Recursion

Instructor: Meng-Fen Chiang

COMPCSI220: WEEK 9

THE UNIVERSITY OF AUCKLAND
Te Whare Wananga o Tamaki Makaurau
NEW ZEALAND

Slides adapted from Mark Wilson, Georgy Gimel'farb, Simone Linz, Tanya Gvozdeva, and Sathiamoorthy Manoharan

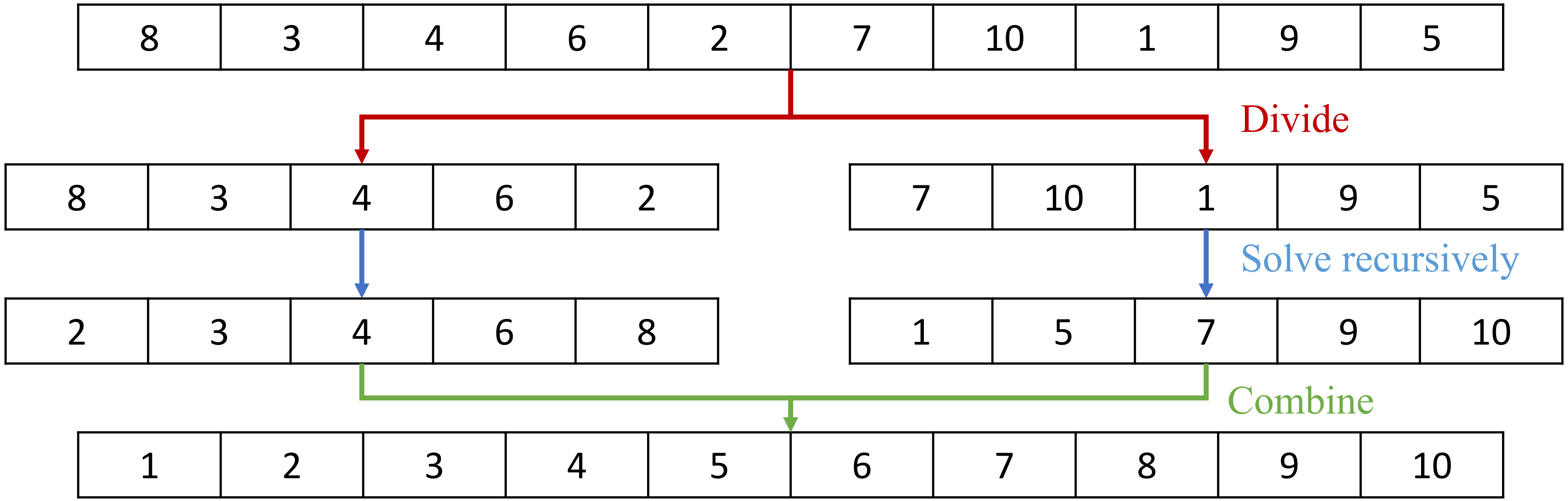
OUTLINE

- Recursion Examples
- Time Complexity Analysis
- Recursion Implementation I: Factorial
- Recursion Implementation II: Factorial
- Complexity Analysis

factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
= 120

Recursion: Illustrative Example

- **Divide** a large problem into smaller subproblems;
- **Recursively solve** each subproblem, then
- **Combine** solutions of them to solve the original problem.



Divide and Conquer: Sorting Algorithms

- **General idea**: divide the input list into two sublists, recursively sorts each sublist, and then combines the sorted sublists to sort the original list.

- **Mergesort** (J. von Neumann, 1945): dividing the list into halves, leave most of the work in combining. Combine the recursively sorted lists with a merge procedure.

- **Quicksort** (C. A. R. Hoare, 1962): most of the work is done in dividing the list, combining is straightforward. Use a pivot element and divide the list into two sublists with all elements smaller and larger than the pivot. Then the two sublists are sorted recursively.

- Each is used as the basis for built-in sorting algorithms in common programming languages.

Time Complexity Analysis

- Running time by a recurrence relation accounts for
  - The size and the number of the subproblems
  - The cost of dividing the problem and the cost of combining the results of subproblems

- Recurrence relation: $F(n) = \psi\big(F(n_0), F(n_1), \dots, F(n_k)\big)$, k≥0 defines a function that calls itself.
  - Non-circular definition: $n > n_k > \cdots > n_0$
  - The recursion stops at some base cases, such as $F(0)$ or $F(1)$

- Example: Factorial with base cases $F(1) = 1$, and the recurrence relation $F(n) = n \times F(n-1)$ for $n \geq 2$

Implicit and Closed-form Formula

- A recurrence relation can either be written in
  - an implicit formula or
  - a closed-form (explicit) formula

- Example

n	0	1	2	3	4	5	6	7	…
$F(n)$	1	2	4	8	16	32	64	128	…

  - Implicit formula: $F(n) = 2F(n-1)$, $F(0)$=1
  - Closed-form formula: $F(n) = 2^n$, $F(0)$=1

Top-down Telescoping

- Consider an implicit formula $T(n) = 2T(n-1) + 1, T(1) = 1$, we can "telescope" the recurrence relations by recursive substitutions:

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

$$
\begin{array}{rcrcl}
T(n) & - & 2T(n-1) & = & 1 \\
2T(n-1) & - & 4T(n-2) & = & 2 \\
4T(n-2) & - & 8T(n-3) & = & 4 \\
& & \dots\dots & & \\
2^{n-2}T(2) & - & 2^{n-1}\,T(1) & = & 2^{n-2}
\end{array}
$$

Summing all rows and gets $T(n) - 2^{n-1}T(1)$

Closed-form formula:

$$T(n) = 2^{n-1}T(1) + 1 + \cdots + 2^{n-2} = 1 + \cdots + 2^{n-1} = 2^n - 1$$

Bottom-up Guessing

- In this method, we guess a pattern starting from the base case and prove it by mathematical induction.

Mathematical Induction

A useful tool to prove a math statement is true for all integers $n \geq n_0$, where n_0 is a non-negative integer, it has three key steps:

1. **Basis**: Prove that the statement is true for n_0.
2. **Induction hypothesis**: Assume that the statement is true for some $n = k$.
3. **Inductive step from $n = k$ to $k + 1$**: If the induction hypothesis holds, prove that the statement is also true for $k + 1$

Example: Bottom-up Guessing

- Suppose we are given $T(n) = 2T(n-1) + 1, T(1) = 1$. Compute first several numbers in the sequence:
  - $T(1) = 1,\ T(2) = 2T(1) + 1 = 3, T(3) = 2T(2) + 1 = 7\ldots$
  - One may guess $T(n) = 2^n - 1$. Prove this is true with Mathematical Induction.

1. **Basis**: Prove that the statement is true for $n = 1$.
   It is true because $T(1) = 2^1 - 1 = 2 - 1 = 1$
2. **Induction hypothesis**: Assume that $T(k) = 2^k - 1$ is true for some integer $k \geq 1$.
3. **Inductive step from k to $k+1$**:
   $$T(k+1) = 2T(k) + 1 = 2 \cdot \underbrace{(2^k - 1)}_{\substack{\text{Induction hypothesis} \\ T(k) = 2^k - 1}} + 1 = 2^{k+1} - 2 + 1 = \underbrace{2^{k+1} - 1}_{\substack{\text{The formula holds} \\ \text{for all } k \geq 1}}$$

Notes on Closed-form Formula

- Not all the recurrence relations have a closed-form formula.
- Linear recurrences which have the form $F(n) = \sum_{i=1}^{K} a_i F(n-i) + g(n)$ for some fixed K and some constants a_i, $g(n)$ is a function of n.
- In this course, we mainly focus on linear recurrences.
- Following examples are linear recurrences or can be converted in the form of linear recurrences:

Tower of Hanoi: $T(n) = 2T(n-1) + 1, T(1) = 0;$

Searching a linked list: $T(n) = T(n-1) + 1, T(0) = 0;$

Insertion sort: $T(n) = T(n-1) + n, T(0) = 0;$

Mergesort: $T(n) = 2T(n/2) + n, T(1) = 0$ makes sense when n is a power of 2.

Question: How to convert the complexity of a divide and conquer algorithm, e.g., Mergesort, to the form of linear recurrence?

Linear Recurrence: Changing variable

- A simpler case – divide by half
  - $T(n) = T(n/2) + 1, T(1) \ = \ 0$. Makes sense for n a power of 2.

- Changing variable by $n = 2^i$ and $U(i) = T(n) = T\left(2^i\right)$.

- This gives a linear recurrence with variable $i = \log_2 n$:

$$U(i) = T\left(2^{i-1}\right) + 1 = U(i-1) + 1, U(0) = T(2^0) = T(1) = 0$$

- By telescoping or guessing, we can get the closed-form formula of $U(i)$:
  - $U(i) = i = T(n) = \log_2 n$

Linear Recurrence: Changing Variable

- Divide and conquer sorting - Mergesort
  - $T(n) = 2T(n/2) + n, T(1) = 0$. Makes sense for n a power of 2.

- Changing variable by $n = 2^i$ and $U(i) = \frac{T(n)}{n} = \frac{T(2^i)}{2^i}$.

- This gives a linear recurrence with variable $i = \log_2 n$:

$$U(i) = \frac{2T(2^{i-1})}{2^i} + 1 = U(i-1) + 1,\ U(0) = \frac{T(2^0)}{2^0} = T(1) = 0$$

- By telescoping or guessing, we can get the closed-form formula of $U(i)$:

$$U(i) = i = \frac{T(n)}{n} = \log_2 n \Rightarrow T(n) = n \log_2 n$$

OUTLINE

- Recursion Examples
- Time Complexity Analysis
- Recursion Implementation I: Factorial
- Recursion Implementation II: Factorial
- Complexity Analysis

```
factorial(5)
= 5 * factorial(4)
= 5 * 4 * factorial(3)
= 5 * 4 * 3 * factorial(2)
= 5 * 4 * 3 * 2 * factorial(1)
= 5 * 4 * 3 * 2 * 1
= 120
```

Recursion Implementations: Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

```
const fac = (n) => {
   const facT = (n, a) => n <= 1 ? a : facT(n - 1, n * a);
   return facT(n, 1);
};
```

What are the time and space complexities of both implementations?

Factorial: First Implementation

Factorial: First Implementation

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

Call Stack – First Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```



```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Call Stack – First Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

5

```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Call Stack – First Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

4
5

```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Call Stack – First Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

3
4
5

```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Call Stack – First Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

2
3
4
5

```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Call Stack – First Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

1
2
3
4
5

```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Call Stack – First Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

2×1
3
4
5

```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Call Stack – First Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

3×2×1
4
5

```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Call Stack – First Factorial

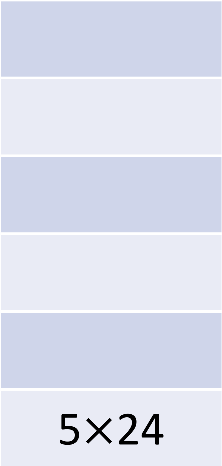
```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

4×6
5

```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Call Stack – First Factorial

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

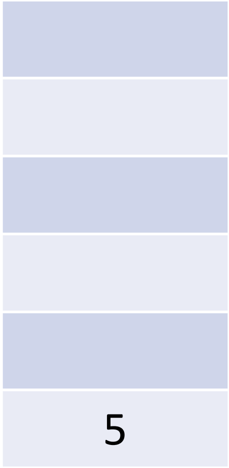



```
p = fac(5)

push 5
call fac


fac:
peek t
if (t <=1) { ret; }
push (t - 1)
call fac
pop rslt
pop t
push t*rslt
```

Factorial: Second Implementation

Call Stack – Second Factorial

```
const fac = (n) => {
   const facT = (n, a) => n <= 1 ? a : facT(n - 1, n * a);
   return facT(n, 1);
};
```




```
p = fac(5)

push 5
push 1
call facT

facT
pop a
pop t
if (t <=1 ) { push a; ret; }
push t-1
push (t * a)
call facT
```

Call Stack – Second Factorial

```
const fac = (n) => {
   const facT = (n, a) => n <= 1 ? a : facT(n - 1, n * a);
   return facT(n, 1);
};
```

1
5

```
p = fac(5)

push 5
push 1
call facT

facT
pop a
pop t
if (t <=1 ) { push a; ret; }
push t-1
push (t * a)
call facT
```

Call Stack – Second Factorial

```
const fac = (n) => {
   const facT = (n, a) => n <= 1 ? a : facT(n - 1, n * a);
   return facT(n, 1);
};
```

5×1
4

```
p = fac(5)

push 5
push 1
call facT

facT
pop a
pop t
if (t <=1 ) { push a; ret; }
push t-1
push (t * a)
call facT
```

Call Stack – Second Factorial

```
const fac = (n) => {
   const facT = (n, a) => n <= 1 ? a : facT(n - 1, n * a);
   return facT(n, 1);
};
```

4×5×1
3

```
p = fac(5)

push 5
push 1
call facT

facT
pop a
pop t
if (t <=1 ) { push a; ret; }
push t-1
push (t * a)
call facT
```

Call Stack – Second Factorial

```
const fac = (n) => {
   const facT = (n, a) => n <= 1 ? a : facT(n - 1, n * a);
   return facT(n, 1);
};
```

3×20
2

```
p = fac(5)

push 5
push 1
call facT

facT
pop a
pop t
if (t <=1 ) { push a; ret; }
push t-1
push (t * a)
call facT
```

Call Stack – Second Factorial

```
const fac = (n) => {
   const facT = (n, a) => n <= 1 ? a : facT(n - 1, n * a);
   return facT(n, 1);
};
```


```
p = fac(5)

push 5
push 1
call facT

facT
pop a
pop t
if (t <=1 ) { push a; ret; }
push t-1
push (t * a)
call facT
```

Call Stack – Second Factorial

```
const fac = (n) => {
   const facT = (n, a) => n <= 1 ? a : facT(n - 1, n * a);
   return facT(n, 1);
};
```

120

```
p = fac(5)

push 5
push 1
call facT

facT
pop a
pop t
if (t <=1 ) { push a; ret; }
push t-1
push (t * a)
call facT
```

Complexity Analysis

- First implementation’s space complexity is O(n)
- Second one’s space complexity is O(1)
- Both have the same time complexity O(n)

Head Recursion

- There is work done after the recursive function call.
  - The return value of the recursive call is multiplied by n and the result is returned by the caller.

```
const fac = (n) => n <= 1 ? 1 : n * fac(n - 1);
```

Tail Recursion

- There is no work done after the recursive call – we return the result of recursion.
- Recursion is at the tail end of all work, and thus called tail recursion.
- This is a lot more space efficient than head recursion.

```
const fac = (n) => {
   const facT = (n, a) => n <= 1 ? a : facT(n - 1, n * a);
   return facT(n, 1);
};
```

SUMMARY

- Recursion Examples
- Time Complexity Analysis
  - Top-down telescoping
  - Bottom-up guessing
- Recursion Implementation I: Factorial
- Recursion Implementation II: Factorial
- Complexity Analysis

```
factorial(5)
    factorial(4)
        factorial(3)
            factorial(2)
                factorial(1)
                    return 1
                return 2*1 = 2
            return 3*2 = 6
        return 4*6 = 24
    return 5*24 = 120
```