Mathematical Software Programming (02635)

Lecture 11 — November 20, 2025

Instructor: Martin S. Andersen

Fall 2025

Announcements

Course evaluation

Survey open from November 18 until November 28 (both days included).

Your feedback is valued

- What activities/exercises helped you learn the material?
- Which concepts/exercises/lectures/assignments/... were difficult? What can be improved?
- Is there anything that you expected to learn in this course but did not?
- Do you feel that your programming skills have improved throughout the course?
- Did the Autolab exercises help you learn?

This week

Topic

- Recursion

Learning objectives

- Compare iterative and recursive solutions for simple problems
- Analyze the runtime behavior and the time and space complexity of simple programs

Recursive functions

A recursive function is a function that calls itself during its execution.

Example 1: Factorial (single recursion)

Base case: $f_0 = 1$
Recursive case:

$$f_n = n \cdot f_{n-1}, \quad n \geq 1$$

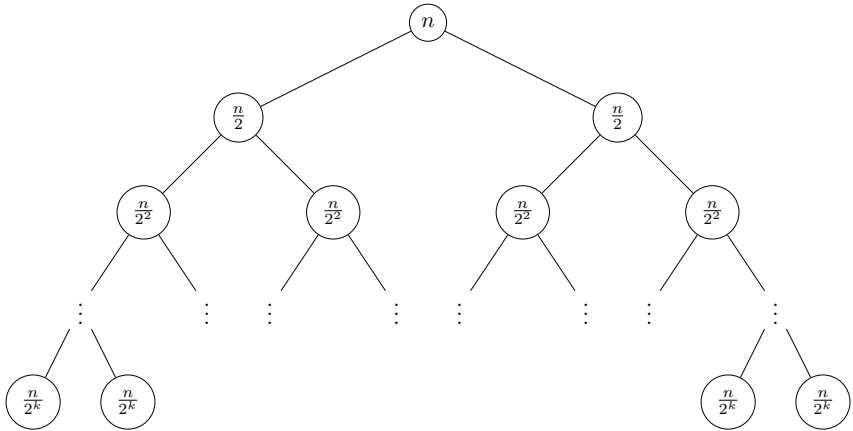
Example 2: Fibonacci numbers (multiple recursion)

Base cases: $f_0 = 0$ and $f_1 = 1$
Recursive case:

$$f_n = f_{n-1} + f_{n-2}, \quad n \geq 2$$

Divide and conquer

Break problem into subproblems and combine answers



Example 1: power function

The function x^n (with $n > 0$ and integer) can be expressed as

$$x^n = \begin{cases} x^{n/2} \cdot x^{n/2} & n \text{ even} \\ x \cdot x^{(n-1)/2} \cdot x^{(n-1)/2} & n \text{ odd} \end{cases}$$

Example 1: power function (version 1)

Non-recursive implementation of power function x^n ($n \geq 0$ integer)

```
double power_v1(double x, unsigned int n) {
   double val = 1.0;
   for (int i=0; i<n; i++) val *= x;
   return val;
}
```

What is the space/time complexity?

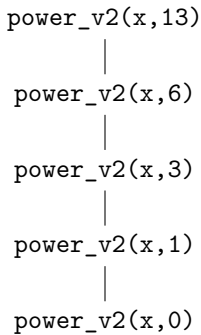
Example 1: power function (version 2)

Recursive implementation of power function x^n ($n \geq 0$ integer)

```
double power_v2(double x, unsigned int n) {
    double val;
    if (n == 0)
       return 1.0;
    val = power_v2(x, n/2);
    if (n%2 == 0)  // n is even
       return val*val;
    else           // n is odd
       return x*val*val;
}
```

What is the space/time complexity?

Example 1: power function (version 2)



What is the depth of the recursion as a function of n?

Example 1: power function (version 3)

Non-recursive implementation of power function x^n ($n \geq 0$ integer)

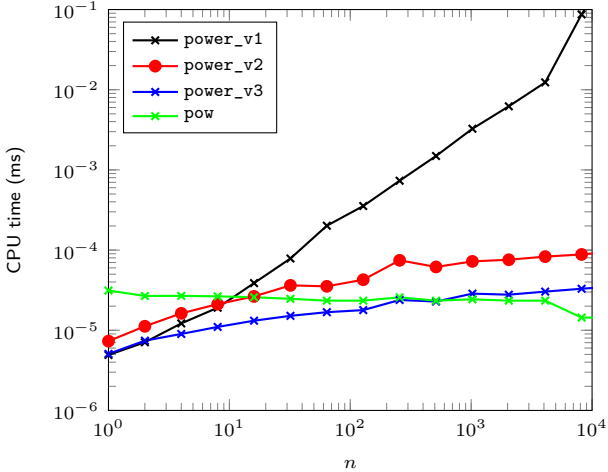
```
double power_v3(double x, unsigned int n) {
   double val = 1.0;
   while(n != 0){
      if(n%2 == 0) {    // n is even
         x = x*x;
         n = n/2;
      }
      else {            // n is odd
         val = val*x;
         n = n-1;
      }
   }
   return val;
}
```

What is the space/time complexity?

Example 1: complexity

Function	Space complexity	Time complexity
`power_v1`	$O(1)$	$O(n)$
`power_v2`	$O(\log n)$	$O(\log n)$
`power_v3`	$O(1)$	$O(\log n)$

Example 1: experiment



Example 2: summation

Recursively divide summation into two partial sums

$$\sum_{i=1}^{n} x_i = \sum_{i=1}^{\lfloor n/2 \rfloor} x_i + \sum_{i=\lfloor n/2 \rfloor + 1}^{n} x_i$$

```
#define Nbase 128
double recursive_sum(int n, const double * x) {
    double psum = 0.0;
    if (n > Nbase)
        return recursive_sum(n/2,x)+recursive_sum(n-(n/2),x+n/2);
    for (int k=0;k<n;k++) psum += x[k];
    return psum;
}
```

- also known as *pairwise summation* or *cascade summation*
- can be parallelized
- worst-case error bound is *better* than for sequential summation ($O(\log n)$ vs. $O(n)$)

Example 3: Walsh–Hadamard transform

Define $H_k \in \mathbb{R}^{2^k \times 2^k}$ as

$$H_k = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}, \qquad k \geq 1$$

with $H_0 = 1$

$$H_0 = 1, \qquad H_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \qquad H_2 = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}, \quad \dots$$

H_k is symmetric and orthogonal (involutory): $H_k H_k = I$

Example 3: Walsh–Hadamard transform

Divide and conquer

$$H_k x = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad x_1, x_2 \in \mathbb{R}^{2^{k-1}}$$

Recursive formulation of $x \leftarrow H_k x$

$$\begin{aligned} x_1 &\leftarrow \frac{1}{\sqrt{2}} H_{k-1} x_1 \\ x_2 &\leftarrow \frac{1}{\sqrt{2}} H_{k-1} x_2 \\ x_1 &\leftarrow x_1 + x_2 \\ x_2 &\leftarrow x_1 - 2x_2 \end{aligned}$$

Cost of $x \leftarrow H_k x$ grows as $O(n \log(n))$ where $n = 2^k$ since

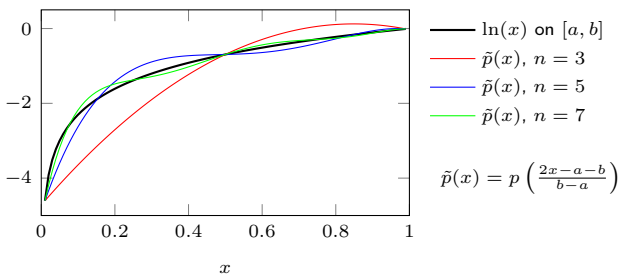
$$C_k = 2C_{k-1} + 5 \cdot 2^{k-1} = 2^k \left(1 + \frac{5k}{2}\right), \qquad C_0 = 1$$

Example 4: Chebyshev series expansion

$$p(x) = \sum_{k=0}^{n} c_k T_k(x), \quad x \in [-1, 1].$$

- $c = (c_0, c_1, \ldots, c_n)$ is a given vector of length $n + 1$
- $T_k(x)$ is the kth Chebyshev polynomial of the first kind, defined recursively as

$$T_0(x) = 1, \qquad T_1(x) = x, \qquad T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x), \quad k \geq 1.$$



Example 4: Chebyshev series expansion

Recursive definition of Chebyshev polynomials can be expressed in matrix form:

$$\underbrace{\begin{bmatrix} 1 & & & & & & \\ -2x & 1 & & & & & \\ 1 & -2x & 1 & & & & \\ & 1 & -2x & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & -2x & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} T_0(x) \\ T_1(x) \\ T_2(x) \\ T_3(x) \\ \vdots \\ T_n(x) \end{bmatrix}}_{t} = \underbrace{\begin{bmatrix} 1 \\ -x \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}}_{r}$$

Clenshaw's algorithm uses this relation to evaluate $p(x)$ efficiently:

$$f(x) = c^T t = c^T L^{-1} r = b^T r = b_0 - b_1 x, \qquad L^T b = c$$

- We only need the first two entries of $b = L^{-T} c = (b_0, b_1, \ldots, b_n)$
- Requires $O(n)$ operations and $O(1)$ space

Today's exercises

Analyze the time and space complexity of recursive implementation of Fibonacci function

```
unsigned long fibonacci(unsigned long n) {
    unsigned long Fn;
    if ( n == 0 )
        Fn = 0;
    else if ( n == 1 )
        Fn = 1;
    else
        Fn = fibonacci(n-1) + fibonacci(n-2);
    return Fn;
}
```

Today's exercises (cont.)

Analyze call stack (aka program stack)