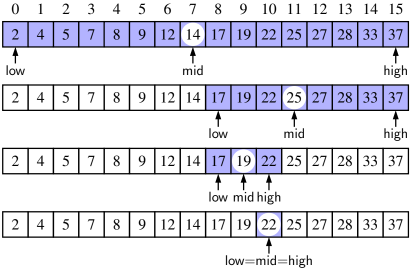
Binary search




```java
boolean binarySearch(int[] data,int target,int low,int high) {
  if (low>high) return false;
  else {
      int mid = (low + high) / 2;
      if (target == data[mid]) return true;
      else if (target < data[mid])
          return binarySearch(data, target, low, mid - 1);
          else
              return binarySearch(data, target, mid + 1, high);
  }
}
```

-

Complexity of Binary Search (Or, why binary search is good)



- number of candidates starts with n
- next call the candidates size reduced by half, and so on.

$$n, \frac{n}{2}, \frac{n}{2^2}, \ldots, \frac{n}{2^h} \tag{2}$$

It stops when

$$\frac{n}{2^h} \approx 1 \tag{3}$$

$$h \approx \log n \tag{4}$$

Analysis using the substitution method

1. Write the recurrence relation

$$T(n) = \begin{cases} 1, & \text{if n=1} \\ T(n-1)+1, & \text{if } n > 1 \end{cases} \tag{5}$$

2. Guess its closed-form upper bound

$$T(n) = O(n) \tag{6}$$

   i.e., there exists a constant c such that

$$T(n) \leq cn \tag{7}$$

3. Prove it using mathematical induction

Prove that $T(n) = O(n)$

By the definition of the Big O notation, It is equivalent to proving that there exists a constant c such that

$$T(n) \leq cn \tag{8}$$

Prove using mathematical induction

- Base case: when n=1. it is trivially true: $T(1) = 1$.
- Induction step: suppose that it is true for $n-1$. i.e.,

$$T(n-1) \leq c(n-1) \tag{9}$$

We need to prove that it is also true for n. i.e.,

$$T(n) \leq cn \tag{10}$$

The proof goes as follows:

$$\begin{aligned}
T(n) &= T(n-1) + 1 && (11)\\
&\leq c(n-1) + 1 && \text{(by induction assumption)}\\
&= cn - c + 1 && (12)\\
&= cn - (c-1) && (13)\\
&\leq cn && \text{(this is true when } c \geq 1\text{)}\\
& && (14)
\end{aligned}$$

An incorrect proof

- You must prove the exact form of the induction hypothesis.

$$T(n) = 2T(n/2) + n \tag{15}$$

- Incorrect proof that T(n)=O(n):
- Guess that

$$T(n) = O(n) \tag{16}$$

i.e., There exists c such that

$$T(n) \leq cn \tag{17}$$

- Induction step: Assume that it is true for $n/2$, i.e., $T(n/2) \leq cn/2$.

Incorrect proof (cont.)

- We need to prove that it is also true for n. i.e.,

$$\begin{aligned} T(n) &= 2T(n/2) + n && \text{(recurrence relation)} \\ &\leq 2(cn/2) + n && \text{(inductive assumption)} \\ &= cn + n \\ &= O(n). && \text{(Definition of Big O)} \end{aligned}$$

- What is wrong here is that we need to prove $T(n) \leq cn$, not $T(n) \leq (c+1)n$.
- We can not use Big O notation here.
- We need to have a fixed constant c, not a moving c.

A Few Examples of Recurrence Relations and Algorithms

Recurrence Relation	Solution	Example Algorithms
T(n)=T(n-1)+1	$T(n) = O(n)$	factorial
T(n)=T(n-1)+n	$T(n) = O(n^2)$	selection/insertion sort
T(n)=2T(n-1)+1	$T(n) = O(2^n)$	fib (bad)
T(n)=T(n/2)+1	$T(n) = O(\log n)$	binary search
T(n)=2T(n/2)+1	$T(n) = O(n)$	binary sum
T(n)=2T(n/2)+n	$T(n) = O(n \log n)$	merge sort

Reverse an array

-

0	1	2	3	4	5	6	7
4	3	6	2	7	8	9	5

5	3	6	2	7	8	9	4

5	9	6	2	7	8	3	4

5	9	8	2	7	6	3	4

5	9	8	7	2	6	3	4

```
void reverseArray(int[] data,int low,int high) {
    if (low < high) {
       int temp = data[low];
      data[low] = data[high];
      data[high] = temp;
      reverseArray(data, low + 1, high - 1);
    }
}
```

Inefficient recursive programs:Fibonacci example

$$fib(n) = \begin{cases} 0, & \text{if n=0} \\ 1, & \text{if n=1} \\ fib(n-1) + fib(n-2), & n > 1 \end{cases} \tag{18}$$

```
long fibonacciBad(int n) {
   if (n <= 1)      return n;
   else return fibonacciBad(n-2) + fibonacciBad(n-1);
 }
```

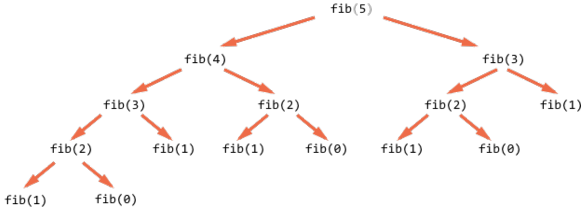
Complexity of Fib

```
long fibonacciBad(int n) {
   if (n <= 1)      return n;
   else return fibonacciBad(n-2) + fibonacciBad(n-1);
 }
```

- c_n: number of calls

$$c_2 = 2 + 1 = 3 \tag{19}$$
$$c_3 = c_2 + c_1 + 1 = 3 + 1 + 1 = 5 \tag{20}$$
$$c_4 = c_3 + c_2 + 1 = 5 + 3 + 1 = 9 \tag{21}$$
$$\ldots \tag{22}$$

- $c_n > 2c_{n-2} > 2 \times 2c_{n-4} > \ldots 2^{n/2}$



Fib example

- Recurrence relation

$$T(n) = T(n-1) + T(n-2) + 1 \tag{23}$$

- Guess: We simplify it into

$$T(n) \geq 2T(n-2) + 1 \tag{24}$$

Then

$$
\begin{aligned}
T(n) &= T(n-1) + T(n-2) + 1 && (25)\\
&\geq 2T(n-2) + 1 && (26)\\
&= 2 * (2T(n-2*2) + 1) + 1 && (27)\\
&= 2 * 2T(n-2*2) + 2 + 1 && (28)\\
&\dots && (29)\\
&= 2^{n/2}(T(n-2*n/2) + \cdots + 2 + 1 && (30)\\
&= 2^{n/2}(T(0) + \cdots + 2 + 1 && (31)\\
&= 2^{n/2} + \cdots + 2 + 1 && (32)
\end{aligned}
$$

So it should be bigger than $2^{n/2}$. We can also show that it is smaller than $O(2^n)$.

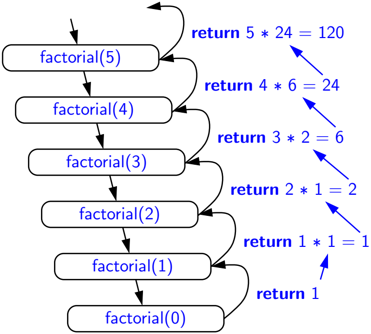
Good Fib program

```java
public static long[] fibonacciGood(int n) {
  if (n <= 1) {
    long[] answer = {n, 0};
    return answer;
  } else {
    long[] temp = fibonacciGood(n - 1);
    long[] answer = {temp[0] + temp[1], temp[0]};
    return answer;
  }
}
```

What is the complexity?

Drawbacks of recursion

- Implementation of recursion relies on a stack data structure
- Stack incur additional memory cost
- Hence there is a need to remove recursion before running
- One type of recursion is particularly easy to remove– tail recursion



Tail recursion

- A recursion is a tail recursion if the recursive call is the very last operation
- the return value of the recursive call (if any) immediately returned by the enclosing recursion.

Tail recursion example

```
void reverseArray(int[] data,int low,int high) {
    if (low < high) {
       int temp = data[low];
      data[low] = data[high];
      data[high] = temp;
      reverseArray(data, low + 1, high - 1);
    }
}
```

Example for not tail-recursive

```
int factorial(int n) {
   if (n == 0) return 1;
   else  return n * factorial(n-1);
 }
```

It is not tail-recursive because there is a multiplication operation after the recursive call

Remove Tail Recursion-arrayReverse

```
public static void reverseArray(int[] data, int low, int high) {
  if (low < high) {
    swap(data, low, high);
    low++;      high--;
    reverseArray(data, low, high);
  }
}
```

Expand the recursion:

```
public static void reverseArray(int[] data, int low, int high) {
  if (low < high) {
    swap(data, low, high);
    low++;
    high--;
    if (low < high) {
      swap(data, low, high);
      low++; high--;
      reverseArray(data, low, high);
     }
  }
}

public static void reverseIterative(int[] data) {
  int low = 0, high = data.length - 1;
  while (low < high) {
    swap(data, low, high);
    low++;      high--;
  }
}
```

Remove Tail Recursion-binarySearch

```
boolean binarySearchIterative(int[] data, int target) {
    int low = 0;
    int high = data.length - 1;
    while (low <= high) {
      int mid = (low + high) / 2;
      if (target == data[mid])    return true;
      else if (target < data[mid])
        high = mid - 1;
       else
        low = mid + 1;
    }
    return false;
 }
```

Takeaways

- What is recursion, why recursion.
- Linear and binary recursion, how to write recursive programs, how to avoid infinite recursive calls.
- Complexity analysis of recursive programs. substitution method
- Tail recursion, tail recursion removal.
- Readings: Goodrich et al., P189-P221.