Lecture – Recursion

Linear recursion

- Example – Factorial

  Recursive definition = boundary condition + recurrence relation

$$
\begin{aligned}
n! &= 1 & n &= 0 \\
&= n \times (n-1)! & n &> 0
\end{aligned}
$$

```
unsigned f(unsigned n)
{
   return n==0?1:n*f(n-1);
}
```

Linear recursive process

```
f(3)                     f(3)
→ 3*f(2)                 → n*f(2)       where n=3
→ 3*2*f(1)               → n*n*f(1)     where n=2
→ 3*2*1*f(0)             → n*n*n*f(0)   where n=1
→ 3*2*1*1                → n*n*n*1      ∵ n=0
→ 3*2*1                  → n*n*1
→ 3*2                    → n*2
→ 6                      → 6
```

On storage management

main ⇆ A ⇆ B

main   A   B   ······ Activation records

1 An activation record contains the storage for local variables of the function beig activated.

2 When a function is called, its AR is allocated; when it returns, its AR is deallocated.

- Example (Cont'd)

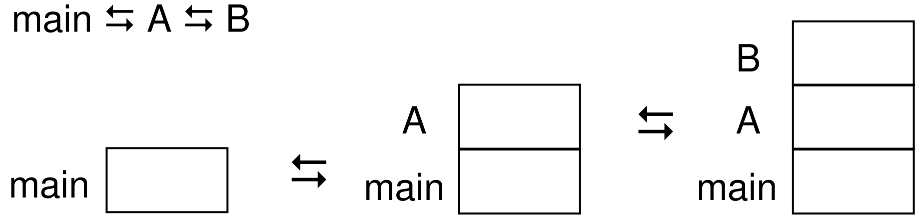
    3 The ARs are allocated and deallocated in the order of
    FILO (First In Last Out), or
    LIFO (Last In First Out)

    On stack

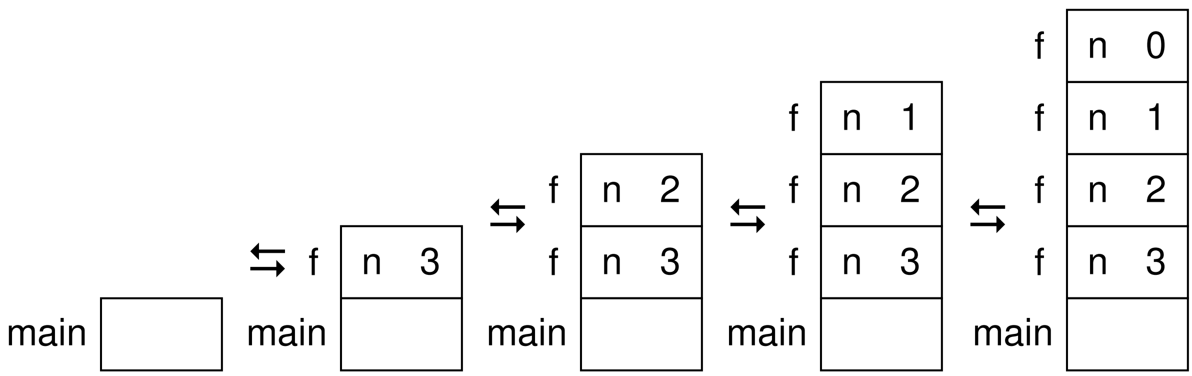
    1 A data structure that works on the priciple of FILO
    2 A data type with two main operatons:
    push (allocation)
    pop (deallocation)

    On runtime stack

    main ⇆ A ⇆ B



In the presence of recursion, the run time stack grows rapidly.



- Example – Exponentiation

$$
\begin{aligned}
a^n &= 1 && n = 0 \\
&= a \times a^{n-1} && n > 0 \text{ is odd} \\
&= (a^2)^{n/2} && n > 0 \text{ is even}
\end{aligned}
$$

```
int pow(int a,unsigned n)
{
   if (n==0) return 1;
   else if ((n&1)==1) return a*pow(a,n-1);
   else return pow(a*a,n/2);  // *
}
```

Linear recursive process

$$
\begin{array}{ll}
\texttt{pow(a,5)} & \texttt{a}^5 \leftarrow \\
\rightarrow \texttt{a*pow(a,4)} & \texttt{a*a}^4 \leftarrow \\
\rightarrow \texttt{a*pow(a}^2\texttt{,2)} & \texttt{a*a}^4 \leftarrow \\
\rightarrow \texttt{a*pow(a}^4\texttt{,1)} & \texttt{a*a}^4 \\
\rightarrow \texttt{a*a}^4\texttt{*pow(a}^4\texttt{,0)} \rightarrow \texttt{a*a}^4\texttt{*1} & \Lsh
\end{array}
$$

The starred line can't be written as **pow(pow(a,2),n/2)**, since

```
pow(a,2)
→ pow(pow(a,2),1)
→ pow(pow(pow(a,2),1),1)
→ ...
```

Eventually, the runtime stack will overflow.

Since $(a^2)^{n/2} = \left(a^{n/2}\right)^2$, we may also write

```
int pow(int a,unsigned n)
{
   if (n==0) return 1;
   else if ((n&1)==1) return a*pow(a,n-1);
   else { int x=pow(a,n/2); return x*x; }
}
```

- Example – Digit sum

$s(d_0) = d_0$
$s(d_k \cdots d_1 d_0) = s(d_k \cdots d_1) + d_0, \quad k \geq 1$

Version A

```
int sum(int n)
{
   return n<10? n: sum(n/10)+n%10;
}
```

Linear recursive process

```
sum(123)
→ sum(12)+3
→ sum(1)+2+3                 // output 1    (in sum)
→ 1+2+3                      // output +2   (in sum)
→ 3+3                        // output +3   (in sum)
→ 6                          // output =6   (in main)
```

Version B – With equation // distinct action on boundary

```
int sum(int n)               // 1 +2 +3   ✗ 1+ 2+ 3
{
   if (n<10) {               // same action on recursion
      printf("%d",n);
      return n;
   } else {
      int d=n%10;
      int s=sum(n/10)+d;     // watch the order of
      printf("+%d",d);       // these two statements
      return s;
   }
}
int main(void)
{
   printf("=%d\n",sum(12345));
}
```

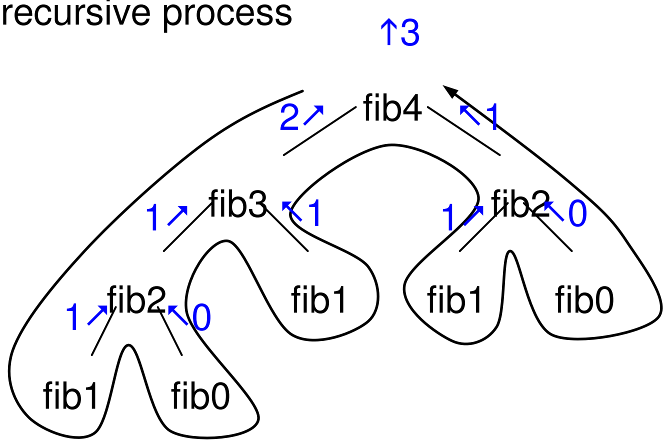
Tree recursion

- Example – Fibonacci numbers

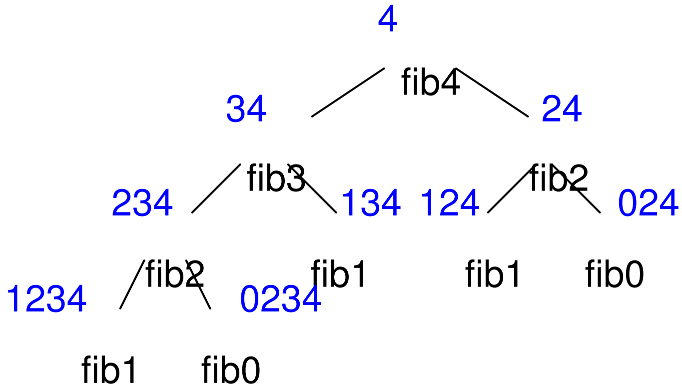
$$\mathrm{fib}(n) = n \qquad\qquad n \le 1$$
$$\quad = \mathrm{fib}(n-1) + \mathrm{fib}(n-2) \qquad n > 1$$

```
unsigned fib(unsigned n)
{
   return n<=1?n:fib(n-1)+fib(n-2);
}
```

Tree recursive process



Runtime stack

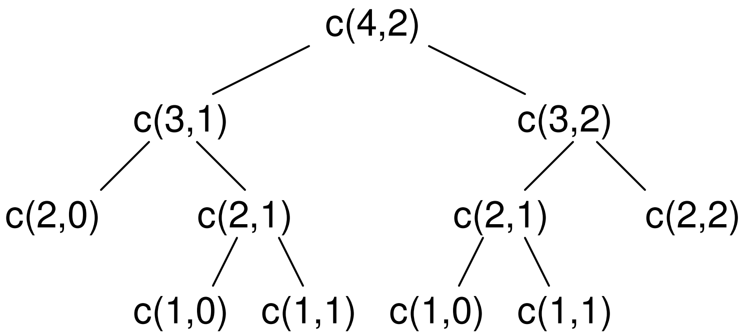


- Example – Combination generation

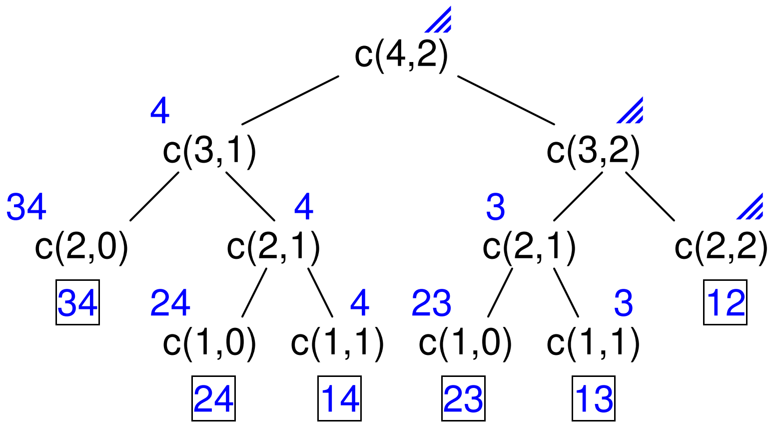
$$c(n,k) = 1 \qquad\qquad k = 0 \text{ or } n = k$$
$$\qquad = c(n-1,k-1) + c(n-1,k) \qquad \text{otherwise}$$

```
int c(int n,int k)
{
   return k==0||n==k?1:c(n-1,k-1)+c(n-1,k);
}
```

Tree recursive process



Use a stack to generate all k-combinations



- Example (Cont'd)

  Version A1 – Global stack

```
struct stack {
   int top;
   int stk[10];    // maximum k = 10
};
stack s={-1};      // or, initialize it in the starred line

int c(int n,int k)
{
   if (k==0||n==k) {
      for (int i=1;i<=k;i++) printf("%d",i);
      for (int i=s.top;i>=0;i--)
         printf("%d",s.stk[i]);
      printf("\n");
      return 1;
   } else {
      s.stk[++s.top]=n;           // push
      int r=c(n-1,k-1);
      s.top--;                    // pop
      return r+c(n-1,k);
   }
}
int main(void)
{
//  s.top=-1;                     // *
   printf("%d\n",c(4,2));
//  s.top=-1;                     // redundant
   printf("%d\n",c(5,3));
}
```

  N.B. It is guaranteed that the stack is empty each time `c` is called.

  Version A2 – Local static stack

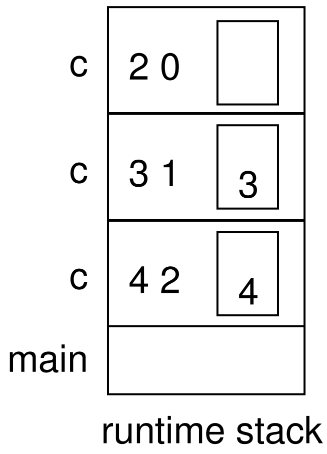
```
int c(int n,int k)
{                                 // NO!
   static stack s={-1};           // static stack s;
   // same code as above          // s.top=-1;
}
```

- Example (Cont'd)

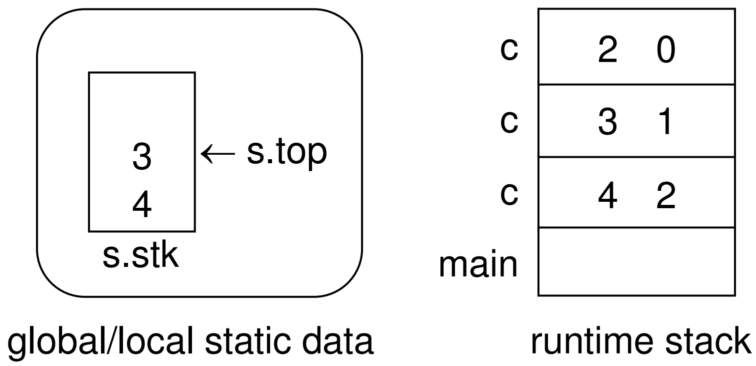
Comments

1 The **`stack`** type itself may also be declared locally.
2 The stack **`s`** is initialized only once.

With local auto stack



With global or local static stack



On global, local auto, and local static variables

Local auto variable
1 Automatic storage duration (lifetime)
2 History insensitive
3 Uninitialized, if w/o initializer

Global/local static variable
1 Static storage duration (lifetime)
2 History sensitive
3 Zero-initialized, if w/o initializer

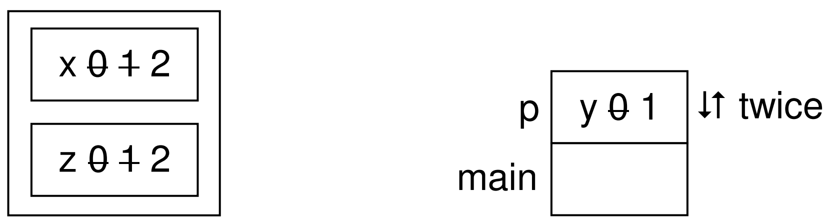
- Example (Cont'd)

In summary,

variable	lifetime	scope
global	static	global
local auto	dynamic	local
local static	static	local

For example,

```
int x=0;                              // global
void p(void)
{
   auto int y=0;                      // local auto
   static int z=0;                    // local static
   x++; y++; z++;
   printf("%d%d%d",x,y,z);            // 111   212
}
int main(void) { p(); p(); }
```



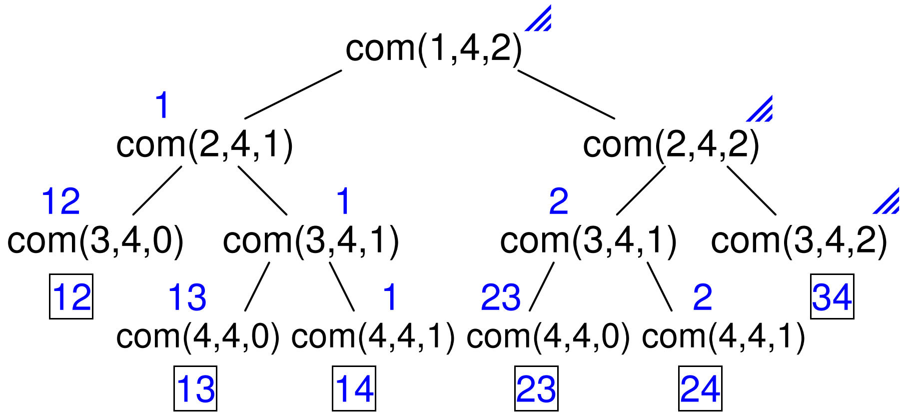
global/local static data   runtime stack

For another example,

```
int a;                  // zero-initialized before main is called
void p(int n)           // usually before main is called
{
   static int b;        // zero-initialized before p is called
   static int c=2;      // initialized before p is called
   static int d=n;      // initialized first time p is called
   int e=n;             // initialized every time p is called
   int f;               // uninitialized
}
int main(void) { p(5); p(6); }
```

- Example (Cont'd)

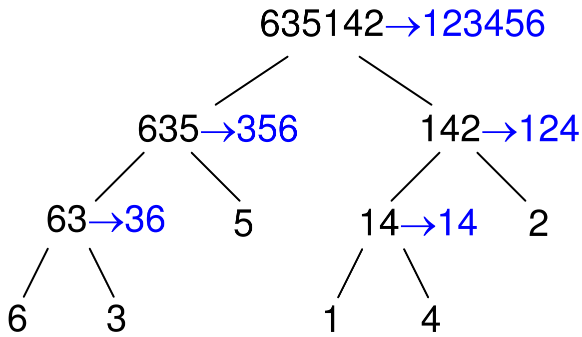
Version B – Generate all k-combinations in lexicographic order




```
int com(int m,int n,int k)
{
   static stack s={-1};
   if (k==0||n-m+1==k) {
      for (int i=0;i<=s.top;i++)
         printf("%d",s.stk[i]);
      if (k!=0)
         for (int i=m;i<=n;i++) printf("%d",i);
      printf("\n");
      return 1;
   } else {
      s.stk[++s.top]=m;           // push
      int r=com(m+1,n,k-1);
      s.top--;                    // pop
      return r+com(m+1,n,k);
   }
}

int c(int n,int k) { return com(1,n,k); }

int main(void)
{
   printf("%d\n",c(4,2));         // remain unchanged
}
```

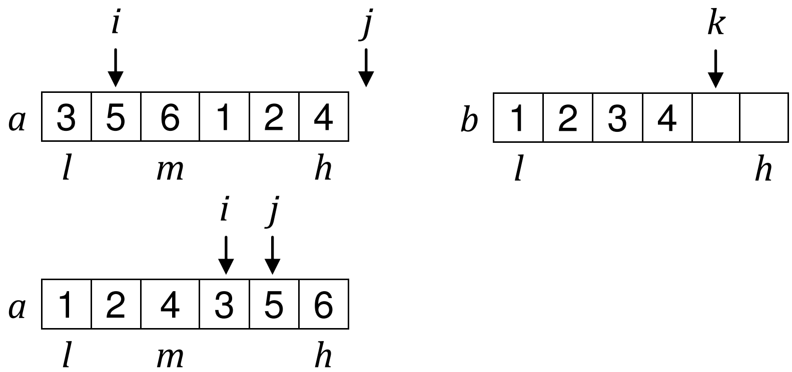
- Example – Mergesort

```
const int sz=20;
int a[sz];

void msort(int l,int h)
{
   if (l<h) {
      int m=(l+h)/2;
      msort(l,m);
      msort(m+1,h);
      merge(l,m,h);
   }
}

int main(void)
{
   // other statements omitted
   msort(0,sz-1);
}
```



Version 1

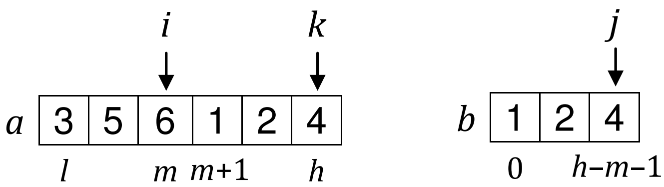
```
void merge(int l,int m,int h)
{
   int b[sz];
   int i=l,j=m+1,k=l;
   while (i<=m&&j<=h)
      if (a[i]<a[j]) { b[k]=a[i]; i++; k++; }
      else { b[k]=a[j]; j++; k++; }
   for (int z=m;z>=i;z--) { j--; a[j]=a[z]; }
   for (int z=l;z<k;z++) a[z]=b[z];
}
```



- Example (Cont'd)

  Version 2 – Half space merging

```
void merge(int l,int m,int h)
{
   int b[sz/2];
   for (int i=m+1;i<=h;i++) b[i-m-1]=a[i];
   int i=m,j=h-m-1,k=h;
   while (i>=l&&j>=0)
      if (a[i]>b[j]) a[k--]=a[i--];
      else a[k--]=b[j--];
   while (j>=0) a[k--]=b[j--];
}
```



Time complexity

Let $t(n)$ = the worst-case time taken by mergesort on n elements
Then,

$$
\begin{aligned}
t(n) &= 1 && n = 1 \\
&= t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + O(n) && n > 1
\end{aligned}
$$

It can be shown that $t(n) = O(n \log n)$

Balanced divide-and-conquer

Mergesort divides the problem into two balanced subproblems, i.e. the sizes of the two subproblems equal or differ by 1.

Q: Why don't we divide it into subproblems of size $n/3$, $2n/3$ or, $n/3$, $n/3$, $n/3$, and so on?

A: It turns out that dividing it into two balanced subproblems is the best strategy.

- Example (Cont'd)

  An extreme unbalanced case: a single-element subproblem
  Observe that this amounts to insertion sort.
  In this case,

$$
\begin{aligned}
t(n) &= 1 & n = 1 \\
&= t(1) + t(n-1) + O(n) & n > 1
\end{aligned}
$$

  It is easily seen that $t(n) = O(n^2)$.

- Example – Quicksort

x	

$\Rightarrow$
partition

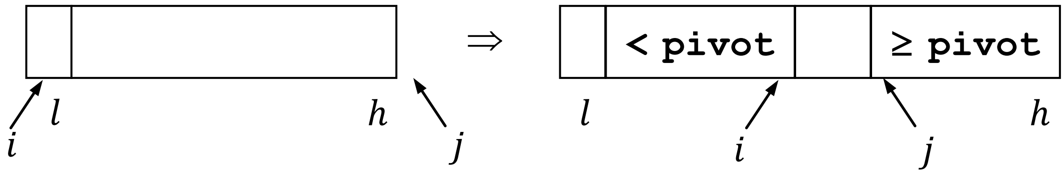
$< x$	x	$\geq x$

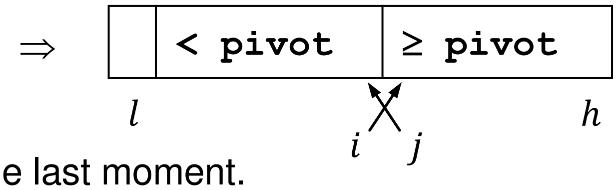
```
void qsort(int l,int h)
{
   if (l<h) {
      int m=partition(l,h);
      qsort(l,m-1);
      qsort(m+1,h);
   }
}
```

Version 1

```
int partition(int l,int h)
{
   int i=l,j=h+1,pivot=a[l];
   while (i!=j+1) {
      do i++; while (i<=h&&a[i]<pivot);
      do j--; while (j>=l+1&&a[j]>=pivot);
      if (i<j) { int z=a[i]; a[i]=a[j]; a[j]=z; }
   }
   a[l]=a[j]; a[j]=pivot;
   return j;
}
```

- Example (Cont'd)

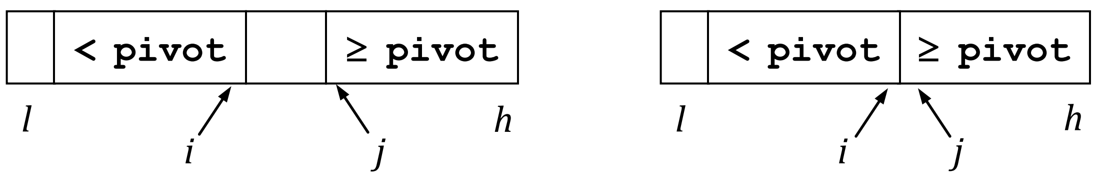




This invariant lasts until the last moment.
In particular, i and j will cross the boundary in case the pivot is the largest or smallest element, respectively.

Version 2 – Lomuto's partitioning algorithm

```
int partition(int l,int h)
{
   int i=l,j=h+1,pivot=a[l];
   while (i+1!=j)
      if (a[i+1]<pivot) i++;
      else {
         j--; int z=a[i+1]; a[i+1]=a[j]; a[j]=z;
      }
   a[l]=a[i]; a[i]=pivot;
   return i;
}
```



The invariant lasts forever for Lomuto's partitioning algorithm. In particular, i and j will never cross the boundary.

- Example (Cont'd)

Comment

Lomuto's partitioning algorithm can easily be modified to partition the array into three regions:

x	

$\Rightarrow$
partition

$< x$	$= x$	$> x$

Time complexity

Let $t(n) =$ the worst-case time taken by quicksort on n elements

$$
\begin{aligned}
t(n) &= 1 & n &= 1 \\
&= \max_{0 \le k \le n-1} (t(k) + t(n-k-1)) + O(n) & n &> 1
\end{aligned}
$$

It can be shown that $t(n) = O(n^2)$.

Randomized quicksort – $O(n \log n)$ excepted running time

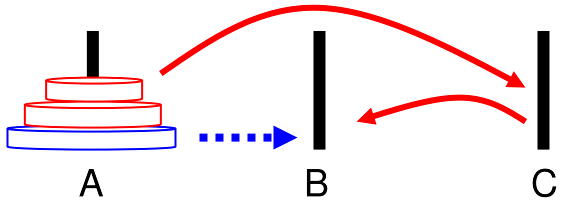
```
int partition(int l,int h)
{
   int p=l+rand()%(h-l+1);
   int z=a[l]; a]l]=a[p]; a[p]=z;
   // proceed as usual
}
```

Deterministic vs randomized algorithms

Deterministic algorithms
1 Have best-case or worst-case inputs
2 For a particular input, the algorithm's behavior is reproducible.

Randomized algorithms
1 All input cases are equal – there are no best-case or worst-case inputs, only "lucky or unlucky probability"
2 For a particular input, the algorithm's behavior isn't reproducible.

- Example – Towers of Hanoi



Divide and conquer

To solve the problem n, $A \to B$, C, solve the three subproblems

1 $n-1$, $A \to C$, B
2 1, $A \to B$, C
3 $n-1$, $C \to B$, A

Why cannot one solve these three "subproblems"?

1 1, $A \to C$, B
2 $n-1$, $A \to B$, C
3 1, $C \to B$, A

```
void hanoi(unsigned n,char a,char b,char c)
{
   if (n>0) {
      hanoi(n-1,a,c,b);
      printf("%c -> %c\n",a,b);
      hanoi(n-1,c,b,a);
   }
}
```

or, assume that $n \geq 1$

```
void hanoi(unsigned n,char a,char b,char c)
{
   if (n==1) printf("%c -> %c\n",a,b);
   else {
      hanoi(n-1,a,c,b);
      printf("%c -> %c\n",a,b);
      hanoi(n-1,c,b,a);
   }
}
```

- Example – Permutation generation

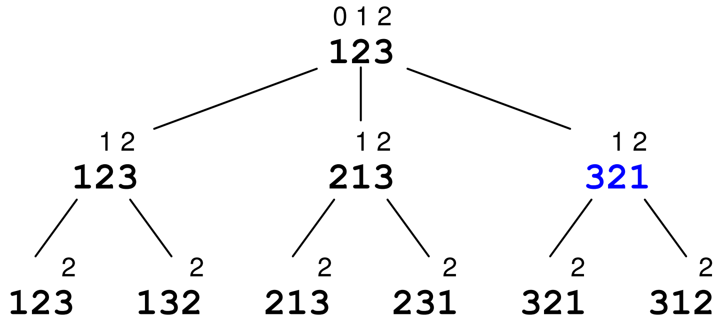
$$p(\boldsymbol{a_0}) = \boldsymbol{a_0}$$

$$\begin{aligned} p(a_0 a_1 a_2 \cdots a_n) = \ & a_0\, p(a_1 a_2 \cdots a_n) + \\ & a_1\, p(a_0 a_2 \cdots a_n) + \\ & a_2\, p(a_1 a_0 \cdots a_n) + \\ & \cdots \\ & a_n\, p(a_1 a_2 \cdots a_0), \qquad n > 0 \end{aligned}$$

```
int a[10];        // at most 10 objects allowed
```

Version A – generate $\mathrm{a}[0..i-1]$ + any permutation of $\mathrm{a}[i..n]$

```
void perm(int i,int n)
{
   if (i==n) {
      for (int j=0;j<=n;j++) printf("%d ",a[j]);
      printf("\n");
   } else {
      perm(i+1,n);        // swap(a[i],a[i]) skipped
      for (int k=i+1;k<=n;k++) {
         int z=a[i]; a[i]=a[k]; a[k]=z;
         perm(i+1,n);
         z=a[i]; a[i]=a[k]; a[k]=z;  // restore a[i..n]
      }
   }
}
int main(void)
{
   for (int i=0;i<10;i++) a[i]=i+1;
   perm(0,2);
}
```



- Example (Cont'd)

$$p(\boldsymbol{a_0}) = \boldsymbol{a_0}$$

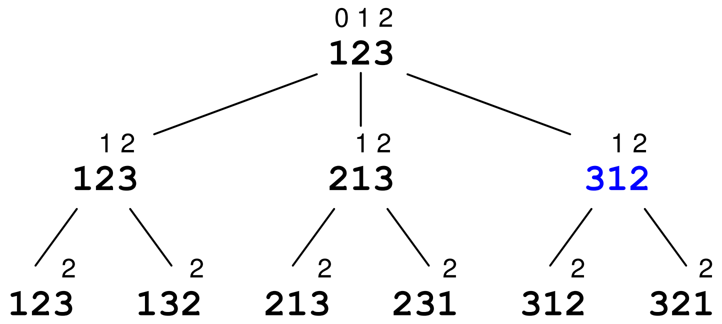
$$\begin{aligned} p(a_0 a_1 a_2 \cdots a_n) = {} & a_0 p(a_1 a_2 \cdots a_n) + \\ & a_1 p(a_0 a_2 \cdots a_n) + \\ & a_2 p(a_0 a_1 \cdots a_n) + \\ & \cdots \\ & a_n p(a_0 a_1 \cdots a_{n-1}), \qquad n > 0 \end{aligned}$$

If $a_0 < a_1 < a_2 < \cdots < a_n$, the permutations will be generated in lexicographic order.

Version B – lexicographic order

// Precondition: $\mathrm{a}[i..n]$ in increasing order

```
void perm(int i,int n)
{
   if (i==n) {
      for (int j=0;j<=n;j++) printf("%d ",a[j]);
      printf("\n");
   } else {
      perm(i+1,n);
      for (int k=i+1;k<=n;k++) {
         int z=a[i]; a[i]=a[k]; a[k]=z;
         perm(i+1,n);
      }
      int z=a[i];       // left rotation to restore a[i..n]
      for (int j=i+1;j<=n;j++) a[j-1]=a[j];
      a[n]=z;
   }
}
```



Mutual recursion

- Example – Even and Odd

  Version 1 – Direct recursion

```
bool even(unsigned n)
{
   return n==0? true: !even(n-1);
}
bool odd(unsigned n)
{
   return n==0? false: !odd(n-1);
}
```

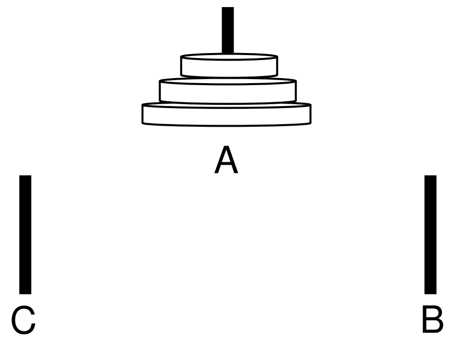
  Version 2 – Mutual recursion

```
bool odd(unsigned);
bool even(unsigned n)
{
   return n==0? true: odd(n-1);
}
bool odd(unsigned n)
{
   return n==0? false: even(n-1);
}
```

  Version 3 – Mutual recursion

```
bool odd(unsigned);
bool even(unsigned n)
{
   return !odd(n);
}
bool odd(unsigned n)
{
   return n==0? false: even(n-1);
}
```

- Example – Cyclic Towers of Hanoi

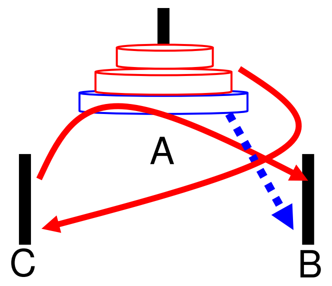


Version 1 – Redundant moves

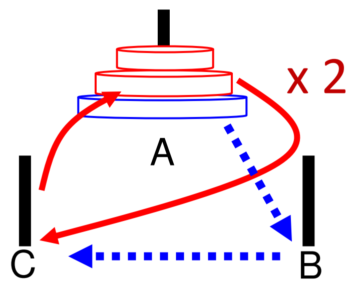
```
void hanoi(unsigned n,char a,char b,char c)
{
   if (n>0) {
      hanoi(n-1,a,b,c);
      hanoi(n-1,b,c,a);
      printf("%c -> %c\n",a,b);
      hanoi(n-1,c,a,b);
      hanoi(n-1,a,b,c);
   }
}
```

Version 2 – Optimal

```
void hanoi2(unsigned,char,char,char);
void hanoi1(unsigned n,char a,char b,char c)
{
   if (n>0) {
      hanoi2(n-1,a,c,b);
      printf("%c -> %c\n",a,b);
      hanoi2(n-1,c,b,a);
   }
}
```



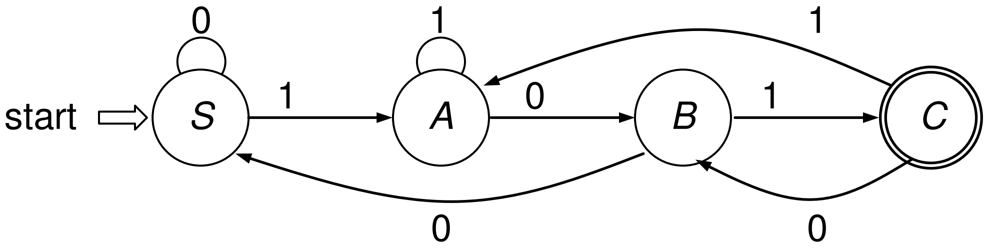

```
void hanoi2(unsigned n,char a,char c,char b)
{
   if (n>0) {
      hanoi2(n-1,a,c,b);
      printf("%c -> %c\n",a,b);
      hanoi1(n-1,c,a,b);
      printf("%c -> %c\n",b,c);
      hanoi2(n-1,a,c,b);
   }
}
```



Indirect recursion

- Example

  Deterministic finite automaton M




```
int main(void)
{
   printf("Enter a binary string: ");
   int ch;
   while ((ch=getchar())!=EOF) {
      ungetc(ch,stdin);
      void S(void); S();
      printf("Enter a binary string: ");
   }
}

void S(void)
{
   switch (getchar()) {
   case '0': S(); return;
   case '1': void A(void); A(); return;
   case '\n': printf("Rejected\n"); return;
   }
}

void A(void)
{
   switch (getchar()) {
   case '0': void B(); B(); return;
   case '1': A(); return;
   case '\n': printf("Rejected\n"); return;
   }
}
```

- Example (Cont'd)

```
void B(void)
{
   switch (getchar()) {
   case '0': S(); return;
   case '1': void C(); C(); return;
   case '\n': printf("Rejected\n"); return;
   }
}

void C(void)
{
   switch (getchar()) {
   case '0': B(); return;
   case '1': A(); return;
   case '\n': printf("Accepted\n"); return;
   }
}
```

Comparison with the goto version (See lecture on statements)

1 Goto version

$$\text{main} \xrightarrow{\text{call}} \text{dfa}(q0) \overbrace{\to q1 \to q2 \to q3 \to q2 \to q3}^{\text{goto}}$$

return

2 Indirect recursion version

$$\underbrace{\text{main} \leftrightarrows S \leftrightarrows A \leftrightarrows B \leftrightarrows C \leftrightarrows B \leftrightarrows C}_{\text{each } \leftrightarrows \text{ is a call and a return}}$$

Recursion vs Iteration

- Iteration
  Based on lower-level assignments (and iteration statements)
  Run faster and use less space

- Recursion
  Based on higher-level concepts, e.g. mathematical definitions
  Runtime and space overheads (of maintaining the runtime stack)

- Example

  Iterative version – O(n) time and O(1) space

```
unsigned f(unsigned n)
{
   unsigned r=1;
   while (n>0) { r*=n; n--; }
   return r;
}
```

f	n	~~3~~ ~~2~~ ~~1~~ 0
	r	~~1~~ ~~3~~ ~~6~~ 6

  Recursive version – O(n) time and O(n) space

```
unsigned f(unsigned n)
{
   return n==0?1:n*f(n-1);
}
```

f	n	0
f	n	1
f	n	2
f	n	3

  The iterative version runs faster, within a constant factor, than the recursive version.

  On the other hand, the recursive version is more mathematics-oriented and hence less error-prone.

- Recursion = Iteration, in computational power

  Rule of thumb Whenever it is easy to express iteratively, do so; otherwise, express it recursively.

  Example
  Iteration   merge, partition
  Recursion msort, qsort