CPT-281 - Introduction to Data Structures with C++

Module 7

Recursion

Dayu Wang

- **Recursive Algorithm**

  → A recursive algorithm solves a problem by breaking that problem into smaller subproblems, solving these subproblems, and combining the solutions.

  → An algorithm that is defined by repeated applications of the same algorithm on smaller problems is a recursive algorithm.

- **Two parts in recursive algorithm**

  → Base case(s)

  → Recurrence relation(s)

- **Designing recursive algorithms**

  → Four steps

  Step 1: Re-define the problem.

  The problem you solve using recurrence relations may or may not be the same as the original problem.

  Step 2: Write the base case(s).

  Step 3: Write the recurrence case(s).

  Step 4: Write the wrapper function.

- **Finding the time complexities of recursive algorithms**

→ [Example] Sum up an array of integers

Step 1: Re-define the problem.

Original Problem	Re-Defined Problem
Let `sum(arr)` be the sum of **all** the values in array `arr`.	Let `sum(arr, i, j)` be the sum of **all** the values in the *contiguous* **segment** of array `arr` from index `i` (inclusive) to index `j` (inclusive).

Step 2: Write the base case.

If `i > j`**, then** `sum(arr, i, j)` **= 0** (the segment is *empty*).

Step 3: Write the recurrence relation.

`sum(arr, i, j) = arr[i] + sum(arr, i + 1, j)`

Sum the **rest** of the segment (*except* `arr[i]`).

Step 4: Wrapper function (how the re-defined problem is converted back to the original problem)

`sum(arr) = sum(arr, 0, arr.length - 1)`

Sum the **entire** array is *equivalent* to sum the segment from index **0** to last index.

→ Writing the code of the designed recursive algorithm

```
/** Sums up the segment of a vector from index i to index j.
    @param vec: vector whose segment is to sum up
    @param i: inclusive beginning index of the segment
    @param j: inclusive end index of the segment
    @return: sum of all the elements in the segment
*/
int sum(const vector<int>& vec, size_t i, size_t j) {
    if (i > j) { return 0; }  // Base case
    return vec.at(i) + sum(vec, i + 1, j);  // Recurrence relation
}
```

```
// Wrapper function
int sum(const vector<int>& vec) { return sum(vec, 0, vec.size() - 1); }
```

→ What is the time complexity of this *recursive* algorithm?

Suppose there are `n` **elements in the segment.**

"`if (i > j) { return 0; }`": $T(0) = 1$

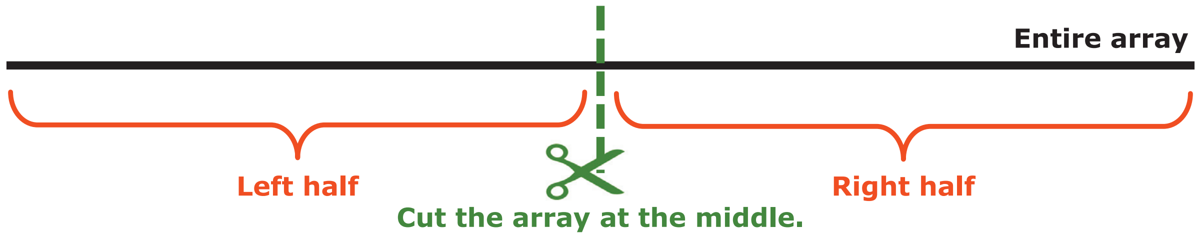
"`return vec.at(i) + sum(vec, i + 1, j);`": $T(n) = T(n-1) + 2$

→ Solving the recurrence relation to find the explicit expression of $T(n)$

Please see lecture notes.

→ **[Example] Finding the length of the <u>longest increasing segment</u> in an array**

There are many ways to solve this problem. Here, let's discuss a function called divide-and-conquer.



The longest increasing segment may appear in the left half.

The longest increasing segment may appear in the right half.

The longest increasing segment may across the middle point.

[4, 2, 3, 6, 5, 3, 3, 4]	[5, 2, 3, 6, 8, 2, 3, 5]	[3, 2, 2, 5, 4, 4, 7, 9]

So, we need to consider 3 candidates: find the LIS on the left side; find the LIS on the right side; find the LIS across the middle point.

How to find the LIS in the left (right) half?

Apply the same "cutting" function recursively...

Until there is only 1 element left.

If there is only 1 element in an array, then the only element itself is considered an increasing segment, which has length 1.

How to find the LIS across the middle point?

We can use `for` loops ($O(n)$).

```cpp
/** Finds the length of the longest increasing segment in a vector.
    @param vec: a non-empty vector of integers
    @return: length of the longest increasing segment
*/
unsigned int len(const vector<int>& vec) {
    if (vec.size() == 1) { return 1; }
    vector<int> left, right;
    copy(vec.begin(), vec.begin() + vec.size() / 2, back_inserter(left));
    copy(vec.begin() + vec.size() / 2, vec.end(), back_inserter(right));
    unsigned int c_1 = len(left);  // First candidate
    unsigned int c_2 = len(right);  // Second candidate
    size_t i, j;
    for (i = vec.size() / 2; i >= 1; i--) {
        if (vec.at(i - 1) >= vec.at(i)) { break; }
    }
    for (j = vec.size() / 2; j < vec.size() - 1; j++) {
        if (vec.at(j + 1) <= vec.at(j)) { break; }
    }
    unsigned int c_3 = j - i + 1;  // Third candidate
    return max({ c_1, c_2, c_3 });
}
```

Base case: $\mathrm{T}(1) = 1$

Recurrence relation: $\mathrm{T}(n) = 2\mathrm{T}\left(\frac{n}{2}\right) + 2n$

Please see lecture notes to understand how to find $\mathrm{T}(n)$.

- **Swap nodes in pairs**

→ Given a linked list, swap every two adjacent nodes and return its head.

You may not modify the values in the list's nodes, only nodes itself may be changed.

Given 1→2→3→4, your function should return the list as 2→1→4→3.

```cpp
/** Swaps nodes in pairs in a singly-linked list of integers.
    @param head: a pointer to the head node of the list.
    @return: a pointer to the head node of the list after swap.
*/
List_Node* swap_in_pairs(List_Node* head) {
    if (!head || !head->next) { return head; }  // Base case
    // Recursively swap the rest of the list (except the first 2 nodes).
    List_Node* after = swap_in_pairs(head->next->next);
    // Then, swap the first 2 nodes.
    List_Node* p = head->next;
    p->next = head;
    // Link the swapped first 2 nodes with the rest part.
    head->next = after;
    return p;
}
```

$\mathrm{T}(0) = 1, \quad \mathrm{T}(1) = 1$

$\mathrm{T}(n) = \mathrm{T}(n - 2) + 4$

Can you find the explicit expression of $\mathrm{T}(n)$?

- **Master's Theorem**

→ **Form:** $\mathrm{T}(n) = 2\mathrm{T}\left(\frac{n}{2}\right) + f(n)$ **and** $\mathrm{T}(1) = constant$.

1) **If** $f(n) > \mathrm{O}(n)$**, then** $\mathrm{T}(n) = \mathrm{O}(f(n))$.
2) **If** $f(n) == \mathrm{O}(n)$**, then** $\mathrm{T}(n) = \mathrm{O}(n \log n)$.
3) **If** $f(n) < \mathrm{O}(n)$**, then?**

   It is a question in your assignment.