Algorithms and Data Structures

Quicksort

Outline

Quicksort

- Basic algorithm
- Performance
- Improvement

Extra topic

- Hidden cost of Recursive programs
  - Call stack
  - While-program with stack
  - Space complexity of Quicksort

Quicksort

(Invented in 1960 by C. A. R. Hoare)

- Easy to implement
- Efficient for various kinds of input
- Little computing resource

O(N logN) time (Average)

O(N^2) time (Worst case)

This can be avoided in practice

Memory Space

- Use a stack of length **O(log N)** on average

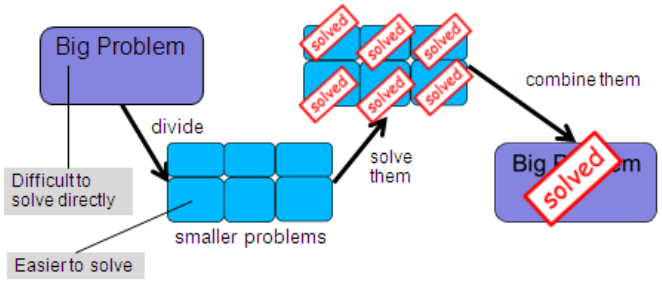
extra working space

Quicksort: basic algorithm

Divide and Conquer + Recursive program

Divide-and-Conquer Algorithm

It is a general approach to solve problems



To solve a smaller problem, we may need to solve more smaller problems.
This mechanism essentially uses recursive calls

13

Recursive Programs

A program (function) that calls **itself**

Recursive call

An image of Recursive Call
Matryoshka doll (Russian toy)

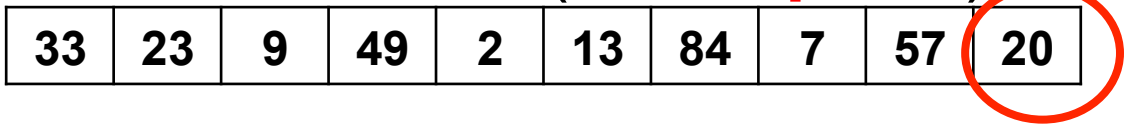
4

Quicksort: basic algorithm

(0) If the length of given array is 1, then do nothing

Otherwise,

(1) Pick an element (called **pivot**) from given array



(2) **Partition** the array into two parts

the part of (≤20) and the part of (20≤)

Repeat (0) - (3) **recursively**

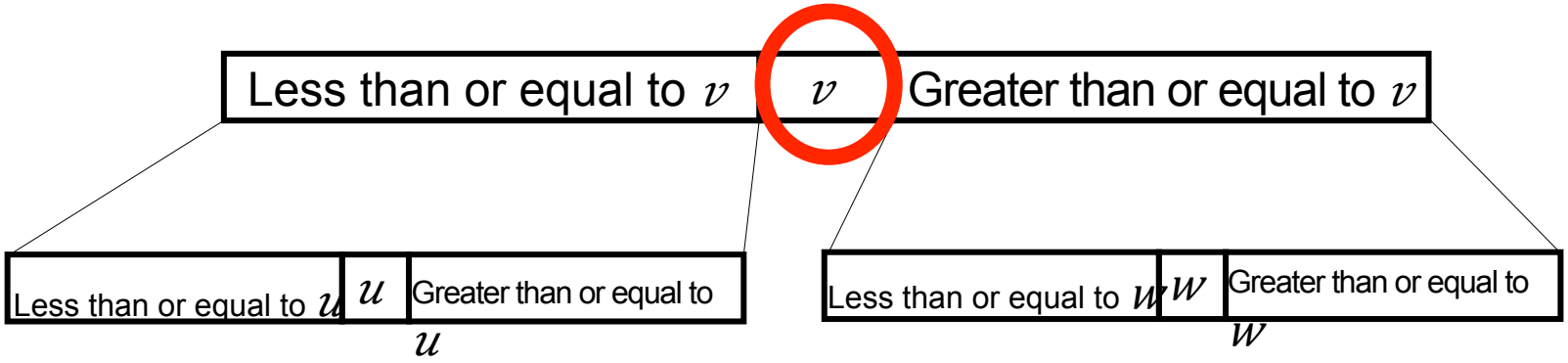
(3) Sort the two small parts and concatenate them

Quicksort: basic algorithm

Divide and Conquer + Recursive Program

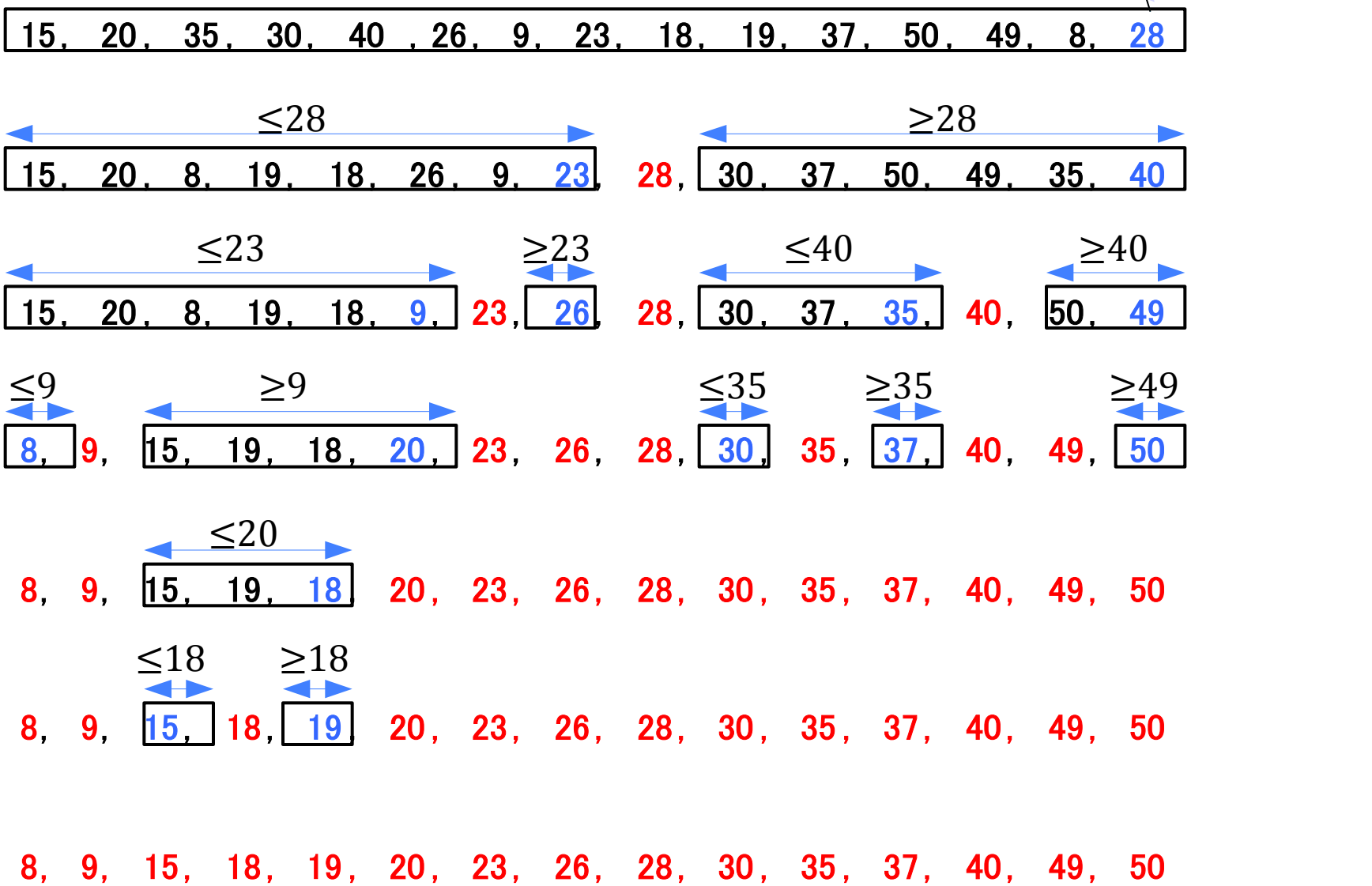
The current big problem (sorting of a big array) becomes **smaller independent problems**

This D&C approach will produce an **efficent algorithm**



Quicksort: basic algorithm

In each step, take the last element as a pivot



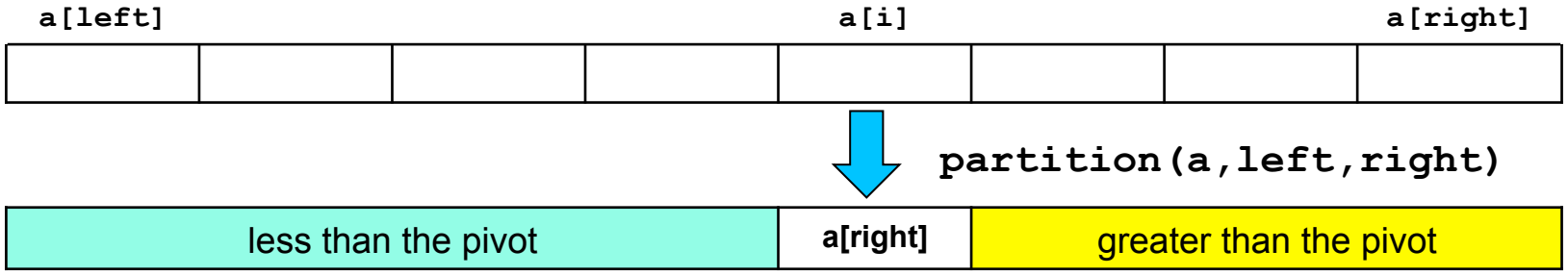
Quicksort: basic algorithm

quicksort.c

```
void quicksort(Item a[],int left,int right){
  int i;
  if (right <= left) return;
  i = partition(a,left,right);
  quicksort(a,left,i-1);
  quicksort(a,i+1,right);
 }
```

Divide the array and return the partition position `i`

`partition(a,x,y)` partitions the array `a[x],..a[y]` taking `a[y]` as the pivot, and returns the partition position `i`

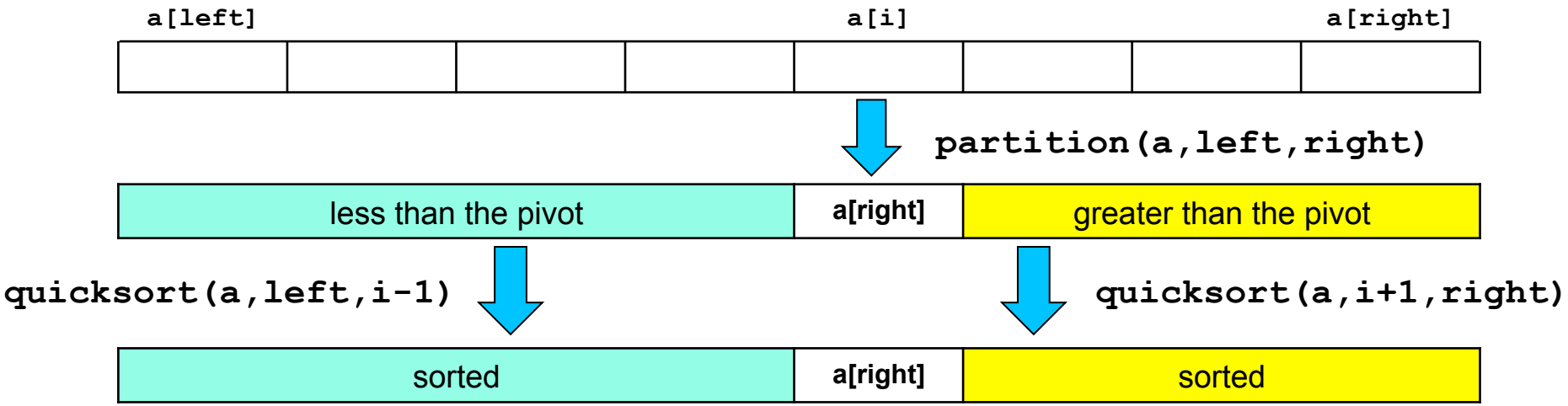


Quicksort: basic algorithm

quicksort.c

```
void quicksort(Item a[],int left,int right){
  int i;
  if (right <= left) return;
  i = partition(a,left,right);
  quicksort(a,left,i-1);
  quicksort(a,i+1,right);
 }
```

`quicksort` are recursively called



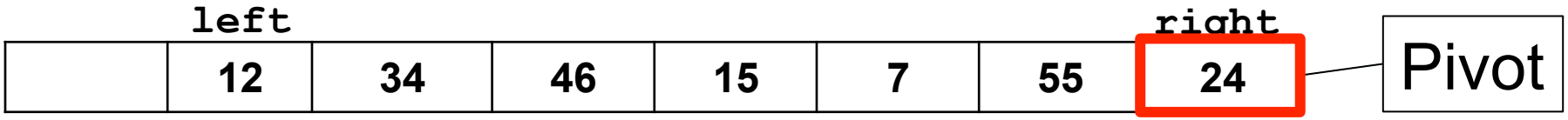
Quicksort (`partition`)

quicksort.c (cont.)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

(0) Take the right most element as the pivot

Set `pivot = a[right]`

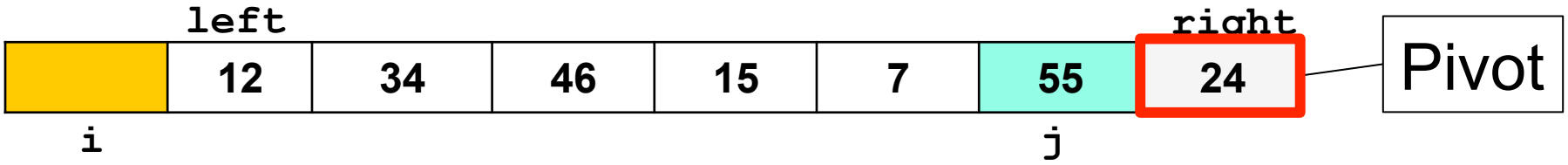


Quicksort (`partition`)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

quicksort.c (cont.)

(1) Set `i = left-1` and `j = right-1`



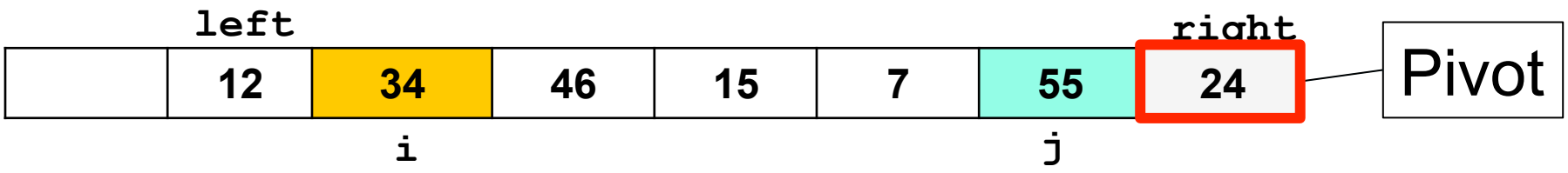
Quicksort (`partition`)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

`quicksort.c` (cont.)

(2) Do `while(less(a[++i],pivot));`

(This loop stops when `i` points to the leftmost element which is greater than or equal to `pivot`. It always stops since `i` points to `pivot` when `i = r`)



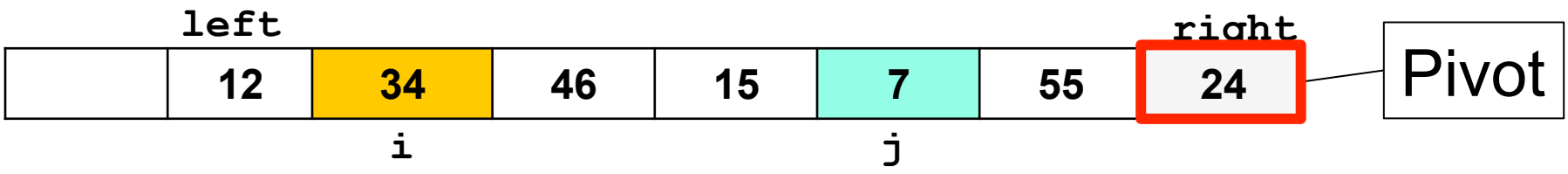
Quicksort (`partition`)

quicksort.c (cont.)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

(3) Do `while(less(pivot, a[j])){j--; ...}`

(This loop stops when `j` points to the rightmost element which is less than or equal to `pivot`, or `i>=j` holds)

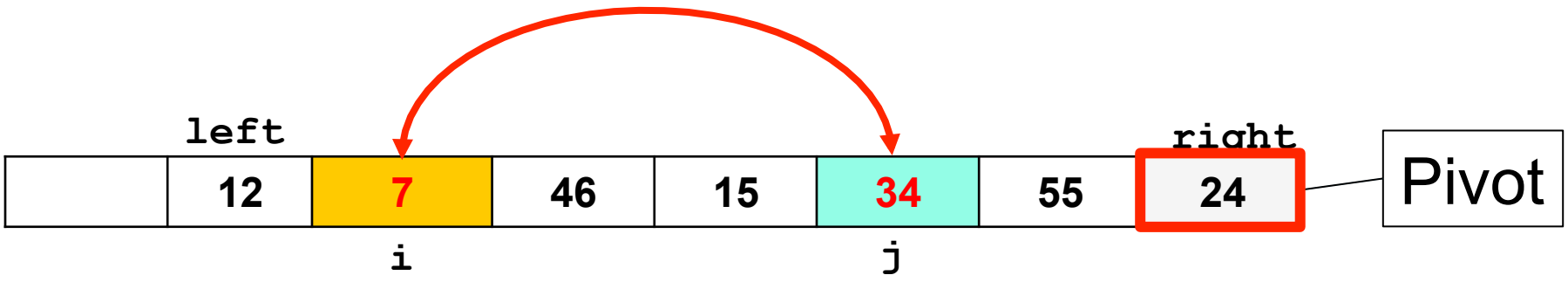


Quicksort (`partition`)

`quicksort.c` (cont.)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

(4) Exchange `a[i]` and `a[j]`

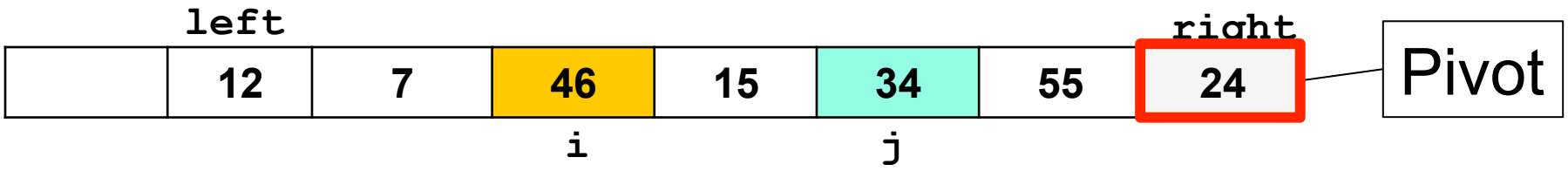


Quicksort (`partition`)

quicksort.c (cont.)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

(2) Do `while(less(a[++i],pivot);` again

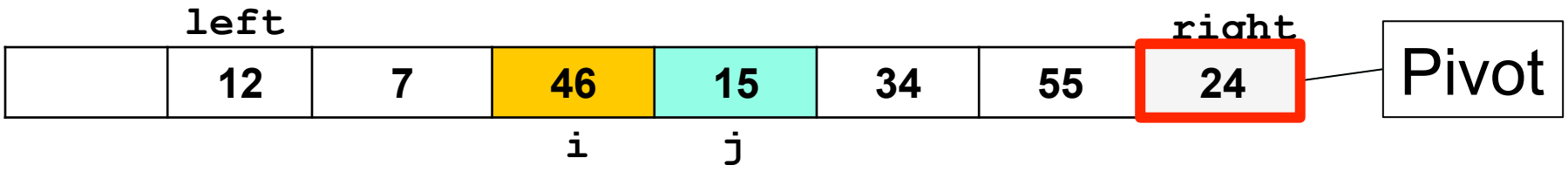


Quicksort (`partition`)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

quicksort.c (cont.)

(3) Do `while(less(pivot, a[j])){j--; ...}` again

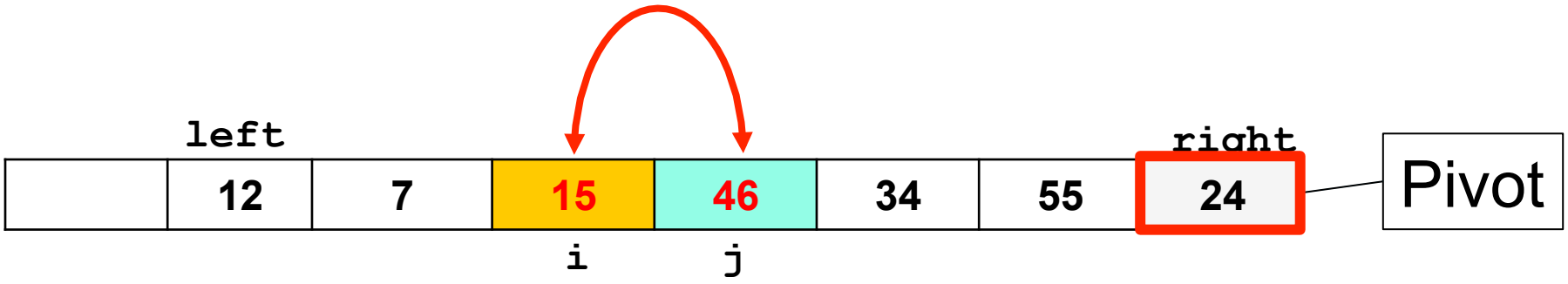


Quicksort (`partition`)

`quicksort.c` (cont.)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

(4) Exchange `a[i]` and `a[j]`

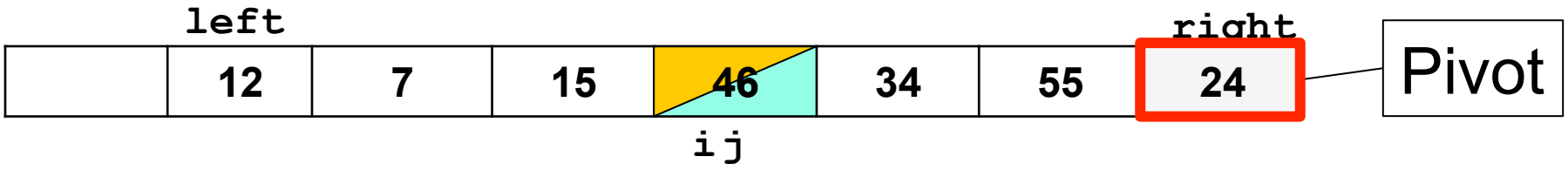


Quicksort (`partition`)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

`quicksort.c` (cont.)

(2) Do `while(less(a[++i],pivot));` again

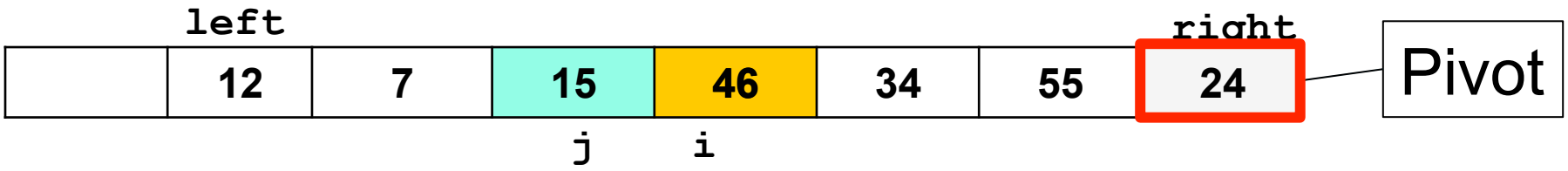


Quicksort (`partition`)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

`quicksort.c` (cont.)

(3) Do `while(less(pivot,a[j])){..if(i>=j) break;};`
Then escape from this **inner** `while`-loop by `break`

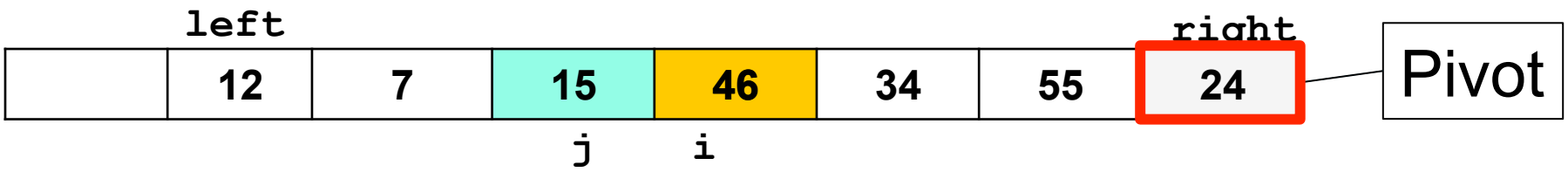


Quicksort (`partition`)

`quicksort.c` (cont.)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

(5) Escape from the **outer** `while`-loop

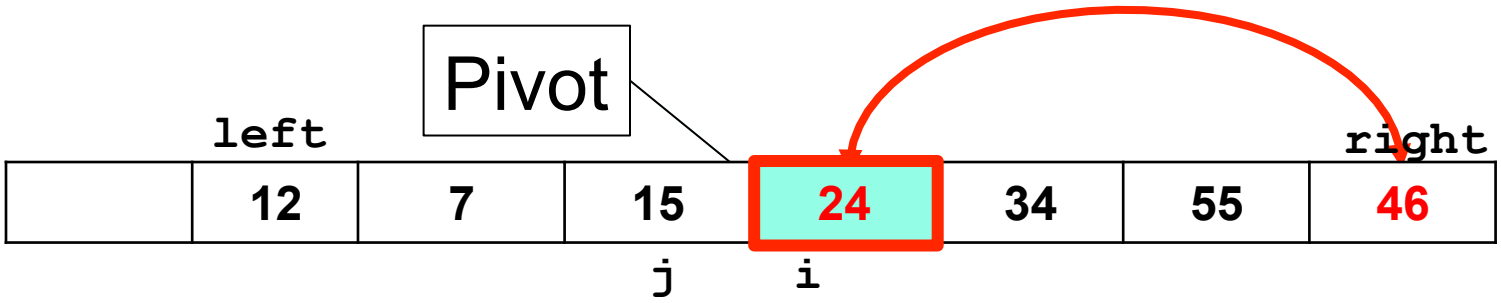


Quicksort (`partition`)

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

(6) Exchange `a[i]` and `a[right]`

(7) Return `i`



The process of quicksort

5, 7, 2, 3, 8, 1, 6, 4	Start quicksorting
5, 7, 2, 3, 8, 1, 6, 4	Start partitioning
5, 7, 2, 3, 8, 1, 6, 4	move `i` and `j` `(a[i]=5,a[j]=1)`
1, 7, 2, 3, 8, 5, 6, 4	exchange
1, 7, 2, 3, 8, 5, 6, 4	move `i` and `j` `(a[i]=7,a[j]=3)`
1, 3, 2, 7, 8, 5, 6, 4	exchange
1, 3, 2, 7, 8, 5, 6, 4	move `i` and `j` `(a[i]=7,a[j]=2,i>j holds)`
1, 3, 2, 4, 8, 5, 6, 7	Finish partitioning

The process of quicksort

5, 7, 2, 3, 8, 1, 6, 4	Start quicksorting
1, 3, 2, **4**, 8, 5, 6, 7	
1, 3, 2, 4, 8, 5, 6, 7	Start quicksorting (1a)
1, 3, 2, 4, 8, 5, 6, 7	Start partitioning
1, 3, 2, 4, 8, 5, 6, 7	Take the new pivot
1, 3, 2, 4, 8, 5, 6, 7	move `i` and `j` `(i>j` holds)
1, 2, 3, 4, 8, 5, 6, 7	Finish partitioning
1, **2**, **3**, 4, 8, 5, 6, 7	Finish quicksorting (1a)

The process of quicksort

5, 7, 2, 3, 8, 1, 6, 4	Start quicksorting
1, 2, 3, 4, 8, 5, 6, 7	Start quicksorting (1b)
1, 2, 3, 4, 8, 5, 6, 7	Start partitioning
1, 2, 3, 4, 8, 5, 6, 7	Take the pivot
1, 2, 3, 4, 8, 5, 6, 7	move `i` and `j` `(a[i]=8,a[j]=6)`
1, 2, 3, 4, 6, 5, 8, 7	exchange
1, 2, 3, 4, 6, 5, 7, 8	Finish partitioning
1, 2, 3, 4, 6, 5, 7, 8	Start quicksorting (2a)
1, 2, 3, 4, 5, 6, 7, 8	Finish quicksorting (2a)

The process of quicksort

5, 7, 2, 3, 8, 1, 6, 4	Start quicksorting
1, 2, 3, 4, 8, 5, 6, 7	Start quicksorting (1b)
1, 2, 3, 4, 5, 6, 7, 8	Start quicksorting (2b)
1, 2, 3, 4, 5, 6, 7, 8	Finish quicksorting (2b)
1, 2, 3, 4, 5, 6, **7**, 8	Finish quicksorting (1b)
1, 2, 3, **4**, 5, 6, 7, 8	Finish quicksorting

Stablility of quicksort

Q: Is quicksort stable?

A stable sorting = it preserves the relative order of items with duplicated keys

Example: 3,1,1,2
(already sorted w.r.t. color-order, where **R** < **Br)**

A: Not stable

Keys can move over other equal keys during the partitioning

Performance of quicksort

Execution of **`quicksort`** ≈ execution of **`partition`**

```
void quicksort(Item a[],int left,int
  int i;
  if (right <= left) return;
  i = partition(a,left,right);
  quicksort(a,left,i-1);
  quicksort(a,i+1,right);
 }
```

Main routine of **`quicksort`**

```
int partition(Item a[],int left,int right){
  Item pivot = a[right];
  int i = left-1, j = right-1;
  while(1){
      while(less(a[++i],pivot));
      while(less(pivot,a[j])){j--; if(i >= j) break; }
      if ( i >= j ) break;
      exch(a[i], a[j]);
    }
  exch(a[i], a[right]);
  return i;
}
```

The body of **`partition`**

The primitive operation of **`partition`** is **comparison**

The main operation of an algorithm, which is counted to estimate the performance of the algorithm

Performance of quicksort

Performance of **`partition`** is **O(N)** (linear order)

4, 3, 7, 2, 6, 5	`i`: check **4 < 5 ?**
4, 3, 7, 2, 6, 5	`i`: check **3 < 5 ?**
4, 3, 7, 2, 6, 5	`i`: check **7 < 5 ?**
4, 3, 7, 2, 6, 5	`j`: check **5 < 6 ?**
4, 3, 7, 2, 6, 5	`j`: check **5 < 2 ?**
4, 3, 2, 7, 6, 5	swap
4, 3, 2, 7, 6, 5	`i`: check **7 < 5 ?**
4, 3, 2, 7, 6, 5	`j`: check **5 < 2 ?**
4, 3, 2, 7, 6, 5	`j` is less than `i`
4, 3, 2, 5, 6, 7	swap

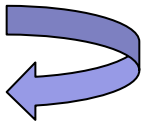
N = 6
`i` : the **yellow** number
`j` : the **green** number

- `i` and `j` approaches each other
- They perform a comparison if they move to their adjacent place
- The process is finished when `i` passes `j`

Performance of quicksort (worst-case)

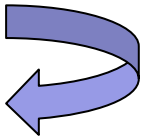
Worst case : $O(N^2)$ (Already sorted array)

1, 2, 3,..,N-2, N-1,N
1, 2, 3,..,N-2, N-1,N
1, 2, 3,..,N-2, N-1,N



partition
(about **N**-comparisons)

1, 2, 3,..,N-2,N-1,N
1, 2, 3,..,N-2,N-1,N
1, 2, 3,..,N-2,N-1,N

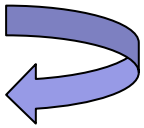


partition
(about **(N-1)**-comparisons)

1, 2, 3,..,N-2,N-1,N

...

1, 2, 3,..,N-2,N-1,N
1, 2, 3,..,N-2,N-1,N



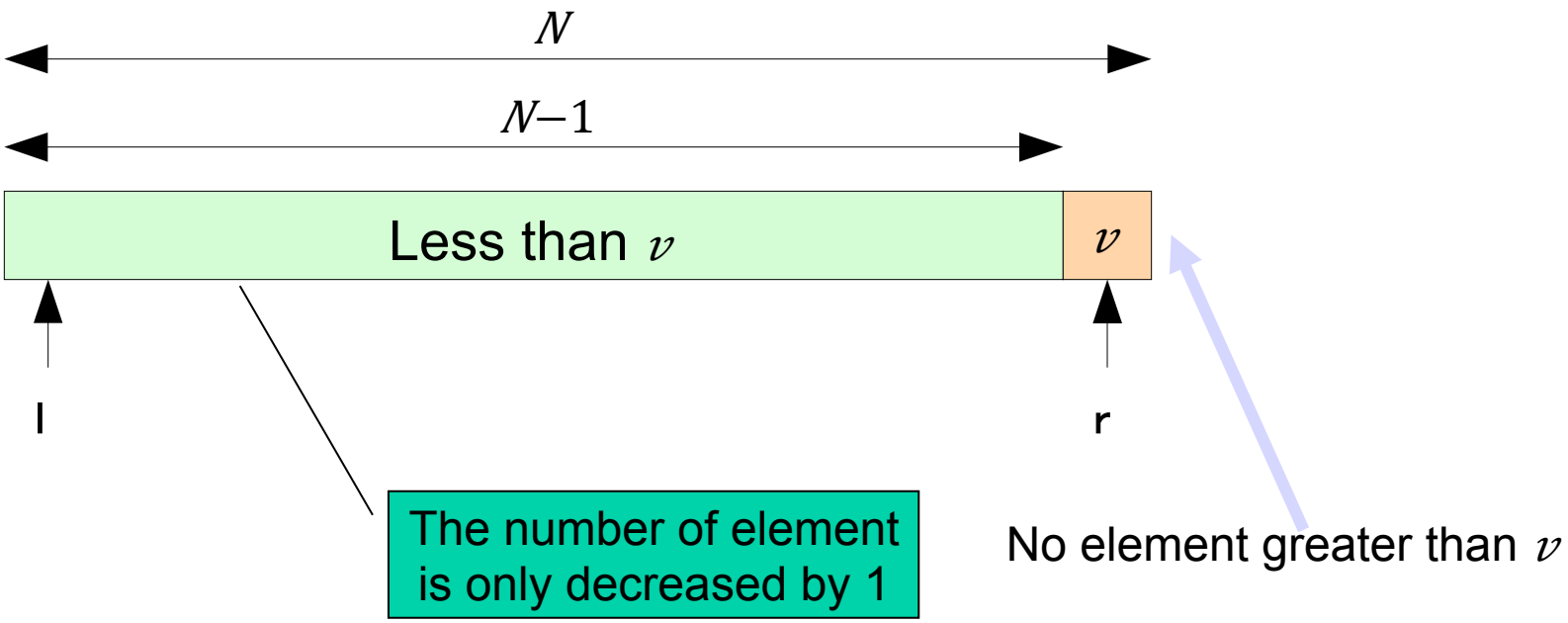
partition
(about **1**-comparisons)

Total: about N+(N-1)+...+2+1 = **N(N+1)/2** -comparisons

Performance of quicksort (worst-case)

When the pivot v is the greatest (least) element

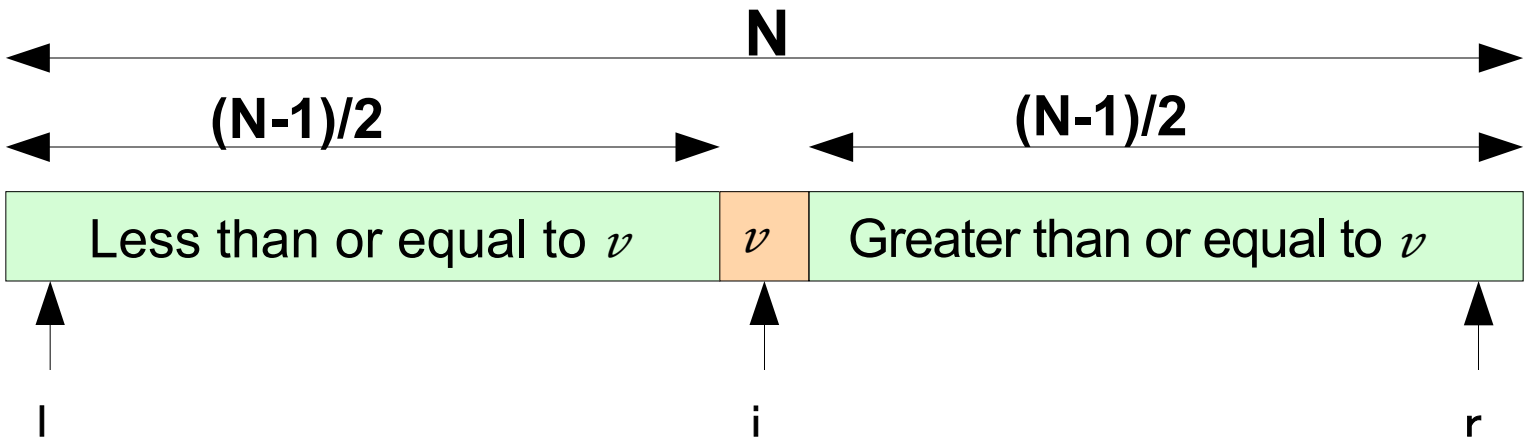
Every element other than v is partitioned into the left (right) partition



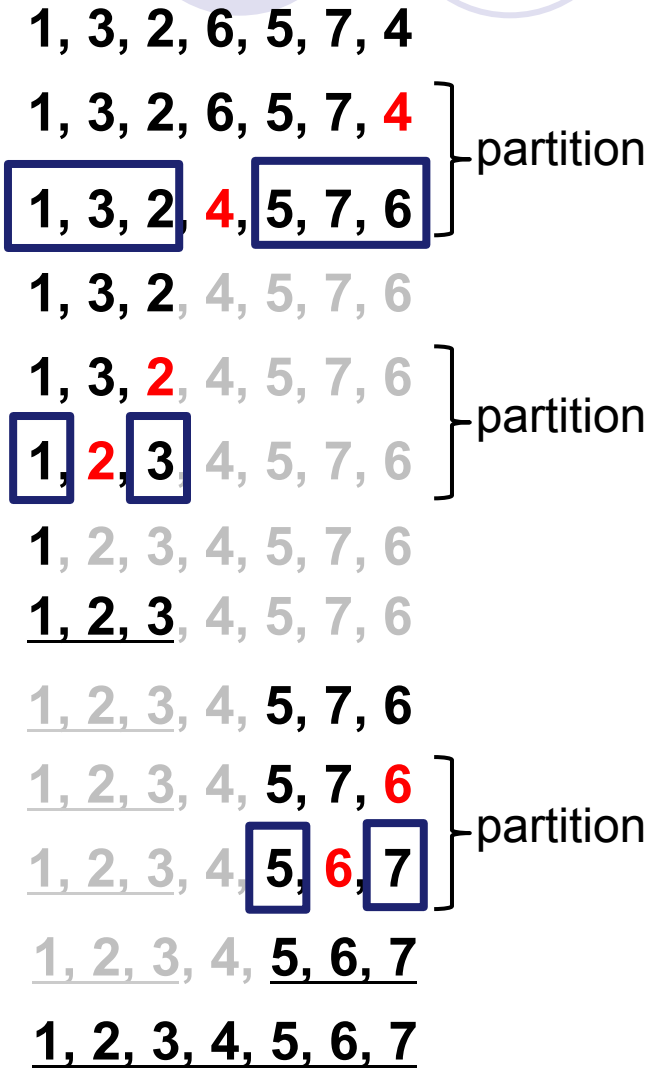
Performance of quicksort (best-case)

Best case : $O(N \log_2 N)$

When the pivot of each step halves the array by **`partition`**



Example of the best case

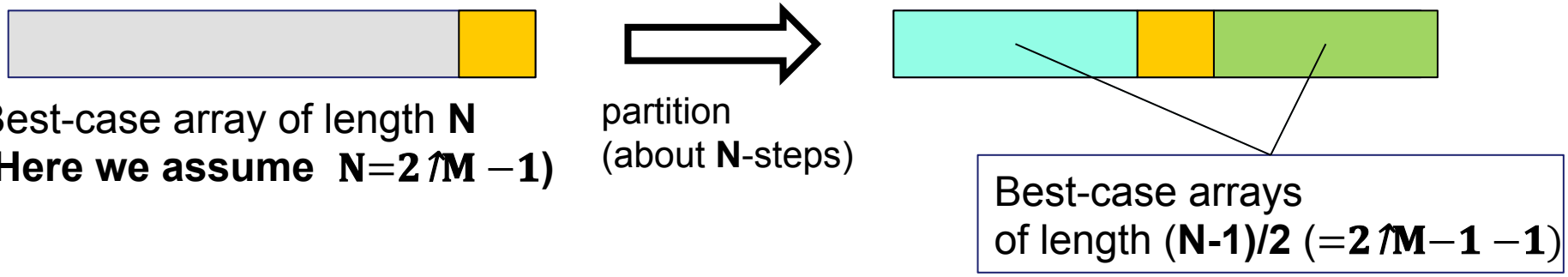


Each step of partition halves the array

In general a best case has length

2^M-1

Performance of quicksort (best-case)



B_N : the total number of comparisons of a **N**-length best-case

We have the following recurrence relations

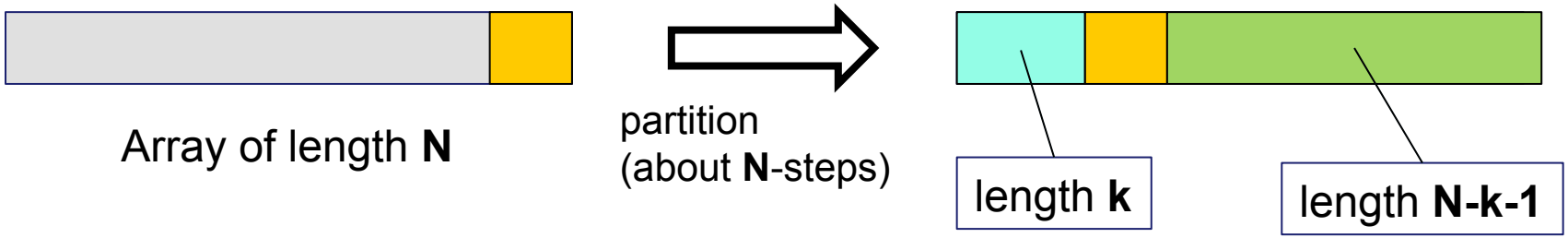
$$\begin{cases} B_1 = 0 \\ B_N = 2B_{(N-1)/2} + N \end{cases}$$

Thus we have

$$B_N = (N+1)(\log_2 (N+1) - 3/2) + 1$$
$$\approx N \log_2 N$$

Performance of quicksort (average)

Average : $O(\mathbf{N \log N})$



$A\downarrow N$: the average total number of comparisons of a **N**-length

$$A\downarrow 1 = 0$$
$$A\downarrow N = N+1/N \sum k=0 \uparrow N-1 \blacksquare (A\downarrow k + A\downarrow N-k-1)$$

see next slide

Thus we have

$$A\downarrow N \approx 2(N+1)\log(N+1) - N \cdot \mathrm{Const}$$
$$\approx \mathbf{N \log N}$$

The pivot is at index k with probability $1/N$

$$\begin{cases} A_1 = 0 \\ A_N = N + \frac{1}{N}\sum_{k=0}^{N-1}(A_k + A_{N-k-1}) \end{cases}$$

From the second equation,

$$A\downarrow N = N+1/N\,(\mathrm{A}\downarrow 0 + \mathrm{A}\downarrow N-1 + \mathrm{A}\downarrow 1 + \mathrm{A}\downarrow N-2 + \ldots + \mathrm{A}\downarrow N-1 + \mathrm{A}\downarrow 0\,)$$
$$= N+2/N\,(\mathrm{A}\downarrow 0 + \mathrm{A}\downarrow 1 + \ldots + \mathrm{A}\downarrow N-1\,)$$

Thus, we have

$$N\,A\downarrow N = N\uparrow 2 + 2(A\downarrow 0 + A\downarrow 1 + \ldots + A\downarrow N-1\,) \quad (1)$$

and

$$(N+1)\,A\downarrow N+1 = (N+1)\uparrow 2 + 2(A\downarrow 0 + A\downarrow 1 + \ldots + A\downarrow N) \quad (2)$$

replace *N* of (1) by *N+1*

By (2) – (1), we have

$$(N+1)A\downarrow N+1 - N\,A\downarrow N = 2N+1+2A\downarrow N$$

Then,

$$A\downarrow N+1 - 1/N+2 = A\downarrow N - 1/N+1 + 2/N+2$$

$$\begin{cases} A_1 = 0 \\ A_N = N + \frac{1}{N}\sum_{k=0}^{N-1}(A_k + A_{N-k-1}) \end{cases}$$

Hence we obtain

$$A_N - 1/N + 1 = -1 + 2\sum_{m=1}^{N} 1/m + 1 \approx -1 + 2\int_{1}^{N} 1/x + 1 \ \mathrm{d}x$$

$$= 2\log(N+1) - 2\log 2 - 1$$

Finally, we have the following:

$$A_N = 2(N+1)\log(N+1) + (N+1)\cdot(-2\log 2 - 1)$$
$$\approx N\log N$$

Performance of quicksort

Time!!

N	Selection (random)	Bubble (random)	Insertion (random)	Quick (random)	Quick (worst)
N=10000	**0.515 sec**	**0.437 sec**	**0.297 sec**	**0.000 sec**	**0.484 sec**
N=20000	**2.078 sec**	**1.640 sec**	**1.156 sec**	**0.000 sec**	**1.859 sec**
N=30000	**4.656 sec**	**3.578 sec**	**2.625 sec**	**0.015 sec**	**4.250 sec**
N=40000	**8.328 sec**	**6.343 sec**	**4.828 sec**	**0.031 sec**	**7.531 sec**
N=50000	**13.031 sec**	**9.875 sec**	**7.218 sec**	**0.031 sec**	**11.796 sec**
N=100000	**53.469 sec**	**39.828 sec**	**28.688 sec**	**0.078 sec**	**44.125 sec**
N=200000	**233.359 sec**	**157.65 sec**	**117.00 sec**	**0.250 sec**	**187.65 sec**

Performance of quicksort

Counter!!

$N = 2^n - 1$	Insertion (random)	Quick (worst)	Quick (random)	Quick (best)
N=3	4	8	5	6
N=7	24	34	27	18
N=15	67	134	87	49
N=63	996	2078	467	319
N=127	4165	8254	1350	766
N=511	67437	131326	6837	4092
N=1023	260976	524798	15947	9211
N=4095	4207595	8390654	75742	45049
	$O(N^2)$	$O(N^2)$	$O(N \log N)$	$O(N \log N)$

Generally, quicksort is faster than insertion sort

Performance of quicksort

Counter!!

$N = 2^n - 1$	Insertion (random)	Quick (worst)	Quick (random)	Quick (best)
N=3	4	8	5	6
N=7	24	34	27	18
N=15	67	134	87	49
N=63				219
N=127				466
N=511	67437	131326	6837	4092
N=1023	260976	524798	15947	9211
N=4095	4207595	8390654	75742	45049
	$O(N^2)$	$O(N^2)$	$O(N \log N)$	$O(N \log N)$

But, for small data, insertion is faster

Generally, quicksort is faster than insertion sort

Exercise 7.1

Make a new driver **`sort_driver2.c`** and a quicksort program **`quicksort.c`**, and compile them

```
$ gcc -o quicksort  sort_driver2.c  quicksort.c
$
```

Submit to CourseN@vi

`quicksort.c, sort_driver2.c`

sort_driver2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "sort.h"

void sort(Item*, int, int);
int c;

void main(int argc, char *argv[]){
  int i = 0;
  int N = atoi(argv[1]);
  int sw = atoi(argv[2]);
  int *a = malloc(N*sizeof(int));
  clock_t start, end;

  switch(sw){
     /* same as before */
  }

  c = 0;
  start = clock();
  sort(a, 0, N-1);
  end = clock();
  printf("\n counter = %d\n", c);
  printf("cpu time=%10.3f[sec]\n",(double)(end-start)/CLOCKS_PER_SEC);
  }
```

Counter `c` (global variable)
It is used to evaluate sorting algorithm

`clock()`
It gets the current time (cpu time)
(This function and the type `clock_t` are defined in `time.h`)
`start` is the time **BEFORE** `sort`
`end` is the time **AFTER** `sort`

Improvement of quicksort

In general, quicksort is fast

But it has some weak points

Small inputs

For small inputs (about 10 elements), **insertion sort is faster**

Already sorted arrays (worst case)

Problem: Choice of pivots

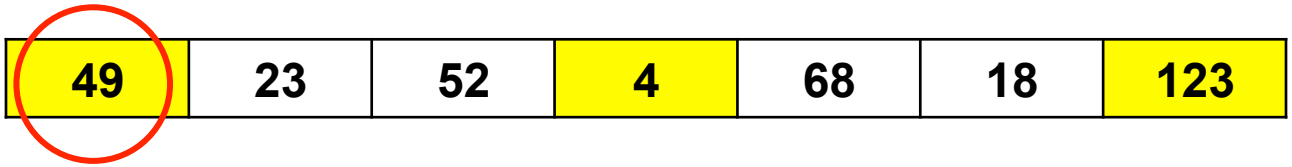
How to choose the pivot so that the file is partitioned into as nearly half as possible

Exercise 7.2

Improve the quicksort program by the following ways

(1) Taking a pivot (it was the rightmost element of the input)

- Take the leftmost, center, rightmost elements of the input
- The middle one of the three is the new pivot
- By this way, we can avoid the case that the pivot is the greatest element



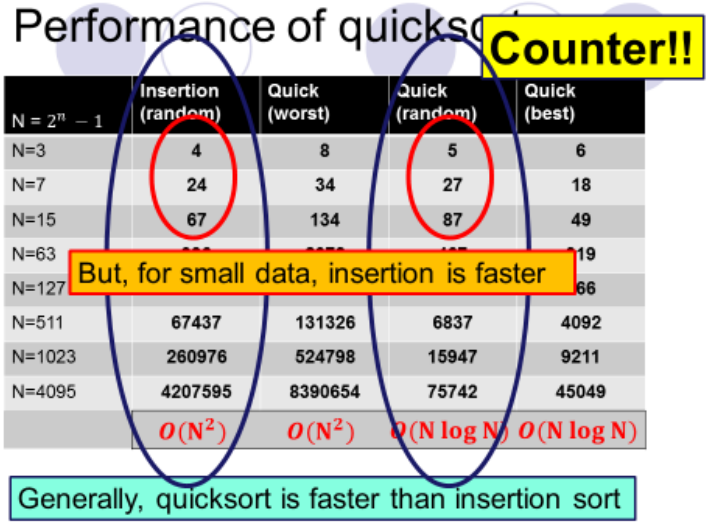
Compare 49, 4, 123
The middle one is 49 (new pivot)

Exercise 7.2

(2) Use insertion sort for small inputs

- Set an integer M
- Check the sizes of inputs before sorting
- If the size is less than M, sort by insertion sort, otherwise sort by quicksort
- Perhaps M = 9 is the best value

Make a modified file **`quicksort2.c`** by applying (1) and (2) to **`quicksort.c`** and check results after compiling



$N = 2^n - 1$	Insertion (random)	Quick (worst)	Quick (random)	Quick (best)
N=3	4	8	5	6
N=7	24	34	27	18
N=15	67	134	87	49
N=63				219
N=127				466
N=511	67437	131326	6837	4092
N=1023	260976	524798	15947	9211
N=4095	4207595	8390654	75742	45049
	$O(N^2)$	$O(N^2)$	$O(N \log N)$	$O(N \log N)$

Submit to CourseN@vi

`quicksort2.c`

Hint of (1)

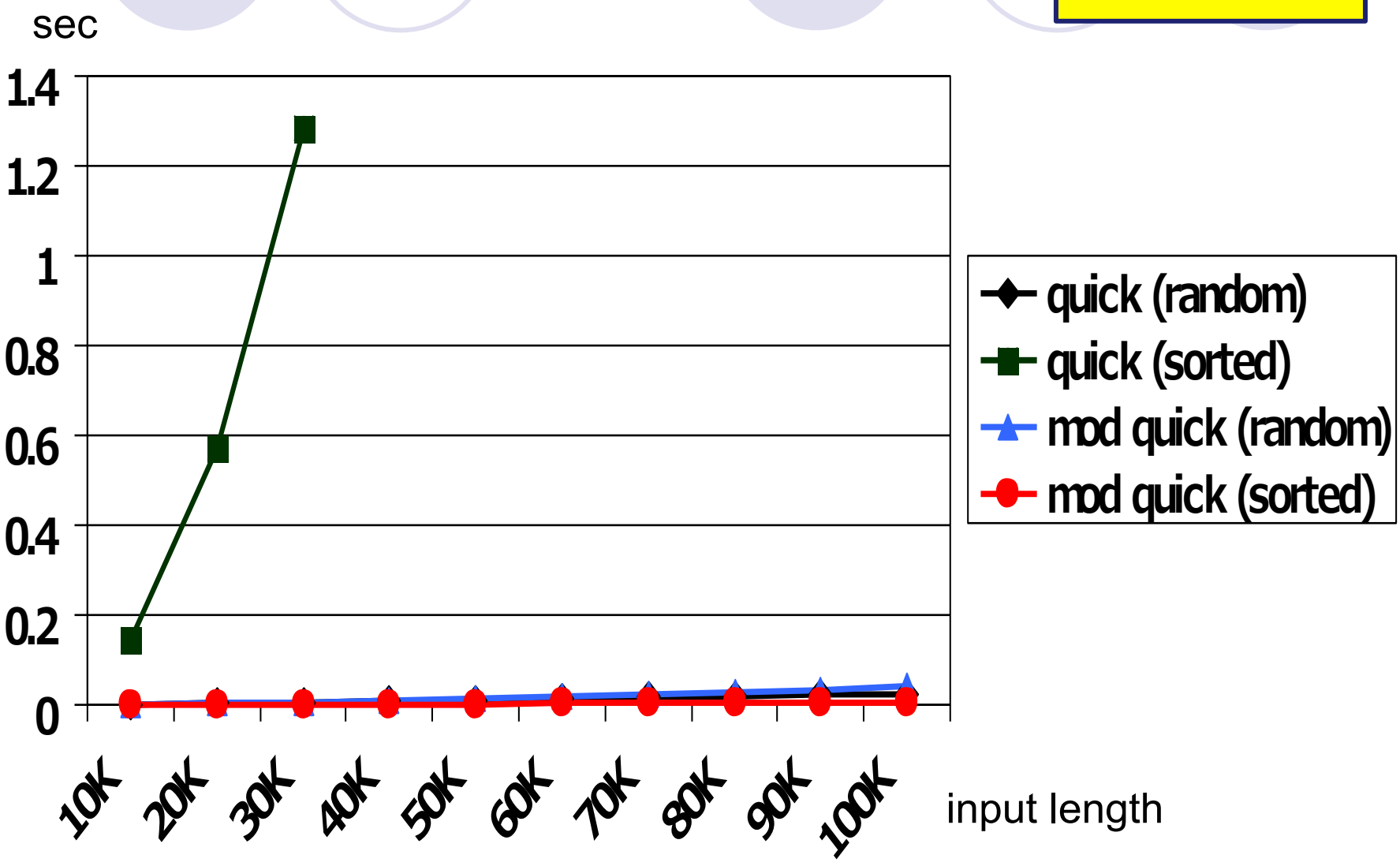
```
int partition(Item a[], int left, int right){
  int mid = (left+right)/2;
  int piv;
  /* Define piv to be the middle value of left, right, mid*/
  Item pivot = a[piv];
  exch(a[piv],a[right]);

  int i = left-1, j = right-1;
  while(1){
    while(less(a[++i], pivot));
    while(less(pivot, a[j])){j--; if(j <= i) break; }
    if ( i >= j ) break;
    exch(a[i], a[j]);
  }
  exch(a[i], a[right]);
  return i;
}
```

Hint of (2)

```
void quicksort(Item a[],int left,int right){
 int i;
 if (right <= left) return;
 i = partition(a,left,right);
 if (right-left <= M) {
    insertion(a,left,right); return;
 }
 quicksort(a,left,i-1);
 quicksort(a,i+1,right);
}
```

Performance of quicksort

Time!!



Outline

Quicksort

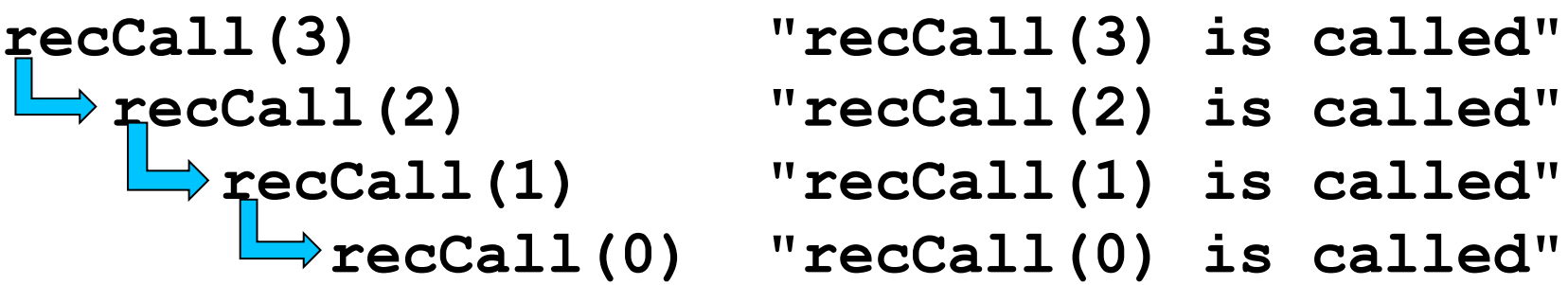
- Basic algorithm
- Performance
- Improvement

Extra topic

- **Hidden cost of Recursive programs**
  - Call stack
  - While-program with stack
  - Space complexity of Quicksort

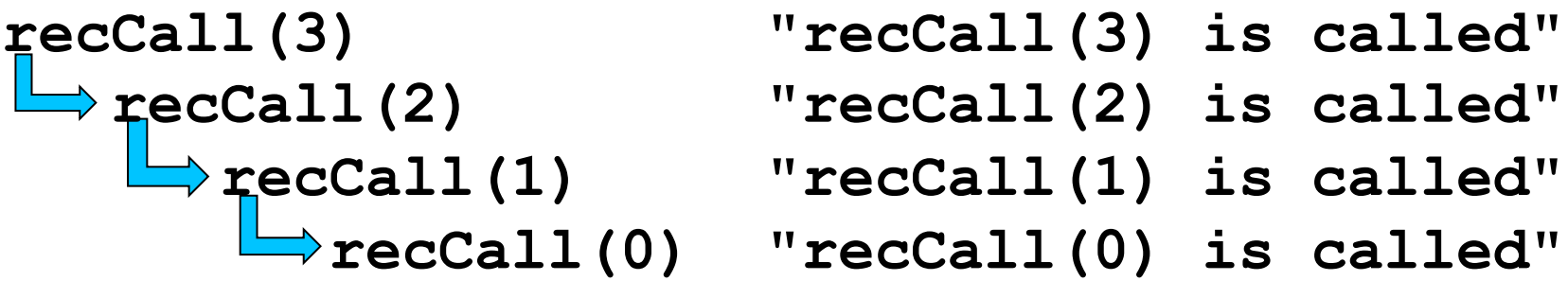
Hidden cost of rec.programs

```
void recCall(int n){
 printf("recCall(%d) is called\n",n);

 if (n == 0) return;
 recCall(n-1);
}
```



Hidden cost of rec.programs

```
void recCall(int n){
 printf("recCall(%d) is called\n",n);

 if (n == 0) return;
 recCall(n-1);
}
```



recCall looks that it **doesn't cousume much memory space**. But...

Hidden cost of rec.programs

```
void recCall(int n){
 printf("recCall(%d) is called\n",n);

 if (n == 0) return;
 recCall(n-1);
}

void main(){
  recCall(50000);
}
```

This limit number depends on PC environments
Take more bigger number if this code successfully finished in your environment

Memory error!

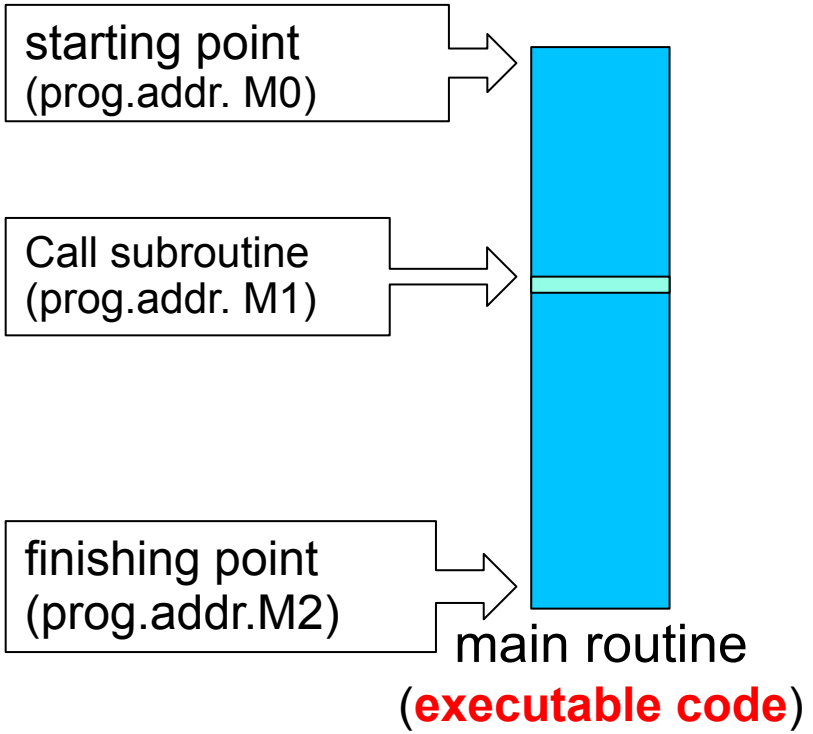
```
$ ./recCall
recCall(50000) is called
recCall(49999) is called
...
segmentation fault
```

Hidden cost of rec.programs

Call-stack

A stack (called **call-stack**) is used during an execution of a code

It contains some ***run-time information***



call-stack

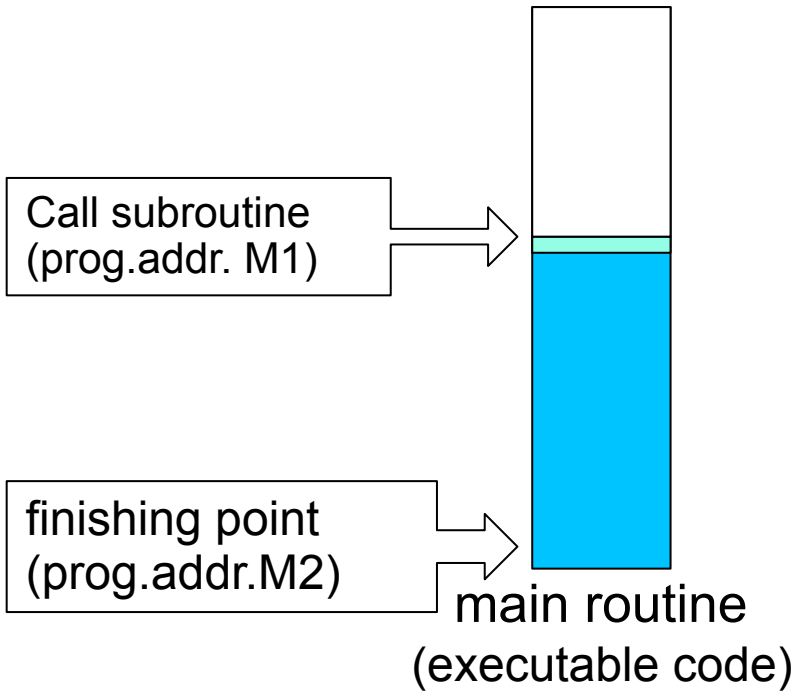
Current prog.addr. = **M0**

Hidden cost of rec.programs

Call-stack

A stack (called **call-stack**) is used during an execution of a code

It contains some ***run-time information***



call-stack

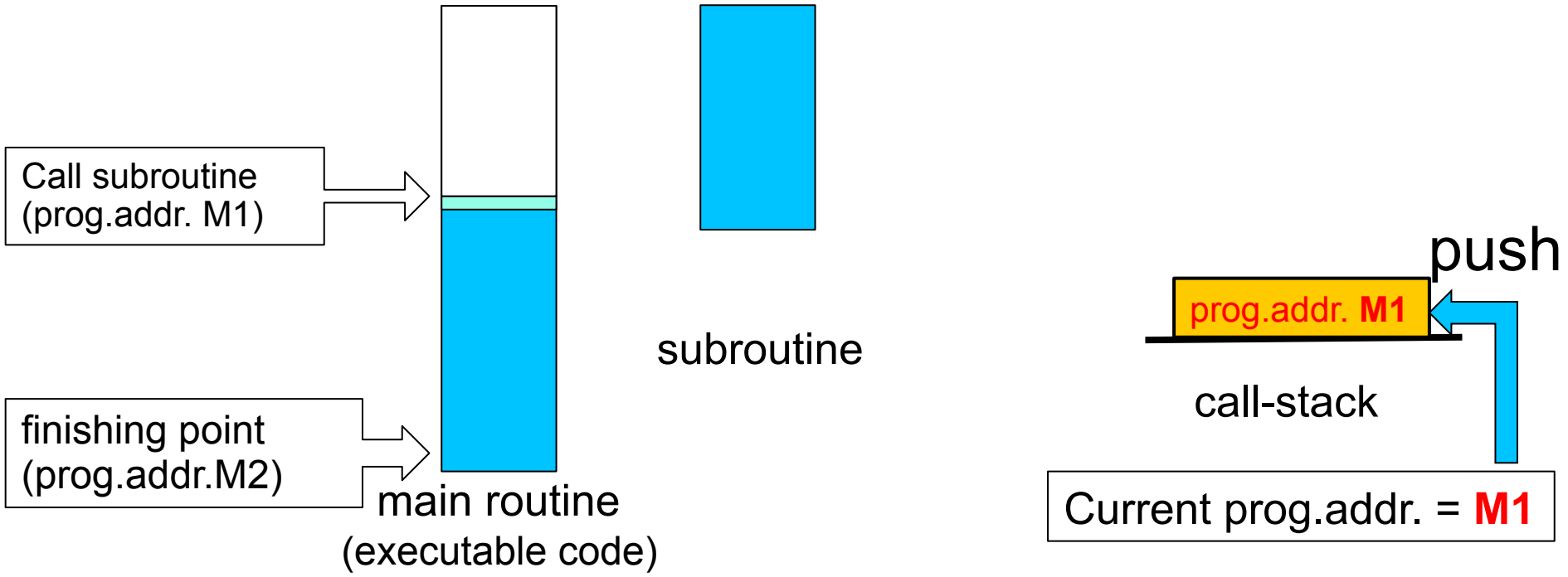
Current prog.addr. = **M1**

Hidden cost of rec.programs

Call-stack

A stack (called **call-stack**) is used during an execution of a code

It contains some ***run-time information***

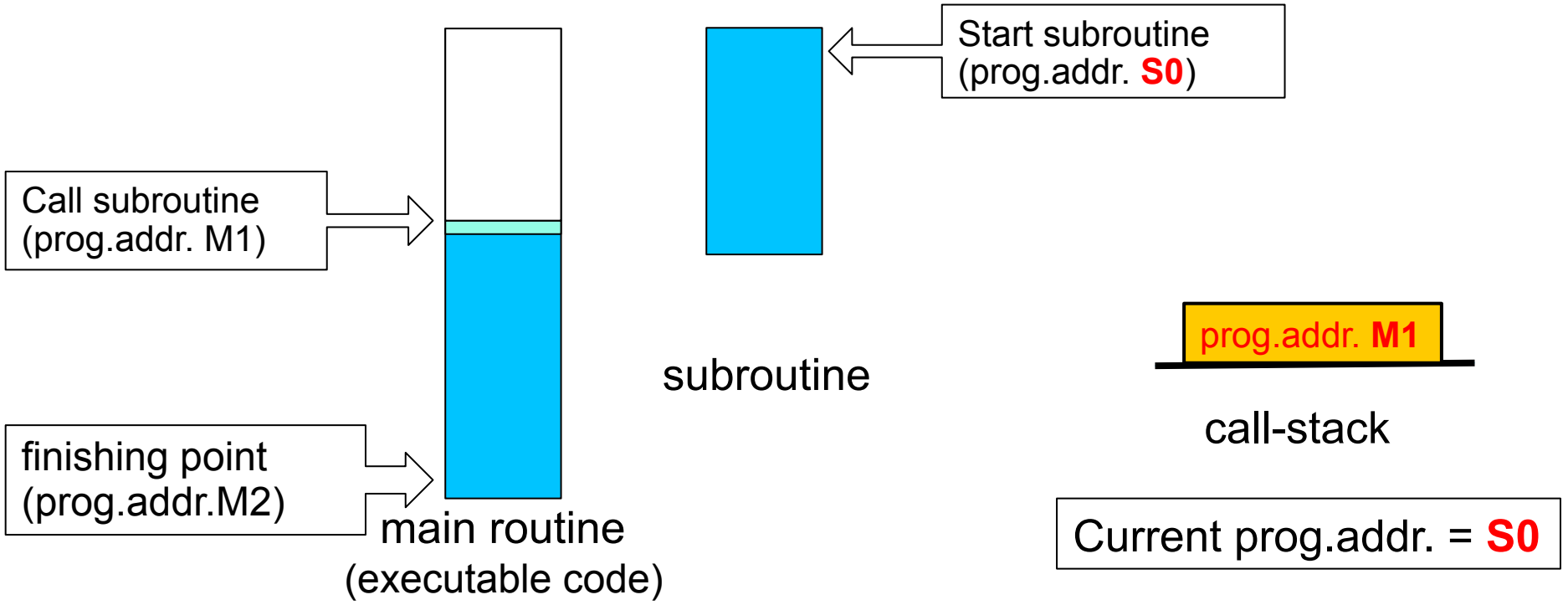


Hidden cost of rec.programs

Call-stack

A stack (called **call-stack**) is used during an execution of a code

It contains some ***run-time information***

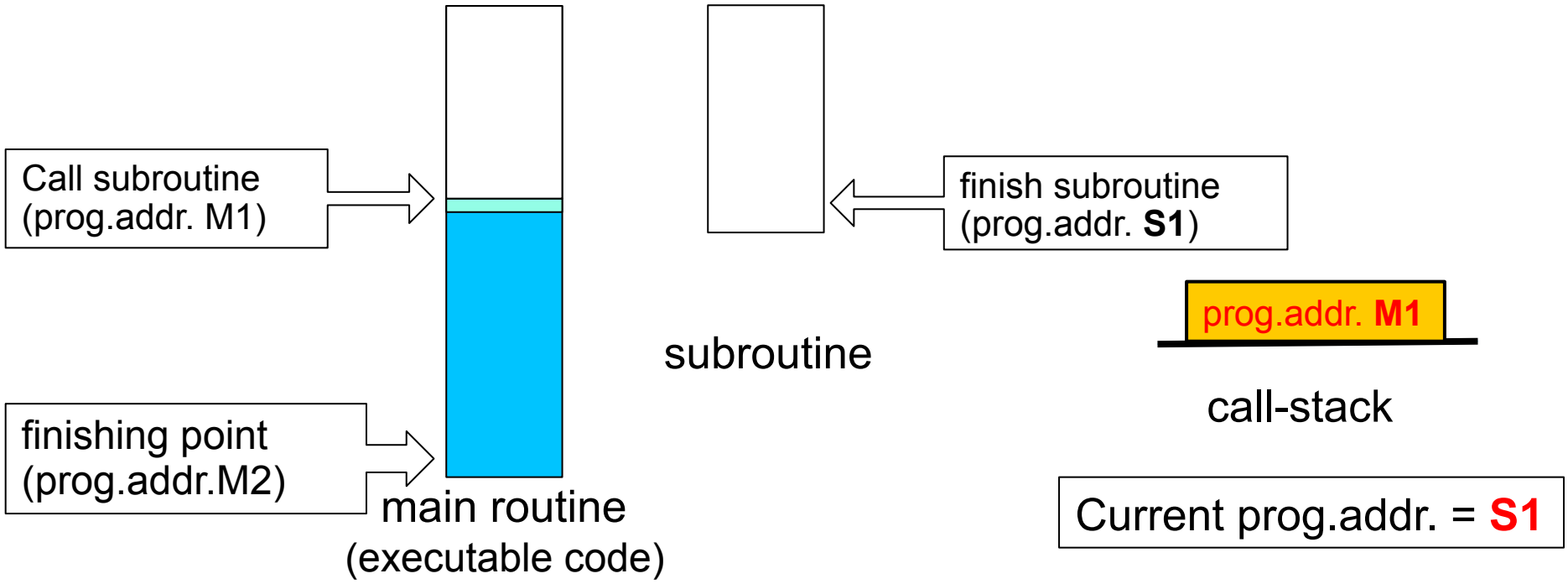


Hidden cost of rec.programs

Call-stack

A stack (called **call-stack**) is used during an execution of a code

It contains some ***run-time information***

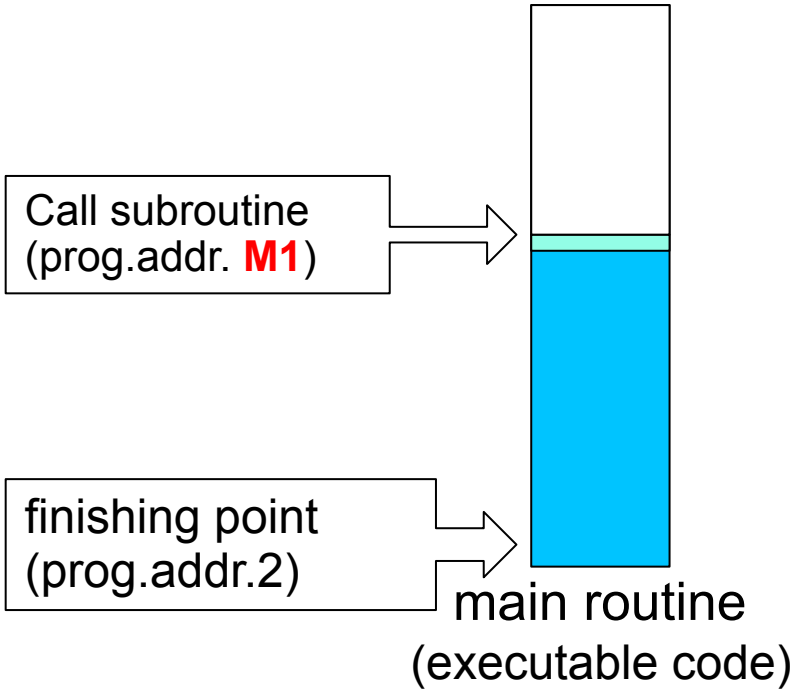


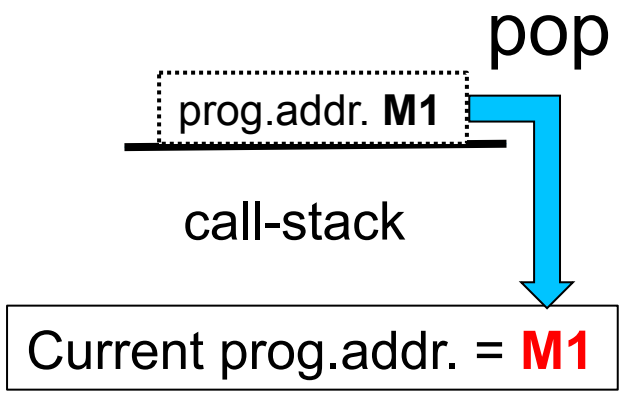
Hidden cost of rec.programs

Call-stack

A stack (called **call-stack**) is used during an execution of a code

It contains some ***run-time information***



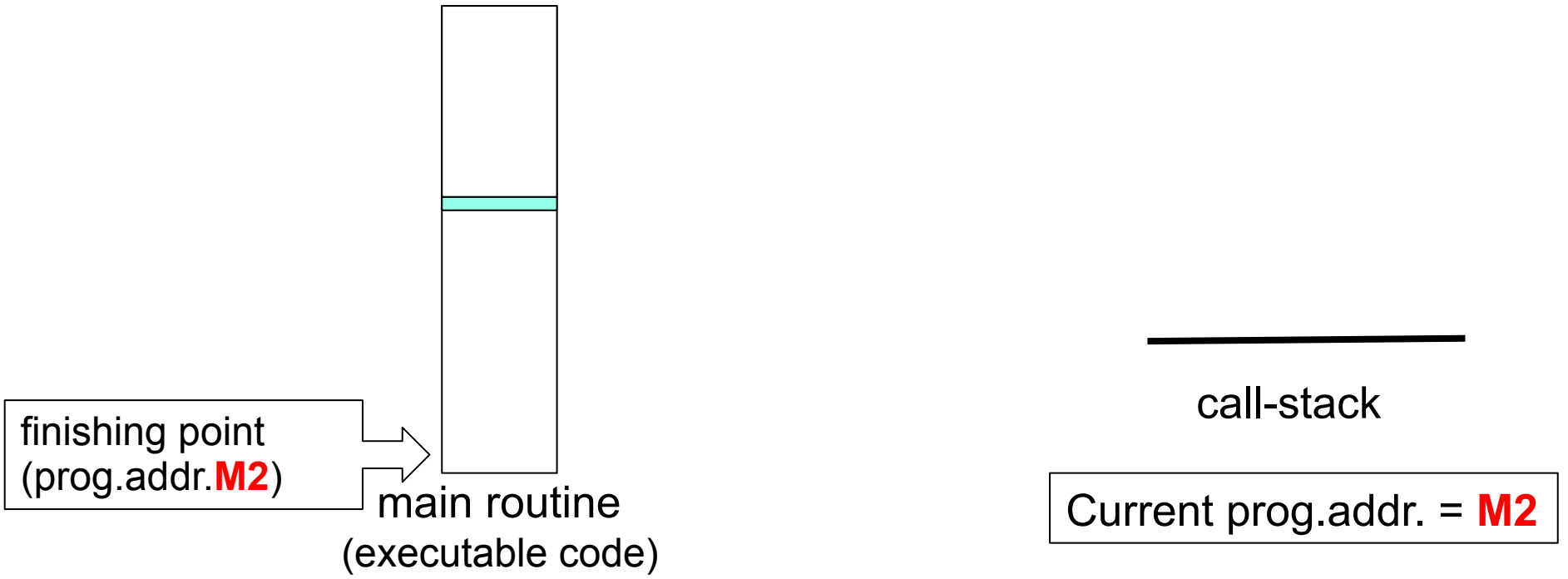


Hidden cost of rec.programs

Call-stack

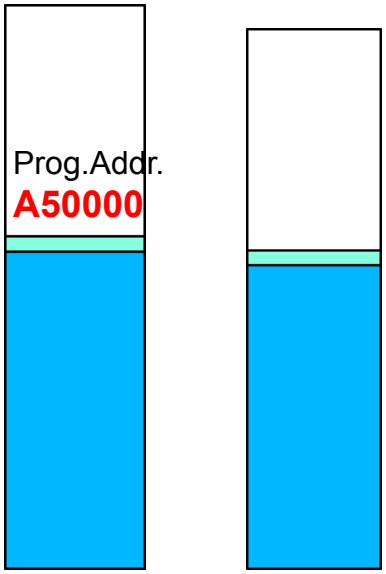
A stack (called **call-stack**) is used during an execution of a code

It contains some ***run-time information***



Hidden cost of rec.programs

What happened in **`recCall`**?



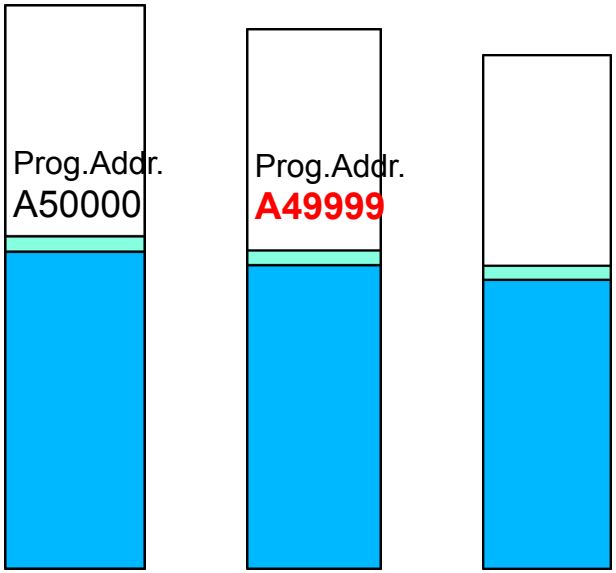
recCall (50000)

recCall (49999)

call-stack

Hidden cost of rec.programs

What happened in **`recCall`**?



recCall (50000)

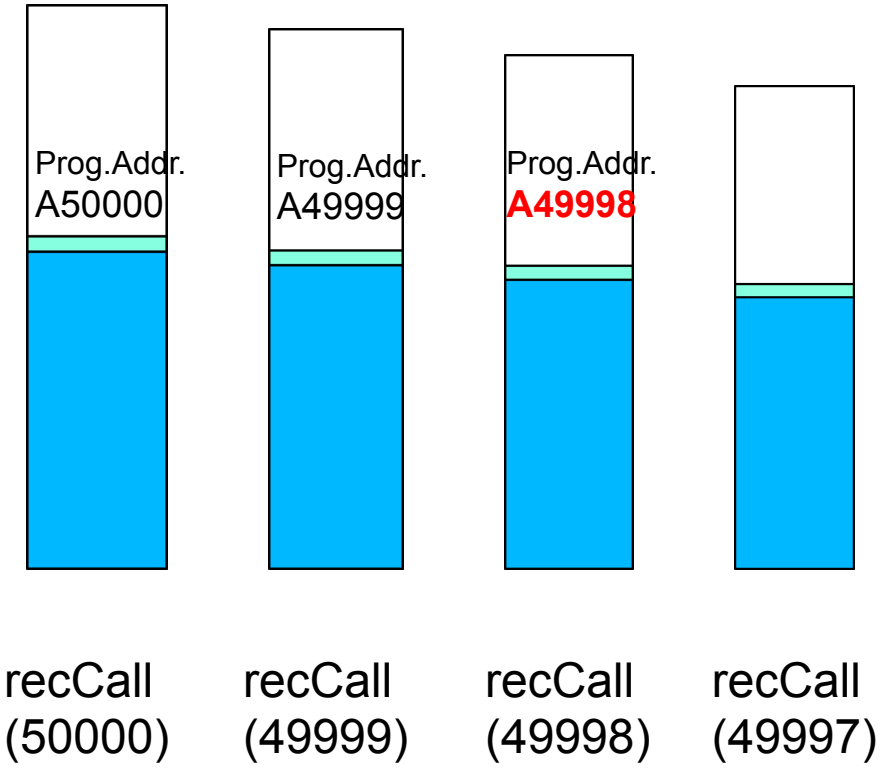
recCall (49999)

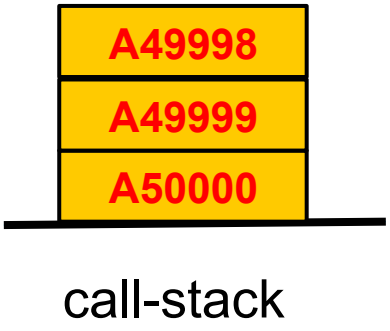
recCall (49998)

call-stack

Hidden cost of rec.programs

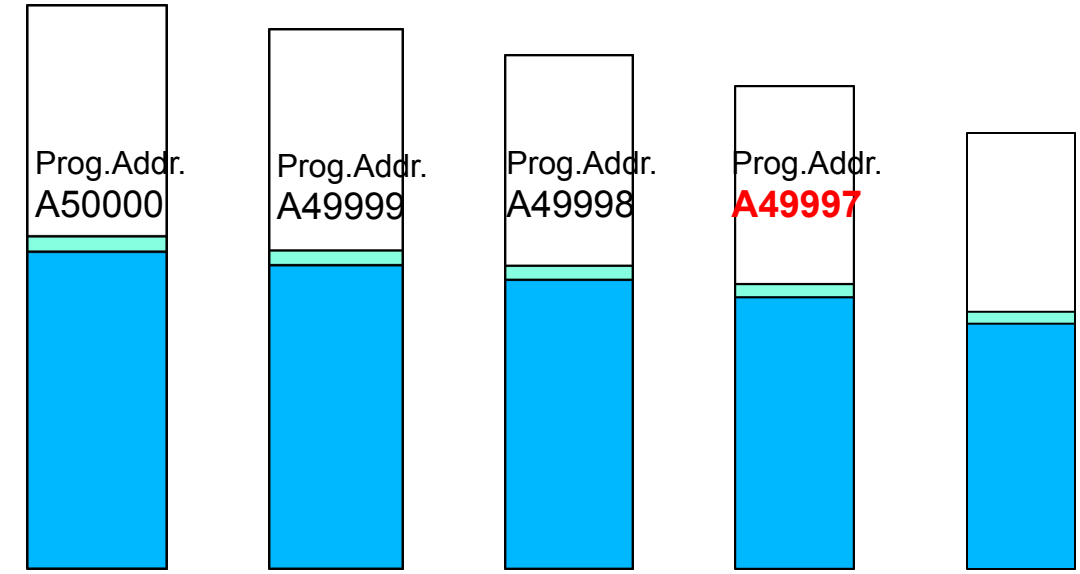
What happened in **`recCall`**?

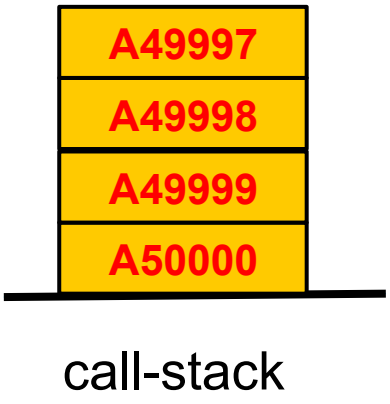




Hidden cost of rec.programs

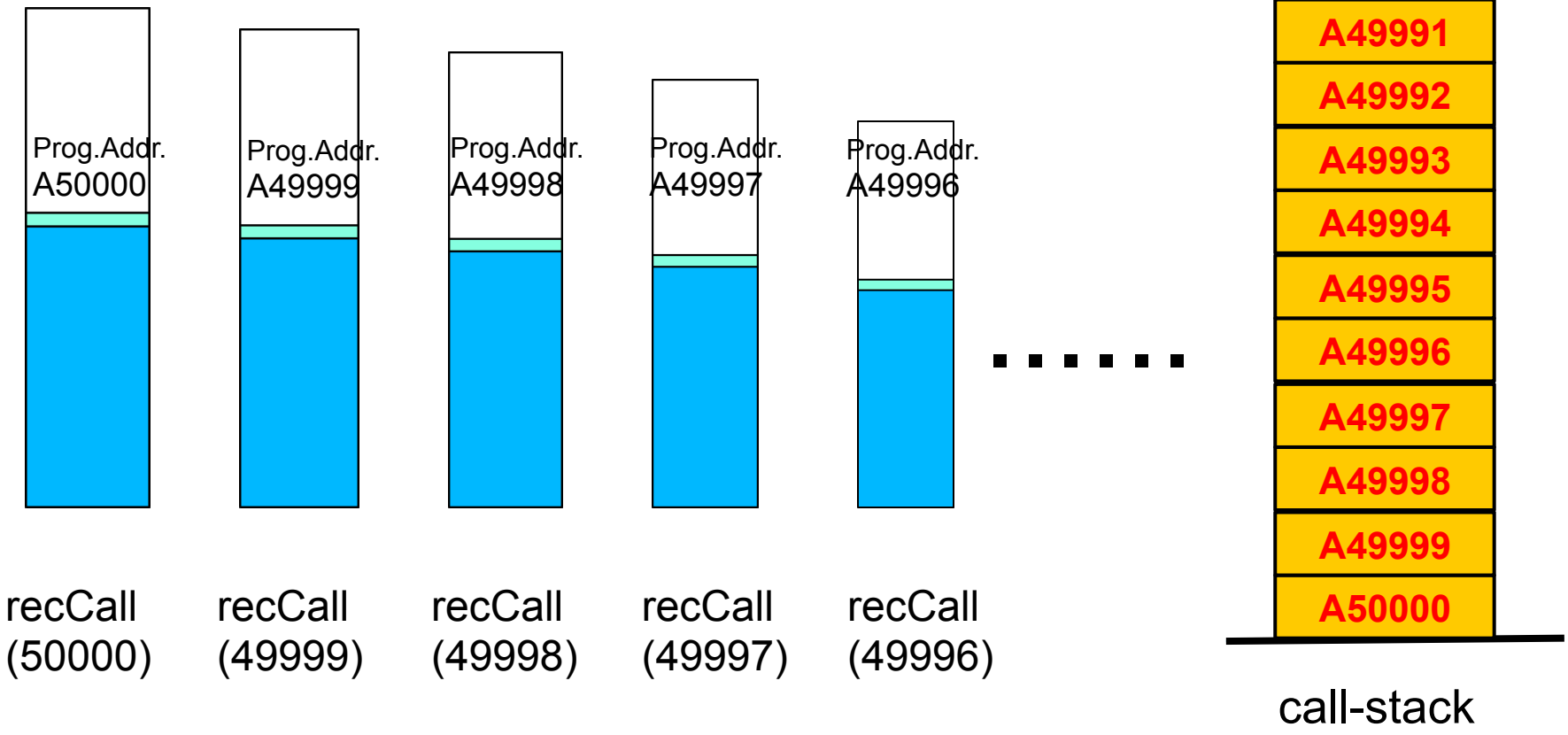
What happened in `recCall`?





Hidden cost of rec.programs

What happened in `recCall`?



Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

A recursive program can be rewritten as a **while-program with a stack**

The stack contains **parameters (of rec.prog.)** that should be performed **in future**
It's NOT a simulation of the call-stack

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

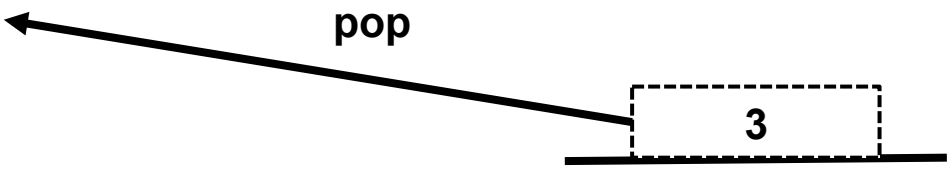
```
recCall_while(3)
```

`recCall(3)`
will be performed

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

```
recCall_while(3)
"recCall(3) is called"
```



`recCall(3)`
is performed

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

`recCall_while(3)`
`"recCall(3) is called"`

push

`recCall(2)`
will be performed

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

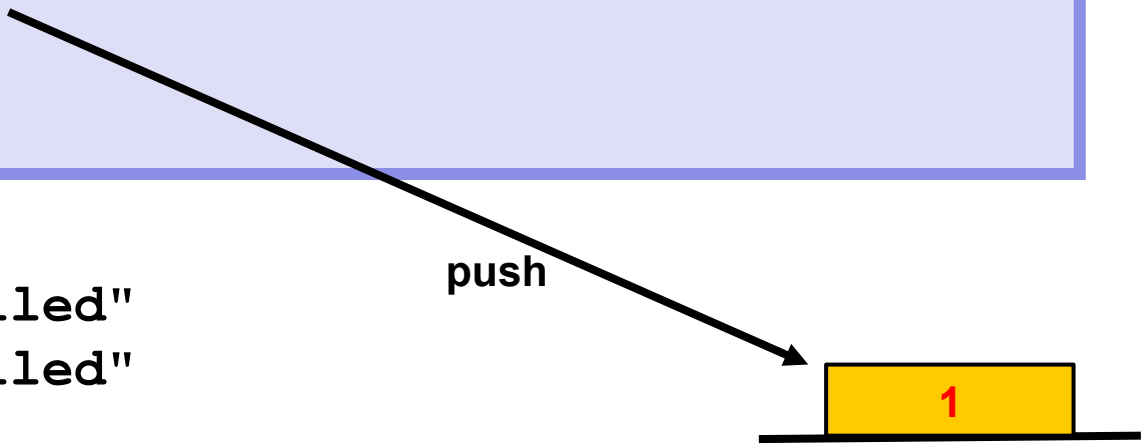
```
recCall_while(3)
"recCall(3) is called"
"recCall(2) is called"
```

`recCall(2)` is performed

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

```
recCall_while(3)
"recCall(3) is called"
"recCall(2) is called"
```



`recCall(1)`
will be performed

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

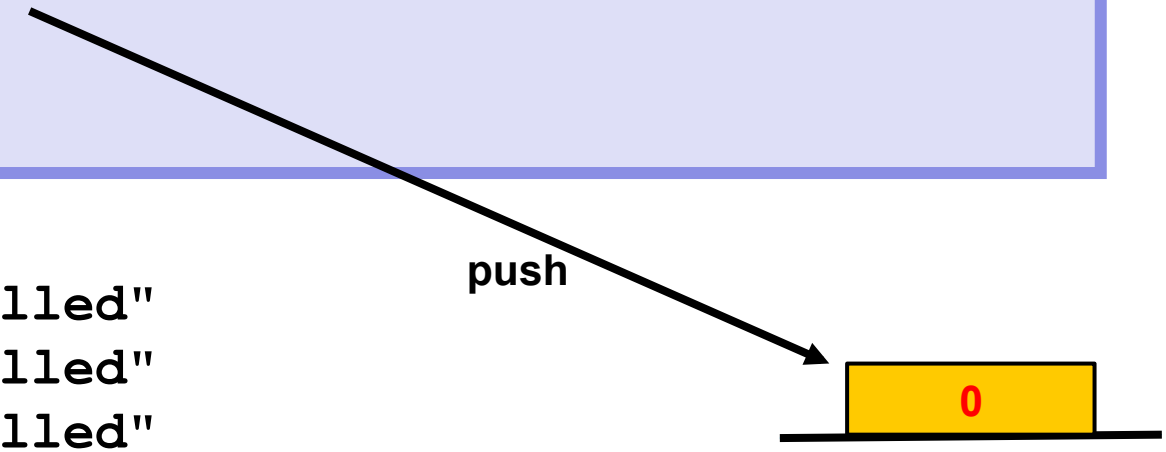
```
recCall_while(3)
"recCall(3) is called"
"recCall(2) is called"
"recCall(1) is called"
```

recCall(1)
is performed

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

```
recCall_while(3)
"recCall(3) is called"
"recCall(2) is called"
"recCall(1) is called"
```

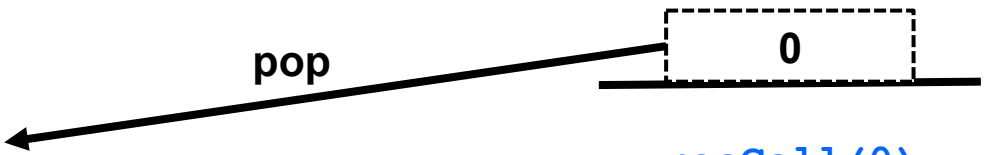


`recCall(0)`
will be performed

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

```
recCall_while(3)
"recCall(3) is called"
"recCall(2) is called"
"recCall(1) is called"
"recCall(0) is called"
```



`recCall(0)` is performed

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

```
recCall_while(3)
"recCall(3) is called"
"recCall(2) is called"
"recCall(1) is called"
"recCall(0) is called"
```

Nothing to do next

Simulating recursion by While+Stack

```
void recCall_while(int n){
  int i;
  STACKinit(1);
  STACKpush(n);

  while(!STACKempty()){
    i = STACKpop();
    printf("recCall(%d) is called\n",i);
    if(i == 0) continue;
    STACKpush(i-1);
  }
}
```

Size of the stack was only 1 !!!

```
recCall_while(3)
"recCall(3) is called"
"recCall(2) is called"
"recCall(1) is called"
"recCall(0) is called"
```

Nothing to do next

Outline

- Quicksort
  - Basic algorithm
  - Performance
  - Improvement
- Hidden costs of Recursive programs
  - Call stack
  - Simulating rec.prog. by while+stack
  - **Space complexity of Quicksort**

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

Non-recursive version

```
void quicksort(Item a[],int left,int right){
  int i;
  if (right <= left) return;
  i = partition(a,left,right);
  quicksort(a,left,i-1);
  quicksort(a,i+1,right);
 }
```

Recursive version

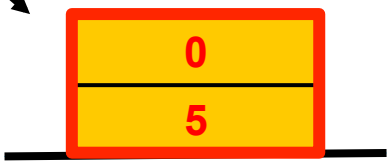
Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

```
quicksort_while(a,0,5)
```

push

x=?
y=?
i=?



quicksort(0,5)
will be performed

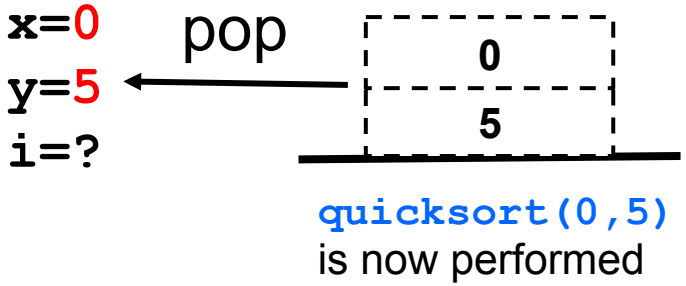
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
13	16	15	12	11	14

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

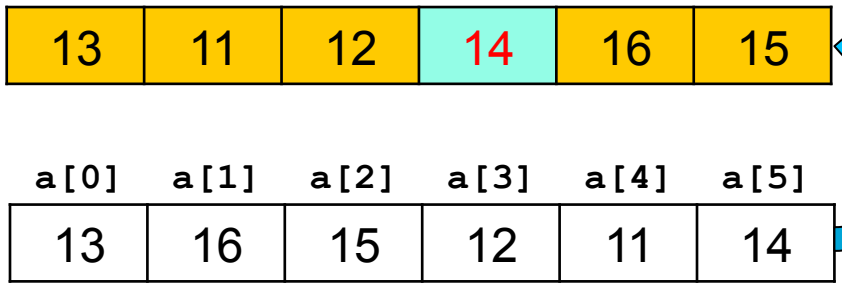
```
quicksort_while(a,0,5)
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
13	16	15	12	11	14



Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```



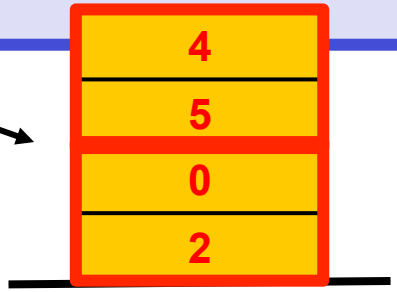
x=0
y=5
i=3

partition(0,5)

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

push



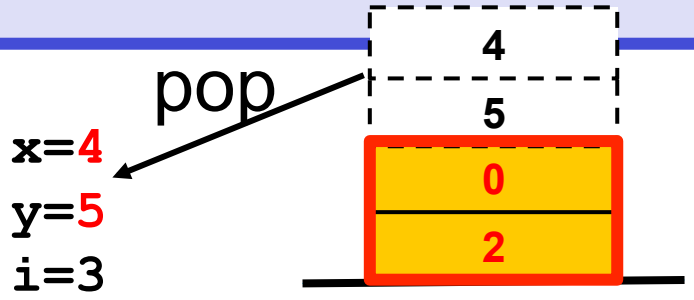
x=0
y=5
i=3

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
13	11	12	14	16	15

`quicksort(0,2)` and `quicksort(4,5)` will be done

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

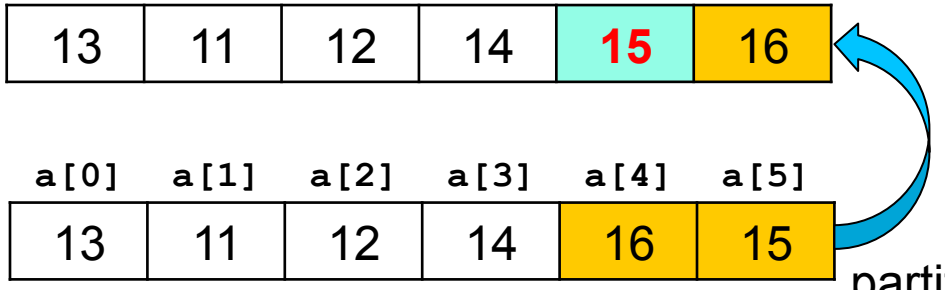


a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
13	11	12	14	16	15

`quicksort(0,2)` will be done

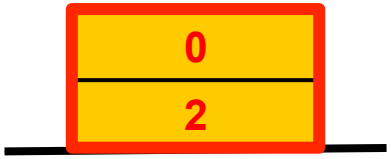
Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```



x=4
y=5
i=4

partition(4,5)

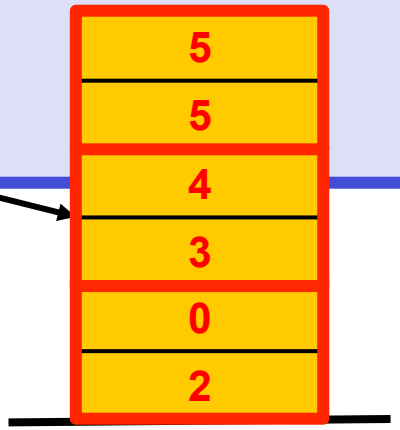


`quicksort(0,2)` will be done

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

push



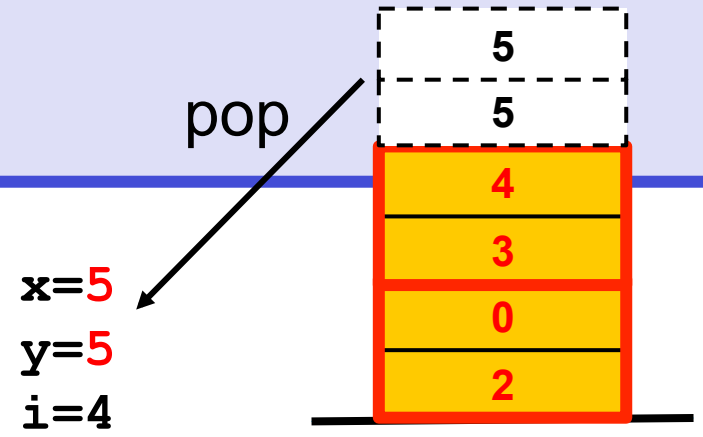
x=3
y=5
i=4

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
13	11	12	14	15	16

quicksort(0,2) will be done

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

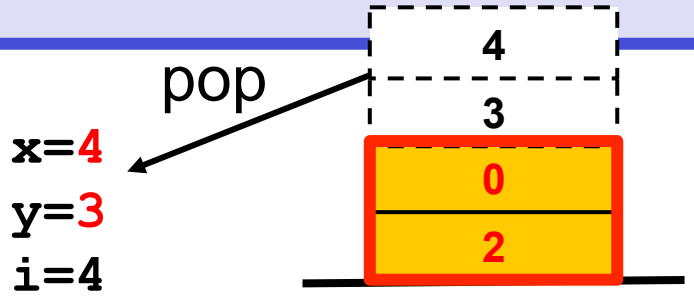


a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
13	11	12	14	15	16

`quicksort(0,2)` will be done

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```



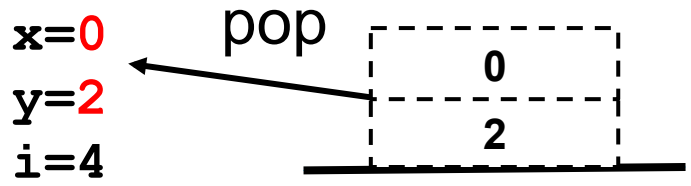
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
13	11	12	14	15	16

`quicksort(0,2)` will be done

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

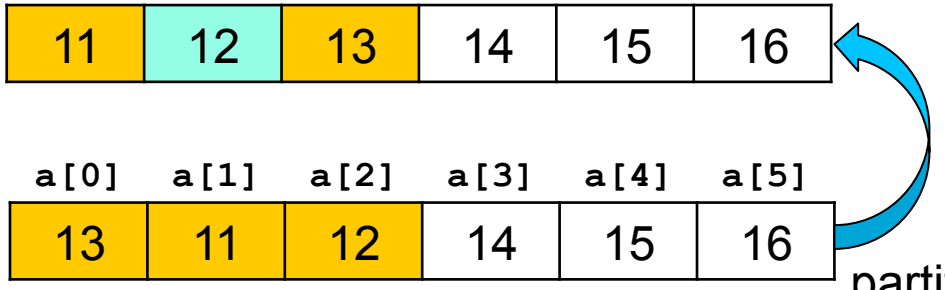
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
13	11	12	14	15	16



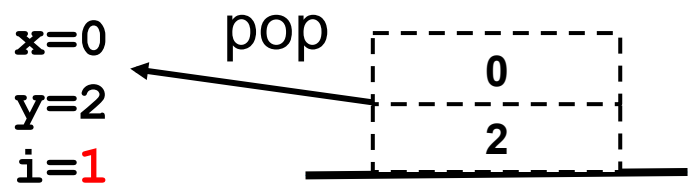
`quicksort(0,2)` is performed

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```



partition(0,2)

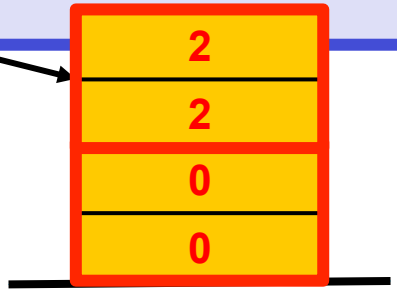


`quicksort(0,2)` is performed

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

push



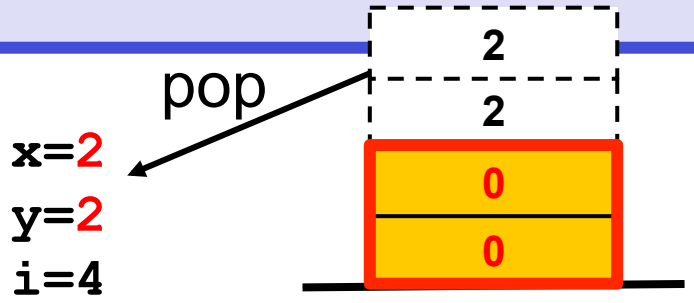
x=0
y=2
i=1

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
11	12	13	14	15	16

`quicksort(0,0)` and `quicksort(2,2)` will be done

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```



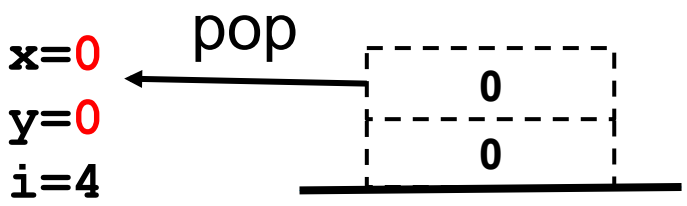
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
11	12	13	14	15	16

`quicksort(0,0)` will be done

Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
11	12	13	14	15	16



Non-recursive version of Quicksort

```
#define STACKpush2(A,B) { STACKpush(B); STACKpush(A); }

void quicksort_while(Item a[],int left,int right){
  int i,x,y;
  STACKinit(1000);
  STACKpush2(left,right);

  while(!STACKempty()){
  x = STACKpop();
  y = STACKpop();
  if (y <= x) continue;
  i = partition(a,x,y);
  STACKpush2(x,i-1);
  STACKpush2(i+1,y);
 }
}
```

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
11	12	13	14	15	16

Nothing to do next

Space complexity of Quicksort

Quicksort is **NOT** in-place sorting

- It requires some extra memory space
  - **O(log N)** (average)
  - **O(N)** (worst)
- Recursive version uses **call-stack**
- Non-recursive version uses a **stack**