TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Applied Algorithms

Nguyễn Khánh Phương

Computer Science department
School of Information and Communication technology
E-mail: phuongnk@soict.hust.edu.vn

Contents

Contents of Chapter 3

Contents of Chapter 3

1. Algorithm diagram

- Divide and Conquer (Chia để trị): consists of 3 operations:
  - Divide: Decompose the given problem S into some problems of same form but with smaller input size (called a subproblem) S_1, S_2, …
  - Conquer: Solve subproblem recursively
  - Combine: Synthesize the solutions of subproblems S_1, S_2,.. to obtain the solution of the original problem S.
- If the subproblem is small enough to be easily solved, then we solve it directly, otherwise: the subproblem is solved again by applying the above procedure recursively (i.e. dividing it again into smaller problem). Therefore, the divide and conquer algorithm is recursive algorithm => to analysis the complexity of the algorithm we can use recursive formula.

1. Algorithm diagram

To get a detailed description of the divide and conquer algorithm, we need to define:

- Critical size n_0 (problems with size less than n_0 don't need subdivision)
- Dimensions of each subproblem
- Number of subproblems
- Algorithm for synthesizing solutions of subproblems.

```
procedure D-and-C(n)
begin
    if (n ≤ n0) then
        Solve the problem directly
   else
   begin
          Divide the problem into K subproblems of size n/b
          for (each subproblem in K subproblems) do D-and-C(n/b)
          Synthesize the solutions of K subproblems to obtain the
          solution of the problem with size n.
   end;
end;
```


1. Algorithm diagram

```
procedure D-and-C(n)
begin
    if (n ≤ n0) then
        Solve the problem directly
   else
   begin
          Divide the problem into K subproblems of size n/b
          for (each subproblem in K subproblems) do D-and-C(n/b)
          Synthesize the solutions of K subproblems to obtain the
          solution of the problem with size n.
   end;
end;
```

Let T(n) be the computation time of the algorithm to solve the problem of size n

- D(n): time to divide
- C(n): time to synthesize the solutions

Then

$$T(n) = \begin{cases} c & when\ n \le n_0 \\ kT\left(\frac{n}{b}\right) + D(n) + C(n) & when\ n > n_0 \end{cases}$$

where c is the constant

Example.

```
procedure D-and-C(int n)
begin
      if (n == 0)  return;
      D-and-C(n/2);
      D-and-C(n/2);
      for (int i =0 ; i < n; i++)
      begin
         Perform operations requires constant time
      end;
end;
```

What is the computation time of this procedure ?

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)=cn^d$ (in other words, $f(n)$ is $\Theta(n^d)$) where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Example: Recursive Binary Search

- Let T(n)= T(n/2) + K. What are the parameters?

  a = 1

  b = 2

  d = 0

  Therefore, which condition applies?

  $1 = 2^0$, case 2 applies

- We conclude that $T(n) \in \boldsymbol{\Theta}(n^d \log n) = \boldsymbol{\Theta}\,(\log n)$

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Pitfalls

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- You cannot use the Master Theorem if
  - T(n) is not monotone, e.g. T(n) = sin(n)
  - f(n) is not a polynomial, e.g. T(n) = 2T(n/2)+ 2^n
  - b cannot be expressed as a constant, e.g. $T(n) = T(\sqrt{n})$

NGUYỄN KHÁNH PHƯƠNG
CS - SOICT-HUST

Master Theorem: Example 1

- Let T(n) = T(n/2) + ½ n^2 + n. What are the parameters?

  a = 1

  b = 2

  d = 2

  Therefore, which condition applies?

  $1 < 2^2$, case 1 applies

- We conclude that $T(n) \in \Theta(n^d) = \Theta\,(n^2)$

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{If } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Example 2

- Let T(n)= 2 T(n/4) + $\sqrt{n}$ + 42. What are the parameters?
  - a = 2
  - b = 4
  - d = 1/2

Therefore, which condition applies?

2 = $4^{1/2}$, case 2 applies

- We conclude that $T(n) \in \Theta(n^d \log n) = \Theta(\log n \sqrt{n})$

Master Theorem

- Let T(n) be a monotonically increasing function that satisfies
  T(n) = a T(n/b) + f(n)
  T(1) = c
  where $a \geq 1$, $b \geq 2$, c>0. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Master Theorem: Example 3

- Let T(n)= 3 T(n/2) + (3/4)n + 1. What are the parameters?
  - a = 3
  - b = 2
  - d = 1

Therefore, which condition applies?

$3 > 2^1$, case 3 applies

- We conclude that $T(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3})$

- Note that $\log_2 3 \approx 1.584...$, can we say that T(n) $\in \Theta(n^{1.584})$????

  No, because $\log_2 3 \approx 1.5849...$ and $n^{1.584} \notin \Theta(n^{1.5849})$

NGUYỄN KHÁNH PHƯƠNG
CS – SOICT- HUST

Contents of Chapter 3

NGUYỄN KHÁNH PHƯƠNG
CS – SOICT- HUST

2.1.Binary Search

Input: An array S consists of n elements: S[0],…,S[n-1] in ascending order; Value ***key*** with the same data type as array S.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

Binary search algorithm: The value *key* either

equals to the element at the middle of the array S,

or is at the left half (L) of the array S,

or is at the right half (R) of the array S.

(The situation L (R) happen only when *key* is smaller (larger) than the element at the middle of the array S)

```
int binsearch(int low, int high, int S[], int key)
```

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

key = 33

low

```
binsearch(0, 14, S, 33);
```

high

2.1.Binary Search

Input: An array *S* consists of *n* elements: S[0],…,S[*n*-1] in ascending order; Value ***key*** with the same data type as array *S*.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```


2.1.Binary Search

Input: An array *S* consists of *n* elements: S[0],…,S[*n*-1] in ascending order; Value ***key*** with the same data type as array *S*.

Output: the index in array if ***key*** is found, -1 if ***key*** is not found

```
int binsearch(int low, int high, vector <int> &S, int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
Call: binsearch (0, S.size()-1,S, key);
```


Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑														↑
lo														hi

binsearch(0, 14, S, 33);

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
↑							↑							↑
lo							mid							hi

binsearch(0, 14, S, 33);

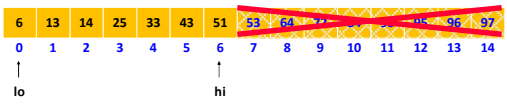
binsearch(0, 6, S, 33);

The section to be investigated is halved after each iteration

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
      int mid = (low + high) / 2;
      if (S[mid]== key) return mid;
      else if (key < S[mid])
          return binsearch(low, mid-1, S, key);
      else
          return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

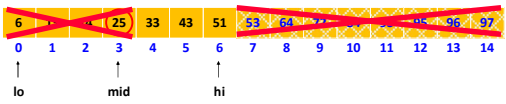


binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
      int mid = (low + high) / 2;
      if (S[mid]== key) return mid;
      else if (key < S[mid])
          return binsearch(low, mid-1, S, key);
      else
          return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33



binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);

The section to be investigated is halved after each iteration

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
      int mid = (low + high) / 2;
      if (S[mid]== key) return mid;
      else if (key < S[mid])
          return binsearch(low, mid-1, S, key);
      else
          return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

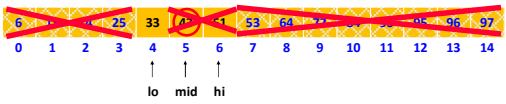
key=33

binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
      int mid = (low + high) / 2;
      if (S[mid]== key) return mid;
      else if (key < S[mid])
          return binsearch(low, mid-1, S, key);
      else
          return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33



binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);
binsearch(4, 4, S, 33);

The section to be investigated is halved after each iteration

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);
binsearch(4, 4, S, 33);

Example: Binary Search

```
int binsearch(int low, int high, int S[], int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]== key) return mid;
       else if (key < A[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

key=33

binsearch(0, 14, S, 33);
binsearch(0, 6, S, 33);
binsearch(4, 6, S, 33);
binsearch(4, 4, S, 33);

Example: Recursive Binary Search

- Let $T(n)= T(n/2) + K$. What are the parameters?

  a = 1

  b = 2

  d = 0

  Therefore, which condition applies?

  $1 = 2^0$, case 2 applies

```
int binsearch(int low, int high, vector <int> &S, int key)
{
   if (low <= high)
   {
       int mid = (low + high) / 2;
       if (S[mid]==key) return mid;
       else if (key < S[mid])
           return binsearch(low, mid-1, S, key);
       else
           return binsearch(mid+1, high, S, key);
    }
    else return -1;
}
```

- We conclude that $T(n) \epsilon \Theta(n^d \log n) = \Theta(\log n)$

Master Theorem

- Let $T(n)$ be a monotonically increasing function that satisfies

  $T(n) = a\,T(n/b) + f(n)$

  $T(1) = c$

  where $a \geq 1$, $b \geq 2$, $c>0$. If $f(n)$ is $\Theta(n^d)$ where $d \geq 0$ then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Decrease and Conquer (Giảm để trị)

- In many problems, it is not necessary to divide the problem into more than one subproblems, we need to reduce to just one subproblem.
- That technique is called Decrease and Conquer

Example for Decrease and Conquer technique is Binary search.

Example 1: https://codeforces.com/problemset/problem/279/B

B. Books

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

When Valera has got some free time, he goes to the library to read some books. Today he's got t free minutes to read. That's why Valera took n books in the library and for each book he estimated the time he is going to need to read it. Let's number the books by integers from 1 to n. Valera needs a_i minutes to read the i-th book.

Valera decided to choose an arbitrary book with number i and read the books one by one, starting from this book. In other words, he will first read book number i, then book number $i + 1$, then book number $i + 2$ and so on. He continues the process until he either runs out of the free time or finishes reading the n-th book. Valera reads each book up to the end, that is, he doesn't start reading the book if he doesn't have enough free time to finish reading it.

Print the maximum number of books Valera can read.

Input
The first line contains two integers n and t ($1 \le n \le 10^5$; $1 \le t \le 10^9$) — the number of books and the number of free minutes Valera's got. The second line contains a sequence of n integers $a_1, a_2, ..., a_n$ ($1 \le a_i \le 10^4$), where number a_i shows the number of minutes that the boy needs to read the i-th book.

Output
Print a single integer — the maximum number of books Valera can read.

Examples

input	output
4 5 3 1 2 1	3

input	output
3 3 2 2 3	1

Example 1: https://codeforces.com/problemset/problem/279/B

- Brute force: O(n^2)

Note about the problem: if the first book that Valera reads is the i^{th} book, then the next book that he reads must be $(i+1)^{th}$, then $(i+2)^{th}$, ... and the process continues until either he finishes reading the n^{th} book or the total reading time will be $> t$ if he reads the next book. Thus, Valera can have a total of n ways for reading as follows:

- 1st way: Start reading the 1st book, and continue to read the other books one by one until if he reads the next book, the total reading time will be $> t$
- 2nd way: Start reading the 2nd book, and continue to read the other books one by one until if he reads the next book, the total reading time will be $> t$
- …..

Go through all these n ways, each taking O(n) time to check the condition of total reading time. The solution of the problem is the maximum number of books that Valera can read such that the total reading time $\le t$.

Speed up ????

Example 1: https://codeforces.com/problemset/problem/279/B

- Binary search: O(*nlogn*)
  - If we use brute force, for each value i (corresponds to the way that reading starts from book i^{th}), we have to calculate the total time Valera reads the books and check if that total time exceeds the given time `t` or not. To avoid having to recompute each value of i, we'll use the array `sum`:
    - `sum[k] = sum[k-1] + time[k]` where `k >= 1`
    - `sum[k] = time[k]` where `k = 1`

    Time to calculate the array `sum` is $O(n)$.

Then the time from starting to read book i^{th} until finish reading book j^{th} is:

`sum[j] - sum[i] + time[i],` where `j > i`

`for i = 1 to n`

for each value i we need to find the largest `j` such that:

`i ≤j ≤n && sum[j]-sum[i]+time[i] ≤ t` → `takes O(n)` — O(n²)

Speed up??? → Binary search: $O(\log n)$ — ***O*(nlog*n*)**

Example 1: https://codeforces.com/problemset/problem/279/B

- Binary search: O(*nlogn*)

`for i = 1 to n`

for each value i we need to find largest `j` such that:

`i ≤j ≤n` and `sum[j]-sum[i]+time[i] ≤ t`

Binary search:
Find in array `sum` from elementh `i`th to `n`th, element `j`th satisfies **above condition**

```
int result = INT_MIN;
for (int i = 1; i <=n; i++) //bat dau doc tu quyen thu i
{
    int j = binary_search(sum, i, n, t + sum[i] - time[i]);
    //j-i+1 la so sach doc duoc neu bat dau doc tu quyen thu i
    result = max(result, j - i + 1);
}
cout << result << endl;
```

Exercise: Write function `int binary_search(int *sum, int low, int high, int target)` return value of index j that satisfies above condition

Binary search on integers

- Consider the function `int f(unsigned int x)` that takes as input a non-negative integer variable `x` and returns an integer. This function is a monotonically increasing function (tăng đơn điệu), that is, for any `unsigned int x`, we always have: `f(x+1) > f(x)`.

Requirements: Find the value n such that the value of function f becomes positive for the first time: i.e. $f(n\text{-}1), f(n\text{-}2), \ldots, f(0) < 0$, but $f(n) > 0$. Because f is a monotonically increasing function, so $f(n) > 0$ then $f(n+1), f(n+2), \ldots > 0$:

i	0	1	....	n-1	n	n+1	n+2	...	
f(i)	< 0	< 0		< 0	> 0	> 0	> 0		

Example: $f(x) = x^2 - 4x - 8$. Then the value need to find is $n = x = 6$

- Easiest way: $O(K*n)$

Brute force: Scan all $n = 0, 1, 2, \ldots$: for each n, we compute the value $f(n)$; stops when $f(n) > 0$

- Binary search: $O(K*\log n)$

where K is the cost to compute the value of function f

Binary search on integers

- Binary search: $O(\log n)$:

n	0	1	....	n-1	n	n+1	n+2	...	
f(n)	< 0	< 0		< 0	> 0	> 0	> 0		

To apply binary search on arrays, we need indicies `low` and `high`:

```
int binary_search(int *Arr, int low, int high, int target)
```

To find values for `low` and `high`, we will calculate the following values in turn:

f(i=0), f(i=1), f(i=2), f(i=4), f(i=8), f(i=16), …. , f(i=high)

i *=2: value of i doubles after every loop

```
low = high/2;
while (low < high) {
    int mid = (low + high)/2;
    if (f(mid) > 0) high = mid;
    else low = mid + 1;
}
return low;
```

where f(high) is the first value that function $f > 0$.

We will apply binary search with the variable `high` being the value we just found, and `low = high/2`

Binary search on integers

Example: Given the function f: {0,..,n-1}→ {T, F} satisfy the condition that if $f(i)$ = T then $f(j)$=T for all $j > i$

i	0	1	....	j-1	j	j+1	j+2	...	n-1
f(i)	F	F		F	T	T	T		T

In the range 0 to n-1, find the value j such that the function f gets the value T for the first time: it means all $f(0), f(1), \ldots, f(j\text{-}1)$ = T.

Answer: Apply binary search: $O(K*\log n)$

where K is the cost to compute the value of function f

```
int low = 0;
int high = n-1;

while (low < high)
{
    int mid = (low + high)/2;
    if (f(mid)==T) high = mid;
    else low = mid+1;
}

if (low == high && f(low) == T)
   cout<<"Chi so nho nhat tim duoc la "<<low;
else
   cout<<"Khong ton tai chi so thoa man de bai";
```


Binary search on integers

Given the function f: {0,..,n-1}→ {T, F} satisfy the condition that if $f(i)$ = T then $f(j)$=T for all $j > i$

i	0	1	....	j-1	j	j+1	j+2	...	n-1
f(i)	F	F		F	T	T	T		T

In the range 0 to n-1, find the value j such that the function f gets the value T for the first time: it means all $f(0), f(1), \ldots, f(j\text{-}1)$ = T.

Answer: Apply binary search: $O(K*\log n)$ where K is the cost to compute the value of function f

Application example: find position of x in array A already sorted in ascending order

Answer: Use binary search as above, with the function f as follows

```
bool f(int i)
{
   return (A[i] >= x);
}
```

```
int low = 0;
int high = n-1;

while (low < high)
{
    int mid = (low + high)/2;
    if (f(mid)==T) high = mid;
    else low = mid+1;
}

if (low == high && f(low) == T)
   cout<<"Chi so nho nhat tim duoc la "<<low;
else
   cout<<"Khong ton tai chi so thoa man de bai";
```

Binary search on real numbers

This is the general version of binary search:

- Given the function *f*: `[low, high]` → {T, F} satisfy that if *f*(*i*) = T then *f*(*j*) = T for all *j* > *i*.

Requirements: In the range `low` to `high`, find the smallest real number *j* such that the function *f* receives the value T for the first time (it means: *f*(*j*) = T as soon as possible).

- We know that when working with real numbers, the range `[low, high]` can be divided infinitely. So instead of finding the exact value of *j*, we can find the real value *j'* that is very close to the correct solution *j*, with an error within an acceptable range, for example error eps = 2^{-30} ($\approx 10^{-10}$).
  - We can do this in $O\left(log\frac{high-low}{eps}\right)$ like doing the binary search on an array:

```
int low = -1000.0;
int high = 1000.0;
double eps =1e-10;

while (high - low >eps)
{
    double mid = (low + high)/2;
    if (f(mid)==T) high = mid;
    else low = mid;
}
cout<<low<<endl;
```


Binary search on real numbers

This is the general version of binary search:

- Given the function *f*: `[low, high]` → {T, F} satisfy that if *f*(*i*) = T then *f*(*j*) = T for all *j* > *i*.

Requirements: In the range `low` to `high`, find the smallest real number *j* such that the function *f* receives the value T for the first time (it means: *f*(*j*) = T as soon as possible).

Example 1: Find the square root of *x*

Then, the function *f* should be

```
bool f(double j)
{
    return j*j >=x;
}
```

Example 2: Find the solution to g(*x*) = 0

Then, the function *f* should be

```
bool f(double x)
{
    return g(x) >= 0.0;
}
```


Example 2: Problem C from NWERC 2006

- NWERC (ACM Northwestern European Programming Contest)

Problem C: Pie

My birthday is coming up and traditionally I'm serving pie. Not just one pie, no, I have a number N of them, of various tastes and of various sizes. F of my friends are coming to my party and each of them gets a piece of pie. This should be one piece of one pie, not several small pieces since that looks messy. This piece can be one whole pie though.

My friends are very annoying and if one of them gets a bigger piece than the others, they start complaining. Therefore all of them should get equally sized (but not necessarily equally shaped) pieces, even if this leads to some pie getting spoiled (which is better than spoiling the party). Of course, I want a piece of pie for myself too, and that piece should also be of the same size.

What is the largest possible piece size all of us can get? All the pies are cylindrical in shape and they all have the same height 1, but the radii of the pies can be different.

Input

One line with a positive integer: the number of test cases. Then for each test case:

- One line with two integers N and F with $1 \leq N, F \leq 10000$: the number of pies and the number of friends.
- One line with N integers r_i with $1 \leq r_i \leq 10000$: the radii of the pies.

Output

For each test case, output one line with the largest possible volume V such that me and my friends can all get a pie piece of size V. The answer should be given as a floating point number with an absolute error of at most 10^{-3}.

Sample input	Sample output
3 3 3 4 3 3 1 24 5 10 5 1 4 2 3 4 5 6 5 4 2	25.1327 3.1416 50.2655

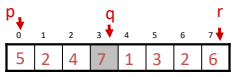
Contents of Chapter 3

Divide and conquer

Problem: Sorting n elements of an array A[0]…A[n-1]

- **Merge sort** on an input sequence A with n elements consists of three steps:
  - Divide: partition A into two sequences A_1 and A_2 of about $n/2$ elements each
  - Conquer: recursively sort A_1 and A_2 by using merge sort
  - Combine: merge A_1 and A_2 into a unique sorted sequence
- **Quick sort**:
  - Divide: partition A into two sets: the first set consisting of "small" elements is placed on the left of the array and the second set consisting of "large" elements is placed on the right of the array.
  - Conquer: recursive sort these 2 sets using quick sort. If set consists of one element, then do not need to do anything as it is already sorted.
  - Combine: the obtained array after conquer step is already sorted, do not need to do anything

Merge Sort

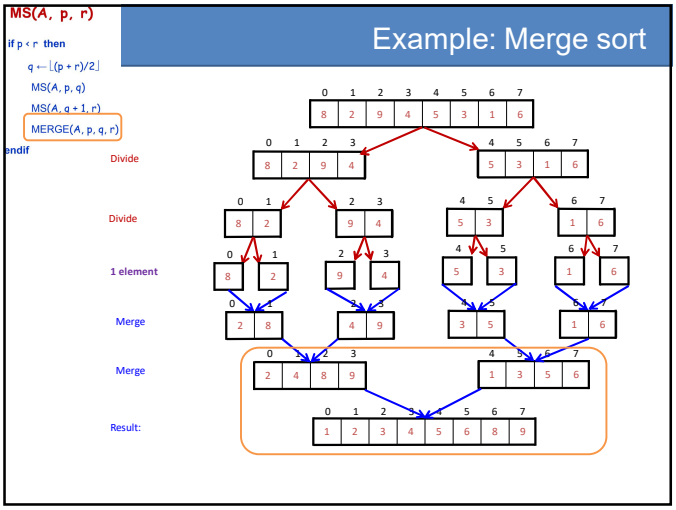


MS(A, p, r)

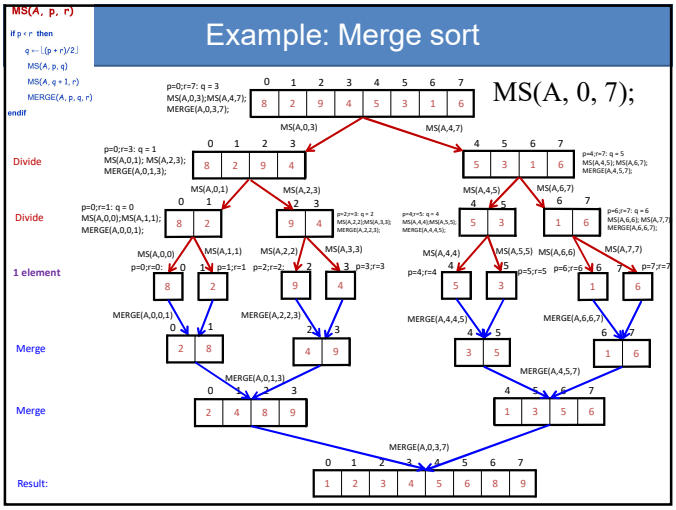
```
if p < r  then                          ▷ Check the base case
    q ← ⌊(p + r)/2⌋                     ▷ Divide
    MS(A, p, q)                         ▷ Conquer
    MS(A, q + 1, r)                     ▷ Conquer
    MERGE(A, p, q, r)                   ▷ Combine
endif
```

- The statement to sort array A of n elements: MS(A, 0, n-1);

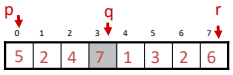
Example: Merge sort



Example: Merge sort



Merge Sort



MS(*A*, p, r)

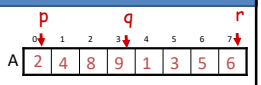
```
if p < r  then                          ▷ Check the base case
    q ← ⌊(p + r)/2⌋                     ▷ Divide
    MS(A, p, q)                         ▷ Conquer
    MS(A, q + 1, r)                     ▷ Conquer
    MERGE(A, p, q, r)                   ▷ Combine
endif
```

Merge 2 ordered subarrays A[p]...A[q] and A[q+1]...A[r] into one array that is ordered in ascending order

Merge 2 ordered subarrays A[p]...A[q] and A[q+1]...A[r] into one array that is ordered in ascending order

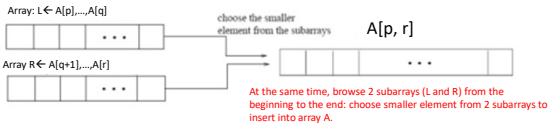
Array A consists of 2 ordered halves:

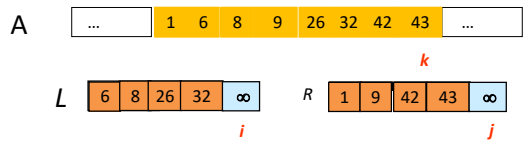
- Subarray 1: consists of elements A[p]… A[q]
- Subarray 2: consists of elements A[q+1]… A[r]



We need to merge these 2 subarrays to get an array that is ordered in ascending order:

Idea 1: we use 2 extra subarrays L, R with size = ½ size of array A



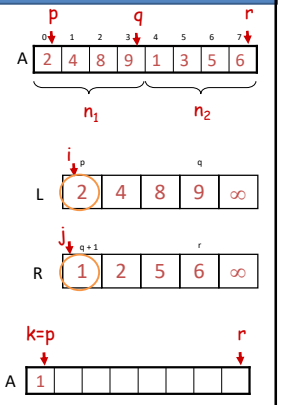


46

Merge - Pseudocode

MERGE(A, p, q, r)

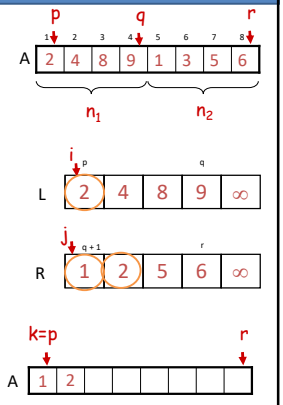
1. Calculate n_1 and n_2
2. Copy the first n_1 elements to $L[1 . . n_1]$ and remain n_2 elements to $R[1 . . n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \le R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$



Merge - Pseudocode

MERGE(A, p, q, r)

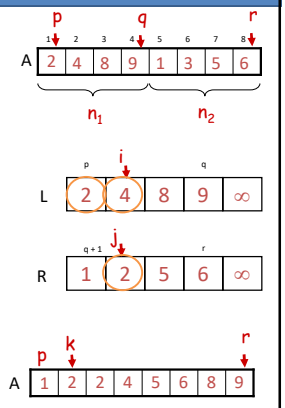
1. Calculate n_1 and n_2
2. Copy the first n_1 elements to $L[1 . . n_1]$ and remain n_2 elements to $R[1 . . n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \le R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$



Merge - Pseudocode

MERGE(A, p, q, r)

1. Calculate n_1 and n_2
2. Copy the first n_1 elements to $L[1 \ldots n_1]$ and remain n_2 elements to $R[1 \ldots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \le R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$



Computation time analysis Merge sort: O(*n*log*n*)

Running time of Merge procedure:

- Initialize (create 2 subarrays L and R):
  - $O(n_1 + n_2) = O(n)$
- Insert elements into array S (the loop **for**):
  - n loops, each loop requires constant time $\Rightarrow O(n)$
- Total running time of Merge procedure: $O(n)$

Running time of Merge Sort:

- **Divide:** calculate q as the average of p and r: O(1)
- **Conquer:** solve 2 sub problems with size $n/2$ each $\Rightarrow 2T(n/2)$
- **Combine:** MERGE 2 sub arrays of n elements requires $O(n)$

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(n) & \text{if } n > 1 \end{cases}$$

Therefore: $T(n) = O(n \log n)$

MERGE(A, p, q, r)

1. Calculate n_1 and n_2
2. Copy the first n_1 elements to $L[1 \ldots n_1]$ and remain n_2 elements to $R[1 \ldots n_2]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r **do**
6. **if** $L[i] \le R[j]$
7. **then** $A[k] \leftarrow L[i]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[j]$
10. $j \leftarrow j + 1$

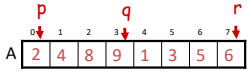
MS(A, p, r)

if p < r **then**
  q ← ⌊(p + r)/2⌋
  MS(A, p, q)
  MS(A, q + 1, r)
  MERGE(A, p, q, r)
endif

Merge 2 ordered subarrays A[p]...A[q] and A[q+1]...A[r] into one array that is ordered in ascending order

Array A consists of 2 ordered halves:

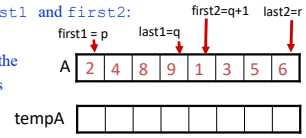
- Subarray 1: consists of elements A[p]… A[q]
- Subarray 2: consists of elements A[q+1]… A[r]



We need to merge these 2 subarrays to get an array that is ordered in ascending order:

Idea 2: use a temporary array tempA with the same size of array A

At the same time scan through each element of two sub-sequences A [p..q] and A [q + 1… r] by using the variable `first1` and `first2`: compare each pair of corresponding 2 elements of 2 sub-sequences, selects smaller element to copy to the sub array tempA. At the end of the loop, all elements of the two sub-sequences have been scanned; then the tempA array contains all the elements of the two sub-sequences, but they are are sorted. Do A = tempA

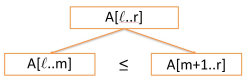


51

Quick sort

The quick sort algorithm is described recursively as following (similar to merge sort):

1. **Base case**. If the array has only one element, then the array is sorted already, return it without doing anything.
2. **Divide**:
   - Select an element in the array, and call it as the pivot p.
   - Divide the array into 2 subarrays: Left subarray (L) consists of elements $\le$ the pivot, right subarray (R) consists of elements $\ge$ the pivot. This operation is called "Partition".

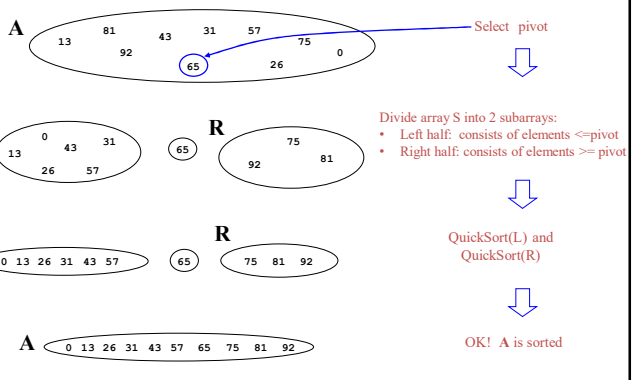


3. **Conquer**: recursively call QuickSort for 2 subarrays $L = A[l \ldots m]$ and $R = A[m+1 \ldots r]$.
4. **Combine**: The sorted array is $L\ p\ R$.

In contrast to Merge Sort, in Quick Sort: division operation is complicated, but the Partition operation is simple.

52

Quick sort: Example



Quick sort diagram

Quick-Sort(*A*, *Left*, *Right*)

```
1.  if (Left < Right ) {
2.          idPivot = Partition(A, Left, Right);
3.          Quick-Sort(A, Left, idPivot –1);
4.          Quick-Sort(A, idPivot +1, Right);
5.   }
```

Function Partition(*A*, *Left*, *Right*) divides *A*[*Left*..*Right*] into 2 subarrays *A*[*Left*..*idPivot* –1] and *A*[id*Pivot*+1..*Right*] such that:

- Elements of *A*[*Left*..*idPivot* –1] is less than or equal to *A*[id*Pivot*]
- Elements of *A*[id*Pivot*+1..*Right*] is greater than or equal to *A*[id*Pivot*].

The statement to sort array A consists of *n* elements: Quick-Sort(*A*, 0, *n*-1)

Quick sort

The selection of the pivot plays a decisive role in the efficiency of the algorithm. It is best if the selected pivot is the one with the average value in the array (we call such an element median). Then, after $\log_2 n$ times of partition, we reach an array of size 1. However, that is very difficult to do. It is common to use the following ways to select the pivot element:

- Select the left most (first) element as the pivot.
- Select the rightmost (last) element as the pivot.
- Select the element in the middle of the array as the pivot.
- Select the median element in the 3 elements: top, middle and the last as the pivot element (Knuth).
- Randomly selects an element as the pivot.

Quick sort

- Computation time:

$$T(n) = T(k) + T(n-k-1) + O(n)$$

k is number of elements on the left.

Worst case:

$T(n) = T(0) + T(n-1) + O(n) = T(n-1) + O(n)$

➔ $T(n) = O(n^2)$

Best case:

$T(n) = 2T(n/2) + O(n)$

➔ $T(n) = O(n\log n)$

Contents of Chapter 3

2.2.3. Multiply 2 integer numbers

```
        9 8 1
      1 2 3 4
    -----------
      3 9 2 4
    2 9 4 3
  1 9 6 2
  9 8 1
-----------------
1 2 1 0 5 5 4
```

- If each operand has n digits, then computation time is $\Theta(n^2)$
- Is there a better way?

2.2.3. Multiply 2 integer numbers

- Problem: Given

  $x = x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ and

  $y = y_{n-1}\, y_{n-2} \ldots y_1\, y_0$

2 positive integer numbers with n digits. We need to calculate

$z = z_{2n-1}\, z_{2n-2} \ldots z_1\, z_0$

representing product of xy with $2n$ digits.

2.2.3. Multiply 2 integer numbers - Karatsuba algorithm (1962)

- We have: $x = x_{n-1}\, x_{n-2} \ldots x_1\, x_0$ and $y = y_{n-1}\, y_{n-2} \ldots y_1\, y_0$

➔ $x = x_{n-1} \times 10^{n-1} + x_{n-2} \times 10^{n-2} + \ldots + x_1 \times 10^1 + x_0 \times 10^0$

$y = y_{n-1} \times 10^{n-1} + y_{n-2} \times 10^{n-2} + \ldots + y_1 \times 10^1 + y_0 \times 10^0$

- Thus: $z = z_{2n-1}\, z_{2n-2} \ldots z_1\, z_0 = x * y$

$$z = z_{2n-1} \times 10^{2n-1} + z_{2n-2} \times 10^{2n-2} + \ldots + z_1 \times 10^1 + z_0 \times 10^0$$
$$= (x_{n-1} \times 10^{n-1} + x_{n-2} \times 10^{n-2} + \ldots + x_1 \times 10^1 + x_0 \times 10^0) \times$$
$$\times (y_{n-1} \times 10^{n-1} + y_{n-2} \times 10^{n-2} + \ldots + y_1 \times 10^1 + y_0 \times 10^0)$$

2.2.3. Multiply 2 integer numbers - Karatsuba algorithm (1962)

We have: $x = x_{n-1}\, x_{n-2} \dots x_1\, x_0$ and $y = y_{n-1}\, y_{n-2} \dots y_1\, y_0$

- Set: $a = x_{n-1}\, x_{n-2} \dots x_{n/2+1}\, x_{n/2}$,
  $b = x_{n/2-1}\, x_{n/2-2} \dots x_1\, x_0$,
  $c = y_{n-1}\, y_{n-2} \dots y_{n/2+1}\, y_{n/2}$,
  $d = y_{n/2-1}\, y_{n/2-2} \dots y_1\, y_0$.

  Then: $x = a\times10^{n/2} + b$;
  $y = c\times10^{n/2} + d$,
- So: $z = x*y = (a\times10^{n/2} + b) \times (c\times10^{n/2} + d)$
  $= (a\times c)\, 10^n + (a\times d + b\times c)\, 10^{n/2} + b\times d.$

➔ To calculate $a\times c$, $a\times d$, $b\times c$, $b\times d$ we must perform 4 multiplications of two $n/2$-digit numbers.

➔ The given problem requires multiplication of two n-digit numbers (x and y): is transferred to problem of calculating 4 multiplications of two $n/2$-digit numbers

2.2.3. Multiply 2 integer numbers - Karatsuba algorithm (1962)

- **Basic case:** The multiplication of two 1-digit integers can be done directly;
- **Divide:** if $n>1$ then the product of 2 integers with n digits can be represented through 4 products of 4 integer numbers with $n/2$ digits: $a*c$, $a*d$, $b*c$, $b*d$
- **Combine:** to calculate $z = xy$ when we already know the above 4 products, we only need to perform additions (can be done in $O(n)$) and multiply with power of 10 (can be done in $O(n)$, by inserting an appropriate number of digits 0 to the right).

  $x = a\times10^{n/2} + b$;
  $y = c\times10^{n/2} + d$,
  $z = x*y = (a\times10^{n/2} + b) \times (c\times10^{n/2} + d)$
  $= (a\times c)\, 10^n + (a\times d + b\times c)\, 10^{n/2} + b\times d.$
- Evaluate the computation time $T(n)$ of the algorithm:
  $T(1) = 1$, $n = 1$
  $T(n) = 4T(n/2) + n$, $n>1$
  Using the master theorem: $T(n) = O(n^2)$

➡ Is it possible to speed up??

2.2.3. Multiply 2 integer numbers - Karatsuba algorithm (1962)

Karatsuba discovered how to perform 2 integer n-digit numbers multiplication requires only 3 multiplications of $n/2$-digit numbers as following:

- Set: $U = a\times c$, $V = b\times d$, $W = (a+b)\times(c+d)$

Then: $a\times d + b\times c = W - U - V$,

And calculate:

$z = x*y = (a\times10^{n/2} + b) \times (c\times10^{n/2} + d)$
$= (a\times c)\, 10^n + (a\times d + b\times c)\, 10^{n/2} + b\times d$
$= U\times10^n + (W-U-V)\times10^{n/2} + V.$

Karatsuba algorithm

- Set: $U = a\times c$, $V = b\times d$, $W = (a+b)\times(c+d)$

Then: $a\times d + b\times c = W - U - V$,

And calculate:

$z = x*y = (a\times10^{n/2} + b) \times (c\times10^{n/2} + d)$
$= (a\times c)\, 10^n + (a\times d + b\times c)\, 10^{n/2} + b\times d$
$= U\times10^n + (W-U-V)\times10^{n/2} + V.$

```
function Karatsuba(x, y, n)
begin
  if n=1 then  return x[0]*y[0]
  else
  begin
      a:= x[n-1] ... x[n/2];
      b:= x[n/2 -1] ... x[0];
      c:= y[n-1] ... y[n/2];
      d:= y[n/2-1] ... y[0];
      U:= Karatsuba(a, c, n/2);
      V:= Karatsuba(b, d, n/2);
      W:= Karatsuba(a+b, c+d, n/2);
      return U*10^n + (W-U-V)*10^(n/2) + V;
  end;
end;
```

Evaluate the computation time $T(n)$ of the algorithm:
$T(1) = 1$, $n = 1$
$T(n) = 3T(n/2) + cn$, $n>1$
Using the master theorem: $T(n) = O(n^{log_2 3})$
As $\log_2 3 = 1.585 < 2$ ➔ already speed up

Contents of Chapter 3

2.4. Multiply matrices

Given $A = \{a_{ik}\}$ and $B = \{b_{kj}\}$ the two matrices of size $n \times n$.

We call the product of two matrices A and B the matrix $C=\{c_{ij}\}$ of size $n \times n$, denoted by $C = AB$, with the elements calculated by the formula

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

2.4. Multiply matrices: direct algorithm

```
procedure NaiveMatrixMultiplication()
begin
   for i=1 to n do
     for j=1 to n do
     begin
        c[i,j]=0;
        for k=1 to n do
          c[i,j] = c[i,j] +a[i,k]*b[k,j];
     end;
end;
```

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Computation time: $O(n^3)$

2.4. Multiply matrices: divide and conquer

- Divide matrix of size $n \times n$ into 4 matrices of size $(n/2) \times (n/2)$. We have

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Where

$$C_{11} = A_{11}B_{11} + A_{12}B_{21},$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22},$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21},$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

8 matrix multiplications of size $(n/2)$x$(n/2)$
4 matrix additions of size $(n/2)$ x $(n/2)$

Let $T(n)$ the computation time when calculating the product of two matrices size $n \times n$ then:

$$T(n) = 8T(n/2) + O(n^2)$$

Apply Master theorem, we have: $T(n) = O(n^3)$

→No more efficient than the `NaiveMatrixMultiplication`

Multiply matrices: Strassen formular

Calculating 7 matrix products :

$P_1=A_{11}(B_{12}-B_{22})$

$P_2=(A_{11}+A_{12})B_{22}$

$P_3=(A_{21}+A_{22})B_{11}$

$P_4=A_{22}(B_{21}-B_{11})$

$P_5=(A_{11}+A_{22})(B_{11}+B_{22})$

$P_6=(A_{12}-A_{22})(B_{21}+B_{22})$

$P_7=(A_{11}-A_{21})(B_{11}+B_{12})$

Then:

$C_{11}=-P_2+P_4+P_5+P_6$

$C_{12}= P_1+P_2$

$C_{21}= P_3+P_4$

$C_{22}=P_1-P_3+P_5-P_7$

==> 7 multiplications and 18 addition/subtraction on matrices of size $(n/2)$ x $(n/2)$

Strassen formular: divide and conquer

Divide and conquer algorithm for matrix product: $C_{n\times n} = A_{n\times n} * B_{n\times n}$

- **Divide**: Divide matrices $A_{n\times n}$ and $B_{n\times n}$ into submatrices of size $(n/2)$ x $(n/2)$
- **Conquer**: Perform 7 multiplications on submatrices of size $(n/2)$ x $(n/2)$ recursively
- **Combine**: Calculate the resulting matrix $C_{n\times n}$ using operations + and – on submatrices of size $(n/2)$ x $(n/2)$

Then, we have the computation time is

$$T(n) = 7T(n/2) + O(n^2)$$

Apply Master theorem, we have: $T(n) = O(n^{\log 7}) = O(n^{2.81})$

→ Speed up when comparing with $O(n^3)$

Contents of Chapter 3

2.5. Binary exponentiation

Problem: Calculate x^n , where x, n are integer numbers

The easy way to implement: $x^n = \underbrace{x*x*x*\ldots.*x}_{n \text{ times}}$

```
int pow(int x, int n)
{
    int res = 1;
    for (int i = 0; i < n; i++) res = res * x;
    return res;
}
```

Computation time: $O(n)$

➔ How to speed up ??

2.5. Binary exponentiation: divide and conquer

Comment (1):

- $x^0 = 1$
- $x^n = x \times x^{n-1}$

➔ Need to write function pow:

- pow(x,0) = 1
- pow(x, n) = x × pow(x, n − 1)

```
int pow(int x, int n)
{
   if (n == 0) return 1;
   return x * pow(x, n - 1);
}
```

➔ Computation time: T(n) = 1 + T(n-1) ➔ T(n) = O(n)

2.5. Binary exponentiation: divide and conquer

Comment (2): $x^n = x^{n/2} \times x^{n/2}$

➔ pow(x, n) = pow(x, n/2) × pow(x, n/2)

pow(x, n/2) is used twice, but we only need to calculate it once:

pow(x, n) = pow(x, n/2)²

➔ We can use it to speed up in case n is even because n/2 is an integer

```
int pow(int x, int n)
{
   if (n == 0) return 1;
   if (n%2 !=0)
     return x * pow(x, n - 1);
   int tmp = pow(x, n/2);
   return tmp*tmp;
}
```

Computation time:

T(n) = 1 + T(n/2) if n is even

T(n) = 1 + T(n-1) if n is odd

= 1 + [1 + T((n-1)/2)]

➔ T(n) = O(logn) ➔ Speed up

2.5. Binary exponentiation

```
int pow(int x, int n)
{
   if (n == 0) return 1;
   if (n%2 !=0)
     return x * pow(x, n - 1);
   int tmp = pow(x, n/2);
   return tmp*tmp;
}
```

The above algorithm can be applied to:

- Calculate x^n when x is a real number, and * is multiplication on real numbers
- Calculate A^n when A is a matrix, and * is multiplication on matrices
- Calculate x^n (mod m) when x is a matrix, and * is congruent integer multiplication m
- Calculate $x*x*\ldots*x$ when x is any element and * is any associative operator.

Computation time: O(f * log$n$) where f is the computation time to implement the operator *

Contents of Chapter 3

2.6. Fibonacci word

The Fibonacci sequence is defined recursively as follows:

- $Fib_1 = 1$
- $Fib_2 = 1$
- $Fib_n = Fib_{n-2} + Fib_{n-1}$

Thus, we obtain Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21…

Using the first condition $F_1 = 5$ and $F_2 = 4$ ➔ we have the sequence 5, 4, 9, 13, 22, 35, 57,…

What happened if first condition is not a number ?

Example: The first condition is a character, the + denotes a string concatenation

- $g_1 = A$
- $g_2 = B$
- $g_n = g_{n-2} + g_{n-1}$

➔ We have sequences on characters:

- ✓ A
- ✓ B
- ✓ AB
- ✓ BAB
- ✓ ABBAB
- ✓ BABABBAB
- ✓ ABBABBABABBAB
- ✓ BABABBABABBABBABABBAB
- ✓ .....

2.6. Fibonacci word

Example: The first condition is a character, the + denotes a string concatenation

- $g_1 = A$
- $g_2 = B$
- $g_n = g_{n-2} + g_{n-1}$

Length of g_n:

- $length(g_1) = 1$
- $length(g_2) = 1$
- $length(g_n) = length(g_{n-2}) + length(g_{n-1})$

➔ $length(g_n) = Fib_n$

So character strings quickly become very long:

- $length(g_{10}) = 55$
- $length(g_{100}) = 354224848179261915075$
- $length(g_{1000}) = 43466557686937456435688527675040625802564660517371780402481729089536555417949051890403879840079255169295922593080322634775209689623239873322471161642996440906533187938298969649928516003704476137795166849228875$

- ✓ A
- ✓ B
- ✓ AB
- ✓ BAB
- ✓ ABBAB
- ✓ BABABBAB
- ✓ ABBABBABABBAB
- ✓ BABABBABABBABBABABBAB
- ✓ .....

- $Fib_1 = 1$
- $Fib_2 = 1$
- $Fib_n = Fib_{n-2} + Fib_{n-1}$

2.6. Fibonacci word

Example: The first condition is a character, the + denotes a string concatenation

- $g_1 = A$
- $g_2 = B$
- $g_n = g_{n-2} + g_{n-1}$

➔ $length(g_n) = Fib_n$

- ✓ A
- ✓ B
- ✓ AB
- ✓ BAB
- ✓ ABBAB
- ✓ BABABBAB
- ✓ ABBABBABABBAB ← g_7
- ✓ BABABBABABBABBABABBAB
- ✓ .....

Requirements: Find the i^{th} character in g_n

For example: find the $i = 3^{th}$ character in g_7 = ABBABBABABBAB ➔ the desired character is B

- Method 1: Easy to find in $O(length(g_n))$ ➔ it becomes slow when n is large
  - Construct g_n, then give the character i th of g_n.

2.6. Fibonacci word

- Method 2: Divide and conquer O(n)
  - Step 1: No need to construct g_n, just find the smallest k such that g_k has length $\geq i$. Then the i^{th} character in g_n is also the i^{th} character in g_k.

    Write the function `int findIndex(int i)` that returns the smallest k such that g_k has length $\geq i$
  - Step 2: From the formular: $g_k = g_{k-2} + g_{k-1}$ ➔ need to determine whether the i^{th} character of g_k belongs to g_{k-2} or g_{k-1}
    - If $i \leq length(g_{k-2})$ then the i^{th} character belongs to g_{k-2}
      - Then the i^{th} character of g_k is the ith character of g_{k-2}. Repeat step 2 with $k = k-2$
    - else the i^{th} character belongs to g_{k-1}
      - Then the i^{th} character of g_k is the (i-$length(g_{k-2})$)th character of g_{k-1}.Repeat step 2 with $k = k-1$

    Repeat step 2 with decreasing k value until $k <= 2$

Write the function `int findLengthTerm(int k)` returns the length of g_k

Write the function `char Fibonacci(int n, int i)` uses two functions `findIndex` and `findLengthTerm` to solve the problem

Assignment: Fibonacci word

For any two strings of digits, A and B, we define $F_{A,B}$ to be the sequence (A, B, AB, BAB, $ABBAB$, ...) in which each term is the concatenation of the previous two.

Further, we define $D_{A,B}(n)$ to be the n-th digit in the first term of $F_{A,B}$ that contains at least n digits.

Example:

Let $A = 1415926535$, $B = 8979323846$. We wish to find $D_{A,B}(35)$, say.

The first few terms of $F_{A,B}$ are:

- 1415926535
- 8979323846
- 14159265358979323846
- $897932384614159265358979323846$
- $1415926535897932384689793238461415926535897932384 6$

Then $D_{A,B}(35)$ is the 35-th digit in the fifth term, which is 9.

You are given q triples (A, B, n). For all of them find $D_{A,B}(n)$.

Assignment: Fibonacci word

Input Format

First line of each test file contains a single integer q that is the number of triples. Then q lines follow, each containing two strings of decimal digits a and b and positive integer n.

Constraints

- $1 \leq q \leq 100$
- $1 \leq length(a), length(b) \leq 100$
- $1 \leq n \leq 2^{100}$

Output Format

Print exactly q lines with a single decimal digit on each: value of $D_{A,B}(n)$ for the corresponding triple.

Sample Input 0

```
2
1415926535 8979323846 35
14159265358979323846264338327950288419716939937510582097494459230781640628620899862803
48253421170679
82148086513282306647093844609550582231725359408128481117450284102701938521105559644622
94895493038196 104683731294243150
```

Sample Output 0

```
9
8
```