UCLA Samueli Computer Science

CS32: Introduction to Computer Science II

Discussion Week 6

Junheng Hao, Arabelle Siahaan

May 10, 2019

Announcements

UCLA Samueli Computer Science

- Homework 3 is **due today 11PM**. (OAO Last call!)
- Project 3 is due on 11PM Tuesday, May 21. (O_O it may be changed!)
- Midterm 2 is scheduled on Thursday, May 23. (@_@, this may not change!)

Outline Today

UCLA Samueli Computer Science

- Recursion (Review)
- Template and STL (Preview)

Recursion

Basics

UCLA Samueli Computer Science

- Function-writing technique where the functions refers to itself.
- Let's talk about the factorial example again!
  - Similar to mathematical induction → Prove `k=1` is valid and prove `k=n` is valid when `k=n-1` is valid.
  - Base cases are important and need to be carefully considered.

```
int factorial(int n)
{
    int temp = 1;
    for (int i = 1; i <= n; i++)
        temp *= i;
    return temp;
}
```

```
int factorial(int n)
{
    if (n <= 1)
        return 1;

    return n * factorial(n - 1);
}
```

Without explicit loops!

Recursion

Pattern: How to write a recursive function

UCLA Samueli Computer Science

- Step 1: Find the base case(s).
  - What are the trivial cases? Eg. empty string, empty array, single-item subarray.
  - When should the recursion stop?
- Step 2: Decompose the problem.
  - Take tail recursion as example.

> ➔ Take the first (or last) of the ***n*** items of information
> ➔ Make a recursive call to the rest of **(*n*-1)** items. The recursive call will give you the correct results.
> ➔ Given this result and the information you have on the first (or last item) conclude about current ***n*** items.

- Step 3: Just solve it! *(Well, easier said than done~)*

Recursion

Smallberg's notes

UCLA Samueli Computer Science

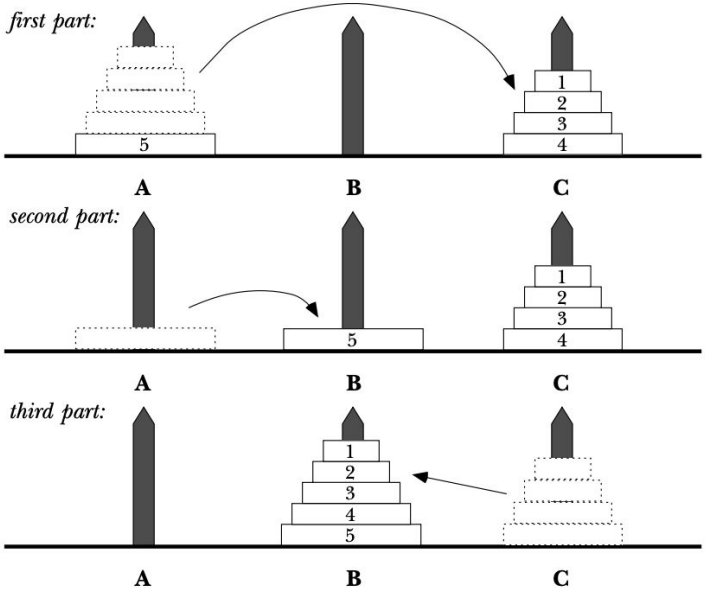
What I've emphasized all week is how to understand a recursive solution to a problem and know it's correct:

1. Identify the base cases (paths through the function that make no recursive calls) and recursive cases.
2. Come up with measure of the size of the problem for which the base case(s) provide a bottom, typically 0 or 1.
3. Verify that if the function is called with a problem of some size, any recursive call it makes is to solve a problem of a strictly smaller size. (Problem sizes should be nonnegative integers.) This proves termination, since a decreasing sequence of nonnegative integers must eventually hit bottom.
4. Now that we've proved termination, verify that the base cases are handled correctly.

Recursion

Hanoi's story

UCLA Samueli Computer Science




```
➔  to hanoi :n
➔  hanoi :n-1
➔  movedisk :n
➔  hanoi :n-1
➔  end
```

```
➔  to hanoi :n-1
➔  hanoi :n-2
➔  movedisk :n
➔  hanoi :n-2
➔  end
```

```
➔  to hanoi :n-1
➔  hanoi :n-2
➔  movedisk :n
➔  hanoi :n-2
➔  end
```

Recursion

Examples

UCLA Samueli Computer Science

- Problem 1: Given an integer array a and its length n, return whether the array contain any element that is smaller than 0.
- Problem 2: Given an integer array a and its length n, count the number of elements that are smaller than 0.
- Problem 3: `pathExists()` function in Homework 2 without stack or queue but with recursion.

```
// a simple function with for loop
bool anyTrue(const double a[], int n)
{
  for (int k = 0; k < n; k++)
  {
    if (a[k] < 0)
      return true;
  }
  return false;
}
```

```
// try: without for loop
bool anyTrue(const double a[], int n)
{
  // recursion implementation
}
```

Recursion

Practice Examples

UCLA Samueli Computer Science

- Practice: Print out the permutations of a given vector (Difficulty: Hard).

```
Input: [1,2,3]
Output: [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]
Implement: void permutation(vector<int>& nums, int start);
```

- Note: Some data structures are easy to implement recursive technique: arrays, trees (will be discussed later).

Recursion

Practice Examples: Merge sort and quick sort

UCLA Samueli Computer Science

Merge sort

1. Find the middle point to divide the array into two halves:

```
middle m = (l+r)/2
```

2. Call mergeSort for first half:

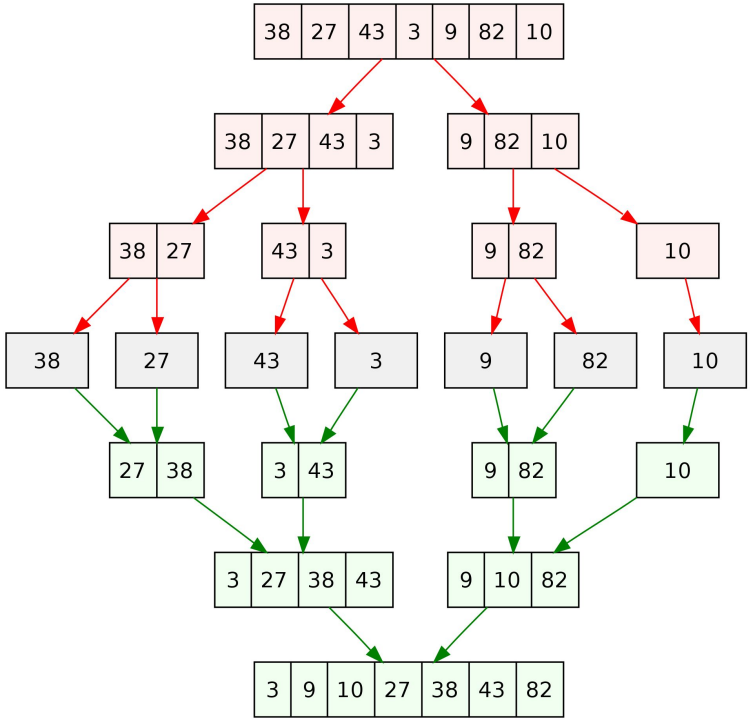
```
mergeSort(arr, l, m)
```

3. Call mergeSort for second half:

```
mergeSort(arr, m+1, r)
```

4 .Merge the two halves sorted in step 2 and 3:

```
merge(arr, l, m, r)
```



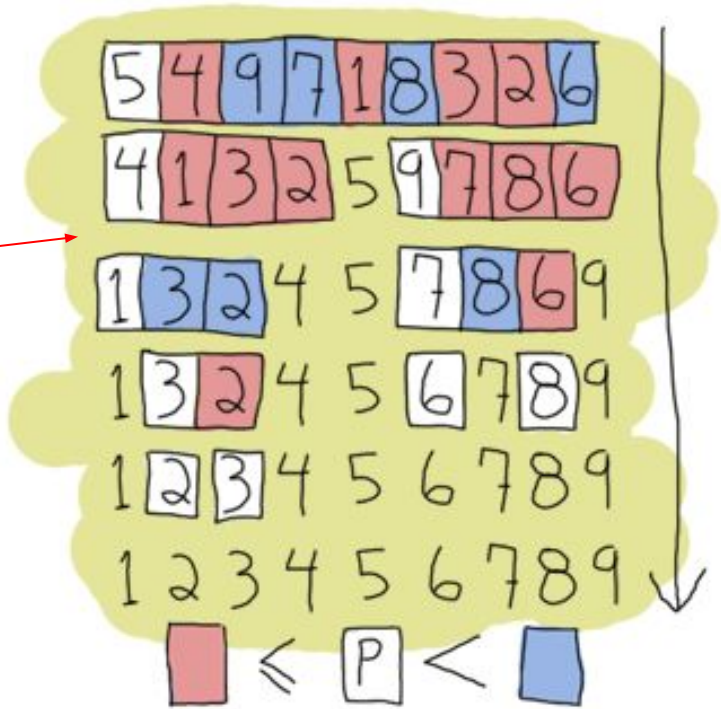
Recursion

Practice Examples: Merge sort and quicksort

UCLA Samueli Computer Science

Quick Sort

Two recursion calls!



Template

Motivation: More generic class

- Think about the `Pair` class. The class should not work only with integers. That is we want a “generic” `Pair` class. (Well, I know you were thinking `typedef` just now~)
- Here we go: `Pair<int> p1; Pair<char> p2;`

```
class Pair {
    public:
       Pair();
       Pair(int firstValue,
            int secondValue);
       void setFirst(int newValue);
       void setSecond(int newValue);
       int getFirst() const;
       int getSecond() const;
    private:
       int m_first;
       int m_second;
};
```

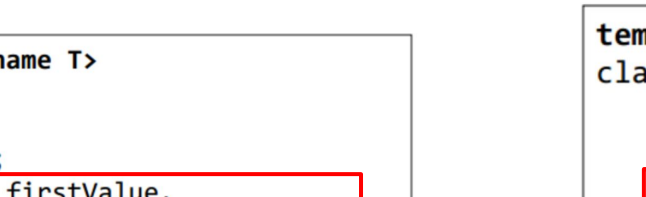
```
template<typename T>
class Pair {
    public:
       Pair();
       Pair(T firstValue,
            T secondValue);
       void setFirst(T newValue);
       void setSecond(T newValue);
       T getFirst() const;
       T getSecond() const;
    private:
       T m_first;
       T m_second;
};
```

Template

Multi-type template

- What if we need pair with different types? (One with `int` value while the other with `string` value)
- Just slightly change your template class and: `Pair<int, string> p1;`

```
template<typename T>
class Pair {
    public:
        Pair();
        Pair(T firstValue,
             T secondValue);
        void setFirst(T newValue);
        void setSecond(T newValue);
        T getFirst() const;
        T getSecond() const;
    private:
        T m_first;
        T m_second;
};
```



```
template<typename T, U>
class Pair {
    public:
        Pair();
        Pair(T firstValue,
             U secondValue);
        void setFirst(T newValue);
        void setSecond(U newValue);
        T getFirst() const;
        U getSecond() const;
    private:
        T m_first;
        U m_second;
};
```

Template

Change member functions in template classes

- Member function should also be edited in template class as well.

```
void Pair::setFirst(int newValue)
{
  M_first = newValue;
}
```

```
template<typename T>
void Pair<T>::setFirst(T newValue)
{
  M_first = newValue;
}
```

Template

Template Specialization

- What if we want a template class with certain data type to have its own exclusive behaviors? For example, in `Pair` class we only allow `Pair<char>` has `uppercase()` and `lowercase()` function but not for `Pair<int>`.

```
template<>
class Pair<char> {
    public:
       Pair();
       Pair(char firstValue,
            char secondValue);
       void setFirst(char newValue);
       void setSecond(char newValue);
       char getFirst() const;
       char getSecond() const;
       void uppercase();
    private:
       char m_first;
       char m_second;
};
```

```
Pair<int> p1;
Pair<char> p2;

p1.uppercase(); //error
p2.uppercase(); //correct
```

Template

Const references as parameters

- When you are not changing the values of the parameters, make them `const` references to avoid potential computational cost. (Pass by value for ADTs are slow.)

```
template<typename T>
T minimum(const T& a, const T& b)
{
  if (a < b)
    return a;
  else
    return b;
}
```

Template

Some notes

- Generic comparisons:
  - `bool operator>=(const ItemType& a, const ItemType& b)`
- Use the template data type (e.g. `T`) to define the type of at least one formal parameter.
- Add the prefix **`template <typename T>`** before the class definition itself and before each function definition outside the class. Also place the postfix **`<T>`** Between the class name and the :: in all function definition.

```
template <typename T>
class Foo
{
  public:
    void setVal(T a);
    void printVal(void);
  private:
    T m_a;
};
```

```
template <typename T>
void Foo<T>::setVal(T a)
{
  m_a = a;
}
template <typename T>
void Foo<T>::printVal(void)
{
  cout << m_a << "\n";
}
```

STL: Standard Template Library

Easy and efficient implementation

- A collection of pre-written, tested classes provided by C++.
- All built using templates (adaptive with many data types).
- Provide useful data structures
  - `vector(array), set, list, map, stack, queue`
- Standard functions:
  - Common ones: `.size(), .empty()`
  - For a container that is neither stack or queue: `.insert(), .erase(), swap(), .clear()`
  - For list or vector: `.push_back(), .pop_back()`
  - For set or map: `.find(), .count()`
  - More on stacks and queues…

STL: Standard Template Library

Notes on `vector` and `list`

- You may only use brackets to access existing items in `vector`. Keep the current size vector in mind especially after `push_back()` and `pop_back()`.
- You cannot access list element by brackets.
- Choose between vector and list:
  - `vectors` are based on dynamic arrays placed in contiguous storage. Fast on access but slow on insertion/deletion.
  - `lists` are the opposite. It offers fast insertion/deletion, but slow access to middle elements.

STL: Standard Template Library

Notes on `size` and `capacity`

- Bonus question: Size and capacity of a vector?

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> myVec;
  // insert only one item
  myVec.push_back(999);
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  // insert 100 items
  for (int i=0; i<100; i++){ myVec.push_back(i); }
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  cout << "max size:" << myVec.max_size() << endl;
  return 0;
}
```

```
size: ?
capacity: ?

size: ?
capacity: ?

max size: ?
```

STL: Standard Template Library

Notes on `size` and `capacity`

- Bonus question: Size and capacity of a vector?

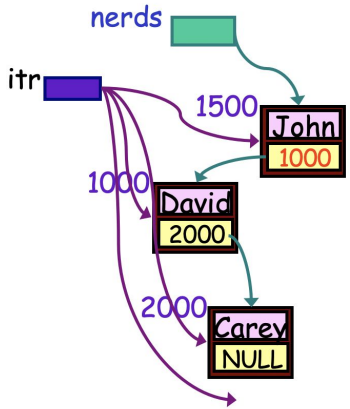
```cpp
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> myVec;
  // insert only one item
  myVec.push_back(999);
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  // insert 100 items
  for (int i=0; i<100; i++){ myVec.push_back(i); }
  cout << "size:" << myVec.size() << endl;
  cout << "capacity:" << myVec.capacity() << endl;
  cout << "max size:" << myVec.max_size() << endl;
  return 0;
}
```

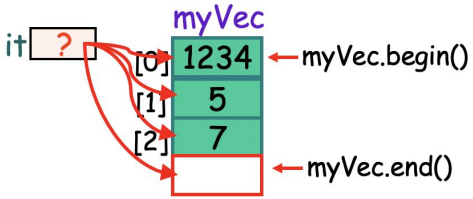
```
→ On my computer:
size:1
capacity:1
size:101
capacity:128
max size:4611686018427387903
```

STL: Standard Template Library

Implementation example: Iterators

- STL Iterators: Use `.begin()` and `.end()`
  - `.begin()` : return an iterator that points to the first element.
  - `.end()`: return an iterator that points to the ***past-the- last*** element.
- A container as a `const` reference cannot use regular iterator but need to use `const` iterator. Example: `list<string>::const_iterator it;`
- Examples






```
void main()
{
  vector<int>   myVec;
  myVec.push_back(1234);
  myVec.push_back(5);
  myVec.push_back(7);
  vector<int>::iterator it;
  it = myVec.begin();
  while (  it != myVec.end() ){
    cout << (*it);
    it++;
  }
}
```

STL: Standard Template Library

Warning: using iterators for changing vector

- It could be dangerous to use iterator to traverse a vector when we have performed insertion/deletion.
- Safe solution: Reinitialize iterators of a vector whenever its size has been changed.

```
// Guess what is the output?
int main ()
{
  vector<int> v;
  v.push_back(50);
  v.push_back(22);
  v.push_back(10);
  vector<int>::iterator b = v.begin();
  vector<int>::iterator e = v.end();
  for (int i = 0; i < 100; i++) { v.push_back(i); }
  while (b != e) {
    cout << *b++ << endl;
  }
}
```

Standard Template Library

How to use STL? No need to recite all of them!

- Remember the basic provided libraries (such as size, etc)
- Check http://www.cplusplus.com/reference/stl/ for more details if needed.

STL: Standard Template Library

Some more topics

- More STL examples, such as `map, set, etc.`
- More STL algorithms, such as `find(), sort(), etc.`

*Inline Functions

Motivation & Examples

- When you define a function as being inline, you ask the compiler to directly embed the function's logic into the calling function (for speed).
- All methods with their body defined directly in the class are inline. Simply add the word inline before the function return type to make an externally defined method inline.
- Inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:
  - Loops, recursion, static variables, etc
- Save time for function call vs large binary executable file?

```
template <typename T>
class Foo
{
  public:
    void setVal(T a);
    void printVal(void){cout<<m_a<<endl;}
  private:
    T m_a;
};
```

```
inline template <typename T>
void Foo<Item>::setVal(T a)
{
  m_a = a;
}
```

UCLA Samueli Computer Science

Break Time! (5 minutes)

Q & A

Group Exercises: Worksheet

UCLA Samueli Computer Science

- Exercise problems from **Worksheet 5** (see “LA worksheet” tab in CS32 website). Answers will be posted next week.
- Questions for today:
  - Code tracing: 1
  - Code writing: 2, 4, 7, 8, 13

Question 1

UCLA Samueli Computer Science

What does the following code output and what does the function LA_power do?

```
#include <iostream>
using namespace std;

int LA_power(int a, int b)
{
   if (b == 0)
       return 0;
   if (b % 2 == 0)
       return LA_power(a+a, b/2);

   return LA_power(a+a, b/2) + a;
}

int main()
{
  cout << LA_power(3, 4) << endl;
}
```

Solution:
- The output of the main routine is 12
- The function returns the result of the multiplication of the two arguments: a*b

Question 2

UCLA Samueli Computer Science

Given a singly-linked list class LL with a member variable *head* that points to the first *Node* struct in the list, write a function to recursively delete the whole list, void LL::deleteList(). Assume each Node object has a next pointer. (Hint: It might be a good idea to use a helper function)

```
struct Node {
      int data;
      Node* next;
};

class LL {
 public: // other functions such as insert not shown
      void deleteList(); // implement this function
private: // additional helper allowed
      Node* m_head;
};
```

Question 2: Solution

UCLA Samueli Computer Science

```
void LL::deleteListHelper(Node* &head) {
      if (head == nullptr)
        return;

      deleteListHelper(head->next);
      delete head;
      head = nullptr;
}


void LL::deleteList() {
      deleteListHelper(m_head);
}
```

Question 4

UCLA Samueli Computer Science

Given a string *str*, recursively compute a new string such that all the 'x' chars have been moved to the end. Use the function header: `string endX(string str);`

Example:
`endX("xrxe")` → `"rexx"`

Question 4: Solution

UCLA Samueli Computer Science

```
string endX(string str) {
  if (str.length() <= 1)
    return str;
  else if (str[0] == 'x')
    return endX(str.substr(1)) + 'x';
  else
    return str[0] + endX(str.substr(1));
}
```

Question 7

UCLA Samueli Computer Science

Implement the function sumOfDigits recursively. The function should return the sum of all of the digits in a *positive* integer. Use the function header: `int sumOfDigits(int num)`

Example:

```
sumOfDigits(176) = 14
sumOfDigits(111111) = 6
```

Question 7: Solution

UCLA Samueli Computer Science

```
int sumOfDigits(int num) {
      if (num < 10)
            return num;
      return num % 10 + sumOfDigits(num/10);
}
```

Question 8

UCLA Samueli Computer Science

Implement the function `isPalindrome` recursively. The function should return whether the given string is a palindrome. A palindrome is described as a word, phrase or sequence of characters that reads the same forward and backwards. Use the function header: `bool isPalindrome(string foo)`

Example:

```
isPalindrome("kayak") = true
isPalindrome("stanley yelnats") = true
isPalindrome("LAs rock") = false (but the sentiment is true :))
```

Question 8: Solution

UCLA Samueli Computer Science

```
bool isPalindrome(string foo) {
      int len = foo.length();
      if (len <= 1)
            return true;
      if (foo[0] != foo[len-1])
            return false;
      return isPalindrome(foo.substr(1, len-2));
}
```

Question 13

UCLA Samueli Computer Science

Implement a recursive function that merges two sorted linked lists into a single sorted linked list. The lists are singly linked; the last node in a list has a null next pointer. The function should return the head of the merged linked list. No new Nodes should be created while merging. Use the following function header: `Node* merge(Node* l1, Node* l2)`

Example:
List 1 = `1 -> 4 -> 6 -> 8`
List 2 = `3 ->9 -> 10`
After merge: `1 -> 3 -> 4 -> 6 -> 8 -> 9 -> 10`

Use the following definition of a Node of a linked list:

```
struct Node {
      int val;
      Node* next;
};
```

Question 13: Solution

UCLA Samueli Computer Science

```
Node* merge(Node* l1, Node* l2){

	// base cases: if a list is empty, return the other list
		if (l1 == nullptr)
			return l2;
		if (l2 == nullptr)
			return l1;

	// determine which head should be the head of the merged list
	// then set the next pointer to the head of the recursive calls
		Node* head;
		if (l1->val < l2->val) {
			head = l1;
			head->next = merge(l1->next, l2);
		}
		else {
			head = l2;
			head->next = merge(l1, l2->next);
		}

	// return the head of the merged list
		return head;
	}
```

UCLA Samueli Computer Science

Thank you!

Q & A