COMP 251: Recurrences

Jérôme Waldispühl & Giulia Alberini
School of Computer Science
McGill University

Based on slides from Hatami, Bailey, Stepp & Martin, Snoeyink.
Outline

• Introduction: Thinking recursively

• Definition

• Examples:
  o Binary search
  o Fibonacci numbers
  o Merge sort (bonus: insertion sort)
  o Quicksort

• Running time

• Substitution method
Course credits

c(x) = total number of credits required to complete course x

\[
c(\text{COMP462}) = ?
= 3 \text{ credits} + \#\text{credits for prerequisites}
\]

COMP462 has 2 prerequisites: COMP251 & MATH323

\[
= 3 \text{ credits} + c(\text{COMP251}) + c(\text{MATH323})
\]

\text{The function } c \text{ calls itself twice}

\[
c(\text{COMP251}) = ? \quad c(\text{MATH323}) = ?
\]

\[
c(\text{COMP251}) = 3 \text{ credits} + c(\text{COMP250}) \quad \text{COMP250 is a prerequisite}
\]

Substitute \( c(\text{COMP251}) \) into the formula:

\[
c(\text{COMP462}) = 3 \text{ credits} + 3 \text{ credits} + c(\text{COMP250}) + c(\text{MATH323})
\]

\[
c(\text{COMP462}) = 6 \text{ credits} + c(\text{COMP250}) \quad + c(\text{MATH323})
\]
Course credits

c(COMP462) = 6 credits + c(COMP250) + c(MATH323)
  c(COMP250) = ?  c(MATH323) = ?
  c(COMP250) = 3 credits # no prerequisite

c(COMP462) = 6 credits + 3 credits + c(MATH323)
  c(MATH240) = ?
  c(MATH240) = 3 credits + c(MATH141)

c(COMP462) = 9 credits + 3 credits + c(MATH141)
  c(MATH141) = ?
  c(MATH141) = 4 credits # no prerequisite

c(COMP462) = 12 credits + 4 credits = 16 credits
A noun phrase is either
- a noun, or
- an adjective followed by a noun phrase

<noun phrase> → <noun> OR <adjective> <noun phrase>

```
<noun phrase>
  \____/<adjective>
     \___/ <adjective>
        \___/<noun>
            |<noun>
             |  big
             |  blue
             |  boat
```
Aside on grammars

- The previous slide was a simplified example of how we can use grammars to define sentences
- Grammars also exist outside of natural language.
- They are commonly used in computer science to represent as tree any kind of sequence of words, events, nucleotides
- Trees contains more information than the string alone.
- You can read more here: https://en.wikipedia.org/wiki/Formal_grammar
- You will discuss this in more detail in COMP 330!

Definitions

**Recursive definition:**
A definition that is defined in terms of **itself**.

**Recursive method:**
A method that calls **itself** (directly or indirectly).

**Recursive programming:**
Writing methods that call **themselves** to solve problems recursively.
Why using recursions?

- "cultural experience" - A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)
- Many programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)
- Recursions can be a good alternative to iteration (loops).
Definition

Definition (recurrence):
A **recurrence** is a function that is defined in terms of
• one or more base cases, and
• itself, but with smaller arguments.

Examples:

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T(n-1) + 1 & \text{if } n > 1 
\end{cases}
\]

Many technical issues:
• Floors and ceilings
• Exact vs. *asymptotic* functions
• Boundary conditions

Note: we will usually express the solution of the recurrence using the *asymptotic* notation.
Iterative algorithms

Definition (iterative algorithm): Algorithm where a problem is solved by iterating (going step-by-step) through a set of commands, often using loops.

Algorithm: power(a,n)
Input: non-negative integers a, n
Output: $a^n$

```
product ← 1
for i = 1 to n do
    product ← product * a
return product
```

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
<td>1</td>
<td>a</td>
<td>$a \times a = a^2$</td>
<td>$a^2 \times a = a^3$</td>
<td>$a^3 \times a = a^4$</td>
</tr>
</tbody>
</table>
Recursive algorithms

Definition (Recursive algorithm): An algorithm is recursive if in the process of solving the problem, it calls itself one or more times.

Algorithm: `power(a, n)`
Input: non-negative integers `a`, `n`
Output: `a^n`
if `(n=0)` then
    return 1
else
    return `a * power(a, n-1)`
Example

power(7,4) calls

\[ 7 \times \text{power}(7,3) \]

\[ 7 \times \text{power}(7,2) \]

\[ 7 \times \text{power}(7,1) \]

\[ 7 \times 1 = 7 \]

\[ 7 \times 7 = 49 \]

\[ 7 \times 49 = 343 \]

\[ 7 \times 343 = 2041 \]
Algorithm structure

Every recursive algorithm involves at least 2 cases:

**base case**: A simple occurrence that can be answered directly.

**recursive case**: A more complex occurrence of the problem that cannot be directly answered but can instead be described in terms of *smaller occurrences* of the same problem.

Some recursive algorithms have more than one base or recursive case, but all have at least one of each.

A crucial part of recursive programming is identifying these cases.
Algorithm binarySearch(array, start, stop, key)

**Input:**
- A sorted array
  - the region \([\text{start} , \ldots , \text{stop}]\) (inclusively) to be searched
  - the key to be found

**Output:** returns the index at which the key has been found, or returns -1 if the key is not in array \([\text{start} , \ldots , \text{stop}]\).

**Example:** Does the following sorted array \(A\) contains the number 6?

\[
A = \begin{bmatrix}
1 & 1 & 3 & 5 & 6 & 7 & 9 & 9
\end{bmatrix}
\]

Call: \text{binarySearch}(A, 0, 7, 6)
Binary search example

We are splitting the array in two at each step.

We are splitting the array in two at each step.

1 1 3 5 6 7 9 9

Search [0:7]

5 < 6 $\Rightarrow$ look into right half of the array

1 1 3 5 6 7 9 9

Search [4:7]

7 > 6 $\Rightarrow$ look into left half of the array

1 1 3 5 6 7 9 9

Search [4:4]

6 is found. Return 4 (index)
Binary Search Algorithm

```c
int bsearch(int[] A, int i, int j, int x) {
    if (i <= j) { // the region to search is non-empty
        int e = (i+j)/2;
        if (A[e] > x) {
            return bsearch(A, i, e-1, x);
        } else if (A[e] < x) {
            return bsearch(A, e+1, j, x);
        } else {
            return e;
        }
    } else { return -1; } // value not found
}
```
Fibonacci numbers

\[
\begin{align*}
\text{Fib}_0 &= 0 \quad \text{base case} \\
\text{Fib}_1 &= 1 \quad \text{base case} \\
\text{Fib}_n &= \text{Fib}_{n-1} + \text{Fib}_{n-2} \quad \text{for } n > 1 \quad \text{recursive case}
\end{align*}
\]

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib_i</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
</tr>
</tbody>
</table>
Recursive algorithm

Compute Fibonacci number \( n \) (for \( n \geq 0 \))

```java
public static int Fib(int n) {
    if (n <= 1) {
        return n;  // Can handle both
                    // base cases together
    }
    return Fib(n-1) + Fib(n-2);  // Recursive case
                                 // (2 recursive calls)
}
```

**Note:** The algorithm follows the definition of Fibonacci numbers.
Recursion is not always efficient!

Note: This is a recursion tree

Question: When computing Fib(n), how many times are Fib(0) or Fib(1) called?
Designing recursive algorithms

• To write a recursive algorithm:
  – Find how the problem can be broken up in one or more smaller problems of the same nature
  – Remember the base case!

• Naive implementation of recursive algorithms may lead to prohibitive running time.
  – Naive Fibonacci ⇒ $O(\phi^n)$ operations
  – Better Fibonacci ⇒ $O(\log n)$ operations

• Usually, better running times are obtained when the size of the sub-problems are approximately equal.
  – $\text{power}(a,n) = a \times \text{power}(a,n-1) \Rightarrow O(n)$ operations
  – $\text{power}(a,n) = (\text{power}(a,n/2))^2 \Rightarrow O(\log n)$ operations
Sorting problem

**Problem:** Given a list of $n$ elements from a totally ordered universe, rearrange them in ascending order.

Classical problem in computer science with many different algorithms (bubble sort, merge sort, quick sort, etc.)
We are expanding a sorted region by traversing the list from left to right and swapping from right to left until the key is at a right place.
Insertion sort (principle)

$n$ elements already sorted
New element to sort

$n+1$ elements sorted
Insertion sort (iterative algorithm)

For $i \leftarrow 1$ to $\text{length}(A) - 1$
  $j \leftarrow i$
  while $j > 0$ and $A[j-1] > A[j]$
    swap $A[j]$ and $A[j-1]$
    $j \leftarrow j - 1$
  end while
end for

• Iterative method to sort objects.
• Relatively slow, we can do better using a recursive approach!
Merge Sort

Sort using a divide-and-conquer approach:

• **Divide:** Divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each.

• **Conquer:** Sort the two subsequences recursively using merge sort.

• **Combine:** Merge the two sorted subsequences to produce the sorted answer.

Note: We will speak more of divide-and-conquer techniques when we will discuss algorithm design.
Merge sort (principle)

Recursive case

- Unsorted array A with $n$ elements
- Split A in half $\rightarrow$ 2 arrays L and R with $n/2$ elements
- Sort L and R
- Merge the two sorted arrays L and R

Base case: Stop the recursion when the array is of size 1.
Why? Because the array is already sorted!
**Merge-Sort** \((A, p, r)\)

**INPUT:** a sequence of \(n\) numbers stored in array \(A\)

**OUTPUT:** an ordered sequence of \(n\) numbers

```plaintext
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
1   if \(p < r\)       
2     then \(q \leftarrow \lfloor (p+r)/2 \rfloor\)       
3       MergeSort (A, p, q)       
4       MergeSort (A, q+1, r)       
5       Merge (A, p, q, r)   // merges A[p..q] with A[q+1..r]

Initial Call: MergeSort(A, 1, n)
```
Procedure Merge

Input: Array containing sorted subarrays \(A[p..q]\) and \(A[q+1..r]\).

Output: Merged sorted subarray in \(A[p..r]\).

Sentinels, to avoid having to check if either subarray is fully copied at each step.
QuickSort

Quicksort(A, p, r)
  if p < r then
    q := Partition(A, p, r);
    Quicksort(A, p, q – 1);
    Quicksort(A, q + 1, r)
  fi

Partition(A, p, r)
  x, i := A[r], p – 1;
  for j := p to r – 1 do
    if A[j] ≤ x then
      i := i + 1;
    fi
  od;
  A[i + 1] ↔ A[r];
  return i + 1

Partition stores all the elements lesser than the pivot, then the pivot, then the other elements
Algorithm analysis

Q: How to estimate the running time of a recursive algorithm?
A:
1. Define a function $T(n)$ representing the time spent by your algorithm to execute an entry of size $n$
2. Write a recursive formula computing $T(n)$
3. Solve the recurrence

Notes:
• $n$ can be anything that characterizes accurately the size of the input (e.g., size of the array, number of bits)
• We count the number of elementary operations (e.g., addition, shift) to estimate the running time.
• We usually compute an upper bound rather than exact count.
• We will introduce later a general method to solve this.
Examples (binary search)

```c
int bsearch(int[] A, int i, int j, int x) {
    if (i<=j) { // the region to search is non-empty
        int e = (i+j)/2;
        if (A[e] > x) { return bsearch(A,i,e-1,x); }
        elif (A[e] < x) { return bsearch(A,e+1,j,x); }
        } else { return e; }
    } else { return -1; } // value not found
}
```

\[ T(n) = \begin{cases} 
    c & \text{if } n = 1 \\
    T\left(\frac{n}{2}\right) + c' & \text{if } n > 1 
\end{cases} \]

Notes:
• \( n \) is the size of the array
• Formally, we should use \( \leq \) rather than =
Example (naïve Fibonacci)

```java
public static int Fib(int n) {
    if (n <= 1) { return n; }
    return Fib(n-1) + Fib(n-2);
}
```

\[
T(n) = \begin{cases} 
    c & \text{if } n \leq 1 \\
    T(n-1) + T(n-2) + c' & \text{if } n > 1 
\end{cases}
\]

What are the value of \(c\) and \(c'\) ?
• If \(n \leq 1\) there is only one comparison thus \(c=1\)
• If \(n > 1\) there is one comparison and one addition thus \(c'=2\)

Notes:
• we neglect other constants
• We can approximate \(c\) and \(c'\) with an asymptotic notation \(O(1)\)
Example (Merge sort)

\[
\text{MergeSort}(A, p, r)
\]
\[
\text{if } (p < r) \text{ then}
\]
\[
q \leftarrow \lfloor (p+r)/2 \rfloor
\]
\[
\text{MergeSort}(A, p, q)
\]
\[
\text{MergeSort}(A, q+1, r)
\]
\[
\text{Merge}(A, p, q, r)
\]

• Base case: constant time \(c\)
• Divide: computing the middle takes constant time \(c'\)
• Conquer: solving 2 subproblems takes \(2 \cdot T(n/2)\)
• Combine: merging \(n\) elements takes \(k \cdot n\)

\[
T(n) = \begin{cases} 
  c & \text{if } n = 1 \\
  2 \cdot T\left(\frac{n}{2}\right) + k \cdot n + c + c' & \text{if } n > 1 
\end{cases}
\]