COMP 251: Recurrences

Jérôme Waldispühl & Roman Sarrazin-Gendron
School of Computer Science
McGill University

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

• Hello!

• Based on feedback from the past year, I will answer questions in the chat periodically, but I will not interrupt the flow of the lecture.

• Please ask your question on Ed! Some questions deserve more time than we can afford during lectures, so it’s best to continue that conversation on the discussion board.
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

\[ c(x) = \text{total number of credits required to complete course } x \]

\[ c(\text{COMP462}) = ? \]

\[ = 3 \text{ credits } + \#\text{credits for prerequisites} \]

COMP462 has 2 prerequisites: COMP251 & MATH323

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

*The function \( c \) calls itself twice*

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

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

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

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

\[ c(\text{COMP462}) = 6 \text{ credits } + c(\text{COMP250}) + c(\text{MATH323}) \]
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(MATH323) = ?
  c(MATH323) = 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
Recursive definition

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>       <noun phrase>
    <adjective>     <noun phrase>
      <noun>
        big           black          dog
```
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 are typically easier to work with than strings.
• 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)
• Recursion is often a good alternative to iteration (loops).
Definition (recurrence):

A **recurrence** is a function is defined in terms of
- one or more base cases, and
- itself, with smaller arguments.

Examples:

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

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
T\left(\frac{n}{3}\right) + T\left(\frac{2\cdot n}{3}\right) + n & \text{if } n > 1
\end{cases}
\]

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

Note: we usually express both the recurrence and its solution using **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>$a^2$</td>
<td>$a^3$</td>
<td>$a^4$</td>
</tr>
</tbody>
</table>
Recursive algorithms

Definition (Recursive algorithm): 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 $a \times \text{power}(a, n-1)$
Example

- \text{power}(7,4) \text{ calls}
  - \text{power}(7,3) \text{ calls}
    - \text{power}(7,2) \text{ calls}
      - \text{power}(7,1) \text{ calls}
        - \text{power}(7,0) \text{ returns } 1
          - \text{returns } 7 \times 1 = 7
            - \text{returns } 7 \times 7 = 49
              - \text{returns } 7 \times 49 = 343
                - \text{returns } 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.
Binary Search

Algorithm binarySearch(array, start, stop, key)

Input: - A sorted array
    - the region start...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[start...stop].

Example: Does the following sorted array A contains the number 6?

A =

    1 1 3 5 6 7 9 9

Call: binarySearch(A, 0, 7, 6)
Binary search example

We are splitting the array in two at each step

Search [0:7]

1 1 3 5 6 7 9 9

5 < 6 ⇒ look into right half of the array

Search [4:7]

1 1 3 5 6 7 9 9

7 > 6 ⇒ look into left half of the array

Search [4:4]

1 1 3 5 6 7 9 9

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

Fib₀ = 0  base case
Fib₁ = 1  base case
Fibₙ = Fibₙ₋₁ + Fibₙ₋₂  for n > 1  recursive case

<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>Fibᵢ</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 ≥ 0)

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

Note: The algorithm follows almost exactly 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.
  – $\text{power}(a,n) = a * \text{power}(a,n-1)$ ⇒ $O(n)$ operations
  – $\text{power}(a,n) = (\text{power}(a,n/2))^2$ ⇒ $O(\log n)$ operations
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.
Insertion sort

$n$ elements already sorted
New element to sort

$n+1$ elements sorted
Insertion sort

For $i \leftarrow 1$ to 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.
Merge Sort - Example

Divide

Merge
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

\[
\text{MergeSort} (A, p, r) \quad // \text{sort } A[p..r] \text{ by divide & conquer}
\]

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

**Procedure Merge**

\[\text{Merge}(A, p, q, r)\]

1. \(n_1 \leftarrow q - p + 1\)
2. \(n_2 \leftarrow r - q\)
3. for \(i \leftarrow 1\) to \(n_1\) do \(L[i] \leftarrow A[p + i - 1]\)
4. for \(j \leftarrow 1\) to \(n_2\) do \(R[j] \leftarrow A[q + j]\)
5. \(L[n_1+1] \leftarrow \infty\)
6. \(R[n_2+1] \leftarrow \infty\)
7. \(i \leftarrow 1\)
8. \(j \leftarrow 1\)
9. for \(k \leftarrow p\) to \(r\) do if \(L[i] \leq R[j]\) then \(A[k] \leftarrow L[i]\)
10. \(i \leftarrow i + 1\)
11. else \(A[k] \leftarrow R[j]\)
12. \(j \leftarrow j + 1\)
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 often aim to compute an upper bound rather than an exact count.
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);
}
```

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)

MergeSort (A, p, r)
if (p < r) then
    q ← ⌊(p+r)/2⌋
    MergeSort (A, p, q)
    MergeSort (A, q+1, r)
    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 \(k \cdot n\)

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

How to solve a recursive equation?
1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Example:

\[ T(n) = \begin{cases} 
1 & \text{if } n = 0 \\
2 \cdot T(n - 1) & \text{if } n > 0 
\end{cases} \]

Guess: \( T(n) = 2^n \) (remember the fibonacci recursive tree)

Base case: \( T(0) = 2^0 = 1 \) ✓

Inductive case:
Assume \( T(n) = 2^n \) until rank \( n-1 \), then show it is true at rank \( n \).
\[ T(n) = 2 \cdot T(n - 1) = 2 \cdot 2^{n-1} = 2^n \] ✓
Running time of binary search

\[ T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
T\left(\frac{n}{2}\right) + 1 & \text{if } n > 1 
\end{cases} \]

Note: set the constant \( c=0 \) and \( c'=1 \)

**Guess:** \( T(n) = \log_2 n \)

**Base case:** \( T(1) = \log_2 1 = 0 \) \( \checkmark \)

**Inductive case:**

Assume \( T(n/2) = \log_2(n/2) \)

\[ T(n) = T(n/2) + 1 = \log_2(n/2) + 1 \]
\[ = \log_2(n) - \log_2 2 + 1 = \log_2 n \] \( \checkmark \)
Running time of Merge Sort

We use a simplified version:

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2 \cdot T \left( \frac{n}{2} \right) + n & \text{if } n > 1 
\end{cases} \]

Simulation:

<table>
<thead>
<tr>
<th>n</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
<th>...</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(n)</td>
<td>1</td>
<td>5</td>
<td>15</td>
<td>39</td>
<td>95</td>
<td>223</td>
<td>511</td>
<td>...</td>
<td>?</td>
</tr>
</tbody>
</table>
Running time of Merge Sort

The graph shows the running time of Merge Sort. The x-axis represents the input size (n), and the y-axis represents the running time. There are three lines on the graph:

- **T(n)**: The blue line represents the running time of Merge Sort, which grows linearly with the input size.
- **Linear time**: The red line represents linear time, which grows at a much slower rate than Merge Sort.
- **Quadratic**: The green line represents quadratic time, which grows significantly faster than linear time.

The graph illustrates that Merge Sort has a linear time complexity, making it efficient for sorting large datasets compared to algorithms with quadratic time complexity.
Running time of Merge Sort

Remember the recursive case: \( T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n \) if \( n > 1 \)

Guess: \( T(n) = n \cdot \log n + n \)

Base case: \( T(1) = 1 \cdot \log 1 + 1 = 1 \)

Inductive case:
Assume \( T(n/2) = \frac{n}{2} \cdot \log \left(\frac{n}{2}\right) + \frac{n}{2} \)

\[
T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n = 2 \cdot \left(\frac{n}{2} \cdot \log \left(\frac{n}{2}\right) + \frac{n}{2}\right) + n \\
= n \cdot (\log n - \log 2) + n + n = n \cdot \log n - n + 2 \cdot n \\
= n \cdot \log n + n
\]

Note: Here, we use an exact function but it will become simpler when we will use the asymptotic notations
Recursion tree

**Objective:** Another method to represent the recursive calls and evaluate the running time (i.e. #operations).

- Value of the node is the #operation made by merge
- One branch in the tree per recursive call
- WLOG, we assume that $n$ is a power of 2

How many operations are required to compute $T(N) = 2 \cdot T(N/2)$?
Recursion tree

Total # operations = total of all rows = $n \times \text{height of the tree}$

Q: How many time can we split in half $n$? A: $\log n$ times

Thus the total # operations for $T(N) = 2 \cdot T(N/2)$ is $n \cdot \log_2 n$
Announcements

• If you haven’t been able to connect to Ed discussion, use the link on the Content section on Mycourses.

• Tutorials and office hours will be announced next week, tentatively.

• Next lecture: Proofs by induction and loop invariants