Recursion Part 1

Today..

- What is a *recursion*?
- Examples of recursive functions
  - Factorials
  - Converting a number into base 7
  - Iterating combinations
  - $r^n \bmod m$

Today..

- **What is a *recursion*?**
- Examples of recursive functions
  - Factorials
  - Converting a number into base 7
  - Iterating combinations
  - $r^n \bmod m$

What is a recursion?

- "Recursion" means to define something in terms of itself.
- Examples:
  - A *folder* is a collection of files and *folder*s.
  - *Words* in dictionaries are defined in terms of other *words*.

Recursive functions

- ***Recursive functions*** are *functions* that are defined in terms of itself.
- Today, we are going to look at some functions which are defined recursively.

Today..

- What is a *recursion*?
- Examples of recursive functions
  - **Factorials**
  - Converting a number into base 7
  - Iterating combinations
  - $r^n \bmod m$

Factorials

- Factorials can be defined recursively:

$$n! = \begin{cases} 1, & n = 1 \\ (n-1)! \times n, & n > 1 \end{cases}$$

- We can code this function like this:

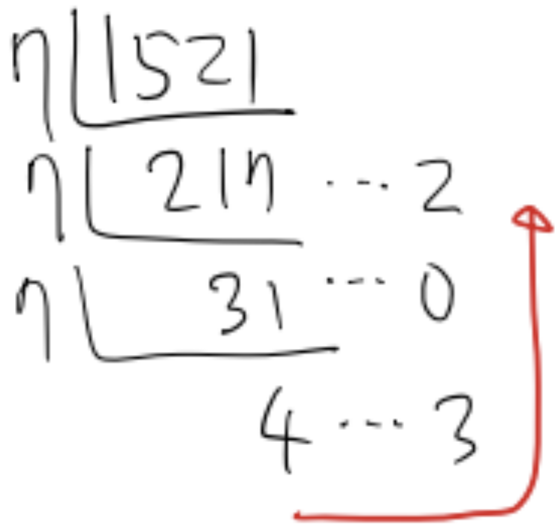
```
long long factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return factorial(n - 1) * n;
  }
}
```

Today..

- What is a *recursion*?
- Examples of recursive functions
  - Factorials
  - **Converting a number into base 7**
  - Iterating combinations
  - $r^n \bmod m$

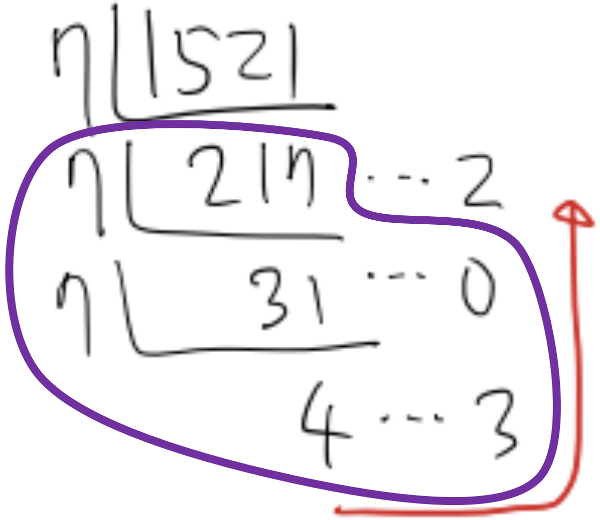
Convert a number to base 7

- How do you convert a given number *n* to base 7?
- Example: when *n* = 1521,



Convert a number to base 7 (cont.)

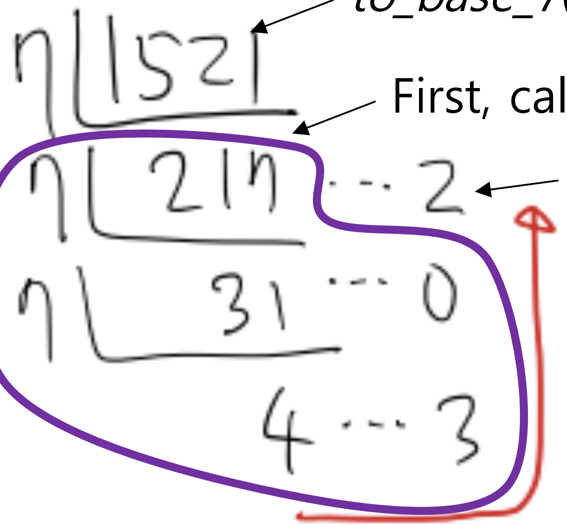
- As $1521 = 7 \times 217 + 2$, we know that the last digit of 1521 in base 7 is 2.
- The remaining digits are the representation of 217 in base 7.



Convert a number to base 7 (cont.)

- From this, we can code

```
void to_base_7 (int n) {
  if(n < 7) {
    printf("%d", n);
  }else {
    to_base_7(n / 7);
    printf("%d", n % 7);
  }
}
```



Convert a number to base 7 (cont.)

- From this, we can code

```c
void to_base_7 (int n) {
  if(n < 7) {
    printf("%d", n);
  }else {
    to_base_7(n / 7);
    printf("%d", n % 7);
  }
}
```

- to_base_7(1521)
  - to_base_7(217)
    - to_base_7(31)
      - to_base_7(4)
        - print '4'
      - print '3'
    - print '0'
  - print '2'

- Therefore "4302" is printed.

Today..

- What is a *recursion*?
- Examples of recursive functions
  - Factorials
  - Converting a number into base 7
  - **Iterating combinations**
  - $r^n \bmod m$

Iterating combinations

- Suppose you have n distinct products, and you want to choose k of them. Of course, there are $\binom{n}{k}$ ways to choose.
- Given n and k, could you print *all possible choices* in lexicographical order?
- Example) $n = 4, k = 2$. Write each product as 1, 2, 3 and 4.
  - All choices: 1,2 / 1,3 / 1,4 / 2,3 / 2,4 / 3,4
  - This is in lexicographical order (think A=1, B=2, C=3, D=4)

Iterating combinations (cont.)

- How to view this problem recursively? First, let's try to define a function

```
void printcomb (int n, int k);
```

- which prints all possible choices, and see whether this function is recursive or not.
- Take an example: Let's call `printcomb(5, 3)` this time.

Iterating combinations (cont.)

- The result is:
  - 1 2 3
  - 1 2 4
  - 1 2 5
  - 1 3 4
  - 1 3 5
  - 1 4 5
  - 2 3 4
  - 2 3 5
  - 2 4 5
  - 3 4 5

In the first group,
we are choosing '1' and
$3 - 1 = 2$ elements from {2, 3, 4, 5}.

Divide the result into 2 groups:
first group contains 1,
second group doesn't contain 1.

In the second group,
we choose 3 elements from {2, 3, 4, 5}.

- Is the function "`printcomb`" is not recursive? Let's see..
- `printcomb` chooses k out of n elements..

Iterating combinations (cont.)

- The result is:
  - 1 2 3
  - 1 2 4
  - 1 2 5
  - 1 3 4
  - 1 3 5
  - 1 4 5
  - 2 3 4
  - 2 3 5
  - 2 4 5
  - 3 4 5

In the first group,
we are choosing '1' and
$3 - 1 =$ **2 elements from {2, 3, 4, 5}.**

Divide the result into 2 groups:
first group contains 1,
second group doesn't contain 1.

In the second group,
we choose **3 elements from {2, 3, 4, 5}.**

- ..and yes! It seems it is recursive, since we can make two groups
- that chooses 2 or 3 elements from the same set $\{2, 3, 4, 5\}$.

Iterating combinations (cont.)

- So, it seems we have to consider
  - n: the total number of elements
  - k: the number of elements that the function should additionally choose
  - S: the already chosen elements
    - In the first group, we have to choose $S = \{1\}$ for all choices.
  - i: the id of the element we are going to consider now.
    - While dividing the result into groups, we considered element $i = 1$ to be chosen or not.

Iterating combinations (cont.)

- Pseudocode of the new `printcomb`:

```
printcomb(n,k,S,i) {
    if(k is 0) print S and return.
    if(i > n) just return, because we need to choose k
    more elements when there is no element left.
    printcomb(n,k − 1,S ∪ {i},i + 1) // choose element i
    printcomb(n,k,S,i + 1) // do not choose element i
}
```

Iterating combinations (cont.)

- Implementation of this pseudocode is like this.

```
void printcomb (int n, int k, vector<int> &S, int i) {
    if(k == 0) {
        for(int x = 0; x < S.size(); x++) printf("%d ", S[x]);
        puts("");
        return;
    }
    if(i > n) return;
    S.push_back(i); printcomb(n, k-1, S, i+1); S.pop_back();
    printcomb(n, k, S, i+1);
}
```

- It is convenient to use a `vector<int>` to store S.
- We can use this function like: `vector<int> S; printcomb(n,k,S,1);`

Iterating combinations (cont.)

- If you don't know about `vectors`, we can still implement this pseudocode. Just use an `int`-type array S and an integer $Ssize$ to denote its size.

```
void printcomb (int n, int k, int S[], int Ssize, int i) {
    if(k == 0) {
        for(int x = 0; x < Ssize; x++) printf("%d ", S[x]);
        puts("");
        return;
    }
    if(i > n) return;
    S[Ssize++] = i; printcomb(n, k-1, S, Ssize, i+1); Ssize--;
    printcomb(n, k, S, Ssize, i+1);
}
```

- We can use this function like: `int S[MAXK]; printcomb(n,k,S,0,1);`

Iterating combinations (cont.)

- Note that the array S is **passed by reference** instead of its values (because the pointer of the array is given)
- So, if we change the content of S, the **change remains on other function calls** too. Just be aware of that.

Today..

- What is a *recursion*?
- Examples of recursive functions
  - Factorials
  - Converting a number into base 7
  - Iterating combinations
  - $\boldsymbol{r^n \bmod m}$

Calculating $r^n \bmod m$

- This problem is back again!
- We actually explained a method using the binary representation of n (12th of July), but nobody managed to use that method.
- Today, we are going to introduce a simpler method.

Calculating $r^n \bmod m$ (cont.)

- Let's define a function that calculates $r^n \bmod m$:

```
int mypower(int r, int n, int m);
```

- Precondition: $r \neq 0$, $n > 0$ and $m > 0$ (to avoid situations like 0^0)

Calculating $r^n \bmod m$ (cont.)

- Like factorials, r^n can be defined recursively:

$$r^n = \begin{cases} r^{n-1} \times r, & n \geq 1 \\ 1, & n = 0 \end{cases}$$

- We can code exactly this idea, which is $O(n)$.

```
int mypower (int r, int n, int m) {
    if(n >= 1)
        return ((long long)mypower(r, n-1, m) * r) % m;
    else
        return 1 % m;
}
```

Calculating $r^n \bmod m$ (cont.)

- We can improve the time complexity into $O(\log n)$ by defining r^n in another way (but also recursively)
- From the Exponential Law: $(r^a)^b = r^{ab}$.
- $\left(r^{n/2}\right)^2 = r^{n/2\times 2} = r^n$
- By **squaring**, we can **decrease $\boldsymbol{n}$ into half**.
  - If n is even, we can apply this formula directly.
  - If n is odd, $n - 1$ is even, so we can calculate r^{n-1} by the formula and then multiply r.

Calculating $r^n \bmod m$ (cont.)

- In short,

$$r^n = \begin{cases} 1, & n = 0 \\ \left(r^{n/2}\right)^2, & n \text{ is even} \\ \left(r^{(n-1)/2}\right)^2 \times r, & n \text{ is odd} \end{cases}$$

- If we compute $r^n \bmod m$ like this, time complexity is $O(\log n)$ since we call the function such that n is at most its half.

Calculating $r^n \bmod m$ (cont.)

- We can implement this easily:

```
int mypower (int r, int n, int m) {
    if(n == 0) {
        return 1 % m;
    }else if(n % 2 == 0) {
        int v = mypower(r, n/2, m);
        return ((long long)v * v) % m;
    }else if(n % 2 == 1) {
        int v = mypower(r, (n-1)/2, m);
        return ((((long long)v * v) % m) * (long long)r) % m;
    }
}
```

Calculating $r^n \bmod m$ (cont.)

- However, we can make the code more simpler! Let's look at the definition we used:

$$r^n = \begin{cases} 1, & n = 0 \\ \left(r^{n/2}\right)^2, & n \text{ is even} \\ \left(r^{(n-1)/2}\right)^2 \times r, & n \text{ is odd} \end{cases}$$

- In the definition when n is odd, notice that we use the fact that $\boldsymbol{n-1}$ **is even**, and use **the formula for the even case**.

Calculating $r^n \bmod m$ (cont.)

- So, let's just write

$$r^n = \begin{cases} 1, & n = 0 \\ \left(r^{n/2}\right)^2, & n \text{ is even} \\ \boldsymbol{r^{n-1}} \times r, & n \text{ is odd} \end{cases}$$

- And still everything is okay, and time complexity is the same. (we are doing exactly the same thing!)

Calculating $r^n \bmod m$ (cont.)

- Now, the code becomes like:

```
int mypower (int r, int n, int m) {
    if(n == 0) {
        return 1 % m;
    }else if(n % 2 == 0) {
        int v = mypower(r, n/2, m);
        return ((long long)v * v) % m;
    }else if(n % 2 == 1) {
        return ((long long)mypower(r, n-1, m) * r) % m;
    }
}
```

Calculating $r^n \bmod m$ (cont.)

- We can generalize this idea to find $f(n)$ (f is a function of n) if..
  - There is an efficient way to find $f(n)$ from $f(n-1)$
  - There is an efficient way to find $f(n)$ from $f(n/2)$
- Then, the time complexity of finding $f(n)$ becomes $O\left(\log n \times (time\ of\ transition)\right)$
- In this slides, $f(n)$ was $r^n \bmod m$.

Calculating $r^n \bmod m$ (cont.)

- CAUTION: However, you **MUST NOT** code like this:

```
int myBADpower (int r, int n, int m) {
    if(n == 0) {
        return 1 % m;
    }else if(n % 2 == 0) {
        return ((long long)myBADpower(r, n/2, m)
                                * myBADpower(r, n/2, m)) % m;
    }else if(n % 2 == 1) {
        return ((long long)myBADpower(r, n-1, m) * r) % m;
    }
}
```

Calculating $r^n \bmod m$ (cont.)

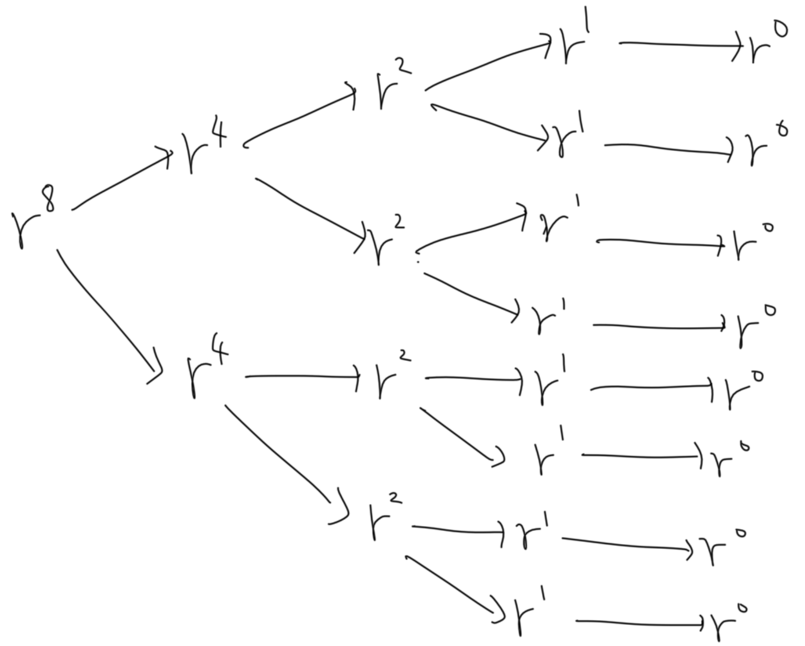
- What is the difference? This code is calculating $r^{n/2}$ **twice** (instead of once) to calculate $\left(r^{n/2}\right)^2 = r^{n/2} \times r^{n/2}$.
- Calling the same function once or twice doesn't seem like a big difference, but recall that this function is **recursive**..
- For example, let's try to calculate r^8.

Calculating $r^n \bmod m$ (cont.)

- If we use the `mypower` function,

$$r^8 \longrightarrow r^4 \longrightarrow r^2 \longrightarrow r^1 \longrightarrow r^0$$

- If we use the `myBADpower` function,



Calculating $r^n \bmod m$ (cont.)

- As n increaes, number of function calls for `myBADpower` increases rapidly.
- So, if you are to use the same function value more than once, (not only for this example)
  - ***DO NOT call the function more than once***!
  - **Store the result in a variable, and use that variable instead!**